The Appsmiths: Community, Identity, Affect and Ideology among Cocoa Developers from NeXT to iPhone

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

In Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy|

by

Hansen Hsu

May 2015

THE APPSMITHS: COMMUNITY, IDENTITY, AFFECT AND IDEOLOGY AMONG COCOA DEVELOPERS FROM NEXT TO IPHONE

Hansen Hsu, Ph.D.

Cornell University 2015

This dissertation is an ethnographic study, accomplished through semi-structured interviews and participant observation, of the cultural world of third party Apple software developers who use Apple's Cocoa libraries to create apps. It answers the questions: what motivates Apple developers' devotion to Cocoa technology, and why do they believe it is a superior programming environment? What does it mean to be a "good" Cocoa programmer, technically and morally, in the Cocoa community of practice, and how do people become one? I argue that in this culture, ideologies, normative values, identities, affects, and practices interact with each other and with Cocoa technology in a seamless web, which I call a "techno-cultural frame." This frame includes the construction of a developer's identity as a vocational craftsman, and a utopian vision of software being developed by millions of small-scale freelance developers, or "indies," rather than corporations. This artisanal production is made possible by the productivity gains of Cocoa technology, which ironically makes indies dependent on Apple for tools. This contradiction is reconciled through quasi-religious narratives about Apple and Steve Jobs, which enrolls developers into seeing themselves as partners in a shared mission with Apple to empower users with technology. Although Cocoa helps make software production easier, it is not a deskilling technology but requires extensive learning, because its design heavily incorporates patterns unfamiliar to many programmers. These concepts can only be understood holistically after learning has been achieved, which

means that learners must undergo a process of conversion in their mindset. This involves learning to trust that Cocoa will benefit developers before they fully understand it. Such technical and normative lessons occur at sites where Cocoa is taught, such as the training company Big Nerd Ranch. Sharing of technical knowledge and normative practices also occurs in the Cocoa community, online through blog posts, at local club meetings, and at conferences such as Apple's WWDC, which help to enroll developers into the Cocoa techno-cultural frame. Apple's relationship with developers is symbiotic, but asymmetrical, yet despite Apple's coercive power, members of the Cocoa community can influence Apple's policies.

BIOGRAPHICAL SKETCH

Hansen Hsu received a B.S. in Electrical Engineering and Computer Science from the University of California, Berkeley in 1999. That year, he joined Apple, Inc. as a software engineer in the Mac OS 9 group. In 2000 he transferred to the Cocoa framework group within the Mac OS X division to work as a Quality Assurance Engineer, contributing to the releases of OS X 10.0 through 10.4 over the next five years. He left Apple in 2005 to study History at the State University of New York, Stony Brook, where he received an M.A. in 2007. With this doctoral dissertation he completes his Ph.D. in Science and Technology Studies at Cornell University.

I dedicate this dissertation to my late mother, Wen Chen Hsu, who always believed in me.

ACKNOWLEDGEMENTS

There are many people without whom I could not have finished this dissertation. I must first thank the members of my special committee, especially my chair, Trevor Pinch, who has been an enthusiastic supporter of me and my work from the moment I arrived at Cornell, provided a positive framework and environment that allowed me to succeed, and who I feel honored to have been taken under his guidance. Likewise, I thank Ronald Kline for being an interlocutor helping me to formulate a larger argument and reminding me to connect my work with key historical literatures, and for providing me with a teaching assistantship for his course, *Inventing an Information Society*, which provided new insights into my project. I thank Rachel Prentice for pushing me hard to think in new, difficult directions, which has greatly improved my work. Last but not least, special thanks must go to Phoebe Sengers, whose generous mentorship, encouragement, and support came at a critical juncture during the writing of the dissertation, and has always made herself available to me no matter how busy her schedule is. Phoebe is also responsible for getting me connected to the network of scholars in Culturally Embedded Computing (CemCom) and the Intel Science and Technology Center (ISTC) for Social Computing, for providing me with funding through her ISTC grant, and for letting me work in the Goldman Social Computing Lab in Gates Hall. Each of my advisors has helped me in innumerous ways, intellectually and personally, and I could not have imagined a more perfect committee to work with. It has been a blessing to work with each and every one of them.

I also wish to thank the many faculty of the Departments of Science & Technology Studies, Communication, Information Science, and Engineering who have provided help and support over the years. I especially thank Bruce Lewenstein, who despite his duties as Chair of the Department of Science & Technology Studies took the time to be the field appointed reader for my dissertation defense. I must also thank Steve Jackson, Tarleton Gillespie, Michael Lynch, Malte Ziewitz, Rebecca Slayton, Park Doing, Suman Seth, Stephen Hilgartner, Sara Pritchard, Peter Dear, Christine Leuenberger, Rachel Maines, and Judith Reppy, whose help and feedback have guided me throughout

my time at Cornell. I also thank the postdoctoral researchers and visiting scholars in S&TS with whom I have had the pleasure of interacting with, including Johanna Crane, Vivian Choi, Jenni Lieberman, and Annalisa Salonius.

I thank the community of Cornell graduate students in Science & Technology Studies with whom I have had the privilege of being part of. My cohort, Tyson Vaughn, Anto Mohsin, and Ilil Naveh-Benjamin, has been with me from the beginning, and we have traveled this journey together. I must also thank the members of our dissertation support group, which has included Angie Boyce, Megan Halpern, and Hrönn Holmer. They have provided key emotional and intellectual support through the early stage of the dissertation writing process. I must credit Hrönn for coming up with the term "Appsmiths," which I have used as the title of my dissertation. Other colleagues in Science & Technology Studies have been important friends over the years: Benjamin Wang, Kasia Tolwinski, Emma Zuroski, Ling-Fei Lin, Honghong Tinn, Nicole Nelson, Harald Kliems, Alexis Walker, Carmen Krol, Bahar Akyurtlu, Darla Thompson, Rob Schombs, Victor Marquez, Janet Vertesi, Lisa Onaga, Anna Geltzer, Hannah Rogers, Kathryn de Ridder-Vignone, Katie Proctor, Owen Marshall, Shoan Yin Cheung, Enongo Lumumba-Kasongo, Christopher Hesselbein, Danya Glabau, Jessy Price, Lisa Avron, Mehmet Ekinci, Jess Polk, and Ranjit Singh. I also thank my fellow Goldman labmates for their conversation and companionship, including Samir Passi, Stephanie Steinhardt, Laewoo (Leo) Kang, and Ishtiaque Ahmed, and the extended Cornell ISTC and CemCom family, including Kaiton Williams, Caroline Jack, Stephanie Santoso, Andreas Kuehn, Vera Khovanskya, Maggie Jack, Nick Knouf, Lucian Leahu, and Jofish Kaye. I also thank additional graduate student colleagues I have had the pleasure to work with at Cornell over the years, including Tony Liao, Matt Bernius, Stephen Purpura, Josh Braun, Dima Epstein, and Ellan Fei-Spero.

I thank the Intel Science and Technology Center for Social Computing, for providing funding through a research assistantship, and additional funding for travel and audio transcription. I especially want to thank Mel Gregg at Intel, with whom I had key conversations that shaped my dissertation in the final stages. I also need to thank the

Charles Babbage Institute, Thomas Misa and Jeffrey Yost for providing the Arthur L. Norberg Travel Fund Award in 2011, which allowed me to pursue archival research at the Institute. I thank as well the Special Interest Group, Computing, Information, and Society (SIGCIS) of the Society for the History of Technology (SHOT), for awarding me the MIT Press Travel Award in 2009 and the UW-Milwaukee School of Information Studies Information History Travel Award in 2010 to attend the SIGCIS session of SHOT, and I thank Thomas Haigh especially for working with me on my paper submissions to SIGCIS and being a generous and effective organizer of the group and the history of computing community. I also thank Atsushi Akera, Nathan Ensmenger, and Alex Bochannek for welcoming me into this community.

I thank my advisors Wolf Schäfer, Iona Man-Cheong, and Don Ihde at Stony Brook University for teaching me how to be a scholar, and Cynthia Kauffman at De Anza College for providing me with the encouragement to pursue academia and leave my industry job.

Without the generous time donated by the many Cocoa and iOS developers participating in my study, this project could not have been possible. I must thank most of all Aaron Hillegass, who gave me unfettered access to his company, the Big Nerd Ranch, where a significant proportion of the fieldwork for this dissertation was done. I also thank the many Big Nerd Ranch employees with whom I worked or interviewed for welcoming me into their lives, including Mikey Ward, Step Christopher, Emily Herman, Nate Chandler, Adam Preble, Joe Conway, Jeremy Sherman, Mark Dalrymple, Christian Keur, Brian Turner, Brian Hardy, Bill Phillips, Eric Jeffers, Owen Mathews, Steve Sparks, Bolot Kerimbaev, Jason Russell, Jami Siedler, Jaye Liptak, Stacy Moore, Agnes Mackintosh, Charles Brian Quinn, Mark Fenoglio, Chris Stewart, Andrew Lunsford, Alex Silverman, Brian Harper, Brandy Porter, James Majors, as well as Mark Sanchez and Michael Ledford of topsOrtho. In addition, I must thank all of my participants who agreed to set aside time in their busy days to conduct interviews with me, including the following Atlanta CocoaHeads and iOS Developer Meetup members, including Robert Walker, Rusty Zarse, Brandon Alexander, Jonathan Freeman, Joe DeCarlo, Mark Rhodes,

Michael Ayers, and R.D. Willhoite. Critical to my study was the participation of the Seattle Xcoders, NSCoder night attendees, and other Seattle-area Cocoa and iOS developers, including Ken Case, Tim Wood, Curt Clifton, Jake Carter, Luke Adamson, Brent Simmons, Gus Mueller, Chris Parrish, Hal Mueller, Daniel Pasco, Bill Moorhead, Michael Burford, Pam DeBriere, Michael Swanson, Michael Eisses, Joe Heck, Jason Lust, Chris Livdahl, Hasan Edain, and many others. I must also thank important developers who I interviewed in the Bay Area, or remotely over iChat, or elsewhere, including Wil Shipley, Dan Wood, Michael B. Johnson, Mike Lee, Tristan O'Tierney, Chris Clark, Sean Heber, Kevin Avila, Layton Duncan, Jonathan "Wolf" Rentzsch, Jonathan Saggau, Craig Hockenberry, Gedeon Maheux, Tim Novikoff, and Ge Wang. I especially want to thank the former Apple employees who participated in my project, including Becky Willrich, John Randolph, Steve Naroff, Blaine Garst, James Dempsey, and Julie Zelinski. Lastly, I must thank Andrew Stone for his enthusiasm, kindness, and hospitality in letting me stay at his home in Albuquerque. I thank my former coworkers at Apple, including Ali Ozer, Mike Engber, Chris Parker, Kristin Forster, Chris Kane, Aki Inoue, Doug Davidson, Mark Piccerelli, Scott Herz, Vince DeMarco, Amul Goswamy, and many others, for introducing me to the Cocoa community in the first place.

I thank Meg Gourley for her hard work helping me transcribe my interviews.

Finally, I must thank my family, including my sister Hana and especially my father, Po Choo Hsu. My father has always been supportive of my decision to leave the technology industry and pursue humanistic scholarship, and without his moral and financial support during these difficult past two years, this dissertation would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

INTRODUCTION

A June 15, 2014, article in the *New York Times* covered Apple's latest Worldwide Developer Conference (WWDC) and the eager anticipation surrounding possible iPhone announcements by Apple CEO, Tim Cook. At the conference, Cook announced the latest version of the iPhone and iPad operating system, iOS 8, calling it "the biggest release since the launch of the App Store." (Richtel and Chen 2014) The mainstream press, and the general public, hoping for news about the rumored Apple smartwatch, was disappointed. However, the Times noted that the big announcement at WWDC 2014 was "not a consumer product, but a set of software tools called a developer's kit, which would help developers build better apps. If the rest of the world yawned, the developers stood, and whooped." To the primary audience of the conference, third party software developers who write applications for the iPhone, iPad, and the Macintosh, WWDC 2014 appeared to be one of the most important in years. Apple was finally going to allow iOS developers a way to create application "extensions," allowing them to customize the behavior of other apps or parts of the system, a capability that Google's competing mobile operating system, Android, had had for years. In addition, Apple announced two new toolkits, HeathKit and HomeKit, to provide centralized, coordinated ways for apps to securely track a user's health and fitness data, or to automate the home. What energized the audience of developers most, however, was Apple's release of a new programming language that it had developed, called Swift, to replace the venerable Objective-C language, which it had relied on for over a decade.

The third party developers of software for particular computer operating systems and platforms, such as Apple's iOS and Mac OS X, are strategically important to their usefulness to users and their success in the market competition between computing platforms When Apple opened up the iPhone in 2008 to third party application development, it catalyzed a frenzy of interest in developing mobile applications, not just on its own platform, but also on the rival Android platform as well. The number of apps available on the App Store grew from about 10,000 in fall of 2008 to well over a million today, as of June 17, 2014 (Steel Media Ventures 2014). By making available a software development

kit, Apple has tapped into the ingenuity of third party developers, who have created apps extending the iPhone and iPad with functions far beyond Apple's original designs. For this reason, the additional capabilities Apple provides for its third party developers can have more significant long-term impact on users' experiences and the technology industry than the features it develops itself. The stakes of developer support are enormous—Microsoft, which has depended on strong developer support of its desktop Windows platform, has struggled with its mobile Windows OS due to a dearth of applications. And despite Android's lead in smartphone marketshare, many mobile developers still prefer to target iOS first, because of the difficulty of targeting the many different versions of the Android OS in the user base (a problem known as "fragmentation"), fears over piracy, (Dredge 2013) and because it has been much easier to generate revenue on iOS than Android. "Benedict Evans, an analyst with Andreesen Horowitz… [noted] that despite there being more than 1 billion Android users worldwide, developers make less on Android than they do on iOS… Evans concludes, 'Google Android users in total are spending around half as much on apps on more than twice the user base, and hence app ARPU (Average Revenue Per User) on Android is roughly a quarter of iOS.'" (Krakow 2014) Arguably, Apple's success in mobile is in no small part due to the millions of apps available for its App Store. Just as Microsoft's dominance in PC operating systems created a cycle in which more applications were available for Windows, locking in more users, and thus creating increased incentives for developers to make software for it first, the same is now true for Apple's iOS, to Microsoft's dismay. This has played a role in the explosive growth of Apple's iPhone and iPad hardware sales, sending its stock into the stratosphere and turning Apple into the largest company in the world, in terms of market capitalization.[1]

---

[1] The number of available apps on Apple's App Store rose from 35,000 in April 2009 to 1.3 million and counting by September 2014 (Statista 2015b). By October 2014, 85 billion apps had been downloaded from the App Store (Statista 2015c). This contributed to fourth quarter revenue of $4.6 billion in a category that included content sales of music, movies, TV shows, and books from the iTunes media store, as well as sales from AppleCare service warranties, licenses, and other services, and Apple did not break down how much of

Apple's third party developers are an important site of entrepreneurial technological activity, quintessential members of what Richard Florida calls the "creative class" in the knowledge economy, (Florida 2002) who have not only weathered, but actually prospered, in the wake of the 2008 financial crisis. Through Apple's iOS App Store, the apps they create are available to a global user base.[2] Media stories of lone programmers who made overnight riches in the initial years of the App Store generated a new gold rush of entrepreneurial activity in IT rivaling that of the dot.com era, making it a mainstream platform for software development for the first time since the days of the Apple II (Wortham 2009). App development has become a big business.[3] Yet despite all the newcomers looking to strike it rich, the iOS developer community did not spring out of

---

this revenue came from the iOS App Store or the Mac App Store. (Statista 2015a) This represented 11% of Apple's total revenue of $42 billion for the fourth quarter. However, the largest portion of Apple's revenue was iPhone sales, accounting for $23 billion in revenue, with iPad sales at $5.3 billion, trailing Mac computer sales at $6.6 billion. (Apple Inc. 2014) Software sales, while profitable on their own, more importantly increase the value of the iOS platform by maintaining and growing hardware sales, which constitute the majority of Apple's revenue. Driven by strong sales of the iPhone 6, Apple's stock most recently peaked at $119, putting its market capitalization at $700 billion for the first time (Huddleston 2014).

[2] Apple says that 1.4 million apps on the iOS App Store are available in 155 countries (Apple Inc. 2015b).

[3] On January 8, 2015, Apple announced that "customers around the world" had spent "nearly half a billion dollars on apps and in-app purchases" in the first week of January 2015, following a "record breaking 2014" in which "billings rose 50 percent and apps generated over $10 billion in revenue for developers." This $10 billion meant that cumulatively, third party developers have earned over $25 billion from sales on the App Store since it opened in 2008, after Apple's 30% cut. Apple further boasts that "the iOS ecosystem has helped create 627,000 jobs in the US alone," (Apple Inc. 2015b) citing two studies on the "app economy." (Mandel and Scherer 2012; VisionMobile Ltd 2014)

nowhere. These newcomers learned how to write apps for Apple devices from an already existing community of developers that write apps for Apple's desktop operating system, Mac OS X, using the Apple software development kit known as "Cocoa." iOS development uses a technology called "Cocoa Touch," which was developed along the same principles as its desktop Cocoa cousin. Software development using Cocoa, whether its desktop or mobile variants, involves learning a programming language unique to Apple's platforms (Objective-C and now Swift) as well as idioms, design patterns, and ways of thinking about software architecture that are particular to it. The existing developer community's expertise in Cocoa is also bound up with a set of norms, values, and practices about how a developer should write code and design apps. Unlike newcomers to iOS, many of these earlier Mac developers came to the Mac OS X platform at a time when it had less than 10% PC marketshare, and developing software for exclusively for it instead of Windows or the Web seemed like a terrible business decision. For these developers, something more than maximizing profit motivated them to devote their careers to Apple's platform: a pleasurable, affective experience writing software using Cocoa technology that they believed was better than that of any other software development environment. Coupled to an ideology of empowering users through software tools, and a romantic individualist identity[4] as an anti-corporate programmer-entrepreneur as hero-rebel-crusader, these independent or "indie" Mac developers formed a close-knit community in the early 2000s, defining themselves against the corporate programmers of the Microsofts and Adobes of the world.

It was this indie Cocoa/Mac community that transmitted its knowledge, practices, and values to newcomers to iOS, through blogs, tweets, mailing lists, books, classes, podcasts, and conference talks. One expert, Aaron Hillegass, whose Cocoa programming books and courses have trained a generation of Cocoa programmers, has had an enormous influence on prevailing norms and practices. The kinds of apps Cocoa programmers write,

---

[4] Thomas Streeter has written about the influence of romanticism and constructions of the romantic individual (as opposed to a rational utilitarian self) in the culture of computing and the internet, and the role it has played in both the revival of neoliberalism and the development of the open source movement (Streeter 2011).

and how they write them, are powerfully shaped by the culture, values, and practices of the developer community with which they interact and learn their craft, as well as by the capabilities of the Cocoa toolkit itself, and the abstract principles that guide Apple's design of them. These two factors are not independent, but linked—third party Cocoa developers and Apple engineers alike exist in a community of practice that transcends the boundary of Apple the corporation; shared ideas that motivate how developers write apps likewise shape how engineers at Apple write the toolkits developers use to write those apps.

This dissertation is an ethnographic and sociological study of this technological subculture: namely the Cocoa developers, third party software programmers who use Apple's Cocoa technology to make apps. The economic boom in computing technologies since the 1980s has drawn many people to become software developers as a way to make a comfortable living, and others hope to found startups and become fabulously wealthy. Yet, prior to the iPhone, Apple's platform was the minority, and developing for it was not seen in the industry as a clear path to such prosperity. What then motivated developers to hitch their wagons to Apple? The answer, as we shall see, is both affective and ideological. Affectively, Cocoa developers wish to maintain the pleasurable experience of programming with Cocoa technology. Ideologically, Cocoa developers came to believe that their life's vocation was to improve society by making software, and that this is best done independently of corporate or managerial control. This project is about the complex interplay between 1) developers' affective experience with Cocoa, 2) a sense of self as a creative, artistic individual and a productive maker empowered by that experience, 3) the moral and ideological commitments tied to those experiences and subjectivities, 4) the technical practices entailed by these commitments, and 5) the sharing of both these practices and the commitments behind them among a community of like-minded individuals, who wish to preserve and spread the technology, the experience of using it, and the identity, values, and practices associated with

it. Each of these aspects interacts with and constitutes the others, in a "seamless web" in which culture and technology are tightly bound.[5]

What's new about apps?

For all the hype about the mobile app revolution, "apps" themselves are not really all that new. Prior to smartphones, "app" was simply an abbreviation for "application program,"[6] a program that is "applied" to a specific use task, or "application" that a computer user is trying to accomplish. According to PC Magazine, an "application program" is "Software that processes data for the user" ("Application Program" 2015) and can refer to any software that is not "system software," which is "Software used to control the computer and develop and run applications" ("System Software" 2015), and which Martin Campbell-Kelly explains, "included the operating systems and programming aids that all users needed to run their computer installations efficiently and to create their own programs." (Campbell-Kelly 2003, 97) According to Campbell-Kelly, the term "applications" was already in use by IBM as early as 1959 with the Program Applications Library for the IBM 1401, though the library actually included systems programs as well as applications programs. Campbell-Kelly notes that this classification scheme became more formalized in 1964 with the announcement of IBM's System/360: "The distinction between systems software and

---

[5] In the introduction of the "schoolbus book," the seminal *The Social Construction of Technological Systems* (Bijker, Hughes, and Pinch 1987, 9–15), editors Wiebe Bijker, Thomas Hughes, and Trevor Pinch use the metaphor of the "seamless web," derived from Thomas Edison's notebooks, to describe the way society and technology mutually construct each other such that the technical and the social are thoroughly intertwined.

[6] Blogger Thom Holwerda claims that the earliest use of the abbreviation "app" that he could find using Google books dates from the June 8, 1981 issue of Computerworld, in which it can be found in numerous job postings, which were written in the style of telegrams, which abbreviated to save on per-character costs. Later uses include 1985's MacApp, Larry Tessler's object-oriented application development framework for the original Macintosh, among others. (Holwerda 2011)

application programs was a natural division that was used in all subsequent software classification schemes." (Campbell-Kelly 2003, 96–97) Since in some respects the category of "applications" is widely used to designate any software that is not systems software, applications programs have been around since digital electronic stored program computers have been available. In another sense, applications are really what computers and software are all about—the hardware and systems software exist only to provide the medium, the infrastructure, for the computer to run the application programs for which it was constructed or acquired. In this sense, applications are, and have always been, central to computing since before there were computers. Michael Mahoney has argued that the multiple histories of computing are really about the multiple histories of the communities of practice that needed computing for particular end-uses, or in other words, applications—business data processing, numerical calculation, military command and control, industrial automation, etc. (Mahoney 2008, 9)

The selling of software to computer users, both systems software as well as applications, has come in different forms, historically. Originally, all software, applications especially, had to be written by users themselves, often with heavy support and training from computer manufacturers such as IBM, though in the 1950s, IBM and other manufacturers began providing some software tools and utilities. (Campbell-Kelly 2003, 29–31) Also in the 1950s, user groups such as SHARE began to cooperatively develop and share programs. (Campbell-Kelly 2003, 31–34) To protect the proprietary interests of member firms, however, SHARE focused on systems software, especially operating systems, rather than application programs, because applications "could contain important hints about a firm's engineering activities." (Akera 2007, 262) The software industry began with the first software contracting companies (then known as "programming services companies," the first one started in 1955), which developed custom software solutions for clients, generally corporate firms or government bureaus that could afford computers. (Campbell-Kelly 2003, 50–51) Pre-packaged software products, which could be purchased or leased as "a discrete software artifact that required little or no customization" (Campbell-Kelly 2003, 99) appeared in 1964, though the industry really got going when IBM was prompted by antitrust investigations to unbundle its software packages from its hardware, selling it

separately as a product, allowing third party software firms to compete with IBM on a more even footing. (Campbell-Kelly 2003, 109–118) These software products targeted primarily corporate or institutional customers, a market known as "enterprise software," and are marketed and sold directly to organizations like capital goods, because of their high costs in marketing and pre- and after-sale support. (Campbell-Kelly 2003, 6) The ability to buy an application as a commodity by consumers has really only existed since the personal computer created a market for mass-market, shrink-wrapped software products that could be purchased literally "off-the-shelf" at retail stores. This PC software industry developed almost completely independently of the corporate enterprise software industry. (Campbell-Kelly 2003, 208) VisiCalc, the first interactive spreadsheet application, became famous as the personal computer's first "killer app," an application so useful that it singlehandedly drove personal computer adoption (Campbell-Kelly 2003, 212–213).[7]

Are mobile apps really that different from personal computer applications? It can be argued that on smartphones and tablets like the iPhone, the experience confronting a user is different—typically, mobile apps are focused to perform only a limited function, such as show the weather, look up restaurants in the area, or check up on stocks, rather than be the kind of kitchen-sink programs with thousands of functions that desktop applications often are. In this sense, mobile apps are more akin to the little "widgets" available on the Mac OS X dashboard (which perform simple functions like stock checker and calculator), or the small "desk accessory" programs on the classic Macintosh (which also contained a calculator). Certainly, the way an app appears on either iOS or Android, as a singular icon and thus implying a single function, lends to the perception of a different ontology than programs on PC platforms, which may have one primary application but come packaged with many auxiliary helper applications or come in a suite in which multiple applications are designed to interact with each other, as in Microsoft Office. However, from the perspective of a software developer, an app is just another piece of software, albeit maybe

[7] Campbell-Kelly contends that VisiCalc's role as a "killer app" has been over-exaggerated, as he argues that word processing and databases combined with falling hardware costs would eventually have introduced microcomputers into businesses.

less complex because of its more focused purpose. In particular, on Apple's iOS platform, which is derivative of its Mac OS X desktop platform, writing an iOS app is remarkably similar to writing a Mac OS X application, and when the iPhone software development kit was first made available, those programmers with years of experience on Mac OS X had a huge advantage in expertise over newcomers, allowing them to maintain intellectual authority among the Apple developer community. I argue, then, that despite the enormous expansion and change in the makeup of the third party Apple developer community in the wake of the iPhone, there is a significant continuity with the much smaller, prior existing culture of Apple developers who programmed using Cocoa technology on the Mac. This dissertation is in part the story of the continuity of the community and its culture over three decades, through two major transitions, the iPhone being only the latest.

A Brief History of Cocoa

Cocoa was not always the dominant technology for third party Apple developers, or even at Apple itself. Cocoa and its developer community have a long history, which I will only sketch briefly here. Cocoa is a set of software libraries (or frameworks, in Apple's parlance) that make up a software development kit (SDK), made up of interfaces into the operating system that allows developers to build applications. The toolkits that make up Cocoa originated on NeXTSTEP, a Unix based operating system created by NeXT, Steve Jobs' second company, founded after his ouster from Apple in 1985. According to former NeXT employees, Jobs had originally intended NeXT to be a better Apple, but NeXT had never lived up to these expectations, and had shut down its hardware business by 1993. However, NeXTSTEP had acquired a reputation among software developers for dramatically enhancing programmer productivity, and had acquired a loyal following. Apple acquired NeXT in 1997, gaining not only Jobs, but NeXT's operating system and development environment, which eventually became Mac OS X and Cocoa, respectively. This allowed NeXT developers such as Andrew Stone and Wil Shipley, who remained loyal to the NeXT platform despite a diminished market for shrink-wrapped software products post-1993, to begin selling applications to Apple's large installed base of consumers after 2001.

The transition to NeXT-based Cocoa technology, however, was not made without difficulty at Apple. For Mac OS X to succeed, it needed to maintain compatibility with existing Macintosh software, written using a developer kit, known as the Macintosh Toolbox, that had been the way developers programmed for Macs since the original 1984 Macintosh. Existing developers with large codebases, such as Microsoft and Adobe, pushed back against Apple's initial plan to replace the Mac Toolbox with Cocoa,[8] which would require them to learn Objective-C, a programming language no other company used, and throw out years of existing code. In response, Apple created "Carbon," an updated variant of the Mac Toolbox that allowed developers to continue using the tools they were familiar with, without requiring that their applications run in a virtual environment that emulated the original Mac. Carbon was a compromise to appease large corporate software firms, without whose support Apple's operating system transition from the original Mac OS to the NeXT-based Mac OS X would not have succeeded. For five years after the initial release of Mac OS X in 2001, Apple supported both the Carbon and Cocoa toolkits for developing applications, and claimed that they were equals.

Within the third party developer community, old NeXT developers embraced Cocoa and very publicly touted its superiority over Carbon, antagonizing many old-time Mac developers, as we will see later. A few Mac developers, mostly small independent shops whose apps were small and thus without large code-bases, were willing to try Cocoa and were converted. Large corporations had a much harder time. Thus, for first half of the 2000s, the Apple Mac developer community was split between two competing technical cultures: Carbon, which represented the old Macintosh way of doing things, and Cocoa, the hot, new, NeXT-based way of doing things. It was clear among many Apple observers that, given how former NeXT engineers were in charge of all the major engineering organizations at Apple, that eventually Carbon would go away and Cocoa would replace it. Former NeXT

---

[8] The frameworks that were later renamed "Cocoa" were referred to by Apple as the "Yellow Box" in 1997, to differentiate them from the "Blue Box," which ran classic Mac OS inside a virtual machine on the NeXT-based operating system, which went by the code name "Rhapsody" at the time.

developers and Cocoa enthusiasts were egging on Apple to do this sooner, rather than later. (Wil Shipley Interview, April 18, 2012) In 2006, Apple announced at WWDC that in its next version of Mac OS X, it would be updating the Cocoa toolkits to run in the new 64-bit mode, but Carbon would remain in the old 32-bit mode. This was a clear signal that the years of Carbon/Cocoa parity were soon to be over. Carbon developers, who had bought into Apple's older message that Carbon would continue for the foreseeable future, were outraged, a few threatening to abandon the platform for Windows. The remaining holdouts, Microsoft and Adobe, were now forced to adopt Cocoa and Objective-C in future versions of their applications.

Two years later, the question became moot, as the release of the iPhone SDK, based explicitly on Cocoa and Objective-C, created an entirely new market for applications with enormous potential. In 2008, Cocoa and Objective-C knowledge suddenly became lucrative and in-demand. The small community of NeXT/Cocoa experts, who had remained devoted to the platform throughout the 1990s despite having almost no market, was now at the center of a gold rush not seen since the dot.com era, when they had only been peripheral players writing web applications using NeXT's WebObjects toolkit. NeXT/Cocoa developers had originally been a minority even among Macintosh developers, but by 2006, when Apple effectively closed the controversy by fiat, they had become the dominant voices among the Mac developer community. In 2008, their longtime expertise with Objective-C and Cocoa put them at the center of the entire mobile app universe.

The history of the Cocoa developer community can thus be separated into three periods. The first is the period of NeXT, from the initial beta releases of NeXTSTEP in the late 1980s, till Apple's acquisition of the company in December of 1996. This initial period can itself be broken up into two periods, with the first lasting till about 1992 or 1993, when NeXT still had a hardware market and its developers were still trying to make shrinkwrapped consumer applications, and the period from about 1993 through 1997, when most NeXT developers became contractors writing custom software for enterprise customers. The years from 1997 through 2001 were a period of transition, as Apple was working on Mac OS X and periodically released developer preview versions, and NeXT developers got used to being hitched to Apple's wagon. However, the second period did not formally begin

until Mac OS X finally shipped in March 2001. This second period was one of gradual integration, and occasional tension, between the NeXT/Cocoa community and the old Mac developer community, when Carbon was still a viable alternative development environment for the Mac with its own devoted community. This was also the period when independent Mac developers began to call themselves "indies." The third and current period starts roughly between 2007 and 2008 with the release of the iPhone, and the announcement and opening of the App Store, which attracts waves of new developers to Cocoa.

Why Cocoa developers?

So, why study Cocoa developers? As this brief history shows, the technological subculture of Cocoa development began as a very marginal one within the computer industry, at NeXT, a tiny company whose ambitions had never matched up to its reality. A very small but devoted user and developer base existed, but NeXT's niche status made many of them fear that this technology that they loved and believed would be the future might eventually die out in the face of competition from Microsoft. But for an accident of history, this might indeed have been NeXT's fate—Apple was originally considering purchasing the BeOS operating system from another company founded by a former Apple executive, Jean-Louis Gassée. According to Issacson, Gassée "overplayed his hand" by asking for too much money, while Jobs "dazzled" the Apple executives with his presentation of NeXT's technology, and did not overreach in the negotiations. (Isaacson 2011, 297–301) Even after the NeXT acquisition, Cocoa developers might simply have remained developers of Apple's Macintosh platform, which still has only a fraction of the marketshare of Microsoft Windows on personal computers. Apple's transformative entry into mobile computing with the iPhone and iPad, however, has put Cocoa programmers front and center in the new mobile app revolution. What was originally a marginal technological subculture is now extremely powerful, serving as the socio-technical basis for Apple's dominance in mobile computing. This has catapulted many small third party companies that were dependent on the Apple ecosystem, which had formerly been a tiny niche, into new growth and prosperity. These include Aaron Hillegass's Big Nerd Ranch, which had been a virtual obligatory passage point for learning Cocoa programming. Though still a small but growing company, its role and influence in training mobile programmers has been enormous. For example, it

was hired to train Facebook's entire mobile division (both iOS and Android) in 2012, in a move critical for Facebook's successful transition from the web to mobile (Big Nerd Ranch 2014a; Isaac 2013; Schramm 2013).

As a company so much in the public spotlight, Apple, along with its founders Steve Jobs and Stephen Wozniak, has been the subject of many popular books and biographies. (Amelio 1998; Blumenthal 2012; Butcher 1988; Carlton 1997; Cringely 1996a; Deutschman 2000; Esslinger 2014; Freiberger and Swaine 2000; Hertzfeld 2005; Isaacson 2011; Jobs and Beahm 2011; Kahney 2004; Kahney 2009; Kahney 2013; Kane 2014; Kawasaki 1990; Levy 1984; Levy 1994; Linzmayer 2004; Malone 1999; Markoff 2005; Moritz 2009; Sander 2012; Sculley and Byrne 1987; Segall 2013; Stross 1993; Wozniak 2006; Young 1988; Young and Simon 2005) Academic work has included business and management literature treating Apple as a case study, alternately a model for success or a cautionary tale, depending upon the date of publication (Castrogiovanni, Baliga, and Jr. 1992; Crossan, Lane, and White 1999; Gordon 1991; Holder and McKinney 1992; G. D. Hughes 1990; Nonala and Kenney 1991; Richardson and Arthur 2013; Rotemberg and Saloner 2000; Schoemaker 1997; Sullivan 1991; Van Horn 1996). Three works analyze Apple as a cultural phenomenon through a literary perspective (Belk and Tumbat 2005; Campbell and La Pastina 2010; Robinson 2013). This literary analysis of Apple's marketing, press pieces, and online blogs about Apple shows how Apple's discourse uses religious and mythological tropes. La Pastina and Robinson in particular connect this to a historical American discourse that links technological progress to religious transcendence (Noble 1999; Nye 1994; Nye 2003). Belk and Tumbat categorize various narratives about Apple into mythological archetypes: creation, hero, Manichean, and resurrection myths. A fourth work, while not dealing with Apple as a religion, nonetheless also analyzes the autobiography of former Apple CEO John Sculley (and two others, CEOs of IBM and Xerox) as works of a literary genre. The relationship between Sculley and Jobs, involving initial seduction by Jobs and infatuation on Sculley's part, and ending in a betrayal leading to Jobs' ouster, fits the trope of a failed romance (Schoenberger 2001).

None of these works discuss Apple developers or software engineers, however. Gideon Kunda's ethnography of an unnamed computer company (most likely DEC) is

illuminating for discussing the various ways engineering employees buy into or resist corporate ideology, which is inculcated through a combination of persuasive and coercive means (Kunda 1992). While I will draw on Kunda's work in more detail later, in regards to how ideology is disseminated in organizational practice, Kunda'sbook, although quite illuminating for analyzing the ideological commitments of Apple employees, does not fully explain why software developers who do not work for Apple, but who work independently are nevertheless committed to writing software for Apple platforms. Although their livelihoods are deeply tied to Apple, many of them chose to attach themselves to Apple at a time when they could have chosen to write software for more lucrative or marketable platforms. While Apple can and does use a combination of persuasive and coercive means to domesticate its developers, because they are third parties who can leave the platform for greener pastures, as well as publicly criticize Apple when it does something they dislike, Apple must rely much more on persuasive means, and use different channels than it would with its own employees. The work of Robinson, Belk and Tumbat, and Campbell and La Pastina on the "religion" of Apple is useful in this regard, as the commitment of both Apple users and developers to Apple can be seen as a kind of search for transcendence and affective connection through consumption of a cult brand. Apple developers certainly are among Apple's most committed users, and many become developers due to their love of using Apple products. However, developers not only consume technology, they also produce it. Because their technological production is explicitly targeted at Apple's platforms, it is highly dependent on it. Apple's technology, then, matters for what its developers can make, and how. Recent ethnographies of software developers, particularly free software, (Coleman 2013; Kelty 2008; Takhteyev 2012) fit into a growing literature on democratized, participatory, commons-based peer-production and Do-It-Yourself making, of which open source software production has become an exemplar. (Benkler and Nissenbaum 2006; Buechley et al. 2009; "DIY Hardware: Reinventing Hardware for the Digital Do-It-Yourself Revolution" 2009; Lindtner, Bogost, and Bleeker 2014; Tanenbaum et al. 2013) Thomas Streeter has connected this concern of computer enthusiasts with transcendence and creation to a Romantic notion of individual selfhood that sometimes support markets and property rights, and at other times opposes them, but always involves a self motivated more by affect,

feeling, and acts of expression rather than rational calculations of utility. (Streeter 2003; Streeter 2011)

As far as I know, only two other academic works have examined third party Apple software developers. The first is Michiel Van Meeteren's self-published thesis, "Indie Fever." (van Meeteren 2008) This work was first pointed out to me by one of my actors in my first preliminary field study, and many of its findings prefigure my own. Van Meeteren studied the independent (or "indie") Macintosh developer community just prior to the introduction of the iPhone, and describes many of its cultural values (what he calls, "sensibilities"), the structure of the community, and its complicated relationship with, and dependence on, Apple. More recently, Qiu, Bopal, and Il Horn have published a study of iOS developers. (Qiu, Gopal, and Hann 2011) Qiu et. al. find a difference between those developers who follow a "professional logic" aligned with the "sensibilities" of the indie community discussed by Van Meeteren, relying on quality and reputation among their peer developers for marketing, and a "market logic" followed by many developers new to the Apple platform, who pursue a strategy of developing as many low quality, low priced apps as possible to see what hits. They have also observed a gradual synthesis of the two logics, with older developers thinking more about explicit marketing, and newer developers beginning to improve the quality of their apps. Qiu et. al. draw their theoretical frameworks primarily from the institutional logic and professional identity literature, but also speak to literatures on information systems and entrepreneurship. Van Meeteren, while drawing on literatures on communities and networks of practice (Brown and Duguid 2000), is embedded in the field of human and economic geography, and he is concerned theoretically with how the indie community is socially coherent, its agency versus the structure imposed on it by Apple, and what this might say about larger questions on information economies and globalization. Neither Van Meeteren nor Qiu et. al. engage with Science and Technology Studies.

Technology, Culture, and Ideology

What is missing in both Van Meeteren and Qiu et. al.'s work is the role of Apple software technology itself. All of Van Meeteren's interviewees are experienced, committed

Mac OS X developers who use Apple's Cocoa toolkit, not its Carbon toolkit or cross-platform technologies, to write software. Do the properties of Cocoa technology matter to these developers? Indeed, Van Meeteren's informants strongly believe it does, indicating that "they'd rather be sheep farmers than step over to another technology…" (van Meeteren 2008, 22) Likewise, Qiu et. al. discuss iOS developers' values as a "professional logic" opposed to pure "market logic." This professional logic can be mapped onto the kinds of cultural "sensibilities" expressed by the indie developers in Van Meeteren's study. But is there a relationship between these cultural "sensibilities" and the technology of Cocoa itself? Both these works treat Cocoa as a black box because neither of them have the analytical tools to examine the role of the technology. Science and Technology Studies, in particular, the Social Construction of Technology (SCOT) (Bijker, Hughes, and Pinch 1987; Bijker 1995; Collins and Pinch 1998a; Pinch and Trocco 2002; Kline and Pinch 1996; Bijker and Law 1992; MacKenzie 1996; Mackenzie 1990; MacKenzie 2001; Kline 2000; Oudshoorn and Pinch 2003), provides the tools allowing us to open the black box of Cocoa, and take the role of the technology seriously.

I stated earlier that this dissertation is about the seamless web of technological subculture, selfhood, practices, moral ethic and ideology among the Cocoa developer community. A central element of this subculture is the role technology plays for the social group. I describe this role using the concept of "technological frame." (Bijker 1995) Wiebe Bijker defines a technological frame as that which "comprises all elements that influence the interactions within relevant social groups and lead to the attribution of meanings to technological artifacts—and thus to constituting technology… these elements include… goals, key problems, problem-solving strategies (heuristics), requirements to be met by problem solutions, current theories, tacit knowledge, testing procedures, and design methods and criteria." (Bijker 1995, 123) This notion of "technological frame" is useful for connecting the cultural and normative aspect of Cocoa developers with the content of the technology into a single socio-technical value system. The technological frame is not purely semiotic but also material, incorporating software artifacts and the larger technological system/infrastructure they are embedded in, as well as the design philosophies behind them.

A useful corollary to the notion of "technological frame" is Bijker's concept of "inclusion," which can be high or low, associated with the relative insider/outsider status of group members and with intellectual/moral commitment to the frame. Members with high inclusion see the technology as unambiguous but differentiated, while those with low inclusion confront a technology as a "take it or leave it" whole, accepted as a given to them or not at all. For those with low inclusion, a technology appears obdurate because it appears to them as monolithic, with no opportunities for their intervention. For those with high inclusion, however, the technology is open to intervention, but it is obdurate in a different way: its meaning is unambiguous (Bijker 1995, 283–5). As we will see, this notion of high and low inclusion can be used to explain the differences between experienced Cocoa Mac developers and recent iOS newcomers.

One problem with the concept of technological frame is that is has a static connotation. The metaphor of "frame" does not easily lend itself to seeing the web of technical-social-cultural relations as evolving. Bijker's original notion of technological frame is of an intellectual understanding that is negotiated between groups in an early stage but then becomes "black boxed" and is likely to remain unchanged over time. I wish to convey the sense that the technological subculture of Cocoa, like subcultures elsewhere, shifts and morphs over time, despite having significant continuity as well. Moreover, I wish to bring in more than just meanings and interpretations of technologies that come from the purely technical, instrumental, problem-solving criteria that "technological frame" seems to encompass. Affective experience, notions of the self, moral commitments, and material, embodied practices are all equally important components of what I will call a "techno-cultural frame."

An alternative theoretical term that could be used to describe the techno-cultural frame of Cocoa is Thomas Kuhn's notion of "paradigm." (Kuhn 1996) A "paradigm" is a coherent set of material practices and conceptual theories held by subgroups of practitioners in a scientific field. Kuhn described science as alternating between long periods of gradual progress following a single paradigm, which he called "normal science," with moments in which the buildup of anomalies within a field become so unsolvable as to constitute a crisis, which can only be resolved through scientific revolutions, when adherents to a radically

new paradigm take over the field. This revolutionary process is inherently social, involving persuasion, conversion, or the dying out of adherents to the older paradigm and their replacement with a new generation. A key claim of Kuhn is that different paradigms constitute incommensurable, or radically incompatible, understandings of the world. He describes a paradigm shift as akin to a Gestalt switch, in which suddenly a person perceives something in a radically different way. While this aspect of paradigms can be useful to explain the conversion experiences programmers go through when learning Cocoa and becoming highly included in its technological frame, if Cocoa programming truly were incommensurable with programming in other environments, it would be almost impossible to learn. While two programming environments may be different, there are often sufficient shared concepts, what Star and Griesemer call "boundary objects," which can be used to translate between the epistemic/social worlds. (Star and Griesemer 1989) Rather, learning Cocoa programming is more a process of being enrolled into a particular epistemic, technical, and moral order.[9] More interesting is the fact that the term "paradigm" and concepts associated with it, such as the "paradigm shift," are in use by the actors themselves. In computer science, programming languages are classified into different families, or "paradigms," in which the style and way of thinking and problem solving afforded by the use of languages in one family is radically different than the others. (Nørmark 2014) Examples of such programming language "paradigms" include procedural programming, object-oriented programming, and functional programming. Arguments over practices and pedagogy, including stakes involving the boundaries of community membership, as we will see in Chapters 4 and 6, often use the language of "paradigms" or of "progress" as rhetorical tools.

Peter Galison has pointed out that Kuhn's radical incommensurability between paradigms leaves no basis for communication or collaboration between different camps of scientists (Galison 1999). And by asserting the primacy of theory, Kuhn's paradigm shift model posits that when theory undergoes a revolution, so too does observation. Galison points out that in physics, the subcultural communities of theorists, experimentalists, and

---

[9] I derive the phrase, "moral and technical order from Chris Kelty (Kelty 2008, 43)

instrumentalists each practice independently of each other, and that breaks in theory do not coincide with breaks in experiment or instrumentation; rather, experimentalists continue to use trusted practices and instruments in order to test the new theory. The concept of mass may indeed mean different things to different theorists; Einstein and Lorentz did not rapidly translate between their own and rival views of mass. Rather, in a localized space, the "trading zone," theorists and experimentalists alike come together to agree on a local working definition in order to communicate across their own subcultures. Although outside the trading zone, the object "mass" may have different meanings to the different parties, inside it the object is stripped of these additional entailments, becoming a kind of simplified "pidgin" language. This is similar to Star and Griesemer's concept of "boundary objects." (Star and Griesemer 1989) Galison draws on the analogy to anthropological studies of traders which converse in a lowest-common-denominator pidgin language that allows two different cultures to coordinate their exchange. Galison's model improves on Kuhn's in that it acknowledges that while there is significant disunity in a scientific discipline, which is composed of many subcultures, these subcultures are able to coordinate and collaborate through pidgin languages and trading zone objects "that carry radically different significance" for each side (Galison 1999, 146). This description can usefully be applied to software programmers who use different programming languages. Programming languages do share many similar features, which in academic computer science may go by standardized names. However, on the level of practice, each language may give a particular concept its own terminology, and moreover each language may implement the concept in a unique way so that it is not an exact equivalent of a similar concept in another language. How then, do programmers who are used to programming with a language in the procedural "paradigm" communicate with those used to the object-oriented "paradigm"? The answer is through boundary objects that, in the domain of trading zones such as classrooms and other pedagogical spaces, have simplified, pidginized meanings, but which, to their own constituencies, have much fuller elaborations that can incorporate incompatible worldviews.

If neither the concepts of "technological frame" nor "paradigm" fully captures the seamless web of technology, affect, identity, and ethics, what can? As I have said already, the "techno-cultural frame" of Cocoa programming practice involves moral and ethical

commitments. Van Meeteren described these as "sensibilities," implying a moral and normative order. This order encompasses identity and modes of being, both individual and communitarian, that are deeply rooted in affect, feeling, commitment, and aesthetic and creative expression, appreciation, and motivation that go beyond rational economic calculation or interest. As a cultural system of values, norms, and beliefs, this is perhaps best described by the term "ideology."

Ideology is a fraught and often imprecise term with a long history in various strands of scholarship, particularly Marxist thought. One of its advantages is its conceptual kinship with religion, (Geertz 1973, 199–200) which it occasionally subsumes. Marx, in *The German Ideology*, located religion, philosophy (both morality and metaphysics), politics and law, and indeed all abstract thought together as "consciousness," which he considered an epiphenomenon ("phantoms" or "echoes") of "real" material-economic processes and interests (Marx 1947). This unfortunately leads to a pejorative definition of ideology as "false consciousness," illusory beliefs as opposed to "science." Raymond Williams characterizes three meanings of ideology in Marxist writing: 1) a system of beliefs characteristic of a social group or class; 2) a system of false or illusory beliefs; 3) the (cultural or symbolic) process of production of meaning and values. Williams notes that often, the first two definitions become conflated—false consciousness is associated with the beliefs of, or promoted by, particular classes to justify existing or revolutionary orders (Williams 1977).

I am not interested in a pejorative concept of ideology as "false consciousness" or "illusion." Without a non-pejorative definition of ideology, it would be impossible to "write works on the ideologies of American businessmen, New York 'literary' intellectuals, members of the British Medical Association, industrial labor-union leaders, or famous economists and expect either the subjects or interested bystanders to credit them as neutral." (Geertz 1973, 200) Rather, I am interested in Williams' third meaning, which is in line with Clifford Geertz's notion of ideology as a cultural system, in which symbolic action works to create meanings that stabilize social order or motivate social action (Geertz 1973). Geertz considers ideologies "systems of interacting symbols… patterns of interworking meanings… [that] transform sentiment into significance and so make it socially available…"

(Geertz 1973, 207) Ideology is more than just an expression of economic interests, it is also a way of repairing or reconciling psychological, social, and cultural "strain." Ideology, like religion, thus draws frequently on metaphor to "symbolically coerce" "discordant meanings" "into a unitary conceptual framework…" (Geertz 1973, 211) Consciousness is itself defined by the construction and manipulation of symbol systems, which pattern and guide human behavior, whether religious, aesthetic, or philosophical (Geertz 1973, 215–7). Whereas science seeks disinterestedness, ideology "names the structure of situations in such a way that the attitude contained toward them is one of commitment. Its style is ornate, vivid, deliberately suggestive: by objectifying moral sentiment through the same devices that science shuns, it seeks to motivate action." (Geertz 1973, 231) Ideology motivates social action by formulating, objectifying, and mobilizing previously private emotions, transforming individual moods into a social force (Geertz 1973, 232).

Hugh Gusterson's work on nuclear weapons scientists uses a non-pejorative notion of ideology in the way I seek to. Drawing on Michelle Rosaldo, he argues that affects are components of the cultural system. Citing Raymond Williams, he says that "we must think of ideologies not only in terms of discourses and ideas but also as 'structures of feeling'—ways of experiencing and living in the world that profoundly reshape our emotions, bodily reflexes, and fantasies as well as our ideas and beliefs." (Gusterson 2008, 42) With this perspective, Gusterson explains the processes and rituals by which people become nuclear weapons scientists. This is not merely a process of learning technical knowledge but a process of ethical reorientation, in which scientists come to accept the central axiom of nuclear weapons, the idea that nuclear weapons are *more* ethical than conventional weapons because they will never be used except as deterrents. Gusterson's work shows that ideologies are not just systems of belief, but involve moral economies, affects, and identity.

The notion of a "moral economy" is related to, and in a way, a subset of, ideology. The term was first used by E.P. Thompson to describe the traditional morality governing what constituted a fair price for bread among eighteenth century English commoners that came into conflict with *laisser-faire* capitalism (Thompson 1971). As used in Science and Technology Studies by Lorraine Daston and Robert Kohler, "moral economy" refers to the system of values and affects that guide work practices among a group. Daston defines a

moral economy as "a web of affect-saturated values that stand and function in well-defined relationship to one another… an organized system that displays certain regularities… a balanced system of emotional forces, with equilibrium points and constraints." (Daston 1995, 4) Kohler's work notes that moral economies are mutually constitutive with material instruments, tools, and practices (Kohler 1994). Gusterson's "central axiom" of nuclear weapons can be described as much as a core part of nuclear weapons' "moral economy" as its "ideology." Inasmuch as the technical culture of Cocoa involves such a "web of affect-saturated values," I will occasionally use the term "moral economy" in this STS-inflected, Dastonian sense, in order to connote how values within the Cocoa culture are arranged in an ordered relation to each other, while I use "ideology" to refer to the larger, overall ethical and affective system, which may be more amorphous, less organized, and contain tensions and contradictions.

Gideon Kunda examines how corporate ideologies function to motivate and sometimes control members of a technical community of practice within a single organization, namely, the corporate firm, based on an ethnography of a large computer company, "Tech," in the 1980s. Kunda also draws on Geertz's definition of ideology as a cultural system of symbols that serve to make sense of the social order. "In corporate contexts such as Tech, [organizational] ideology consists of images of organizational social reality—publicly articulated and logically integrated 'reality claims' concerning the company's social nature and the nature of its members, formulated and disseminated by those who claim to speak for 'the company perspective.'" (Kunda 1992, 52) At Tech, engineers are motivated through the organizational ideology via presentation rituals—meetings and presentations, whether given by top management, in training workshops, or even everyday work group meetings. The official ideology is that the company is like family, with a mission: "its unique blend of business and technological principles provides… a moral purpose; and its economic success and unique social contribution are consistent with the ideological principles of the larger environment—profit, progress, and individualism." (Kunda 1992, 89) The ideology allows control to work as much through normative means via symbolic power rather than coercive means. Engineers are expected to be self-starters, "take initiative" and "do what's right," and discipline is based on peer pressure and

"internalized standards of performance." (Kunda 1992, 90) At presentation rituals, employees shift between role embracement and, during more liminal times of casual conversation prior to, after, or in-between meetings, role distancing (Goffman 1961), which involves taking an "ironic" or "cynical" stance and a self-conscious awareness of the theatricality of the ritual and the undercurrents of tension and power on display. Rather than undermine commitment, however, such role distancing, if properly defined as humor or commonsense, serves to control and preempt dissent, and further serves to support the official ideology of openness and bottom-up decision-making (Kunda 1992, 107–8, 158).

While Kunda focuses on ideology within a single firm or organization, John Seely Brown and Paul Duguid note that knowledge and practices often travel more easily between firms than between departments within firms, and moreover, explain that knowledge travels along networks built by material practice (Brown and Duguid 2001, 204). To explain this, they build on Jean Lave and Etienne Wenger's notion of communities of practice (Lave and Wenger 1991). A community of practice is a community whose membership is based on "participation in an activity system about which participants share understandings concerning what they are doing and what that means for their lives and for their communities." It does not "imply necessarily co-presence, a well-defined identifiable group or social visible boundaries." (Lave and Wenger 1991, 98) For Lave and Wenger, communities of practice are centrally about learning and knowledge sharing of work practices, which simultaneously involves the construction of the identity of the worker as a skilled practitioner. Learning is situated and informal, much of it learned face-to-face from interaction with a master. Thus a major component of learning is how novices or apprentices are socialized into the community through peripheral participation. It is not surprising, given the master/apprentice model, that the examples Lave and Wenger use to illustrate the concept are primarily crafts such as tailoring and butchery. Brown and Duguid expand this notion into the concept of "networks of practice," which "suggest that relations among network members are significantly looser," where members may never know or meet one another (Brown and Duguid 2001, 205). These networks of practice cut across the boundaries of firms, and explain, for instance, how knowledge and acceptance of graphical user interfaces traveled more freely from Xerox PARC to other computer scientists before it

did to other divisions of Xerox. “Common underlying practice… creates social epistemic bonds… If knowledge leaks in the direction of shared practice, it sticks where practice is not shared. People with different practices have different assumptions, different outlooks, different interpretations of the world around them.” (Brown and Duguid 2001, 207) As we will see, the Cocoa community, while amorphous and loosely structured, exhibits aspects of both a network of practice online, in blogs, mailing lists, and Twitter, as well as a community (or perhaps, a network of communities) of practice, composed of local clubs where members are friends, and international conferences where members who otherwise know each other only virtually meet in the flesh. All of the forums that constitute the Cocoa community of practice highlight knowledge sharing as a central activity, knowledge that is both technical and normative: the practices that make a “good” Cocoa developer, and distinguish core, expert members from novices and outsiders.

Ideology has been important to understanding communities of software programmers, particularly those self-styled “hackers” committed to free and open source software (F/OSS). Gabriella Coleman’s ethnography of F/OSS hackers, *Coding Freedom: The Ethics and Aesthetics of Hacking,* is a critical resource in showing how such developers build community, share knowledge, and promote values simultaneously (Coleman 2013). Like Cocoa developers, open source hackers undergo an experience in which they are enrolled into a particular ideology relating programming practices to ethical values (in their case, that code should be “free” as in “speech”), a process central to building their identities as hackers. Also like Cocoa developers, while their day-to-day relations are networked online, they also meet periodically at conferences, whose ritualistic aspect heightens the emotional experience of practicing with likeminded people. Linux hackers similarly to Apple programmers derive affective pleasure from the activity of programming, overcoming the frustration of computers’ obduracy while crafting something useful, often in a blissful, transcendent state of flow. It is the promotion and maintenance of this affective experience, to protect the means of this pleasurable labor from corporate control, that motivates hackers to participate in communitarian governance and legal work. At the same time, hackers also must navigate a tension between elitism, which comes from their ideal of individual meritocracy and ingenuity, (a virtue which is often expressed aesthetically through clever

humor) with the ideal of populism, which causes them to be particularly vigilant over their leaders' perceived abuses of power. Although the ideology of Cocoa programmers is, on first blush, the polar opposite of free software, their similar aesthetic and affective experiences with code shape their identities as heroic, rebellious, meritocratic craftspeople, predicated upon romantic individualism (Streeter 2011), which motivates in both communities a skepticism of large corporations as well as government bureaucracy. Where they differ is on intellectual property, with Cocoa programmers being ambivalent, apologetic, or pragmatic about it (especially with regard to Apple's claims to it), while free software hackers are strongly opposed to it. Cocoa programmers, though they frequently share and make use of open source code, largely create proprietary software for sale, making or hoping to make their primary incomes from it.

Yuri Takhteyev's *Coding Places: Software Practice in a South American City,* (Takhteyev 2012), a recent ethnography of Brazilian software developers, also provides theoretical contributions to understanding programming communities. By examining Rio de Janeiro, a site outside the Euro-American center of programming practice, Takhteyev illuminates how both center and periphery are implicated in constituting a global "world of practice," which he defines as a system of practice in which people, material resources, and symbolic meanings must be "disembedded" (a concept derived from Giddens) from the local circumstances of the center and "re-embedded" in the new locality of the periphery to succeed. From the peripheral vantage point of Brazil, practices and knowledge from Silicon Valley must be reconfigured to fit a Brazilian context, but in order for it to remain "global" and not become provincial, it must still orient towards the center rather than its own locality. For this reason, most code written in Brazil is in English, and the global success of the Brazilian programming language Lua is predicated upon dissemination in English and its orientation towards the needs of Silicon Valley, not Brazilian developers. Thus, in the asymmetric power relations between central localities and peripheral ones, globalization increases the power of the center, often due to the agency of peripheral actors themselves. Globality increases, not decreases, the salience of place. Takhteyev's notion of "worlds of practice," combining material and semiotic elements, encompasses a much more global, interconnected set of communities and networks with a center and peripheries than Lave and

Wenger's more local "communities of practice" or Brown and Duguid's placeless "networks of practice." For my purposes, I take Takhteyev's "world of practice" to describe that shared experiential and epistemic world of all programmers, while I prefer "community of practice" over "networks of practice" to describe the Cocoa community in order to highlight the much deeper social bonds, formed intimately in physical forums, between Cocoa developers. Sharon Traweek defines a community as "a group of people who have a shared past, hope to have a shared future, have some means of acquiring new members, and have some means of recognizing and maintaining differences between themselves and other communities." (Traweek 1988, 6) Cocoa developers, despite their geographic distribution, do indeed fit this definition.

Chris Kelty, who also wrote an ethnography of free software/open source programmers, *Two Bits: The Cultural Significance of Free Software* (Kelty 2008) also offers some useful theoretical insights. It is from Kelty that I derive the term "moral and technical order." (Kelty 2008, 43) Kelty avoids embracing the term "ideology," noting its often pejorative meaning, and that its claim to objectivity can be easily turned upon itself. Kelty prefers Charles Taylor's notion of "social imaginaries," which captures how people imagine moral and social order (Kelty 2008, 40–41). I like how Kelty adds the technical to the mix, but his sense of "moral and technical order" is still based on the Taylorian social imaginary. I prefer ideology to social imaginary because in its non-pejorative formulations, it can accommodate not just the imaginary (in the form of "ideas" or "phantasms") but also affective "structures of feeling" (Williams 1977) and material practices (Althusser 1999).

One way that Kelty notes how the moral/technical order of free software is expressed is through the religious metaphor in programmer discourse. Developers' descriptions of arguments between partisans of rival technologies as "religious" or "holy" wars, are rhetorical moves, according to Kelty. What this metaphor accomplishes is to frame devotion to either side "as a kind of arbitrary theological commitment, at once reliant on a pure rationality and requiring aesthetic or political judgment. Such stories imply that two technologies are equally good and equally bad and that one's choice of sect is thus an entirely nonrational one based on the vicissitudes of background and belief… played out in

dramatic and broad strokes that imply fundamental differences," the exemplar being the platform wars between Apple and Microsoft (Kelty 2008, 67–68).

Kelty's quintessential moral and technical social imaginary is the free software community, which, drawing on Michael Warner's notion of publics and counterpublics, Kelty describes as a "recursive public," a "public that is constituted by a shared concern for maintaining the means of association through which they come together as a public." (Kelty 2008, 28) The public of free software programmers is made possible by and through the technological infrastructure of the Internet and Unix; at the same time, the material and discursive production of the free software community is directed specifically at producing, maintaining, and improving this very same infrastructure that is the condition of its existence. In this way, the free software community is "recursive." This definition of "recursive publics," however, is dependent on a social configuration that is unique to the phenomenon (free software) that Kelty is studying, and limits its usefulness. Developers who do not participate in the production of their own tools and the tools necessary for their communication make up publics, but cannot be said to be recursive. Likewise, other commons-based peer production, such as literary production under the Creative Commons license, is not recursive either—the digital infrastructure is produced by software developers, not the authors who use it. For these reasons, Cocoa developers do not constitute a recursive public.

A different strand of literature dealing with ideology, culture, and computing is that which links computing with the 1960s counterculture and the ideology known alternately as "technoliberalism," "techno-libertarianism," and "cyber-libertarianism." Popular accounts of Steve Jobs' life, Apple, and the personal computer revolution are rife with references to countercultural influences (Freiberger and Swaine 2000; Isaacson 2011; Levy 1984; Markoff 2005; Moritz 2009) Jobs was well known for his love of Bob Dylan and ended his 2005 Stanford commencement speech with a quote from Stewart Brand's Whole Earth Catalog: "Stay Hungry, Stay Foolish." (Jobs 2005) Markoff's *What the Dormouse Said*, in particular, traces the interconnected web of relationships between computer engineers, microcomputer hobbyists, free speech activists, and psychedelic experimenters in the San Francisco Bay Area of the 1960s and '70s (Markoff 2005). Adherents of the New Age

philosophy of Werner Erhard, *est*, appear both at Douglas Engelbart's Augmentation Research Center, where many advances that later influenced personal computing interfaces and networked collaboration were developed, (Bardini 2000, 201–208) as well as at IMSAI, an early microcomputer maker of Altair clones (Freiberger and Swaine 2000, 84, 100–102, 108). Stewart Brand and Ted Nelson were both important figures in making the connection between computing and liberation, in the process changing the meaning of computers from instruments of military and bureaucratic control, to tools capable of liberating and empowering individuals.

In *From Counterculture to Cyberculture* (Turner 2006) Fred Turner locates Steward Brand as the central node in a network that connected engineers with hippies and artists. Turner calls Brand's *Whole Earth Catalog* a "network forum," a space in which people from both the countercultural and computer communities could participate, network, and trade ideas and values. In the pages of the *Whole Earth Catalog,* Brand introduced the cybernetic theories of Marshall McLuhan, Buckminster Fuller, and Gregory Bateson to the counterculture. Fuller's vision of the Comprehensive Designer allowed Brand to imagine individuals solving all the world's problems through the cybernetic mastery of information, becoming "gods." Brand also drew on Bateson's notion of "co-evolution." Brand became heavily involved in what Turner calls, the "New Communalists," the wing of the counterculture that rejected political activism in favor of retreat to rural communes, and an escape into transcendent spiritual experiences. Brand reconfigured the New Communalist figures of the "Cowboy Nomad" and the Native American "Long Hunter" into cybernetic Comprehensive Designers. The key to this reconfiguration was the use of small-scale tools. For Brand, tools created by the military-industrial complex, such as LSD, geodesic domes, and pocket calculators could be appropriated for the goal of liberation and enlightenment. The *Whole Earth Catalog* was created as a bricolage of small-scale tools for would be New Communalists, in which deerskin, calculators, and books by McLuhan stood side-by-side. Turner notes, however, that in turning away from politics into escapism on the communes, the New Communalist movement inadvertently reinforced traditional mores: division of labor took place on traditional gendered lines; rejection of formal governance led to

despotism by charismatic leaders; communes which were composed of fleeing white suburbanites came into conflict with local Latino communities.

By the early 1970s, with the failure of the communes, Brand was looking for a successor to the liberatory dreams of the counterculture. In 1972, he proclaimed computing to be this liberator, the "best [thing] since psychedelics," in his article for *Rolling Stone*, "Spacewar," (Brand 1972) in which he reinterprets the work and play going on at Xerox PARC, which had tight connections to the ARPA-funded computer science community, as a project about bringing computers to the people, in line with the more explicitly political *People's Computer Company*, which provided access to a time-shared minicomputer for free. In 1974, Ted Nelson self-published *Computer Lib/Dream Machines*, a manifesto patterning itself explicitly after the *Whole Earth Catalog*, which asserted that ordinary people can and must understand computers in order to be democratically empowered, and that moreover, computing could be creative and fun. Computers must belong to the people, lest they be disenfranchised by the computer elite (Nelson 1974). Thomas Streeter argues that, by emphasizing fun and creativity, and the exhilarating use of computers for its own sake, Brand and Nelson severed computing from its rationalist and utilitarian association with command and control, and reinterpreted it as a heroic, romantic individualist, artistic pursuit, associated with imagination, creation, and transcendence (Streeter 2011, 44–68).

By focusing on the power of tools, Brand helped to sow the seeds of libertarianism in computing, Turner argues. Brand's Fulleresque cybernetic (male) master of all he surveys was an individual empowered by tools to become "as gods." The New Communalist dream of achieving transcendent, spiritual liberation through psychedelics was similarly individualist. The focus on the heroic individual in Brand's ideology created a blindness when confronted with problems that would be better solved with social cooperation and institutions. In the 1980s and 1990s, Brand created a succession of network forums, physical, textual, social, and for the first time, virtual, on computer networks. These forums, starting with a 1984 Hacker's Conference (attended by both Stephen Wozniak and Richard Stallman, where tensions surrounding intellectual property in software began to surface), on to the online service, the *Whole Earth 'Lectronic Link*, the Global Business Network (GBN), and culminating in *Wired* magazine, continued to equate computing and freedom, but took on

increasingly libertarian overtones. In each of these forums, Brand drew together new networks of people, including EFF founders John Perry Barlow, John Gilmore, and Mitch Kapor, and *Wired* founders Kevin Kelly and Louis Rossetto. WELL members such as Brand, Barlow, and Howard Rheingold become *Wired* contributors. GBN and *Wired* transformed and reconfigured New Communalism to be in line with the pro-business, conservative politics of Newt Gingrich and George Gilder. Corporate IT CEOs were splashed on *Wired*'s cover as "infobahn road warriors," celebrated as the newest incarnation of Brand's Cowboy Nomad/Comprehensive Designer/god.

Thomas Streeter argues that the easy association between the free market and computing and the Internet is a historically contingent construction of the 1980s and 1990s. In the 1980's, the ascendance of neoliberalism during Reagan's presidency and the increasing enclosure of digital information as intellectual property coincided with the introduction of personal computers into peoples' lives. Microcomputers appeared as consumer artifacts for personal use, that individuals could master and conquer. These objects were brought to users by much-celebrated rebel-entrepreneur technologists, legitimating the neoliberal belief that the free market was the source of all technological innovation (Streeter 2011, 69–92). In the mid-1990s, *Wired's* celebration of Netscape sparked the dot.com boom and reframed the meaning of the Internet and the Web in libertarian terms, despite its origins in a publicly funded research network (Streeter 2011, 127–137).

Former *Wired* contributor Paulina Borsook was probably the first to describe the libertarian culture of Silicon Valley as "technolibertarian." Borsook notes that being technolibertarian does not necessarily imply classical libertarianism's focus on individual liberties and aversion to government, though that strain may dominate. Rather, she argues that technolibertarianism is broad enough to encompass a number of worldviews, including "all different colors of free-market/antiregulation/social Darwinist/aphilanthropic/guerilla/ neo-pseudo-biological/atomistic threads." (Borsook 2000, 8) It comes in two forms, a political form, and a philosophical one. Political technolibertarians can easily be registered Democrats or Greens as much as Republicans or Libertarians. Rather, political technolibertarianism can manifest as much as an indifference to government as much as

active hostility to it, in part because technologists see technology development as a much more effective way to effect social change than politics. "Because of their conceptual dismissal of government, technolibertarians typically can't be bothered to engage in conventional political maneuvers—and so as political entities are largely rendered invisible. And because they are invisible, they know little about affecting government." (Borsook 2000, 11) Until the Microsoft anti-trust trial, Bill Gates himself "couldn't be bothered interacting with Congress…"[10] Borsook notes that this dismissal of government is especially egregious in regards to local government, with almost no lobbying or donations to politicians in Santa Clara County up to 2000 (Borsook 2000, 12). She suggests that this disdain of locality is a result of escape into cyberspace: "The desire to slip the surly bonds of earth [virtuality]… means using computers to overcome boundaries of time and space and physical limitations… with the blinding consequences of ingoring the ways, good and real, that we are all grounded in time and space and the realm of the senses. Local politics is all about really being there." (Borsook 2000, 14) On the other hand, Borsook describes philosophical libertarianism as a worldview that in its most "virulent form" is "a kind of scary, psychologically brittle, prepolitical autism. It bespeaks a lack of human connection and a discomfort with the core of what many of us consider it means to be human. It's an inability to reconcile the demands of being individual with the demands of participating in society, which coincides beautifully with a preference for, and glorification of, being the solo commander of one's computer in lieu of any other economically viable behavior." (Borsook 2000, 15) Philosophical technolibertarianism has two wings, the "Gilders," more connected to classical free-market libertarianism and traditional political conservatism, such as founding *Wired* executive editor and evangelical Christian Kevin Kelly, and the "Ravers," neo-hippies like Grateful Dead lyricist and EFF founder John Perry Barlow, whose "antigovernment stance is more hedonic than moral, more lifestyle choice than policy

---

[10] "Bill Gates did not make a good impression the first time he testified on Capitol Hill [in 1998 in front of the Senate Judiciary Committee]… He made sure everyone knew he was not interested in playing the political game." (Simon and Mershon 2014)

position." (Borsook 2000, 14–17) Ravers are the romantic, countercultural wing, searching for freedom, community, and transcendence in utopias from cyberspace to Burning Man.

It is this kind of technolibertarianism, which combines market-based technological production with countercultural artistic creation, that Fred Turner has examined more recently, by studying the prevalence of Google employees (including founders Sergei Brin and Larry Page) at Burning Man (Turner 2009). Turner argues that the commons-based peer production celebrated at the bohemian Burning Man festival, done for the sake of art, not money, creates a "cultural infrastructure" to support similar commons-based peer production at Google, where employees are encouraged to work on their own creative projects in an effort to spur bottom-up innovation. Such production is celebrated at both places as spiritual and transcendent, a "vocational ecstasy" in the form of "Silicon Pentacostalism." The difference, however, is that creative production and innovation at Google is monetized for profit.

Another more recent account of technolibertarianism is Thomas Malaby's ethnography of Linden Lab, the company that created the online world Second Life (Malaby 2009). Malaby draws extensively on Turner and shows how technolibertarian ideology undergirds both Second Life and Linden Lab itself. Second Life, unlike most other online worlds, is designed so that users create content for themselves, using tools provided to them in the virtual world by Linden Lab. Linden advertises the participatory nature of Second Life. Malaby notes, however, that there is still a hierarchy—not all tools available to Linden Lab are available to the users, and Linden Lab reserves the right to make world-changing decisions. Similarly, Linden Lab prides itself as a relatively "flat" organization, with all employees able to contribute to decisions, and company philosophy emphasizing openness and collaboration. However, the reality is that CEO Philip Rosedale has final authority. Malaby thus notes a symmetry between the social organization of Second Life with Linden Lab itself. Malaby's account has certain parallels with my work on Cocoa developers. Firstly, both Second Life users and Cocoa developers use tools provided by a proprietary platform vendor (Linden Lab and Apple) in their own creative production. Both app development and the culture of Second Life celebrate democratized creation through giving people access to tools, that quintessentially Brandian conceit. Nevertheless, because both

platforms are proprietary, the rhetoric of democratization and participation goes only so far—both platforms are, in reality, highly controlled. Yet, both platform vendors need their users/developers in order to make their platform more attractive, and this means that they need to make certain accommodations. Both Second Life users and Cocoa developers exist in a symbiotic, mutually dependent, though highly asymmetric relationship with their platform vendor. The technolibertarian ideology of personal empowerment through access to tools runs strong in both communities and helps maintain cohesion between users and the platform vendor, as both sides see themselves working towards shared goals.

In this dissertation, I use "technolibertarianism" to refer to an ideology that does not necessarily imply free-market fiscal conservatism and a preference for smaller government, though many of my actors are indeed classically political libertarians in that sense. Rather, I use it to mean this larger belief that technological innovation is a superior means to enact social change than traditional politics or activism, through the empowerment of individuals through "access to tools," the subtitle of Stewart Brand's *Whole Earth Catalog*. Drawing on Thomas Streeter (2011), I argue that this is an ideology based on the construction of a romantic individual identity, a heroic, creative, expressive, artistic, and often rebellious individual who makes and plays with technologies not for purely rational, utilitarian reasons (including making money) but for its own sake, for the affective pleasure of tinkering, the joy of making, or the spiritual experience of transcendence it may bring. It is this individual at the center of this ideology, who must be free from the shackles of institutional bureaucracies, be they government or corporations. Yet, at the same time, it celebrates this individual if he (usually a he), in the process of pursuing his own edification through technological making, happens to create a corporation as a by-product, as long as this corporation is seen as empowering other individuals by giving them tools, rather than controlling them.[11] This is one of the governing ideologies of Silicon Valley startups from the PC revolution to the dot.com boom on to today's mobile gold rush. Moreover, it is the

---

[11] In practice, this tension between empowerment by a corporation and control by it may be difficult to navigate, but at least in its purest imagined form, this is the way technolibertarians think it "ought to be."

central ideological axiom of Apple itself, present especially in its marketing slogans throughout the years: "The Power to Be Your Best," "Think Different," and "Changing the World, One Person at a Time."[12]

Before concluding this section, I must turn back to the question with which I began it. What term best describes the seamless web of affect, identity, normative values, practices, and technology? Ideology seems to be the most useful term, simply because it is broad enough to encompass all of these aspects. My use of ideology is not primarily about Politics in the geopolitical or partisan sense (though it sometimes can be), but more about the moral logic that drives a culture and community. In this sense, moral economy might be a better term, as it well describes the intricate web of a relational ethical system, and I see moral economy as an important component of ideology, though it does not necessarily get at affective, aesthetic, or material aspects. Neither term incorporates the technological component of what I am describing, which is crucial to the affective experience of those using Cocoa, which constitutes the material basis for the ideology. In a sense, this is a strength, for ideology and moral economy can be in play when technology is not integrally involved. However, some ideologies, such as technolibertarianism, do indeed entail particular ethical and political orientations to technology as well as society. The concept of technological frame can add technology back in, but at the cost of losing other aspects that can be incorporated into ideology, such as affect, ethics, and identity. The concept of paradigms, with its association with epistemic worldviews as well as material practices, might also be useful, particularly in discussing pedagogy, but its assertion of radical incommensurability between paradigms is a problem, leaving no room for communication between boundaries. Galison's notion of the trading zone helps resolve this issue of communication across subcultures. While Galison speaks of trading between subcultures, Star and Griesmer talk of translation across social worlds. Similarly, Takhteyev uses the

---

[12] As a child, I had an Apple bumper sticker, probably from my family's 1988 purchase of a Macintosh Plus, that read "Changing the world, one person at a time," which our family pasted on top of the computer desk. A picture of this bumper sticker can be found at (King 2012).

term “world of practice” to denote a complex of material and discursive practice and experience. Certainly, what I am describing in the social experience Cocoa programmers is indeed a cultural, social, technical, and emotional world. However, I need a term that somehow connotes “worldview,” structures of belief and values that encompass the sense of both “ideology,” and “paradigm.” For lack of a better term, I will describe the seamless web of Cocoa community, identity, and practice as a “techno-cultural” frame, which incorporates elements of ideology, moral economy, and technological frame. I will use each of these subordinate concepts where appropriate—I will use technological frame when discussing Cocoa developers’ relative inclusion or exclusion in the community, its practice, way of thinking, and orientation to Cocoa technology; I will use moral economy to describe its interrelated system of ethics, and ideology to discuss incorporation of this normative system with affect and subjectivities into a larger socio-political worldview.

Software Engineering, Maintenance, and Object-Oriented Programming

According to Borsook, the technolibertarian ideology of high tech pervades Silicon Valley, and indeed, Takhteyev might argue, the global “world of practice” of programmers everywhere. Yet here I return to a question I posed earlier—what is the role of Cocoa technology itself? For Cocoa developers, it not only matters that they have “access to tools,” but also what those tools are—Cocoa itself. To them, the properties of Cocoa and the practices associated with using it make a difference, both on an affective basis, in which they claim it is more enjoyable to work with than other development tools, as well as on a rational or utilitarian basis, in which they claim it is technically superior to other tools.

The basis for this claim lies in Cocoa’s object-oriented nature. Object-oriented programming is a methodology for programming and programming language design that emerged in the late 1960s and early 1970s. Unlike traditional procedural programming, in which a programmer envisions a program as a set of processes and procedures through which program execution flows, in object-oriented programming a program is envisioned as a configuration of black-boxed objects, which have often hierarchical relations to each other, and cause things to happen by sending messages to each other. The term was coined by Alan

Kay at Xerox PARC to describe his Smalltalk programming language and user environment, and although Smalltalk was not the first object-oriented language,[13] concepts it pioneered have been exceedingly influential in the development of later languages, such as Java, Python, Ruby, as well as Objective-C, the language at the heart of Cocoa. In fact, Objective-C was originally developed to add a simple Smalltalk-like layer on top of the industry-leading C programming language, which is not object-oriented. Steve Job's NeXT chose Objective-C to be the foundation for its operating system's software libraries, which later developed into Cocoa at Apple.

That a technology originating at Xerox PARC became the centerpiece of yet another of Jobs' ventures is no accident. In an interview for the PBS documentary, "Triumph of the Nerds," Jobs acknowledged that he had seen both object-oriented programming and networking at PARC during his famous visit in the late 1970s, but had failed to see its importance at the time: "One of the things they [PARC] showed me was object orienting [sic] programming… The other one they showed me was a networked computer system... [but] I didn't even see that… I was so blinded by the first thing they showed me which was the graphical user interface." (Cringely 1996b)[14] Jobs had not realized that PARC's Smalltalk System was an integrated whole—its graphical user interface was built with object-oriented libraries and techniques, and the entire system, not just the programming language or the user interface, was what Smalltalk was. Jobs had only grasped the technology at the superficial level, and at Apple, his engineers replicated the look and feel of the graphical user interface on the Lisa and Macintosh but did it with radically different, non-object-oriented technologies. This was largely a necessity, however, as the Macintosh was supposed to be a mass-market consumer device, its hardware was considerably less

---

[13] Joline Zepcevski argues that Simula-67 was the first language to contain the necessary features to today be classified as "object-oriented." (Zepcevski 2012, 226–227)

[14] Michael Hiltzik's book on Xerox PARC says much the same thing: "As for Jobs, he was so 'saturated' by the power of the user interface he had seen that he ignored the other two phenomena he was being shown: object-oriented programming, which was the essence of Smalltalk, and networking." (Hiltzik 1999, 343)

powerful than the Alto's (originally shipping with only 128K of memory) and Smalltalk's object-oriented environment was resource intensive and inefficient and would not have run on the original Mac. This strategy worked successfully in the short run to ship the Mac, but the limitations of the Mac's programming environment, the Macintosh Toolbox, would come to haunt Apple in the 1990s as it tried to modernize its operating system. After Jobs left Apple to form NeXT, he hired top computer science researchers such as Avie Tevanian from Carnegie Mellon. By the late 1980s, object-oriented programming had become the next big thing in computer science, with its own conferences, and NeXT made it a centerpiece of its new platform. In a way, Jobs may have been trying to recreate a PARC-like environment at NeXT, by hiring, in his opinion, the best and brightest and let them decide what technologies should go into a bleeding edge computer. NeXT's focus on networking, which it marketed as the new paradigm of "interpersonal computing," was really a take on Xerox's "personal distributed computing." (Lampson 1986; Thacker 1986) Similarly, NeXT's graphical user interface was built using object-oriented libraries, just as Smalltalk was, and it was written with a language, Objective-C, that was explicitly modeled after Smalltalk.

Why is object-oriented programming, and Smalltalk-based technology in particular, so important to Cocoa programmers? To answer this question, we must look at the context in which object-oriented programming came about, during the so-called "software crisis" of the 1960s,[15] alongside the emergence of "software engineering." As we will see in chapters 2, 3, and 6, Cocoa developers claim that because Cocoa is object-oriented, it can improve programmer productivity by an order of magnitude, through reducing complexity and

---

[15] Thomas Haigh argues that the software crisis was in reality more a discursive construction by Dijkstra to overstate crisis language in the 1968 NATO Conference proceedings during his 1972 Turing Award acceptance speech. Dijkstra did this to criticize the "failure of computer companies to recognize the mathematical nature of software development and their insistence on hiring insufficiently intelligent people to do it." (Haigh 2010) Talk of the crisis is later seized upon by historians of computing in connection with software engineering, but Haigh argues that this is overblown with regard to the concerns of actual software practitioners.

increasing the maintainability of programs. Historian Michael Mahoney understood object-oriented programming to be a particular response to the software crisis. "One only has to read Doug McIlroy's 'On Mass-Produced Software Components,' presented to the first NATO Software Engineering Conference in 1968, to see where the conceptual roots of object-oriented programming lie." (Mahoney 1993, 778) A recent dissertation by Joline Zepcevski argues that both structured programming and object-oriented programming methodologies developed concurrently in the 1970s to tackle the software crisis, Smalltalk especially. Both of these methodologies sought to address the issues of the complexity of software, and its successful verification. Both approaches encouraged increasing modularization and reuse of code, and while these principles were previously voluntary practices to be encouraged, new languages designed specifically to support the new methodologies (such as Pascal and Ada for structured programming, Smalltalk and C++ for object-oriented programming) began to enforce them more rigidly. Nevertheless, Zepcevski argues that only object-oriented programming represented a change in worldview from the earlier procedural programming paradigm: instead of viewing a program as a flow of processes, it transforms programs into systems of dynamically interacting, communicating objects (Zepcevski 2012).

Zepcevski's work draws on a growing literature on the software crisis and software engineering in the history of computing. (Abbate 2012; Ensmenger 2010; Ensmenger and Aspray 2002; MacKenzie 2001; Mahoney 1990; Mahoney 2002; Mahoney 2004; Slayton 2013a) In the late 1960s, the computer industry began to note that software development costs were outpacing hardware costs, there seemed to be a perennial shortage of skilled programmers, and several large, highly complex software projects failed spectacularly, including OS/360, IBM's operating system for its next generation System/360 series of computers. What she identifies as the problem of managing overwhelming complexity in programming was a central concern for figures influential in the development of software engineering, especially Fred Brooks, the IBM software manager who oversaw OS/360 and wrote the influential *The Mythical Man-Month* (F. P. Brooks 1995) as a post-mortem of the project, which became a canonical text in software engineering. Brooks wrote in 1987 that complexity was one of the "essential" traits of programming that meant that no technology

or technique, “No Silver Bullet,” could ultimately solve the software crisis. (F. P. Brooks 1987)

Brad Cox, the creator of the Objective-C programming language, disagreed. Cox claimed that object-oriented programming (combined with a market of off-the-shelf software object components that programmers could buy) was not only a silver bullet, but would usher in a “software industrial revolution,” taking programming out of the realm of craft and into the era of manufacturing (Cox 1990b; Cox 1990a). Historian Michael Mahoney has pointed out how the dream of “automatic programming” and the “software factory” has a long history, which became especially pronounced with the software crisis (Mahoney 2002). Nathan Ensmenger has cited Brad Cox as an example of the emphasis in software engineering towards the development of technologies and methodologies to aid in the management of unruly programmers, as programmer unmanageability was one of the key perceived sources of the software crisis (Ensmenger and Aspray 2002, 15–16). Indeed, the metaphor of “engineering” itself, chosen in part to give programming an air of quantification, rigor, discipline, professionalization, masculinization, and thus status, lent itself easily to discourse involving the industrial revolution, manufacturing, interchangeable parts, and automation. Sociologist Phillip Kraft warned in the 1970s that many of the techniques proposed by software engineering advocates, such as structured programming, a set of formalized, disciplinary programming practices advocated by computer scientist Edsgar Dijkstra, would increasingly routinize and deskill programmers (Ensmenger 2010, 231–2). Ensmenger, examining the managerial rhetoric of such efforts, largely agrees that control over labor was a primary motivation of software engineering, and that in corporate settings, managers did indeed increase a measure of control, but feels that ultimately, the rhetoric was overblown, as programming continues to require craft skill (Ensmenger 2010, 47–49, 243). For Ensmenger, programming simply proved intractable to managerial attempts to discipline it.

Donald Mackenzie and Janet Abbate, however, interpret the discourse of software engineering differently. Both note that “Software engineering advocates such as Dijkstra and Brooks identified as programmers themselves and had no desire to downgrade their peers.” (Abbate 2012, 107) Mackenzie points out that “Dijkstra recoiled at the analogy of

the factory. When asked his profession, he was proud to declare himself simply a 'programmer': for him, programming was intrinsically a demanding activity… The discipline needed for successful programming was not organizational and managerial, in Dijkstra's opinion, but intellectual." (MacKenzie 2001, 40) Mackenzie further notes that Harlan Mills' adoption of structured programming at IBM resulted in the opposite of deskilling: "The tasks that programmers were left with, however, after the discipline of structured programming had been imposed, were far from routine. Mills's [sic] cleanroom demanded more, not less, from its programmers…" (MacKenzie 2001, 57) Similarly, Abbate states, "Unlike most industrial or office automation, software innovations were created and adopted by programmers themselves, not managers… Rather than a serious goal, deskilling functioned more as a handy cultural trope that marketers could use to appeal to potential customers." (Abbate 2012, 85) "Many programmers actively took up techniques such as structured programming as a way of easing their work and enhancing their own value in the job market. Rather than making programmers obsolete, software engineering methods became simply another skill that programmers could claim." (Abbate 2012, 107–108) As we will see in chapter 3, my own interviews with Cocoa programmers, who have embraced a cultural-technological frame that deeply embeds many software engineering principles, corroborates Abbate's argument. Software engineering techniques, despite deskilling or routinizing rhetoric, discipline the programmer in the name of building professional skill more than managerial docility. Both structured programming and object-oriented programming methods are standard curricula in computer science, part of the trade knowledge of the profession, and have increased, rather than decreased, the skill involved in programming. Experienced programmers today largely see the acceptance of these methodologies as evidence of progress in the field, not as the triumph of managerial interests.

Why have such methods been so widely accepted in programming? Another essential characteristic of software that made it difficult to write, Brooks asserted in "No Silver Bullet," was its changeability, in order to respond to new feature requests from users, and new hardware that it must support. This need to change is rooted in software's inseparability from its larger social context: "In short, the software product is embedded in a cultural

matrix of applications, users, laws, and machine vehicles [hardware]. These all change continually, and their changes inexorably force change upon the software product." (F. P. Brooks 1987, 12) Although software is more easily changed than hardware, in practice it is not "infinitely malleable" as Brooks would have it. It is in fact because, as Nathan Ensmenger puts it, "Software is history, organization, and social relationships make tangible," (Ensmenger 2010, 227) that software is obdurate, constrained by design decisions made in the social, institutional, and material context in which it was first written. Because of the long lifecycles of many programs, choices made today that limit future flexibility and malleability in the name of expedience or efficiency have deleterious effects decades into the future. It was this problem of "legacy software" that caused the panic over the Y2K bug: the COBOL programs that ran much of the nation's financial infrastructure had been written in the 1960s, when it was necessary to save memory by omitting the first two digits of a year, but these programs' lifecycles lasted into the 1990s and past the year 2000, requiring expensive rewriting of all of this stable, working, infrastructural code. Ensmenger calls this the problem of software maintenance, and argues that "the software crisis of the late 1960s was essentially a maintenance problem." (Ensmenger 2010, 224–225) Although maintenance is low-status and boring, as programmers would rather be makers of the new rather than repairers of the old, because even new programs take a long time to write while goals are constantly changing, the line between creation and repair can become blurry in software, as bug fixing is always the final step towards releasing a "new" software product. And as we will see in chapter 6, experienced software developers argue that adopting disciplined practices up front to make software more "maintainable" will not only pay off in the long run, but is seen as taking a step towards increasing one's professional skill and standing in the community of practice. This, in turn, can make the difference between being accepted or excluded as a member of the professional community. Such "best practices" can exist independently of particular technologies, but sometimes are strongly associated with particular programming languages, development environments, and conventional coding idioms. Such is the case with certain software design practices and Cocoa technology. Cocoa programmers assert that the reason Cocoa is a superior tool for writing software is because of Cocoa's dynamic and flexible object-oriented design, and the conventional practices and idioms associated with its use, which they claim help make their software

more maintainable. Practices that emphasize the virtue of maintainability are part and parcel of the techno-cultural frame associated with Cocoa programming.

Methodology

I came to science and technology studies out of a program in history, and I began my studies with a much stronger sense of the disciplinary ways of thinking of historians. However, when I began this project, being in science and technology studies allowed me to ask contemporary questions and to study contemporary phenomena. I intended for the project to be equal parts historical and equal parts contemporary. I attempted to find sources on NeXT at the Charles Babbage Institute and Stanford Library's Special Collections, but beyond a few NeXT brochures in the Michael Mahoney papers and a draft of a proposed paper to the third History of Programming Languages conference on Objective-C, and a cache of NeXTWORLD magazines that I later also found online, I found little of direct relevance to my dissertation, although I did find a lot of materials on object-oriented programming. I did eventually find a few sources to conduct oral histories of NeXT developers and a few NeXT employees. (I will discuss my interview methods below.) Given only a handful (less than ten) oral history interviews pertaining to NeXT, and the dearth of written sources, I felt that I could not accomplish a sufficiently robust history of NeXT or the NeXT developer community in the current project, other than the rough outline given in chapter 2, which is drawn largely from the NeXTWORLD magazines, the NeXT brochure, and excerpts from my interviews. I thus decided to focus more on the contemporary ethnographic and interview-based portion of the project. The result is that I see this work as more a work of sociology of technology with ethnographic and historical components. Chapter 2 is the most historical chapter, being based as much on documents (magazine articles and marketing materials from the 1990s) as on oral history interviews, and is closest to telling a chronological narrative, for the purposes of providing necessary historical context for the rest of the dissertation. Other chapters, especially chapter 6, draw on secondary historical literature as well as on interviews. Despite my extensive use of historical (both primary and secondary) sources, this dissertation is not primarily a work of history. It does not directly address many questions of central concern to historians of computing and software raised by Michael Mahoney, such as the social or institutional

context of hardware or software, communities of computer users in the professions, or the history of software engineering or computer science as a discipline or programming as a profession (Mahoney 1993, 779–80). Rather, I have used my oral histories to provide context on the life and career trajectories of my participants. Nevertheless, this dissertation does touch on some themes outlined by Mahoney: it documents the practice of a specific community of programmers, in particular, the tacit knowledge, the tricks of the trade, the "body of techniques, and the habits of thought" of Cocoa programmers (Mahoney 1993, 774). It does this by opening the black box of Cocoa software, by reading the software artifact that is Cocoa, through engagement in the practice of Cocoa programming itself. Moreover, Mahoney argues that all software is legacy software, which bears the traces of its history.[16] On one level, the development of Cocoa and Objective-C, and associated disciplinary practices, were an attempt to address the problem of maintenance of legacy software, to improve its reliability and flexibility. On another level, Cocoa itself is legacy, in the sense that its design bears the mark of its origins at NeXT and its passage through a period where it was applied primarily to the problems of large enterprises and institutions, not consumers with smartphones. Cocoa must also be understood in the ideological context of the computer counterculture and utopian digital technolibertarianism.

This dissertation draws primarily on semi-structured interviews and ethnographic participant observation. As I had been an Apple employee myself till 2005, I began recruiting participants for my project through my network of social contacts that I had formed at Apple. After being turned down for permission to interview Apple employees (and thus my former coworkers still working at Apple) by vice presidents Scott Forstall and Bertrand Serlet because of Apple's secrecy policies, I refocused my project on third party developers who did not work for Apple. My network of contacts at Apple, and my own former membership in the Cocoa team, was still very useful in helping me to find and gain access to third party developers. As a person formerly from the Apple engineering culture, I

---

[16] "Legacy software is not just old code, but rather a continuing enactment, an operative representation, of the domain knowledge and practice embodied in it. (Mahoney 2008, 15)

already knew who the big names in the community were, and who wrote the important indie apps, as I had been responsible for testing these apps at Apple. One Apple friend invited me to a lunch at a restaurant near Apple's campus, where I met the employees of a new iOS startup. I went to conferences such as MacWorld Expo and Apple WWDC and made connections with developers face-to-face. In the case of Aaron Hillegass, I connected with him through the social media site LinkedIn, as we had other connections in common through mutual friends at Apple. In other cases, I cold e-mailed prominent developers such as Wil Shipley and Ken Case, to try to set up interviews. While in Seattle, I attended local Cocoa developer clubs to meet people. Having obtained these initial connections, I acquired other participants through snowball sampling.

My interviews began with a standard informed consent process that allowed me to record the audio interview. In many cases, the participants are well-known members of the community who write public blogs, and I requested the ability to use their real names if they allowed, otherwise I could grant them confidentiality. The interview then proceeded in a semi-structured manner—I had a rough outline of the types of questions I wanted to ask, but was not rigid, allowing the conversation to proceed organically based on the participant's responses. Given that I was interested in both historical and contemporary issues, I began every interview by asking about their biography and career trajectory: how did they first learn programming? How did they first get into the Macintosh, iPhone, or other Apple products? When did they first learn to program for Apple? What was that process like? That naturally proceeded to more sociological questions: what do they like or dislike about programming for Apple? What do they think of their relationship with Apple? How do they participate in the life of the community? If they teach, what methods do they use? Many of these interviews were conducted concurrently with my participant observation (described below), and thus my questions gradually changed over time to encompass issues that I observed in the field. The interviews were later transcribed and coded using Atlas.ti software with grounded theory methodology.

My interviews are thus a combination of sociological/ethnographic interviews, and oral history interviews. Interviews, of course, cannot be taken to be objective accounts, but are subject to not only the participant's bias but also lapses in memory. With oral history

accounts, I have tried to corroborate factual assertions with the few written sources available to me, which included an archive of NeXTWORLD magazines at Charles Babbage Institute (and found online), popular journalistic accounts of Apple and NeXT, as well as my own memories from my time at Apple.

My ethnographic work began with the developers I had met at the lunch near Apple. One of them invited me to come over to his workplace the next week, and I spent the next two weeks hanging out there, observing and listening to what was going on, sometimes interviewing them, taking field notes, sometimes recording their conversations (with permission). The following summer, I obtained permission to do the same at another iPhone startup in Palo Alto, and spent more than two weeks observing the office, taking field notes, and observing and recording meetings, which I had obtained permission to do. Given the relatively short span of my time in these initial studies, and my computer's inability to run the necessary software so I could engage in programming work at the time, I could only observe but not directly participate in the work of these developers. I found this to be extremely limiting—most of the time, I just saw people staring at computer screens, and without writing code myself, I could not get a deep understanding of what they were actually doing. In my subsequent study of the Big Nerd Ranch, which was accomplished in two separate trips, the first of four months, the second, seven, I resolved to become personally engaged in the work of the company, which I did after some effort. This meant that after a period of time, I became a full employee of Big Nerd Ranch, doing equivalent work to many of its other employees, and eventually even (albeit briefly) attaining the same pay. While working, I would try to take field notes during moments of transition—such as right after lunch, or at the end of the work day, or possibly after an interesting conversation. Fortunately, I was relatively free to do so, although I did gradually come to have significant coding work. I would also write field notes every night after returning to my apartment to summarize what I thought of the day.

To a significant extent, the seeds of my interest in this study originated in my first year as software engineer at Apple, fresh out of college. From the time my family had purchased a Macintosh Plus when I was in fifth grade, I had been an Apple fan, a zealot, even, especially during my undergraduate years when it seemed as if Apple might go out of

business and the Macintosh platform would disappear. It had been a goal of mine as an electrical engineering and computer science undergraduate to get my dream job, to work at Apple, at the height of the dot.com boom, when Apple's future was still uncertain and was not, as it is today, a hot place for young engineers to go. After a long search, I was finally hired at Apple on October 4, 1999, by the Macintosh engineering group, to work on the next version of the classic Mac OS, 9.1. My experience in the team immediately disabused me of any notion that Apple was a land of unicorns and rainbows, milk and honey. Morale was low, as the engineers who had worked on OS 9 felt continually slighted by upper management, who favored the up-and-coming next-generation operating system, the NeXT-based Mac OS X. Within six months, my manager would leave and most of my group would disband, as management would decide that it was now time to shift resources to OS X. Yet despite the clear message that OS X represented "progress" and "the future" for the Mac, the engineers in the OS 9 group made coherent, rational arguments for why certain aspects of OS X actually represented, to them, a step backwards, as well as why OS 9 could be retrofitted with the "modern" features that motivated why it was being replaced, and thus, should continue to be worked on in parallel to OS X in the short term. After my group disbanded, I found a new position as the Quality Assurance engineer (the software tester) for the Cocoa framework group, at the heart of the OS X division. Joining this group was almost like joining a different company, as the cultures were so different. If my former group represented the old, pre-Jobs 1990s Apple, the Cocoa division maintained practices and a culture going back to NeXT. The engineers in the Cocoa group had their own technical arguments over why their technology, and hence, OS X, was superior to classic OS 9 and thus why it should replace it. It was this experience of witnessing two competing technological visions within the company, one ascendant, the other, receding, that I began to understand that autonomous technological progress in the absence of social conflict and struggles for power was a myth. OS X was winning out at Apple because those who controlled Apple actively pursued policies that favored it. Did this mean that it really wasn't a better operating system than OS 9? I wasn't ready to go that far. My own experiences developing an application using Cocoa versus with the classic Mac Toolbox showed me that Cocoa was a way better way to write a program with a graphical user interface. On a technical level, I believed OS X certainly was better than OS 9 in most ways, yet it did not

give me the same warm, pleasurable feeling as my childhood Mac. This raised questions in my mind about the nature of technological development, technical cultures, and power. And it made me question my own beliefs and about, and attachment to, Apple and its technology. To an extent, then, this project has been auto-ethnographic—in explaining the motivations of the Cocoa community, I am by proxy trying to explain and analyze my own attitudes.

Given my own former employment at Apple, and my close proximity to the social world of Apple development, a challenge in my work is the problem of acquiring the "strangeness" necessary to question what is taken as common sense in the community. Because I sometimes share this "common sense," it can be tempting to present the views of my informants at face value without sociological analysis. Have I "gone native"? In my case, I was never not a native, as the culture I was studying was my own original culture. The academic field of Science and Technology Studies is in fact a second (or possibly third) culture I have been initiated and acculturated into, through which I now approach my former one. Nevertheless, I have still encountered plenty of strangeness as I traveled to Seattle and Atlanta, places where I was unfamiliar, and got used to the rhythms of working at the Big Nerd Ranch. Nonetheless, my primary objective is precisely to present and explain the techno-cultural frame of Cocoa developers on their own terms, in the way they understand it and themselves, why they love Cocoa technology and how their work with it gives meaning to their lives. To understand their relationship to Cocoa technology, I had to become (or rather, become again) a Cocoa developer myself, in order to fully engage in their experiential world. It is because of my experience as a Cocoa programmer that I am able to tease out the socio-cultural stakes at play in technical arguments over programming language syntax or choices in design patterns.

John Tresch has argued, using Kuhn, that "going native" is in fact necessary in anthropology in order to fully understand a (especially non-Western) culture's phenomenal world without imposition on the phenomena of analytical categories (Tresch 2001). Harry Collins has similarly argued for the necessity of going native in science studies: "In this kind of research it is necessary to be sucked into the science. First, to elicit revealing reactions one has to be caught up… Respondents must become something closer to colleagues—people who are happy to argue physics, not just react as though filling out a

questionnaire. Second, if one adopts the approach known as 'participant comprehension,' …the object will be to become as like the actors one is studying as possible; the idea is to come to understand the society under investigation from the inside. The aim is to come to see the world through the same sets of categories as the actors one is investigating." (Collins 2011, 319–20) Similarly, my own nativeness was a benefit in the field. Entering the field after a time spent in academia, there was still a brief period of adjustment and strangeness to some extent, but the process was significantly truncated. Relying on my previous knowledge, I began with at least a level of expertise Collins calls, "interactional expertise," in which I could fully engage in the discourse of actors using native terms. This level helped me easily gain rapport, as I already understood the vocabulary and the categories of the actors without too much explanation. Additionally, it took very little for me to acquire "participant comprehension." When my fieldwork was restricted to strict observation instead of participation, however, I could only remain at this interactional level. However, once I began to engage more fully in their work as a full participant, I began to acquire what Collins calls "contributory expertise" and "experience-based expertise."

Nativeness presents its own problems, of course. Collins notes that although the social analyst must go native in the field, she must also achieve distance upon returning out of it. "The social analyst has to analyze sociologically, and this sometimes means analyzing in ways that are not natural to the members of the society being analyzed. Sometimes the social analyst has to disagree with the native members." (Collins 2011, 320) Lacking strangeness, it can be easy to take the actors' categories as natural. I have attempted, to analyze phenomena in the field using analytical concepts drawn from STS and elsewhere. Such categories as "technological frame," "communities of practice," and "techno-libertarianism" are analyst's categories, though drawn from my interpretation of actors' categories. "Paradigm" presents a challenge as it is both an actor's and analyst's category; as an actor's category, it draws on its Kuhnian meaning as an analytical category, but it is not precisely the same. I try to avoid using paradigm as an analytical category, but in cases where I find it unavoidable, I will endeavor to clarify when I use "paradigm" as an actor's versus an analyst's category. In the paragraphs that follow, I will lay out some of the strategies I have used to acquire some analytical distance from my actors.

This dissertation focuses on an in-depth analysis of the Cocoa developer community, rather than comparing it on an equal basis with other communities, say, Windows or Android programmers. In doing this, I try to avoid, as much as I am able, in taking the views of Cocoa programmers, especially about the technology of Cocoa, as "objective" truths. As an example, one of my central concerns is the claim that Cocoa makes software developers more productive, relative to other development environments. My own prior programming experiences at Apple have led me to believe that there is something to this claim. However, it is fully possible that programmers with high inclusion in alternative technological frames might be just as productive as any Cocoa programmer. I believe it is likely that Cocoa has productivity advantages in comparison with most procedural environments, and possibly even in comparison with some object-oriented environments such as C++ or Java, but in comparison with newer or similarly flexible languages and toolkits such as C#, Python, or Ruby, Cocoa may not have any advantage. Regardless, the truth value of any of these claims is irrelevant; my purpose in this dissertation is not to establish whether Cocoa is actually better or more productive, but to ascertain the reasons why Cocoa developers believe it to be so.

Because my primary aim is to explain the central normative and affective logic of the Cocoa community, I have focused my recruiting of participants on the core of the Apple developer community, especially independent developers located in the Seattle area, which Van Meeteren (2008) has already highlighted in his work. (Given the small size of the core of this community, we share a number of the same participants, and it was one of those participants that I recruited who alerted me to his participation in Van Meeteren's previous study.) Being located in Seattle did allow me, however, to include a small handful of less-highly included participants, newer iOS developers who had converted from Windows or other platforms. Again, given the focus of the dissertation on explaining the worldview of the highly included core of the community, and the limited amount of time I had to transcribe and code over 50 interviews, almost all of which were over two hours long (the longest reaching nine hours), I necessarily had to focus on certain perspectives over others.

One way to avoid certain blindnesses associated with getting too close to my participants is to look for areas of disagreement among them. One way I do this is by

studying a technical controversy in the community, the topic of chapter 6. Another way I do this is by looking for areas where developers are critical of Apple. In interviews, I asked both core and peripherally located participants about their disagreements with Apple. With few exceptions, my participants voiced little criticism of Apple's Cocoa frameworks; rather, various critiques were made of the Objective-C language, of Apple's Xcode development tool compared to Microsoft Visual Studio, or of various Apple policies pertaining to their control of the platform. These are all interesting and worthwhile critiques, but aside from the critiques of the language, which I address somewhat in chapter 6 from the perspective of aesthetics and boundary work, these other issues are full topics in their own right that deserve their own treatment, which I plan to return to later. Of special interest to me is criticism of Apple from highly-included Apple developers themselves. I had intended the relationship between the developer community and Apple to be a more prominent theme in the dissertation, but I made an editorial decision to keep the dissertation's core topic and focus manageable and have cut the intended full chapter, reintegrating the material in abridged form into chapter 5.

Another way to address the problem with being too close to my participants is to juxtapose their views with those marginal, outside, or new to the community. Given time, resource, and access constraints, it was not possible do an equivalent treatment of the Windows, Android, or Ruby communities; nor is this dissertation designed to be comparative to that extent. It would have been beneficial, however, to try to obtain some access to Macintosh developers who programed using the alternative, classic Mac OS 9 based-Carbon toolkit, who would undoubtedly have offered a useful counterpoint. Unfortunately, by the time I conducted my fieldwork, most of this community had either converted to other platforms (including Cocoa) or retired, as this technology is no longer supported by Apple, and I did not readily find any developers from my network of contacts who not only had formerly used Carbon but had refused to switch to Cocoa. I was successful, however, in gaining access to some developers who had experience with Windows, Android, or Ruby development, although these developers were all also iOS or Cocoa developers. In the case of Windows, I have three interviews of former Windows programmers who were highly included in that sphere before converting to iOS, including

one former Microsoft employee. One of them informed me that the Windows community does not really exist in the same way as the Cocoa community, with little actual social interaction outside of professional conferences. These informants provided useful points of comparison between Cocoa and other programming communities. Given my time constraints, I have only begun to analyze this material, however, and not all of the questions raised by this data were relevant to the central themes of this dissertation. I anticipate that this material will yield more insights in the future.

Another difficulty for maintaining analytical distance is the sheer technical depth of some of the actors' concepts, which are difficult to explain to persons without a computer science background. I have attempted to explain in detail a few concepts that are relevant to the analytical points I want to make. Inevitably, though, with some less central concepts, I may keep them black-boxed and not subject them to sociological analysis nor explain them in deep technical detail, as they are not of primary importance to the analysis.

Summary of the Dissertation

Anthropologist Sharon Traweek wrote that ethnographies usually describe "four domains of community life." (Traweek 1988, 7) These include ecology, or "the group's means of subsistence, the environment that supports it, the tools and other artifacts used in getting a living from the environment;" social organization, or "how the group structures itself;" the developmental cycle, or "how the group transmits to novices the skills, values, and knowledge that constitute a sensible, competent person;" and cosmology, the "group's system of knowledge, skills, and beliefs, what is valued and what is denigrated." Each chapter of this dissertation focuses on one or more of these anthropological categories of analysis. No single chapter is devoted to Cocoa programmers' ecology, but because Cocoa technology itself is part of Cocoa programmers' means of subsistence, descriptions of their ecology is present throughout the dissertation.

Chapters 1 through 3 focus primarily on the cosmology of Cocoa programmers, and in particular, why they are committed to developing software for Apple platforms in general, and with Cocoa technology, in particular. Chapter 1 focuses on the affective, normative, and ideological aspects of this cosmology, especially the construction of the vocational "indie"

developer identity, its connection to Silicon Valley technolibertarianism and its doctrine of social change through individual empowerment by tools, and the particular role Steve Jobs and Apple play in this cosmology, as exemplar and mythic hero. Apple also plays an integral role in Cocoa developers' ecology, however, providing them with the tools and infrastructure to make their livings as indie developers, and this ecological dependence on the Apple corporation is ideologically reconciled through Apple's exceptional status in their cosmology, which differentiates it from all other corporations.

Chapters 2 and 3 look at the technical aspect of Cocoa developers' cosmology, in particular, the rational, instrumental, or utilitarian arguments Cocoa developers use to explain why they believe Cocoa is a better programming environment than others. Chapter 2 provides the historical background for these arguments, rooted in the discourse of software engineering and maintenance that came out of the so-called "software crisis" from the late 1960s through the 1970s, elaborating how NeXT created the technologies that became Cocoa in the late 1980s and early 1990s and marketed it based on the advantages of object-oriented programming, drawing on the earlier software engineering discourse. Chapter 2 also discusses the early formation of the NeXT developer community, its transformation into the Cocoa community and its sense of vindication that the technology they loved not only survived but has become the dominant technology in mobile development. Chapter 3 examines the particular technical characteristics of Cocoa, how these characteristics make programmers more productive, but also counter-intuitively, how they also require not only significant learning, but to a large extent, a kind of mental conversion or transformation in the way a programmer thinks. This is illustrated by explaining the importance, and the difficulty, of what are called "design patterns" in Cocoa programming.

Chapter 3 thus touches on the developmental cycle of Cocoa programmers, which is more fully the topic of chapter 4. In chapter 4, I look at how new Cocoa programmers are taught by a company that specializes in programmer training called the "Big Nerd Ranch." This chapter is an ethnographic and auto-ethnographic look at the Big Nerd Ranch's signature iOS Programming Bootcamp, revealing how students are trained, and how the norms and values of Cocoa developers are transmitted to them in the classroom.

Chapter 5 focuses on the social organization of the Cocoa community. Here I look at both its organization as a networked public online, and as a loose network of local communities in physical space, focusing special attention on the Seattle Cocoa community. I argue that despite the ideological emphasis of Cocoa developers on independence and individual production of apps and participation in the market, this is made possible by close collegial relationships among indie developers in Seattle, which facilitates sharing of knowledge, practices, and code, as well as occasional business partnerships, governed by a moral economy that transcends market competition. Chapter 5 also looks at the close symbiotic, but sometimes fraught, relationship that the Cocoa community has with Apple, which provides the ecological supports for its existence but also wields the power to put them out of business. Despite the extreme asymmetry of this power relationship, the symbolic discursive power of some influential members of the community can influence decisions at Apple, in part because the Cocoa community itself includes Apple employees who have social relationships with developers outside the company.

Chapter 6, the final chapter, examines cosmology, developmental cycle, and social organization simultaneously through a case study of a technical controversy in the Cocoa community over Apple's introduction of a new syntactic innovation in Objective-C, known as "dot notation," that older Cocoa programmers resisted partly because it was considered a foreign element associated with what they considered "inferior" languages such as C++ and Java. This was felt to be a concession to make Objective-C more palatable to newcomers, and resistance or acceptance of dot notation became a form of boundary work against such newcomers. To some extent, the controversy also represented a struggle between prominent members of the community with Apple itself over social reproduction of Cocoa developers.

Chapter 1: "Indie" Cocoa Developers: Pleasure, Vocation, and Ideology

In 2008, Apple opened up the iPhone to third party application development, sparking a "gold rush" of entrepreneurial activity in mobile software applications. "The rush to stake a claim on the iPhone is a lot like what happened in Silicon Valley in the early dot-com era," claimed a partner with the venture capital firm Kleiner Perkins, which started a $100 million "iFund" for iPhone applications. (Wortham 2009) Programmers flocked to Apple's platform in droves. Nevertheless, these latter-day forty-niners did not find Appleland completely unoccupied. Developers for Apple's Mac OS X personal computer operating system were among the first to explore making apps for the iPhone. Because iPhone and OS X development both use variants of Apple's Cocoa technology, these existing Cocoa experts tried to ensure, through their blogs and Twitter posts, that their community's values, practices, and ideology, in other words, their techno-cultural frame, would continue to be the dominant moral and technical order for the much expanded iPhone developer community.

This chapter explores this techno-cultural frame, especially its ideology, the affective pleasure that binds Cocoa developers to use of Cocoa technology, and the construction of the subjective identity of a Cocoa programmer. These are all components of what Sharon Traweek calls the "cosmological" component of a group's culture, in this case, the culture of the Cocoa community of practice.

The Cocoa developer community has a long history, which I will only sketch briefly here. Cocoa is a set of software libraries (or frameworks, in Apple's parlance) that make up a software development kit (SDK), interfaces into the operating system that allows developers to build applications. The toolkits that make up Cocoa originated on NeXTSTEP, the Unix based operating system created by NeXT for its black-colored computers. However, NeXTSTEP had acquired a loyal following among a small niche of software developers, who praised it for dramatically enhancing their productivity as programmers. Apple acquired NeXT in 1997, gaining

not only Jobs, but NeXT's operating system and development environment, which eventually became Mac OS X and Cocoa, respectively. This allowed the devoted cadre of NeXT developers to begin selling applications to Apple's large installed base of consumers. Most of these developers worked individually or in small-companies independent of large corporate software firms, and they began to call themselves "indie Cocoa developers." It was this indie Cocoa community that served as the core of the burgeoning new iPhone developer community in 2008, now known as the "iOS" developer community. (After Apple released the iPad in 2010, which runs the same operating system as the iPhone, it now refers to the OS for both devices as "iOS.")

What is particularly striking about NeXT developers is how fervently committed they were to using NeXT's toolkits to write software, considering that NeXT had almost no marketshare, and developers had to survive by taking contracts for large financial firms, where NeXT had discovered a market for its software. NeXT developers were known to be fanatical about NeXTSTEP:

> People who write software on NeXT… would rather be sheep farmers than have to program in some other environment." (Dan Wood, Interview, April 9, 2012).

As we saw in the introduction, Michiel van Meeteren also quoted a Cocoa programmer saying this, and apparently it had become something of popular saying amongst them (van Meeteren 2008, 22). This statement is performative, and the playful reference to sheep farming is deliberately outlandish. By focusing on the irrationality of NeXT programmers' stubbornness, it emphasizes their deep conviction to peers in order to enact an identity of moral superiority and separateness from other programmers who deign to use lesser environments. As we will see, until the iPhone, NeXT and Cocoa developers' commitment was proven greater the more a developer gave up the higher earnings they might obtain in greener pastures. During the height of the dot.com era, NeXT programmers could have joined Internet startups (and undoubtedly, many did), but those who remained on the tiny NeXT platform had to find a way to justify their decision. This justification was not based on

rational market choice, but was articulated affectively, involving a calling to a higher purpose:

> In 2000—you had to be in it because you loved what you were doing, because there was no other reason to be there! (Ken Case, Interview March 23, 2012)

It is not strictly true that NeXT developers largely sat on the sidelines of the dot.com boom. NeXT had come out with one of the first object-oriented backend web development environments, WebObjects, in the mid-1990s, built upon the same design principles as the desktop application frameworks that would later become Cocoa. Some significant corporations relied on WebObjects-based solutions for their e-commerce, including Dell until the Apple purchase of NeXT made it a conflict of interest. WebObjects was a much-needed success for NeXT, and if the acquisition had not happened, it is likely that NeXT would have survived into the 2000s relying on it as its primary product. NeXT developers would have been able to continue developing using NeXT-based technologies, and would probably have made good money doing it, but this would have been for corporate enterprise software. Moreover, WebObjects competed in a crowded field with a host of other web environments, especially those based on Java, Microsoft ASP, and PHP, which most of the dot.com startups were using. NeXT would have continued to be seen as a marginal technology in the industry. NeXT developers worked on contracts for already large enterprises, while the startups stuck to industry-standard solutions like Java. Thus, while many programmers joining startups during the dot.com bubble had hoped to become overnight millionaires, NeXT programmers largely worked on steady, but profitable contracts from existing large institutions, forgoing much of the dot.com hype and benefiting from the Internet boom less directly. This is very different from the experience of Cocoa programmers during the iPhone gold rush of 2008-10, where they were now at the center of tech startup activity and investor speculation.

My point is that NeXT and later Cocoa programming until 2008 was largely articulated as a labor of love and devotion for what was a marginal, even obscure

software technology, despite the fact that it was possible to make a comfortable living doing it. Programmers who wanted to strike it rich in 2000 joined Internet startups programming in Java, rather than work as contractors writing web backends in WebObjects. In 2002, they would be even less likely to consider writing consumer applications for Mac OS X, a platform dwarfed in marketshare by Windows, as a sure way to retire early, especially by taking risk onto themselves without investors. While issues of money were not unimportant to NeXT and Cocoa developers before 2008, it certainly was not the only or even primary motivation, as it would have been much easier to make money doing traditional Web or Windows development. This equation certainly changed after 2008, especially among most of the newcomers hoping to get in on the ground floor of the "mobile revolution." Nevertheless, my focus in this chapter is not primarily on these newcomers, but on the old guard of the Cocoa community, the true believers that had stuck with NeXT and Apple through tough times and were developing exclusively with NeXT/Cocoa long before iPhone apps were seen as the surest way to get rich quick. Where did this devotion to Cocoa come from? What sorts of affective pleasures, normative values, and ideological commitments motivate indie Cocoa developers? These are the questions I will examine.

Pleasure in Cocoa Programming

> AppKit [the user interface component of Cocoa on Mac] [is] a joy to use versus other things." (Chris Parrish, Interview March 2, 2012)

In Gabriella Coleman's study of free software hackers, she quotes a Python programmer, Espe*,* who describes the purity of coding in Python (a high-level object-oriented programming language) as reaching a transcendental state: "I… felt the pure abstract joy of programming in a powerful way—the ability to conjure these giant structures, manipulate them at will, have them contain and be contained by one another." (Coleman, 2013, 95) This programmer wrote Python code for the "joy of programming," "rooted in deep pleasure" of "unencumbered exercise of ample creativity." His reverence for Python was that it enabled him to "reach the elusive quality of perfection." (Coleman 2013, 97) Elsewhere, Coleman describes this

transcendental pleasure in programming as an experience of "flow" (Csikszentmihalyi 1994), a blissful "deep hack mode" where self-awareness is obliterated (Coleman 2013, 13).

Espe contrasted this experience of pleasure, order, and productive creativity in programming in Python with the frustration and chaos of programming in another language, Perl. Python programming was a "high tower of control and purity" compared to Perl's "bubbling pool of vagary and confusion" that was the "big ball of mud." (Coleman 2013, 95–96) Another programmer explained that Perl's critics deride it as "ugly, difficult to learn" and enforcing "bad habits." (Coleman 2013, 96) Coleman has noted that the pleasure of programming depends in large part on the tension between pleasure and frustration, and that overcoming frustration is part of the pleasure of programming itself. This frustration frequently stems from the material agency of the computer hardware, but also the constraints imposed by existing, "legacy" software infrastructures upon which higher level software, including applications, are built. Such software is obdurate in a different way—frequently encoding the social and institutional relationships that existed among the software's users and programmers at the time of its creation into a durable and agential form that frequently outlasts its original social context. This, in a nutshell, is the problem of software maintenance (Ensmenger 2010). Programming languages and APIs also exhibit their own form of constraints and affordances—they make possible or easy the ability to express certain ideas quickly in code what is impossible or difficult to express otherwise. Languages express different ways of approaching problem solving, and different programmers express strong preferences for particular languages because these best match how the programmer has become accustomed to thinking, reducing frustration and increasing pleasure.

Programmers who use the Cocoa APIs have until recently predominantly used a language called Objective-C to write their code. Because Objective-C, Python, and Ruby were all influenced by the Smalltalk object-oriented programming language, they all exhibit similar traits. All of these languages are classified as "dynamic," roughly meaning that they allow the objects that make up programs to alter their

properties, behaviors, or relationships dynamically while a program is running, which increases the expressivity and flexibility of certain kinds of code, increasing developer productivity. Moreover, Apple has designed the Cocoa APIs to be ordered and coherent. As a result, Cocoa programmers have commonly expressed a similar pleasure in Cocoa programming (and its precursor, NeXT programming) as *Espe* did of Python. This pleasure has been experienced so strongly that many Cocoa programmers have decided to avoid programming in other environments where possible, resulting in many of them releasing software exclusively for Apple's platforms. Many of them also have exhibited a strong tendency to try to "evangelize," in other words, convince others to write software for Apple so that they too, can experience the same pleasure. Indeed, Apple encourages this attitude by releasing new frameworks and APIs that offer developers powerful new capabilities or more convenient ways to do things they were already doing, reducing everyday frustrations and increasing their pleasure. Mark Dalrymple is an instructor at the Cocoa training company, Big Nerd Ranch, who wrote its *Advanced Mac OS X Programming* guide. For Dalrymple, Cocoa's conveniences allow him to achieve his aims with minimal effort:

> "What makes a programming language fun, or what makes a toolkit fun? And for me it's a combination of mastery… how well do I know the tools? It's like a musical instrument… Same with Objective-C. So I've achieved mastery in the language, …so… going from, here is what I want to do thought-wise, to the code that does it, is a very direct process. It's not error-prone… the results are fairly fast to get. I can go from idea to something running… fairly quickly…" because the surface area of the language is very small…" (Mark Dalrymple, Interview, April 11, 2012)

Dalrymple's proficiency with Cocoa allows him to get to the result quickly. The Cocoa toolkit has become an extension of his mind, like a musical instrument. When Cocoa developers contend that Cocoa is easier to use than other programming toolkits, they do not mean that it has completely deskilled programming into a turnkey activity; they mean that Cocoa has been honed to keep frustrating distractions at a minimum, allowing them to get on with their work. For Cocoa programmers, then, less frustration means more productivity, and more mastery, and

therefore, more fun. The Cocoa programmer is more like the Python programmer, who revels in the elegance and abstract purity of his programming environment, rather than the Perl programmer, who value the ability to express algorithms in cleverly terse ways. These are two contrasting sources of pleasure in programming. Cocoa and Python programmers see the freedom of Perl as debilitating, because by offering too many ways to do the same thing, it introduces unnecessary complexity and confusion. While Dalrymple is partly saying that his mastery and proficiency are the source of his pleasure (which a programmer expert in any language could say), he is also claiming that Cocoa/Objective-C has properties that allows him to get his results, meaning a completed application, not just one algorithm or code module, quickly. While Coleman points out that overcoming frustration is a necessary component of the pleasure of programming, Dalrymple's quote shows that not all frustration is equal. Unnecessary frustration caused by arbitrary complexity in one's programming tools or environment is seen as inefficient, getting in the way of the act of creation, and thus inhibits pleasure.

The idea that Cocoa/Objective-C, at least for a proficient programmer, is pleasurable precisely because it is less frustrating appears often when Cocoa programmers compare it to their experiences with other programming tools, environments, or platforms. Chris Parrish is an independent Cocoa developer living in the Seattle area who used to work on InDesign at Adobe, writing code in C++. Parrish described his experience with C++ as frustrating and complicated, which made him feel unintelligent:

> It was like the overhead of becoming competent enough to produce stuff in Objective-C was so low—it was like, this isn't a big deal. I was picturing the nightmare that is C++. [I thought] I'm just not as smart as these [C++] guys. …So I was picturing Objective-C would be another whole huge complicated mess, and then when I realized how it was just super simple… it's all straightforward, no big surprises. (Chris Parrish, Interview, March 2, 2012)

What differentiated his attitude from that of his fellow programmers who seemed to love C++ was that code was just a means to an end—the application itself,

whereas the C++ lovers seemed to attain pleasure in the writing of the code itself, and the pleasure in the mastery of C++'s arcane complexity:

> I just like to actually do stuff, like, I like to produce the result, rather than just the process of making stuff… I don't need to write code, if I could still make cool stuff without ever writing code, I'm cool with that… (Chris Parrish, Interview, March 2, 2012)

What Parrish reveals here is that, like other Cocoa developers, his priority is making an application that can be used by end-users. Although it is probably an exaggeration that he would prefer not to write code, for Parrish, the pleasure exists not primarily in the technical mastery of the clever hack, but the beauty of the finished product.

Other Cocoa programmers have expressed similar sentiments as Parrish and Dalrymple that Cocoa's efficacy at helping them achieve their ends is a source of their pleasure in it. Brent Simmons, the Seattle-area indie developer who originally wrote NetNewsWire, a newsreader app, sums this up: he likes Cocoa "because I can get my work done." (Brent Simmons, Interview, February 17, 2012)

This concern for the end product (the app) and not the code itself, makes Cocoa developers pragmatists when it comes to proprietary versus open source code. Tristan O'Tierney is a former Apple engineer and a co-founder and former CTO of Square. His view is that he will use whatever tool, open source or proprietary, which best helps him get the job done. Like Dalrymple, Parrish, and Simmons, O'Tierney feels that most of the time, Apple's frameworks and APIs help him write apps more quickly and conveniently. However, occasionally, Apple's solutions are not the best ones, and when he sees this is the case, O'Tierney has no problem writing his own frameworks and sharing them with others:

> You can also find open license code that does almost what you want. Maybe you just have to tweak it. […] The code is not what matters. Especially the reusable stuff. Because what matters is the user experience.

> […] in the end, what matters is that you deliver the final experience. And that is unrelated to the code. If you have code that helps you draw a button, there's no reason to do that a million times over.
>
> […]Just give out the code that is really unrelated to our secret sauce. The secret sauce was […] just our drive for [user] experience, for making good quality stuff.
>
> (Tristan O'Tierney, Interview, January 7, 2009)

O'Tierney's attitude towards the purpose of open-source is one I often heard myself when I was an Apple employee, which is not surprising given O'Tierney's own experience there. This view of the role of open source separates out infrastructural software code from application code, which interacts with the user. In this view, software infrastructures, such as low-level operating systems, are the expert domain not of Apple, but the open source community. Apple's expertise is instead in user interfaces. Lower-level functionality is merely the means to the larger ends of an artistic vision of a user experience. This means that Apple can leverage the work of the open source community for the infrastructure in order to focus its own talent on the user interfaces of its operating system and applications. This creates a hierarchy in which infrastructural software is seen as less interesting than applications that interact with users. For O'Tierney, this value system translates to third party development as well. If possible, a Cocoa developer should spend as little time as possible getting basic functionality to work and more time on getting the user experience "right." This means that if the developer can delegate this responsibility to code he does not have to write, whether it be an open source library, or a new framework from Apple, he should. "Less interesting" work should be delegated to someone else, which in practice means code that has no user-interfacing component, and thus does not express an application's overall vision. The focus on the user makes Cocoa developers app-centric; all other software layers exist only to serve the application, which ultimately serves the user. In the next section, we will look more closely at Cocoa developers' emphasis on the usability and aesthetic look and feel of their applications, which often differentiates them from other programmer cultures.

Commitment to Aesthetics and Usability

Cocoa developers, like open source hackers, may work on software to "scratch their own itch" and fulfill their own needs, and both derive pleasure from the writing of code. However, Cocoa developers are first and foremost Apple users, and have self-selected Apple because they believe in the values that ostensibly guide Apple's product design: that technology should be easy to use, not frustrating or overly complex, and that when designed correctly, can even be a source of pleasure or "delight." Thus, they seek not only to experience pleasure in the act of creating an application, but also to create a pleasurable experience in the act of using it. Cocoa developers recognize that they take cues from Apple on the value of the quality and usability of software:

> I think we do share at least a few values with Apple and Apple employees. Most developers I know are hugely committed to quality. The most important thing is to make what you make really, really good. And we define good in much the same way that Apple does. …User experience is paramount.
>
> […] We choose to adopt those ideals, we probably had them in the first place anyway, which is why we're attracted to [the Mac]. (Brent Simmons, Interview, February 17, 2012)

Others note that Apple has been a trend setter for aesthetics and design that has often been missing from other technology companies:

> [Apple's] focus on aesthetics and usability… I think they've been purveyors of good design. …Back in the day… when you turn on a Mac it smiles at you… it had a lot of personality to it. (Chris Livdahl, Interview, March 28, 2012)

For this developer, good design does not create merely an aesthetic response, but humanizes the computer, generating a personal connection to it that is missing from the experience of using other computers. The machine is no longer seen as a cold, unfeeling thing, but acquires a "personality" associated with the Macintosh's graphical user interface. A common trope associated with the Macintosh is that its users have grown sentimental attachments to the machines in ways that Windows PC users, who treat their computers as instrumental work machines and cheap

commodities, do not. The humanizing quality of Apple's interfaces attracted like-minded developers:

> For me, it's about being humane… the devices, the Mac, and the general approach of most developers in this platform, is to recognize our users as human beings, worthy of respect, and to build things that treat them that way. (Curt Clifton, Interview, March 23, 2012)

Thus, for many Apple developers, the Macintosh was the only computer platform that they experienced pleasure using, and this motivated them to develop exclusively for it.

> We were all on the Mac already, because it worked, and felt better. If we didn't care about that, we'd be on Windows… The [Macintosh] attracted a certain type of developer, and we all loved working on the Mac for that reason. (Gus Mueller Interview, February 21, 2012)

NeXT developers felt similarly about their platform, and when Apple bought NeXT, these sensibilities merged. Because both NeXT and the Macintosh had low marketshare, programming exclusively for either platform was a risky business decision, and for a developer to go Mac-only meant that they put their love of Apple's platform above the greener pastures of the Windows or Web development markets.

> Going back a ways… you had to be a person who was willing to try to make a living, or wanted to develop software for this minority platform, that many people would go, 'Why are you doing this?' It's not only Apple's image [as a beleaguered company]… but just the realities of the size of the market, really. There's only so many people with Macs. There's all these other people [on Windows]—why are you choosing to do that? Isn't that a bad business decision? There's lots of ways you can argue around that, but certainly I think it takes a certain type of person who's interested in going down that road.
>
> (Chris Parrish Interview, March 2, 2012)

If, for many free software hackers cleverness or efficiency matter most, for Cocoa programmers, providing a pleasurable experience to their applications' users is paramount. Although ingenuity in coding itself is still valued, for Cocoa programmers, an application's design, in terms of visuals, how a user uses the app,

and how the application's architecture is planned, all matter more than the raw efficiency of algorithms. Cocoa developers explicitly model their efforts to create elegantly designed software on Apple's:

> The reason we're probably attracted to this platform, has to do with things there might be about elegance and aesthetic and design sense, and usability, like those certain features that make a good Mac app a good Mac app, a good iOS app… Some people are drawn to that sensibility. (Chris Parrish, Interview, March 2, 2012)

This motivated many to see themselves as artists:

> I get along with [our designer] so well because we're both trying to express our vision. He's trying to express it in Photoshop, and I'm trying to express it in Objective-C…
>
> My vision is not threatened by your vision… the existence of Cezanne and the existence of Monet does not lessen the impact of Van Gogh. An individual artist and an individual vision stands on its own merits.
>
> (Mike Lee, Interview, July 23, 2008)

In order to put this quote in context, at the time of this statement, Lee, who had made his name in the Cocoa community working with indie Seattle developer Wil Shipley, was now CTO of a Palo Alto startup that made a Twitter client for iPhone. An executive at this startup had made public statements disparaging Twitterific, a competing application written by the developer Craig Hockenberry that was considered by many Cocoa developers to be one of the best on Apple's platforms. Lee's statement about artistic vision was an attempt to distance himself from his executive's remarks, which he considered to be counter to the norms of collegiality among indie Cocoa developers. Cocoa developers, according to Lee, should be neighborly and supportive of each other's work, even if their apps compete directly with each other in the marketplace. His executive, who had been a merger and acquisitions lawyer prior to co-founding the company, was playing by the cutthroat rules of the market and violated this collegial norm of the community. By proclaiming his identity as an artist, Mike Lee was asserting that despite two products actually competing in the rational marketplace, on a somewhat higher, artistic plane, they are not competing at all, allowing developers to freely coexist

with each other. As "artists," their visions singularly stand on their own, despite the fact that only one of them might get a customer's money. Lee even claimed that if one of his customers was unsatisfied with his product, he would be happy to point them to a competing one. Moreover, Lee argues that his creative work as a programmer is not dissimilar from the work of his fellow employee, a graphic designer who creates user interface elements in Photoshop. This is explicit identity work. Both Objective-C and Photoshop, in Lee's eyes, are artist's tools, like a brush or pen, and both exist to help artists express their creative visions. For Lee, making an app, though it has an instrumental purpose, is still an aesthetic act of design. In fact, an app's instrumental purpose is part of this design—the way each different app helps users accomplish their tasks is part and parcel of the app's overall "vision." Lee, coming out of the tight-knit, collegial indie Cocoa community was running into conflicts with the more cutthroat Silicon Valley culture that was driven much more explicitly by market and money concerns.

Lee's erstwhile competitor, Craig Hockenberry, was well respected in the Cocoa community in part because his app's user interface was carefully considered. Twitterific represented what made Cocoa developers different from developers on other platforms: their perfectionist concern with the aesthetics of their apps.

> There's another thing that differentiates Mac and Windows developers, I think there's more attention to detail, in general, with Mac applications. Just because the customers are more accustomed to having them. The Apple apps are all finely tuned and they spend a lot of time thinking about UI [user interface]. Your competition in the Mac space… you're not going to come out with something that, yeah, OK it's functional, but it doesn't look good! (Craig Hockenberry, Interview, January 7, 2009)

Hockenberry notes that this concern with aesthetics is derivative of Apple itself. Apple sets the standard with is own applications, which become exemplars for design in the Apple software market. In large part, third party developers like Hockenberry self-select to write apps for Apple platforms because they are attracted to Apple's design aesthetics and seek to emulate Apple and achieve the same high standards. Furthermore, Apple's users, and thus developers' customers, similarly

expect that apps on Apple platforms aim for high standards of usability and aesthetic beauty, and spend their money accordingly. Furthermore, Apple itself pushes these aesthetic standards by rewarding select applications with the annual Apple Design Award. For Apple developers and users, design, aesthetics, and usability are a primary reason they use and develop for Apple platforms. This concern has become a key boundary marker versus developers for competing platforms, with Microsoft often the key foil:

> I think there were some rare Windows developers who wanted their stuff to look good, but they were pretty rare. And just, they didn't seem to get that aesthetics helps usability, or can if done well, and with keeping usability a priority, aesthetics can help… in a good design, it's not just a gimmick.
>
> (Brent Simmons Interview, February 17, 2012)

This kind of boundary work against "those Windows developers" who are purported to not care about aesthetics or usability in their products is fairly common among Cocoa developers. However, sometimes Cocoa developers can take aesthetics too far, to the point where some apps are marketed on useless aesthetic flourishes, or "eye candy" that does not contribute to usability or function. Simmons and others noted the example of a disk burning application whose main selling point was the cute animation it made while burning. This was gimmickry, and aesthetic fetishism gone amok. Simmons here is referencing that infamous app, and rhetorically linking its overemphasis on aesthetics with Windows developers' under-emphasis, making them two sides of the same coin. The properly balanced Cocoa developer, Simmons is implying, does not need to resort to gimmicky animations to sell her app, but uses them judiciously, intelligently, and tastefully to make her app easier, and more pleasurable, to use.

Because usability is of such extreme importance for Cocoa apps, Cocoa developers expect each other to take responsibility for their applications' user interfaces, and this is enforced through peer pressure. This is particularly true because many Cocoa developers are indies who work alone, and thus cannot rely on a design or UX (user experience) department to handle art duties. Even for Cocoa

developers that work with designers, however, they are still expected by peers to have a sense of what good user interface (UI) might look like. According to Rusty Zarse, who runs the Atlanta iOS Developer Meetup, in other developer communities, developers are more likely to see a division of labor between programmers and designers, and thus disavow responsibility for UI:

> So in the Microsoft community, I would say half of the developers I worked with, at least, would say I'm not a UI guy… They just wouldn't take responsibility for it… and didn't feel competent… didn't show an interest. And I don't think I've ever had an iPhone developer ever say that same statement. Or a Cocoa developer say, I'm not responsible for the usability or the aesthetic of this app. They're responsible for the behavior as well as the aesthetic and so I think it definitely permeates in. Because when someone builds an iPhone app and its clunky looking… when they show the app, the first thing that their peers in the group are going to say is, 'hey did you think about doing this, and changing those things around, and polishing up those edges, it looks kind of clunky.' And then there's always the UI people that will say you need a designer. You need to find a graphic designer to help you out."
>
> (Rusty Zarse Interview, September 25, 2012)

For Zarse, taking responsibility for an app's aesthetics and UI clearly differentiates Cocoa developers from Windows developers. This trait has almost become a stereotype. Cocoa developers are often seen as spending inordinate amounts of time trying to adjust the pixels in a button to get it just right.

Although up to this point, we have discussed aesthetics and usability of the end product of Cocoa developers' work, their apps, these values also apply to the tools Cocoa developers use to make these apps. In this way, Cocoa developers are themselves users of Apple programming tools. In the same way that they experience pleasure using Apple hardware and applications, they say they experience a parallel pleasure using Apple's tools to create apps, a pleasure connected to the usability of those tools. Cocoa developers thus understood themselves as users as well as producers, and in this sense, they appreciated Apple's own attention to detail with designing its frameworks and developer tools. Curt Clifton, a programmer at Seattle's OmniGroup, noted this parallel explicitly:

> The ease of development…these are, for the most part… humane tools to develop with… And so [Apple] tend[s] to treat the developer with respect… the frameworks really are kind to us." (Curt Clifton, Interview, March 23, 2012)

Clifton's use of the word "humane" is a direct reference to the book, *The Humane Interface,* by Jef Raskin, the HCI researcher who was the original leader of the Macintosh team before it was taken over by Steve Jobs. The "respect" he referred to is from the perspective of the "interface" that Cocoa tools present to developers. In other words, Clifton is saying that Apple tries to make an effort to ensure Cocoa developers' interactions with Apple's programming tools (which include the Cocoa frameworks and APIs themselves) work in a way that could be called "easy to use." By qualifying this with, "for the most part," however, Clifton hints that the reality may not live up to this ideal. During 2011, many developers I spoke with complained about the buggy state of Apple's primary development tool, Xcode. Nevertheless, these developers still felt that the Cocoa frameworks themselves were excellent tools. Clifton's "respect" is also not referring to the way Apple as a corporation treats its third party developers. Many iPhone developers have complained, often publicly, of Apple's draconian and sometimes arbitrary App Store approval process, and other woes. Nevertheless, what Clifton is referring to here is how Apple's own software engineers have designed the Cocoa frameworks to interact with its users, which are Cocoa developers themselves. It is Apple's engineers, who try to treat their users (third party developers) with respect, via the tools they make for them.

The usability of Cocoa itself, like the idealized usability of apps written with it, is often rhetorically conflated with aesthetics. Brent Simmons described Cocoa in terms that evoked a feeling of technological sublime (Nye 1994):

> "You… can't help but just marvel at the elegance of it. …Cocoa certainly does [have a great elegant design]; and understanding that design and… its beauty… is a really, really good feeling. And that goes beyond just knowing how to get something done… that's an actual… aesthetic response." (Brent Simmons, Interview, February 17, 2012)

Brent Simmons has thus articulated a similar transcendent appreciation for what the hacker in Coleman's study described for Python programming. Unlike open source hackers, however, the motivation is not to participate in the construction of the tools, but to use them to make pleasurable experiences for everyday people, like Apple does. And as Steve Jobs was known to drive engineers at Apple to strive for perfection, if one wanted to emulate Apple, one had to become perfectionist as well:

> [You] produce the best of the best and settle for nothing less and you're passionate about what you do.
>
> (Rusty Zarse, Interview, September 25, 2012)

Zarse thus summarizes the moral attitude expected of Cocoa developers. Proper Cocoa developers ought to care deeply about "getting it right," making the highest quality applications that are not only easy to use, but evoke feelings of pleasure and comfort in their use. A 2014 ad that Apple showed to its developers at its developer conference proclaimed that Apple's goal was to "delight" its users with its products, and exhorted developers to do the same with theirs. For developers like Zarse, this striving for perfection also requires that developers carry within themselves a deep affective commitment to their work. It requires "passion." For many Cocoa developers, this means that software development cannot be approached as a simple nine-to-five job. One's work and one's career as a software developer must become part of one's very identity. As we see in the following section, Cocoa developers consider app development to be both a craft and a vocation.

Craft and Vocation

In *The Craftsman,* Richard Sennett defines craftsmanship as "the skill of making things well" and the human desire "to do a job well for its own sake." One story told about Steve Jobs by original Macintosh engineer Andy Hertzfeld was that Jobs wanted the team to rewire the Mac's original circuit board to make it look prettier, even though no user would ever see it. Jobs justified this on the principle of craftsmanship, noting "I want it to be as beautiful as possible, even if it's inside the box. A great carpenter isn't going to use lousy wood for the back of a cabinet, even

though nobody is going to see it." (Hertzfeld 2013a) Craftsmanship is not limited to manual labor but accompanies all forms of skilled labor that unify mental head work and embodied hand work, including programming, medicine, art, parenting, and diplomacy (Sennett 2008, 8–9). Craftsmanship is concerned with pride in the quality and excellence of one's work, a tendency that can lead towards obsession towards correctness, but is tempered with pragmatic concern towards functionality, the need to actually finish a product so it can be used for the purpose it was made. (Sennett 2008, 45–46) Steve Jobs, despite his famous perfectionism, encapsulated this tension with an aphorism, "real artists ship" (as in, "ship" their products") (Hertzfeld 2013b; Hertzfeld 2013c). Craftsmanship, as learned skill, is learned from others, thus requiring a community to transmit them to the next generation (Sennett 2008, 21–22, 51). Sennett described Linux programming as a craft due to the way that programming practice is continually opening up; even as problems are solved, new ones are being discovered, so that the skill of programming never atrophies or routinizes but must constantly evolve (Sennett 2008, 26). Sennett notes within both the Linux programming and Wikipedia communities a tension between a concern for quality, with its tendency towards elitism, and its democratic commitment to openness and knowledge sharing, a tension also noted by Gabriella Coleman in her study of Linux programmers (Coleman 2013, 120–122; Sennett 2008, 25–26).

As we have seen, Cocoa programmers are also extremely concerned about producing quality work in their apps, and thus see programming itself as an edifying, pleasurable activity of self-actualization. As Coleman has pointed out, learning and self-cultivation of skill is heavily valued in the open source programmer community. "Free software developers have come to treat the pursuit of knowledge and learning with inestimable high regard—as an almost sacred activity, vital for technical progress and essential for improving individual talents." (Coleman 2013, 119) It is likewise in the Cocoa programming community. An Atlanta area Cocoa programmer explicitly spoke of programming in the language of craft and apprenticeship:

> …Like other crafts of days of old, blacksmithing, or whatever, where there is some sense of respect for… the masters of the craft. …The apprentice wants to always strive to become that master, so that he can

> be the master for another apprentice coming along… For any craft you've got to spend time outside of [work]—you always want to improve your craft and you always have that kind of respect for other people that have built something really successful.
>
> (Robert Walker Interview, May 19, 2012)

Note that for Walker, craftsmanship implies a moral exhortation to perfect one's skill that normatively suggests that the craftsman spend time outside of normal working ours on self-cultivation of this skill. As we will see in a moment, this insistence that one's own leisure time be spent improving the craft is a key to the idea of one's craft as one's vocational calling. If one is not interested in spending time outside paid work hours doing the work, then the work is just a job, not a vocation.

Others explicitly posed the craft model of programming against what they considered to be the industrial model, which they associated with large corporate software firms: "We don't want the automobile industry to be the software industry. We want it to be the individual artisan." (Wil Shipley Interview, April 18, 2012) Wil Shipley is a developer who co-founded the company OmniGroup in Seattle and later set out on his own to make the digital bookshelf app, Delicious Library. For developers like Shipley, the ideal economic form for a creative programmer is to write software on one's own or in small-groups with like-minded friends, independent of corporate employers. Within the Cocoa community, such developers are called "indies," and we will examine this group in more detail in the next section. For Shipley, indie developers are not routinized or deskilled laborers on an assembly line, but craftsmen and artisans who go where their passions take them. OmniGroup began in just this way when Shipley and his friends Ken Case and Tim Wood from the University of Washington got together to write NeXT software together in the 1990s, and Shipley had the freedom to take on whatever projects he thought were fun. Over time, Ken Case and Tim Wood decided to focus on responsibly building a stable company with a structure and organization, rather than as simply a place to have fun coding with friends, and Shipley left to pursue his own projects. Although OmniGroup is still considered an "indie" company by most of the Cocoa community,

it probably has close to a hundred employees today, and Shipley left because he felt that it had gotten too rigid and bureaucratic for his tastes.

"Indie" Cocoa programmers consider their work to be a vocation. Because it can be highly pleasurable, app development blurs the line between labor and leisure, work and play, in a way that exemplifies the kind of intellectual work central to the knowledge-based "New Economy" driving the rise of what Richard Florida calls the "creative class." (Florida 2002) "It doesn't feel like work. You're playing all day long." (Robert Walker Interview, May 19, 2012) To an extent, programming languages, tools, and environments can be thought of as "hedonizing technologies" (Maines 2009), although the products of this labor are not inconsequential to developers. Indeed, not all Cocoa programmers are professionals; many pursue it as a hobby; some have corporate jobs writing code in other environments but work on iOS app projects in their spare time. Many spoke of having become a professional Cocoa developer only first by exploring and playing around with Cocoa on the side. Mike Lee, who apprenticed himself to Wil Shipley at his post-Omni company Delicious Monster, noted:

> "What I really wanted to do was be a programmer. And I had been doing web stuff for quite a while. But I really wanted to get into application development. And so I studied programming, …during my down time." (Mike Lee, Interview, July 15, 2008)

In this way, what started out as a hobby becomes a vocation. If work is play, the money one receives from performing it becomes almost incidental. "I do programming a lot for fun… I'm enjoying this, the fact that I'm getting paid for this is amazing." (Mark Dalrymple, Interview, April 11, 2012) Dalrymple called people like him who program for pleasure, "recreational programmers." This differentiated them from purely professional programmers who treated it merely as a nine-to-five job. "I'm a programmer, my nine-to-five is to execute code for this particular purpose; once that's done the computer is hung up… and when I go home I have no interest in the technology outside of my job. …[But] those folks tend not to be community leaders, because they have other interests outside of this community." (Mark Dalrymple, Interview, April 11, 2012) Robert Walker noted his opinion that

programming was a career that chooses you, not the other way around, (Robert Walker, Interview, May 19, 2012) explicitly invoking the language of vocation (Shapin 2008; Weber 1946). Another programmer I interviewed had the opinion that if one did not love programming, one should not do it as a job. Andrew Stone, a veteran Cocoa developer and neo-hippie counterculturalist, stated transcendent reasons for being a programmer: "My resonance with the Apple came from this psychedelic wisdom that this actually was the future. […] I came in for spiritual reasons… The financial success, that's awesome… But that's not what hippie-kids care about. For me, and our generation, it's more about this sense that my life actually mattered." (Andrew Stone, Interview June 7, 2011) For such developers, Cocoa programming allows them to pursue careers doing what they love. Daniel Pasco, founder of the indie development company Black Pixel, asserts, "We're here to make stuff. And… to make a living doing it… The goal is to actually have a rewarding life doing what we do…" (Daniel Pasco, Interview, March 28, 2012)

The word "vocation" implies religious overtones. To claim that one's job is a vocation is an ideological act that frames work, and thus profit-making, as a way to achieve a higher, transcendent purpose for one's life beyond mere worldly material accumulation. Max Weber explained in *The Protestant Ethic and the Spirit of Capitalism* that earthly success for the Puritan founders of America was not an end in itself, but a sign that a person was of the Elect, in a Calvinist religion in which one was constantly anxious about one's Predestined salvific status (Weber 1958). Although capitalism itself moved beyond this Calvinist way of thinking, hard work in one's God-given vocation continued to be equated with virtue in American capitalist ideology, and in this ideology, the wealth that inevitably resulted from hard work was merely the signifier of this virtue. In this way, although wealth is not itself the end, it is also not incompatible with one's vocation, but is a necessary by-product. Nevertheless, ideologically one cannot claim that wealth is actually the ends that work is intended to achieve; rather, the process of work itself is what is virtuous. This is what makes it a vocation, that the worker is called to do this, having been blessed with the talent and the passion to do so. Once wealth becomes the goal, the

work is no longer vocation but mere wage labor. This is why it is so important to programmers' sense of self and their purpose in the world to assert the vocational aspect of their work; they need to believe that there is a greater meaning to their labor beyond mere capital accumulation.

"Play" and "tinkering" with technology is one form of masculinity in Western culture, one that may offer pleasure through dominance over machines and technical competence (Wajcman 1991). An alternate form of technological masculinity might be one focusing on logic and analytical thinking. In the 20th Century, amateur ham-radio was a distinctly masculine hobby (Haring 2003), and there was considerably continuity between radio hobbyists and the first personal computer hobbyists. One psychoanalytic analysis suggests that men's fascination with creating technology derives from "womb envy." (Kleif and Faulkner 2003, 213) Sherry Turkle argued that men's fascination with computers represented a "flight from relationships with people" into an intimate one with the machine (Turkle 1984, 216), and also showed that boys enjoyed the feeling of mastery and power over the virtual world inside the computer. Wendy Faulkner argues that the power men feel when working with technology compensates for lack of power, and anxiety over uncertainty, experienced in dealings with people (Faulkner 2000b; Faulkner 2000b; Kleif and Faulkner 2003). Technology is much more predictable and controllable than human relationships. Since the 1980s, the "nerd" or "geek" has emerged as a cultural stereotype of an anti-social young man who spends all of his time with computers, electronic games, or genre-based media. Data reveals that the 1980s were a high-water mark for women's participation in computing (Hayes 2009), while Hilde Corneliussen shows how media portrayals of computing overestimated men's participation while underestimating women's (Corneliussen 2009).

Reinterpreting programming as vocational craft also genders this work in additional ways. The normative view that ideal programmers should be consumed by passion to code even outside of their job suggests that coding work has higher value than human relationships, including family. Mike Lee revealed this attitude during one of my interviews with him, in which he criticized a former female coworker who

he felt was not dedicated enough to programming because her first priority was her children. While Lee acknowledged this was her free choice, in his eyes, this made her a bad software engineer, and in his then current position where he made hiring decisions for his startup, prioritizing family would count against a candidate. Lee felt that software engineering made "a more valuable contribution to society than having children." (Mike Lee, Interview, July 23, 2008) Lee claimed that this attitude was not sexist, but applied equally to male or female candidates, and he felt that one of his male college interns similarly was not being a good programmer because he made no effort to socialize with Lee and the startup's other employees after hours. Nevertheless, he might overlook dedication to family if the developer in question had sufficient experience and reputation. Lee sought to hire a former Apple employee who insisted on working normal hours. This engineer's ex-Apple status and his expertise with Cocoa gave him a pass on the required performance of dedication to code, where Lee was concerned. The view that Cocoa programmers are craftsmen with complete dedication to their craft, requiring constant labor outside of normal hours, implies a traditionally gendered division of labor in which social and domestic work within the family is taken up by the programmer's spouse, allowing the craftsman to pursue his programming, which is seen as the more valuable contribution to society.

At the time I interviewed him, Lee was the CTO of a Palo Alto iPhone app startup with only a dozen employees, all male except for an administrative assistant, with a number of them in their early twenties. This gave the startup a distinct frat-house atmosphere. Some employees favored a highly-caffeinated soft drink called "Bawls," and jokes centering on the double-entendre were frequent. In this environment where boys could be boys, women employees, if there had been any, could have easily felt excluded. The atmosphere I witnessed at the startup could be described along the same lines as the sexist "brogrammer" culture of Silicon Valley startups that has been publicized recently (Hicks 2012; Parish 2014; Raja 2012). Things were not always this way. Programming had originally been considered feminized work (Light 1999), but efforts to raise its professional status in the 1970s

ended up excluding women (Ensmenger 2009). Today, the cultural association of programming with men is firmly entrenched. This state of affairs has not gone unnoticed among progressive male programmers, who lament it but often feel helpless to fix it. A 2014 podcast produced by a Cocoa developer focused on the problem of sexism in tech (Ritchie 2014). Christina Dunbar-Hester's study of a low-power FM radio activist group shows how deep-seated gender identities can hamper inclusion even among activists committed to equality. Technical experts in the "Geek Group" were mostly men, and performance of technical competence was experienced as performance of masculine identity, even among the few experts who were women, who had to negotiate a delicate balance between their feminine identity and their technical masculine one. This had the effect of dissuading women novices from wanting to acquire technical competence if it meant having to compromise on their femininity. Dunbar-Hester concludes that "In spite of the intentions of this small group of activists, the gendered technical experiences and skills that they bring to their site of work tend to overwhelm the ideal of equality, and even to reinforce the gendered divisions between them…" (Dunbar-Hester 2008, 223)

As we will see in the next section, the gendered view of independent programmers as lone individuals who contribute to society through making technology is associated with the ideology of technolibertarianism, which sees social change as being better effected through technology rather than bureaucratic politics or social activism. It also elevates the figure of the entrepreneur over the large monopolistic corporations that are seen as in cahoots with government. In the indie worldview, the action of thousands of independent entrepreneur-programmers, working through the market, will usher in a new utopia in which innovation thrives and society benefits.

Indies and Technolibertarianism

"Indie" developers like Pasco, Stone, Walker, and Dalrymple program for its own sake, for the pleasure of making apps for users. The money is supposedly incidental, except for the fact that it supports their livelihoods doing what they love,

as they claim that they would do it for free as a hobby anyway. These programmers might all make considerably more money if they joined a startup in traditional Silicon Valley fashion, but instead, forgoing potential earnings by rejecting corporate control is what gives them the prestige amongst their peers in the Cocoa developer community. It shows that they are more devoted to their art than becoming instant millionaires through a sudden acquisition or IPO. Their social capital in the Cocoa community derives from the more edifying purpose of their creative labor.

"Indie" developers are programmer-entrepreneurs who are independent of corporate software firms and work on their own self-directed software projects as they please. Because going into business alone is risky, it typically requires some saved up capital accumulated from a prior job, as well as already developed programming skills and the computer hardware to program on. All the indie developers I have spoken with come from middle to upper-middle class backgrounds. Most spoke of childhoods or adolescence tinkering with and possibly programming personal computers, which means that at early ages, they already had begun to acquire both the skills and access to the material artifacts, the capital goods, necessary for a life of programming computers. Overwhelmingly, indie developers are Caucasian, with a few exceptions, such as Mike Lee, who is half-Asian and originally from Hawaii. The vast majority are men, especially the older generation of Cocoa Mac OS X developers. These Mac Cocoa developers also tend to be of middle age or older, in their upper forties or late fifties. A few independent iPhone developers I encountered have been women, but these women are not well known in the community for famous applications, nor do they have must-read blogs or wide Twitter followings. The famous names in the Cocoa indie community are almost all men, with the exception of Erica Sadun who not only writes a personal blog but also was editor and senior writer of the Apple fansite, The Unofficial Apple Weblog or TUAW, at http://www.tuaw.com/editor/erica-sadun/, accessed February 7, 2012. Sadun is probably better known for her blog posts than for her apps, however.

The term "indie" is an actor's category, used to describe artists and smaller companies in the film, music, and video game industries, which are "independent" of

the dominant corporate firms. The term connotes an artistic and cultural authenticity that comes from creative autonomy from the profit-maximizing interests of corporate content producers, who are concerned with a lowest common denominator mass-market blockbuster or chart-topper. Similar logic applies to "indie" software developers.

According to indie developer Brent Simmons, the term "indie" came into use in the Mac developer community around 2002 or 2003, only a year or two after the release of Mac OS X, when development of consumer applications using NeXT-derived Cocoa technology became possible. As is common in the community, this first occurred on blogs, an Internet medium that was also gaining widespread traction in that same era. "We didn't call them Indie developers in those days, I think that started in 2002 or 2003, I think it was a blog post by Buzz Anderson, actually, that it got us to stop using the word 'shareware' and move to the word 'Indie.' Because the term 'Shareware developer' was [used] throughout the '90s..." (Brent Simmons, Interview, February 17, 2012)

The term indie replaced the term "shareware." In the 1980s and 1990s, avocational programmers often wrote software and freely distributed it over BBS or commercial online services, or at local user groups such as the Berkeley Mac User's Group, by passing out floppies. Users were encouraged to donate $5 or $15 to the author by mailing in a check, if they found the software useful to them. Shareware was a 1980s-era compromise in the emerging dispute among hackers over intellectual property. As discussed by Fred Turner (2006), and shown in the documentary, *Hackers: Wizards of the Electronic Age* (Florin 1985), the 1984 Hacker Conference convened by Stewart Brand included commercial PC game developers, Apple engineers such as Steve Wozniak, as well as free software pioneer Richard Stallman. At the conference, the idea that information (software) should be free (both to acquire and to further modify) seemed to conflict with the notion of the programmer as creative auteur, whose creative work should be protected as well as compensated. Shareware was a middle ground: software, produced by individuals, was distributed for free (though not its source code); users who felt that its author

should be compensated for their work would voluntarily give them a donation to keep working on it. For a lucky few whose applications became widely used, the authors were able to make a commercial business out of shareware; but this very success stretched the economic model of gifting rather than payment. The more successful shareware packages began to require registration keys to unlock full functionality, or timers that would shut down full functionality after a trial period. Nevertheless, for a shareware author looking to commercialize and compete on a level field with corporate firms, the barriers were significant. Corporate firms sold software in shrink-wrapped packages and dominated expensive retail shelf space in brick and mortar stores. Van Meeteren's work on Cocoa indies argues that it was the advent of the commercial internet, and the dot.com boom which created the infrastructure of e-commerce and electronic payment and distribution of software, that made the indie possible as an economic entity. (van Meeteren 2008) Freed from the burdens of either competing for retail space, and relying on mail-in donations for payment, small operation programmers could become a more stable business. It was in this new economic environment that the term "indie" began to replace "shareware" to describe small operation Mac programmers.

Indies are the logical endpoint of the vocational drive among Cocoa programmers—making apps of one's own creation. Its ideology disavows money as an indie's primary motivation. Rather, pursuing one's passion for programming as a way to make manifest one's creative vision is seen as the ultimate raison d'être of the indie. This is coupled to a belief that making software will help people become more productive or enrich their lives, and thus improve society. In this way, a lone individual writing code, working through the mechanisms of the market as a small businessman, makes a contribution to society without recourse to politics.

> "Indie is to me, it's just an ethos. …you're part of a culture of… I'm not in this for the money, I'm in it to make something cool, and to make the whole environment better for everyone…
>
> (Wil Shipley Interview, April 18, 2012)

Independence from corporate control is required for the creative autonomy necessary to be an indie:

> "What is Indie?" …If your agenda is… to have complete creative control, and that takes precedence over what will make us the most money—and you have the freedom to make those choices—that is the definition of Indie. It doesn't mean broke or small. It means that you're actually calling your own shots, and not beholden to someone else. (Daniel Pasco Interview, June 12, 2009)

The "indie ethos" also encapsulates all of the previous values Cocoa programmers profess: vocational and craftsperson identity, which focuses on the pleasure of making and on quality, self-cultivation of skills and knowledge, and a commitment to a community of practice in which this knowledge is shared. Indies also share a belief in the empowering (and democratizing) effect of technology on individuals, and seek to participate in that empowerment through making apps for themselves and others. This latter value, we will see, is one heavily promoted by Apple and is central to Apple's own corporate identity.

Being "indie" connotes small-scale, though not necessarily individual, production of apps. Indies, from the perspective of the Cocoa community, can be companies started by a two or three like-minded developers, such as OmniGroup or Black Pixel, that later grew to about a hundred employees. At this size, it can be difficult to articulate why a company of OmniGroup's size is an indie while smaller ones might not be. For one, the company must be founded and controlled by developers (and sometimes user experience designers), not by a "business person." Thus, unlike many other technology startups, those who hold the "indie" identity reject funding from angel investors or venture capitalists, seeing such money as coming with strings attached, giving away creative control to the money people. Indies are about making whatever apps the employees themselves want to make—they are not founded for growth, to attain an IPO or become an acquisition target, but simply to make enough profit to be self-sustaining. The goal is to be a small business, like a country store, or in the case of OmniGroup or Black Pixel, a medium-sized, privately-owned business, in perpetuity. This is markedly different from the mindset

of most Silicon Valley entrepreneurs, whose goal is to found and grow the next Facebook or Instagram and make a billion dollars; either result would be seen by indies as "selling out." This category is somewhat fluid—Instagram may have been considered an indie until it was acquired. For Cocoa developers, what constitutes being or remaining "indie" is continuing to retain creative control over one's business and products.

Of course, this does not mean that money does not matter to an indie. Despite common assertions that "we're not out to make money," many of the well-known indie developers easily make hundreds of thousands of dollars a year, enough to afford expensive toys like Tesla electric sports cars. Each of these indies, however, would claim that they might have made much more money working for a company like Microsoft, joining a VC-funded startup, or selling their company off. "We actually tell people we're not interested in being acquired; we're not interested in being invested in," proclaims Daniel Pasco of Black Pixel. (Daniel Pasco Interview, March 28, 2012) What matters to developers who call themselves "indies" is that they reject the potentially higher earnings they could achieve by selling to a larger company or accepting investment capital in order to maintain control over their own work.

Indies must care about profits to sustain their small businesses. Indies worry about cash-flows a great deal, which means that practically speaking, most indies are not completely self-sufficient through sales of their own apps, but supplement their income with corporate contracts. Even OmniGroup and Black Pixel, companies well known in the Cocoa community for original applications, have relied on contracts for a significant portion of operating income. The iPhone boom has resulted in enormous demand for skilled Cocoa developers from corporations which want a "mobile app" presence in the same way they all suddenly needed a website during the dot.com boom. For many indies, contracting is a lot more secure and lucrative than trying to make one's own app, as an expert iOS programmer can command a rate anywhere from $100 to $150 an hour (Patel 2010). Because indies have rejected investment capital, they are self-funded, and this involves considerable financial risk. Most

would-be indies start by writing an app in their free time off work, and only those lucky enough for their apps to succeed are able to quit their day jobs. Others decide that they want to go indie ahead of time, and save up money from either a day job or contracting to build a reserve of capital on which to sustain themselves while they work on their app full-time. However, the days of the iPhone App Store gold rush, when stories abounded of programmers making thousands of dollars selling apps written in a weekend, are long over. The App Store is crowded with apps that do similar things, and unless one is featured prominently by Apple, or cracks one of the top 25 lists in iTunes, it is difficult to rise above a handful of downloads a day. Would-be app developers can easily spend months slaving away, only to find, once their app is on the store, that they are making only a few hundred dollars a month, and have to go back to a regular job or take a contract. Says one successful developer, "It's either feast or famine. It's hard to go indie on iOS. …I mean that's like winning the lottery, right?" (Gus Mueller, Interview, Feburary 1, 2012) The only true indies are those who have managed to make their app work self-sustaining. For every indie who has made it, there may be ten more programmers working on apps on their free time, eking out a few hundred downloads a day. Despite this risk, however, indies are constantly striving to shake off their corporate clients and become fully independent and self-sustaining. While the actual number of successful indies is dwarfed by the majority of those trying to make it, their influence on the Cocoa community is magnified through their blogs, Twitter feeds, and conference presentations, and it is the voices of these prominent indie Cocoa developers that set the agendas of the community's discourse.

For a number of indie Cocoa developers, Apple's opening of the iPhone to third party development through the App Store has unleashed a wave of entrepreneurship that they see as an indie revolution. Before the iPhone, being a Cocoa indie developer meant catering to the Macintosh's relatively small marketshare, and releasing Mac-only software meant dedication and devotion to the platform. Apple's iPhone App Store, which takes care of digital distribution of software for the developer, has significantly reduced the barriers to entry for

independent software entrepreneurship. Many Cocoa developers saw this as a boon, a way to democratize programming for the masses: “This is like my wildest dreams come true. Millions and millions of indies!” (Andrew Stone, Interview, June 7, 2011) This has convinced some that the future belongs to such individual, decentralized production of software, replacing large corporate production of software: “It’s not driven by [the large software firms] anymore. It’s, what is the next Tiny Wings going to be?” (Wil Shipley, Interview, April 18, 2012) It is particularly striking how much the iOS “revolution” sparked these utopian visions among the core Cocoa developers despite the subsequent sobering realization that the vast majority of independent iOS developers with their own apps could not sustain themselves. Longtime indie Cocoa developer Brent Simmons noted by 2014 “almost all the iOS developers [in the Seattle area] are making money either via a paycheck (they have a job) or through contracting… Some money for iOS development is coming from companies like Omni that do create products—but most of it appears to be coming from corporations that need apps (or think they do). Places like Starbucks and Target. The dream of making a living as an indie iOS developer isn’t dead… but, if I’m right, hardly anyone believes in it any more. [sic]” (Simmons 2014) What is important is how committed the core members of the Cocoa community, who saw themselves as a revolutionary vanguard, were to this vision of utopia, even in its failure to materialize.

Many Cocoa developers see the iPhone App Store as Apple’s response to a huge demand among iPhone users to extend and customize the iPhone’s functionality. When the original iPhone was released in 2007, Apple’s policy was that developers would not be allowed to develop apps that ran “natively” on the device, but rather could only write web applications that were tweaked to run well in the iPhone’s web browser. Much of Apple’s developer community understood the iPhone to be not just a cell phone, an iPod, or a web browser, but a fully-fledged mobile Macintosh computer. Once hackers discovered how to “jailbreak” the iPhone, effectively circumventing Apple’s security protections and allowing programmers to write software for it, an underground market of apps written for jailbroken iPhones

sprouted. Responding to this user appropriation, in 2008 Apple announced that it would provide an officially sanctioned Software Development Kit (SDK) for third party developers to use, and an App Store that would allow developers to sell their apps to users who did not jailbreak their phones, legitimizing the app market but also putting it fully under Apple's control. This "curated" app market has turned out to be hugely profitable for both third party developers and Apple itself (as Apple takes 30% of app sales revenues.) By opening up the iPhone with an SDK, Apple allowed developers to extend the iPhone to do things Apple never originally intended.

This view of the App Store as empowering and democratizing small-scale technology creators has a lot in common with the DIY Maker and Hackerspace movement. The emergence of indie mobile app development has largely coincided with the emergence of the Maker movement, and there are Hackathons centering on iOS app production ("iOSDevCamp" 2014). I do not claim that indie iOS or Mac development is a subset or extension of the Maker movement, as there are some notable differences. Much of the Maker movement is aligned ideologically with the open source software movement, as the recent controversy over DARPA funding of hackerspaces shows (Savage 2013). Cocoa developers' reliance on Apple for proprietary tools and hardware thus contradicts the value of open participation in production and repair of both hardware and software. Despite this, much of the rhetorics and ideologies informing Makers and indie Cocoa developers are similar.

The DIY Maker movement has been hailed as democratizing production and transforming passive consumers into participatory producers, with a particular focus on technical education and pedagogy (Ames et al. 2014; Tanenbaum et al. 2013). DIY makers see their work as self-actualizing craft (Sivek 2011). Made possible by the availability of open and affordable technologies such as 3D printing and Arduino circuit boards, DIY making started out as a hobbyist practice, but has now generated VC funded startups hoping to sell products to consumers. Indeed, much of Maker culture seems to harken back explicitly to the 1970s counterculturally-inflected personal computer hacker/hobbyist culture, where open sharing of computer hardware knowledge produced Apple Computer. Nostalgia for that time is prevalent

among the promoters of DIY Making. "…if you look back into time, you see what's happening in the 60s. The 60s brought advances in computing. Its pioneers were people like Wozniak and Steve Jobs. They were makers, hackers, academics, and entrepreneurs. But this time around it's different. You have Kickstarter and VCs... Hardware startups today can really make anything possible. Today, DIY means that anyone can take a product to the market, with the support from the crowd." (quoted in Lindtner, Hertz, and Dourish 2014, 441) Nor do Makers necessarily see business as incompatible with openness: "That the commitment to countercultural ethics was not perceived as antithetical to structures of the market economy is what we would like to emphasize here. Many… considered such alignments essential in order to move DIY making beyond a hobbyist practice." (Lindtner, Hertz, and Dourish 2014, 442) Nevertheless, the Maker movement's emphasis on pleasure and self-actualization, originating in a privileged class in the West, sits uneasily with a burgeoning field of Makers in the developing world. Silvia Lindtner has pointed out that among Chinese makers, an explicit focus on business was nothing to be ashamed of, and disagreed with Mitch Altman's exhortation that DIY making had to be about doing what you love. (Lindtner, Bogost, and Bleeker 2014)

Like the mobile app craze, the hype surrounding DIY making for solving society's problems taps into technological utopianism (Sivek 2011). Making and apps both focus on self-actualization and the empowerment of the individual, and both fit into a discourse about decentralized production in the Knowledge Economy. Among technologists, technological utopianism has combined with neoliberalism into what critics have variously called *cyber/techno-liberalism/libertarianism* (Borsook 2000; Turner 2006; Malaby 2009). Borsook has documented Silicon Valley's disengagement and distrust of government, favoring technological innovation as the proper way to intervene in, and improve, society. The emergent and self-organizing properties of technology are seen as similar to the (ostensibly natural) workings of the market, and technolibertarians see both as superior means to enact change over what they see as the corrupt give and take of Beltway politics. Indeed, in an interview with Steven Levy in 1983, Steve Jobs remarked, "I'm one of

those people that think Thomas Edison and the light bulb changed the world more than Karl Marx ever did." (Bilton 2014) Borsook notes that technolibertarianism animates both free software hackers and Microsoft employees, (Borsook 2000, 24–26) and its primary ideological proponent has been *Wired* magazine, particularly in its early years. Paulina Borsook, a former contributor to *Wired,* has also grouped technolibertarians roughly into two categories: *gilders* (cultural conservatives like George Gilder and *Wired* co-founder Kevin Kelly) and *ravers* (counterculturalists like EFF co-founder and Grateful Dead lyricist John Perry Barlow, also a former *Wired* contributor). *New York Times* columnist David Brooks has argued that bohemian counterculture has merged with bourgeois capitalism to produce the new Information Age ruling class (D. Brooks 2000). John Markoff noted the countercultural connections with the early personal computer industry. (Markoff 2005) Indeed, Apple is the poster child of counterculturally inflected corporations, celebrating in its famous Think Different ad campaign, "The crazy ones. The misfits. The rebels. The troublemakers." In the 1990s, NeXT and its developer community continued to have ties both to the counterculture and to technolibertarianism. NeXT stayed afloat financially due to an investment from Ross Perot, earning him an endorsement for President from Steve Jobs in the 1992 election. (Ruby and Jobs 1992, 33) John Perry Barlow was a contributor to *NeXTWorld* magazine, itself a precursor to *Wired* (its first issue ran a cover story on futurist Alvin Toffler). Independent NeXT developer Andrew Stone, a neo-hippie himself, became a personal friend to Barlow and EFF co-founder John Gilmore. In 1992, he threw a rave party after the NeXTWorld Expo conference. Like Stewart Brand and Timothy Leary, Stone has connected computer use to psychedelic and transcendent experience:

> …A transformation that occurs…by the Will of God… at times in our life when we work on software for four days and don't sleep… these states of consciousness… they call it flow… when you get that passion that drives you crazy, you do awesome work…
>
> Everybody who's creative knows what I'm talking about… To find meaning in being a tech… that's our identity… doing this project… it's about liberation… we're after the magic!

(Andrew Stone Interview, June 7, 2011)

For Andrew Stone, Jewish, Hindu, and Zen Buddhist mysticism mixed freely with cybernetic and psychedelic modes of expanding human consciousness, explicitly evoking the experience of flow, (Csikszentmihalyi 1994) shared also by free software hackers (Coleman 2013, 11–13) and machine gamblers (Schüll 2012).

Fred Turner's *From Counterculture to Cyberculture* explains the connections between countercultural and technolibertarian ideals during the emergence of the personal computer and Internet industries. Values traveled and transformed along networks of people, with Stewart Brand and his *Whole Earth Catalog* crossing the boundaries between different communities. Turner argues that the *Catalog* served as a *network forum*, a "place where members of these communities came together, exchanged ideas and legitimacy, and in the process synthesized new intellectual frameworks and new social networks." (Turner 2006, 72) Turner contends that network forums have properties of both Susan Leigh Star and James Griesemer's notion of "boundary objects," which "can be a media formation such as a catalog or online discussion system around or within which individuals can gather and collaborate without relinquishing their attachment to their home networks" and Peter Galison's notion of a "trading zone," which is a site "where representatives of multiple disciplines come together to work and, as they do, establish contact languages for purposes of collaboration." (Turner 2006, 72) Within Brand's network forums, cybernetic ideas and technologies from the military-industrial complex mixed with the drugs and buckskins of the New Communalist hippies, the non-activist, utopian branch of the counterculture. From this juxtaposition, Brand proclaimed that the use of tools would empower humans to master their environment, liberate them from their bureaucratic oppressors, and elevate them into latter day gods. This empowered human merged the notion of a "Comprehensive Designer" who surveyed the world through information with the frontier image of the lone Cowboy Nomad. The bricolage of the *Catalog* served to legitimize cybernetics among the counterculture and bohemian art worlds. After the breakup of the commune movement, Brand began to travel in new networks with the hackers of the

computer liberation movement, and into research labs like Xerox PARC. This second legitimacy exchange transformed the PC nerds into the cool inheritors of the countercultural radicals. In 1984 Brand hosted a hacker conference, attended both by Richard Stallman, as well as Apple co-founder Steve Wozniak. After this, Brand extended the *Whole Earth Catalog* into cyberspace with the *Whole Earth 'Lectronic Link* (WELL), which brought together a network of countercultural technology enthusiasts, including Barlow and Kevin Kelly. By the 1990s, Brand had embraced entrepreneurialism and created the Global Business Network. Both the Electronic Frontier Foundation and *Wired* grew out of these networks. Each of these network forums, from the *Whole Earth Catalog*, the Hacker Conference, the WELL, the GBN, and *Wired*, brought together disparate communities into a shared sense of purpose involving tools and technologies, creating a new community. The communities created by each previous network forum would help constitute the next. (Turner 2006)

Turner shows how the blindnesses of *Wired* technolibertarianism can be located in Brand's version of countercultural New Communalism. For one, by rejecting politics and governance, the communes ended up falling back on charismatic leadership, producing autocratic systems and falling apart once the leaders departed. Traditional gender norms were reinforced. Most communards were middle class white escapees from the suburbs, and the communes often ran into conflict with local communities of blacks and Latinos. Moreover, Brand envisioned power as held by individuals, and amplified by tools: "personal power is developing—power of the individual to conduct his own education, find his own inspiration, shape his own environment, and share his adventure with whoever is interested. Tools that aid this process are sought and promoted by the WHOLE EARTH CATALOG." (Brand 1968) Tools would enable a cybernetic mastery over one's environment, conceived of as an information system. Thus empowered, the "Cowboy Nomad" would "consume knowledge and information and carry it with him on his migrations" and "become a member of an information-oriented, entrepreneurial elite." (Turner 2006, 88) This has become clear in the information-

based New Economy, in which employment is increasingly insecure and based on networks. “However, to the degree that the libertarian rhetoric of self-reliance embraces a New Communalist vision of a consciousness-centered, information oriented elite, it can also permit a deep denial of the moral and material costs of the long-term shift toward network modes of production and ubiquitous computing. For Stewart Brand and, later, for the writers and editors of *Wired*, the mirror logic of cybernetics provided substantial support for this denial… As taken up by the New Communalists, this vision produced two contradictory claims, one egalitarian and the other elitist… those who could most successfully depict themselves as aligned with the forces of information could also claim… to have a ‘natural’ right to power, even as they disguised their leadership with a rhetoric of systems, communities, and information flow.” (Turner 2006, 260) Thus, a central contradiction in techno-libertarianism is the duality of control and empowerment: an empowered individual can use his mastery to control others. Steve Jobs’ quest for perfection has justified Apple’s draconian levels of control over its technology.

Indeed, this tension between elitism and egalitarianism is a central one in Apple’s corporate message, and it reflects in the indie Cocoa developer community as well. As we have seen, indie Cocoa developers celebrate independence from control by corporations, and yet, unlike the open source and maker movements, is relatively content with consuming tools provided by Apple, a critical dependency on the largest corporate IT company. In his ethnography of Linden Labs, the company behind the online world Second Life, Thomas Malaby describes a similar dependence on proprietary tools, despite an ideology of access to tools and open participation in creation. Unlike other massively multiplayer online games, Second Life is based explicitly around users creating their own worlds, tying into a similar discourse of participatory peer production that motivates DIY making and open source. However, access to tools is controlled by Linden Labs, which paternalistically decides what tools users ought to have access to, for their own protection: “ ‘Most game developers don’t release all of their tools because so many of them are just one-offs that they do really quickly… and therefore have a lot of

holes in them in terms of the user perspective [and] can be dangerous.' …An emerging tension appeared around Linden Lab between tool users and the tool creators…" (Malaby 2009, 60) This same dichotomy exists in the Cocoa app development world, where Cocoa developers are mostly tool users, consuming what is provided by Apple.

The Ideology of Apple and the Mythology of Steve Jobs

How, then, do Cocoa developers justify this dependency? Indeed, Cocoa developers have frequently griped about the state of the Xcode IDE or other tools that Apple provides. Yet, on the whole, they claim that the Cocoa frameworks provide the best, and most enjoyable, tools for programming on any platform. Richard Sennett, in *The Craftsman,* noted that the Wikipedia community must deal with the tension between maintaining the quality of the content presented and the egalitarianism of participatory production (Sennett 2008, 25–26). Within the Cocoa community, the elitism of craftsmanship wins out. Cocoa developers have deliberately chosen to use Macs and iPhones because they believe that Apple has designed them better than anyone else could. This concern for quality also animates their own desire to be independent and have complete creative control of their own products, but it does not mean participatory design. "So in the end, you need some sort of benevolent dictator, because design by committee does not work." (Tristan O'Tierney, Interview, January 7, 2009) "I joke that Apple is like the Soviet Union but with way better products." (Brent Simmons, Interview, February 17, 2012) Cocoa developers tend to see Apple as an enlightened philosopher-king, whose mandate is maintained as long as Apple continues to give them high-quality tools and products, and addresses their concerns. Moreover, Cocoa developers are not primarily concerned with participating in the development of their tools—they are concerned with making their own apps, crafting pleasurable user experiences. Lower level details should be delegated to Apple: "I would say that there's lots of advantages to letting developers worry a lot more about what matters, like… the experience, and cleaning up all the… UI issues… than having to worry about how

am I going to make this fast, or… run on multiple platforms." (Tristan O'Tierney, Interview, January 7, 2009)

Indie Cocoa developers acquiesce in Apple's control as long as they trust that Apple is benevolent, has their best interests at heart, and shares their values. How is this trust created and maintained? Certainly, listening to developers' feedback and improving their tools to make them more powerful or convenient is one way. Longtime Apple developers have learned that, over time, their concerns will eventually be addressed, though maybe not immediately. However, they also understand that at other times Apple pursues its own interests, which sometimes runs counter to their own. In these cases, developers must trust that in the big picture, Apple shares their values and that they have the same goal: to empower users by creating easy to use, and experientially pleasing technologies.

Earlier, we saw that some developers defined "indie" to mean whether a developer had complete creative control, and that the size of a company did not matter. If this is the case, by extension, Apple, despite its status as a billion-dollar corporation, is actually the quintessential indie company—after Jobs' return to the company, he had essentially complete creative control. In this way, in developers' minds, Apple is transformed into an indie like them.

For this to be effective, Apple's developers must be convinced that ideologically, they and Apple have the same basic mission. This is not that difficult to do, because most indie Cocoa developers are Apple users first, and they are self-selected. In this way, the quasi-religious devotion Apple engenders among its users is also true of its developers.

Much has been written about Apple users as a "cult-like," a metaphorical religion (Belk and Tumbat 2005; Campbell and La Pastina 2010; Kahney 2004; Robinson 2013). Robinson has examined Apple's use of religious tropes in its marketing, drawing on a long American historical tradition in locating transcendence in technological progress (Noble 1999; Nye 1994; Nye 2003). This "religion of technology" is not merely a cynical ploy to sell more products to consumers, but

constitutes an emotionally persuasive ideological system that gives Apple's leaders, users, and third party developers a sense of identity, belonging, and purpose. The "religion of technology" has motivated a whole generation of Silicon Valley technologists to devote their lives to making individually empowering tools, which they see as their contribution to social change, giving their lives higher meaning. Technology is their way of, in Steve Jobs' words, putting "a dent in the universe." (Sutter 2011) This kind of technology worship constitutes a form of technolibertarian ideology. It posits that the best way to enact social change is not through the messiness of political engagement or social activism, but to work on technologies that are seen as the solutions to every problem. This view justifies a retreat into individual engagement with machines or virtual worlds rather than people or institutions. If political libertarians put their faith for social good in the efficiency of the self-organizing market, technolibertarians put their faith into technology, which, if seen through the technologically deterministic lens of such commenters as Kevin Kelly and Ray Kurzweil, takes on a self-organizing, even natural inevitability.

In life, Jobs enacted this ideology through his own charismatic leadership, drawing in both his employees and the wider public. Former NeXT and Apple employees spoke of Jobs' powerful effect on them: "

> I really believed in what we did… Steve [Jobs] had a way of… making you feel like you were doing something… important… worthwhile …noble. […] The technology was really great, but… Steve… infused that company with a sense of purpose. (Julie Zelinski, Interview, April 24, 2012)

Third party developers felt this too:

> I think the campfire around the NeXT is a campfire around Steve. How can you be more of a fanboy than, "you're right! The Mac does suck! Let's design something better!" (Andrew Stone, Interview, June 7, 2011)

Jobs famously developed this cult of personality in his famous Keynote speeches at conferences, especially MacWorld Expo and Apple's own Worldwide Developer Conference (WWDC), which became legendary for his big reveal of

revolutionary new products. While Jobs was undoubtedly a master showman, these Keynotes had the ritual quality of a church revival meeting, in which the audience's reaction was carefully manipulated by the presentation, scripted to feel unscripted, casual, and intimate.

For people who did not know Jobs personally, including most Apple developers and users, Jobs' charismatic authority is supplemented by his status as an exemplar of the virtuous technological life, especially since his death. Accounts of Jobs' life have been extremely popular, and these cannot be easily separated from the story of Apple itself (Deutschman 2000; Isaacson 2011; Moritz 2009; Young and Simon 2005). These accounts fit rather neatly into established mythological and religious tropes (Belk and Tumbat 2005). Jobs begins as a troubled youth, searching for meaning in countercultural pilgrimages to India and an Oregon commune. Apple's founding is a typical creation myth, birthed in the proverbial garage. Manichean battles with corporate bureaucracies ensue, some external (IBM, Microsoft, Google and Samsung), some internal (board members, Apple CEO John Sculley). After losing one internal battle, Jobs is exiled from Apple from 1985-1997, where he is a voice in the wilderness, crying out against the sins of Microsoft-dominated mediocrity. Then, as Apple itself falls from grace, Jobs triumphantly returns as its savior, ushering in a second golden age.

The story of Steve Jobs and Apple has become the new myth of our information age, speaking to technologists of many stripes. Silicon Valley entrepreneurs hoping to become the next Facebook see Apple as the progenitor of the technological rags-to-riches story. DIY Makers see themselves in the early Apple, with its origins in the hobbyist culture, although they identify more with Wozniak, the quintessential hacker/trickster figure. And indie developers see in Jobs' attention to aesthetics, commitment to quality, and his remaking of Apple in his own image, their own aspirations to vocational craftsmanship and creative autonomy. Commitment to the highest standards of quality brooks no compromise, which is equated with mediocrity.

In the wake of Jobs' death, many Apple developers noted that Jobs' greatest legacy may not have been the technologies he shepherded into the world, but Apple itself. Jobs, an admirer of the counterculture, Bob Dylan, Zen Buddhism, and *Autobiography of a Yogi*, infused his values into his company and his successors. Most Apple developers remain confident that Apple will continue to innovate as long as it remains true to these values. Their trust in Apple is also faith. Umberto Eco once compared the Macintosh to Catholicism, and MS-DOS to Protestantism or Calvinism (Eco 1994). As a Catholic myself, my own interpretation of Eco's statement filters through my experiences as both a Catholic and an Apple fan. Eco was referring to the differences between the user's interaction with the two respective platforms' interfaces, the graphical user interface (GUI) of the Mac versus the command-line interface (CLI) of DOS. Eco asserted that the Mac's GUI was "cheerful, friendly, conciliatory; it tells the faithful how they must proceed step by step to reach—if not the kingdom of Heaven—the moment in which their document is printed. It is catechistic: The essence of revelation is dealt with via simple formulae and sumptuous icons. Everyone has a right to salvation." The GUI, like Catholicism, carefully lays out instructions for laypeople without requiring them to understand deeply, and in this fashion, promises to make salvation (or computing) accessible to all. DOS, however, "allows free interpretation of scripture, demands difficult personal decisions, imposes a subtle hermeneutics upon the user, and takes for granted the idea that not all can achieve salvation. To make the system work you need to interpret the program yourself: Far away from the baroque community of revelers, the user is closed within the loneliness of his own inner torment." (Eco 1994) The command-line may allow for more freedom, but it requires considerably more effort, and indeed struggle and study, on the part of the user. This means that not all users can necessarily achieve their goal; it is not universally accessible as is the "Catholic" GUI. Eco remarks that Windows represents a kind of Anglican-like schism from the Mac, allowing the possibility of return to direct interaction with the Word (the command-line).

While Eco was making a statement about user interactions, from my perspective, the Catholic metaphor could apply as well to Apple's social organization and its relationship to its users and developers. As noted earlier, even among their fans, Apple and Steve Jobs were understood to act in autocratic, though in their eyes, mostly benevolent, fashion, as a Platonic philosopher king. Certainly, the hierarchical Catholic Church fits the benevolent monarch trope. As an Apple user, I myself do not always agree with Apple's decisions, but I have faith that overall Apple will remain true to the values that drew me to its products; I recognize that in order to have the user experience I prefer, I have to sacrifice some flexibility. Similarly as a Catholic, I may not always agree with the doctrines of the Church hierarchy, but I prefer to remain in the fold, in part because being Catholic is part of my identity, but more so because I have faith that the Church as a whole, despite the temporal shortcomings of its administrators, has holy intentions.

Indie Cocoa developers, with their devotion to Apple tools but their insistence on independent creation of apps, reconcile this tension ideologically—they have already chosen Apple because they share its mission of empowering and delighting users with easy to use technology, and they agree that this democratizing mission must to some extent be top-down, to ensure the highest levels of quality. Yet, it is not completely top down, for developers have asserted their prerogatives to extend the iPhone and iPad with their own apps, albeit within Apple's control. Practically, developers reconcile this tension by maintaining that their job (indeed their vocation) is to create the overall vision of their apps, and craft the user experience, and as many tasks unrelated to this ought to be delegated to Apple—whether lower level engineering, which Apple's Cocoa frameworks handle, or business tasks like distribution and payment, which Apple's App Store takes care of.

As we have seen, indie app development is just one kind of technological production in a continuum from open source peer production to VC-backed entrepreneurship, all of which are aspects of today's techno-utopian countercultural capitalism, with its focus on creativity, pleasure, and higher purpose in work. This ideology has helped propel Apple's profits to become the largest technology

company in the world. Apple's indie app developers share in this ideology, and despite their small size, can have outsize leverage on the user experiences of iPhone customers who download their apps. What users experience as the iPhone is not made solely by Apple, but is co-produced alongside millions of third party app developers.

Chapter 2: Revenge of the NeXT Nerds: Object-Oriented Programming, the Quest for Productivity, and the Vindication of the NeXT Community

In the previous chapter, we examined the affective, normative, and ideological components of the techno-cultural frame of Cocoa programming. We have not yet looked at the technological component of this frame, Cocoa itself. The next chapter (Chapter 3) will discuss this, focusing on the technical arguments for why Cocoa programmers believe that Cocoa technology provides a superior way to write applications software. We will see that much of that discourse focuses on the productivity of the programmer. Cocoa is better because it makes programmers more productive, Cocoa adherents argue. In this chapter I provide some historical context for understanding this discourse. The chapter follows several interweaving narratives: a business history of Steve Jobs' NeXT and its creation of the technology that later became Cocoa, and a story of the early third party NeXT developer community, which formed the core of the later Cocoa community. Interleaved through these narratives is an analysis of the productivity discourse that NeXT used to market its proto-Cocoa software platform, and that third party developers used to justify their devoted commitment to it, despite a danger that NeXT might fail and their livelihoods might be on the line. This productivity discourse stemmed from NeXT's embrace of object-oriented programming as the basis of its operating system. I argue that NeXT software's promise of order of magnitude improvements in programmer productivity can be located in widespread 1980s-era hype over object-oriented programming's potential to solve the so-called "software crisis" that had been first articulated in the late 1960s. I end this chapter with the recollections of Cocoa developers who began programming on the NeXT and stayed loyal to the platform till the present. In their eyes, the success of Cocoa today, which powers the iPhone and its App Store, proves that NeXT software's advantages for productivity were ahead of its time, vindicating their devotion to it through difficult times.

NeXTSTEP and Object-oriented programming

The history of Cocoa as a cult technology seems to have created a community in which the oldest members are also its most die-hard adherents. Before it was ever known as Cocoa or was the developer kit for the iPhone or the Mac, the technologies were part of NeXTSTEP, the operating system developed by NeXT. Much less has been written about NeXT than about Apple. According to biographies and other accounts of Job's ouster at Apple, (Deutschman 2000; Isaacson 2011; Stross 1993; Young 1988), Jobs founded NeXT both to redeem himself and to one-up the Apple executives who had kicked him out. After poaching key members of the Macintosh team to help start NeXT, Jobs attracted many of the best and brightest of the computer industry and academic research to his new company, among them Avie Tevanian, a Carnegie Mellon computer science Ph.D. who created the Mach kernel that would become the basis of NeXT's operating system, and Bertrand Serlet, a researcher at Xerox PARC. NeXT was supposed to be a better Apple, and Jobs wanted to make bleeding-edge computers that would be light-years ahead of the Macintosh which only a few years before, Jobs had shepherded into the world. Blaine Garst, a former NeXT and Apple engineer, noted that Jobs' motivation to beat Apple was part of NeXT's mission statement: "When he formed NeXT he wanted to build a better Apple, and that was actually in the… one-paragraph mission statement and it was basically 'Here at NeXT we're trying to build a machine that is better, the better machine. A better version of Apple for you and your friends and colleagues' use.'" (Blaine Garst Interview, April 13, 2012)

In pursuit of this goal, NeXT's hardware, the NeXT Cube, would be state of the art, including a Digital Signal Processor (DSP) chip that would give the computer advanced sound and graphics capabilities for a desktop machine. Jobs also insisted, in a move that proved to be too forward thinking, to not include a floppy drive but use instead a magneto-optical drive, which turned out to be too slow and expensive to be feasible. All of these capabilities would be housed in a sleek, black metal cube, manufactured in a state-of-the-art automated factory in California.

The NeXT computer's software would be based on the Unix operating system, an operating system first created at AT&T in the 1970s for minicomputers, that had,

by the 1980s, become popular for desktop scientific workstations such as those sold by Sun Microsystems. Because Apple had previously achieved significant success in education, and Jobs believed that NeXT's best shot would be to initially target American higher education as its key market, Unix's heavy use in academic computer science likely played a role in this decision. Because Unix had run on larger computers from the beginning, it was technically more sophisticated than the disk operating systems that ran on early microcomputers, including that of the Macintosh. However, in the 1980s, Unix was still primarily a command-line based operating system, lacking the Macintosh's friendly graphical user interface. NeXT's version of Unix would build a graphical user interface on top of BSD, the open source Berkeley Standard Distribution of Unix popular in academia, along with Tevanian's Mach kernel, making it the best of both worlds. The new operating system would be called NeXTSTEP.

There was one other key piece of NeXT's technology: object-oriented programming. Like Unix, by the mid-1980s, object-oriented programming was a bleeding edge topic that was in vogue in academic computer science, and beginning to make in-roads into industry. Object-oriented programming is a methodology whereby programs are conceived of as arrangements of objects which communicate with each other. Programmers could better model real-world systems by modeling them as objects in code and combining them together in a complex, dynamically interacting system. In this way, object-oriented programming differs from procedural programming, the way all programs had been conceptualized before. Procedural programs are conceived of as processes, flows of operations between loops and branches, not as arrangements of abstract things. (Hence, procedural programs are depicted graphically using flow charts, while object-oriented programs are depicted using object diagrams.) What computer scientists consider to be the first object-oriented programming language, Simula-67, was created in 1967 for simulation. In the 1970s, Alan Kay at Xerox PARC coined the term "object-oriented" to describe his own language, Smalltalk, that he hoped would be a learning tool for children. Smalltalk, which ran on the Xerox Alto, was more than just a language, it also

incorporated a development environment and the first graphical user interface (GUI), which, famously, was copied by Steve Jobs' team at Apple for the Lisa and later the Macintosh. In those visits to Xerox PARC, Jobs and Apple had focused on the GUI, but had ignored object-oriented programming, as the technology was based on adding levels of abstraction that would have required significantly more powerful hardware than the personal computers Apple would be selling. However, by the late 1980s, at NeXT, the academic computer scientists Jobs was hiring had convinced him that object-oriented programming was the wave of the future, and that it had to be the basis for NeXT's development environment.

NeXT eventually adopted Objective-C, a language developed by Brad Cox, an outspoken advocate of object-oriented programming. One of its advantages over Smalltalk was compatibility with C, the procedural language developed at AT&T in conjunction with Unix, and which by the 1980s had become the dominant language in the computer industry. C++, also developed at AT&T as an object-oriented superset of C, would become the most popular object-oriented language by the 1990s because of its compatibility with C. Why did Objective-C, which also was an object-oriented superset of C, not become popular? One reason was that Objective-C's object-oriented half was based on Smalltalk, a language radically different from C. The resulting "bracket" syntax looked nothing like the procedural C half of the language, and many C programmers had difficulty getting used to it. C++, on the other hand, became popular in part because it did not force object-oriented concepts wholesale on C programmers, but allowed them to adopt it piecemeal, and its object-oriented dot syntax seemed to be a natural extension of C's (Zepcevski 2012).

Using the Objective-C language, NeXT created a number of object libraries that developers could use themselves to help them build programs without having to rewrite a lot of basic functionality for themselves. For example, to facilitate usage of the NeXT's DSP and sound capabilities, NeXTSTEP provided the SoundKit. An early DBKit provided rudimentary database capabilities. The FoundationKit, later known simply as Foundation, contained the fundamental objects which all Objective-C programs needed to run. Most importantly, for building applications with graphical

user interfaces, NeXT provided the Application Kit, known more commonly as the AppKit. Influenced by the GUI object libraries of Smalltalk, the AppKit provided a higher level, more abstract, and much faster way to create graphical applications than the low-level, procedural Pascal and C-based Macintosh Toolbox interfaces on the Apple Macintosh. A major contributor was that NeXT made programming itself partly graphical. Its development environment included Interface Builder, an application that a developer could use to graphically "wire up" interface objects, such as buttons or menus, to the actions or commands they were intended to execute. This graphical method, while not removing the need to write code completely, seemed to many NeXT programmers to be a revolution in the way they designed programs—they could now prototype a user interface using Interface Builder, allowing them to change and iterate over the design quickly, and then turn the prototype into the real application itself by writing the code the user interface linked to, rather than being forced to rewrite the actual application from scratch.

NeXT's marketing made a big deal of these capabilities, pointing out how the AppKit libraries and Interface Builder could improve programmer productivity by an order of magnitude. For example, a brochure advertising the advantages of NeXTstep software (the capitalization of operating system's name changed a few times) had an entire spread devoted to Improv, an innovative new spreadsheet from Lotus, which was written using NeXT's object kits from the ground up. Improv was widely praised and fondly remembered by former NeXT users for its innovation. Subsequent pages of the brochure focus on NeXTstep's development advantages, and how it allowed such innovative software to be produced so quickly:

> "On one level, NeXTstep is the user interface that makes all NeXT computers so very intuitive and visually interesting. On another, it's a development environment that revolutionizes the way software is conceived and created. In fact, it's the entire reason why the companies we just mentioned [Lotus, Ashton-Tate, WordPerfect, Adobe] could create such extraordinary software in a fraction of the time it would have taken with other computer systems.
>
> But even more revolutionary is the fact that NeXTstep is just as accessible to you. So, for example, if you're creating customized

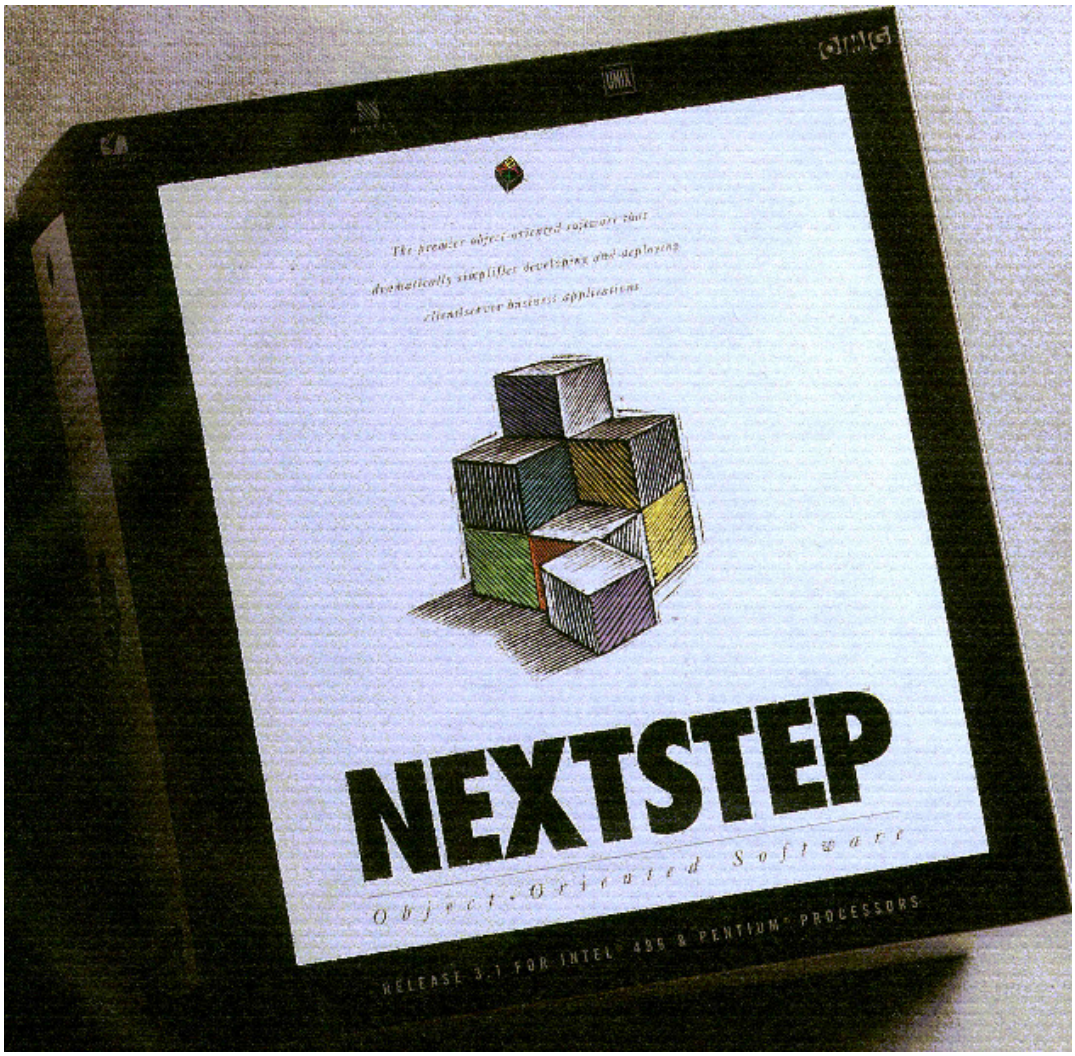
> software for people who take care of personnel, customer service or payroll, you can use the same tools Lotus used to create Improv…
>
> The NeXTstep environment is an object-oriented world. It's purely graphical, making Unix® easier to work with than DOS, OS/2,® Macintosh® or Windows™ environments. And it runs on every NeXT computer.®
>
> One of the most extraordinary parts of NeXTstep is Interface Builder,™ which lets you create an elegant application interface using little more than the mouse. You can choose from a palette of interface objects (such as menus, buttons and sliders) provided by the Application Kit. ™ Then edit, link and arrange them the way you want them to appear in your finished application. In addition, you can easily build new palettes of objects that you design yourself. Or add your own customized objects to the NeXT Application Kit.
>
> So with Interface Builder, you can rapidly generate a graphical front-end to a corporate database. You can also do some fast prototyping of new applications—which makes it that much easier to test your software with the people who will ultimately use it.
>
> And the interface you create, which may have taken 90% of your time previously, now takes less than 5%…
>
> Most important, the programs you create are… real, industrial-strength applications—every bit as fast as the applications you buy off the shelf, and every bit as complete.
>
> Applications you develop with NeXTstep are modular, too, so you can reuse portions whenever you see fit. And they're extremely easy to maintain. Now, when you update, there's no need to rewrite your whole application—you simply update the parts you want to change.
>
> In the words of the NeXT Development Team at Lotus, 'NeXTstep is the best development environment available on any personal computer today.' There really has never been an environment anything like NeXTstep. And no machine is built to support it like a NeXT computer.
>
> (NeXT Brochure, NeXT Computer, Inc. 1990, Box 26. *Software and Peripherals,* Mahoney Papers, Charles Babbage Institute.)

A few things stand out in this ad. Most striking is that the ad emphasizes how easy, and how simple, it is for the user to create his or her own custom software. There are a number of reasons for this. Firstly, despite the few large applications from corporate firms touted by the ad, NeXT had a very small marketshare and very few applications, and a lot of the time, users had to create their own. Secondly,

however, the emphasis given to Interface Builder and its graphical way to build programs was in part the logical extension of the graphical dream: to democratize the next hurdle in computing, programming. This would be how Steve Jobs could one up his previous creation, the Macintosh. While the Mac had democratized computer use through the graphical user interface, NeXTstep, through the combination of object-oriented programming kits and the graphical programming tool Interface Builder, would bring programming to the masses. This was not too far from Alan Kay's dream of making programming accessible to children with Smalltalk, or Seymour Papert's vision of using programming as an educational children's tool with Logo. And indeed, Brad Cox, who created Objective-C, had also hoped that, in conjunction with a market of off-the-shelf, reusable software object modules, users could easily write their own programs: "The programmer shortage can be solved as the telephone-operator shortage was solved: by making every computer user a programmer." (Cox 1990b, 30) NeXT, and Steve Jobs, was only the latest in a line of those who envisioned a future where users were empowered because they could write their own computer programs. Nevermind that this utopia was overblown—NeXTstep programming then, and Cocoa programming now, is not easy, Interface Builder not withstanding. Indeed, Interface Builder did not remove the need to write code completely. As we will explore later, the effect of Interface Builder, AppKit, and other technologies which make certain tasks more convenient (and thus more productive) for the programmer was not to deskill those tasks, but rather, to increase the amount of knowledge and understanding necessary. Because the AppKit and Interface Builder hide a lot of things from the programmer, in order for the programmer to know what to do when he or she actually needs to write code, the programmer actually needs to understand what the AppKit is doing on a much deeper level than if the programmer wrote everything him or herself.

Secondly, a conclusion that can be reached by the end of the ad is that NeXT is selling object-oriented programming itself—or at least, the advantages of NeXTstep as an object-oriented environment. It touts the modularity, reusability, and maintainability of applications written with NeXTstep—all advantages associated

with object-oriented programming. The fact that object-oriented programming is itself the product is made a lot more clear by the following ad from NeXTWorld magazine.



THE WORLD'S ONLY OBJECT-ORIENTED SYSTEM SOFTWARE IS NOW AVAILABLE IN THIS BOX.

Byte Magazine said it's "...probably the most respected piece of software on the planet." *Business Week* said that it's "...years ahead of its potential rivals, such as Microsoft Corp.'s Cairo..." NEXTSTEP™ for Intel® 486 and Pentium® Processors is the worlds' first object-oriented system softwa

Now it's here. Polished and perfected. Shrink-wrapped and shipping. In user and developer versions. For deskt and portable systems.

Figure 1: NeXTSTEP 486 advertisement

(*NeXTWORLD Magazine*, October 1993, p. 24)

The ad makes clear that NeXTSTEP is the world's "only object-oriented system software." At this time, rivals Apple and IBM were teaming up to create their own object-oriented operating system, Taligent, while Microsoft attempted its own project, Cairo, neither of which shipped to the public. The small letters on the box box above the cubic building blocks, which represent objects, say "The premiere object-oriented software that dramatically simplifies developing and deploying client/server business applications."

This ad must be put into context with the state of NeXT's business at this time. The ad is for NeXTSTEP 486, a version of the operating system that ran on the Intel processors used in IBM compatible PCs. By 1992, it was clear that NeXT's hardware business was a failure. Less than 50 thousand machines were ever shipped. (Wil Shipley Interview, April 18, 2012) NeXT had made significant mistakes. According to Randall Stross, Steve Jobs, having made his millions at Apple so quickly, had not been running NeXT in a frugal fashion, and had burned through much of its funding (and a considerable amount of his own fortune) pursuing perfection (Stross 1993). In order to settle a lawsuit with Apple, NeXT had signed a non-compete agreement, which meant that it could not sell machines in the same price range as Apple. Nevertheless, NeXT tried to make a general-purpose computer with a graphical user interface, but much more advanced, and more expensive than the Macintosh. NeXT initially targeted universities, but its machines were too expensive to compete with Macs and IBM PCs, yet slower than the Sun and HP workstations at a similar price. It tried to target desktop publishing, a market already cornered by the Macintosh, and networked office computing with a buzzword that failed to catch on, "interpersonal computing." However, because of their price premium, it was not clear why customers should choose NeXT machines over Macs or IBM compatibles. Unsure if they were PCs or workstations, NeXT computers were solutions looking for a problem. These problems were only exacerbated by a disastrous deal with computer retailer Businessland.

While NeXT was still trying to sell hardware, it needed third parties to write desktop productivity applications to become a viable consumer platform, and Steve

Jobs had used his industry fame and his considerable charisma to convince some large software publishers, such as Adobe and Lotus, to make applications for the end-user, including Lotus's innovative NeXT-only spreadsheet, Improv. However, it was NeXT users at many of NeXT's academic installations who needed no convincing to tinker with NeXT's included developer tools to write apps. These users became some of the first independent NeXT developers. One of the first was Andrew Stone of Albuquerque, NM, the counterculturalist and friend of EFF founders John Perry Barlow and John Gilmore who we met in chapter 1. Stone was a former architect who had been introduced to NeXT at a Macintosh User's Group, attended a NeXT Developer Training camp, and by 1992 had decided to try his hand at selling his apps over the nascent Internet, distributing software packages for free but selling license codes over e-mail. Stone became well known in the NeXT developer community for his drawing application, Stone Design, and for a database, Dataphile. (Andrew Stone, Interview, June 7, 2011.) Other small-scale operations included Glenn Reid's RightBrain, which offered a desktop publishing alternative to Aldus Pagemaker, and Lighthouse Design. Co-founded by Jonathan Schwartz, Roger Rosner (now at Apple), and Kevin Steele, among others, Lighthouse produced an office suite of applications, which included a spreadsheet and a presentation tool, as well as an innovative program called Diagram! which could be used to quickly draw graphs and flow-charts.

Although NeXT made its money from selling boxes, NeXT developers argued that, the sleekness of the hardware notwithstanding, NeXT's biggest competitive advantage was its object-oriented development environment. Eventually, NeXT, decided to hone its marketing to focus on this message, arguing that the productivity advantages of NeXTSTEP's development environment were ideal for rapid application prototyping, and "custom mission critical applications." This term referred to the kinds of problems that were critical to the core business or mission of large Fortune 500 corporations or government institutions, so important to their existence that the cost of a solution was not a primary factor. NeXT would close down its money-losing hardware business, which included selling its state of the art,

automated but mostly idle California factory to Canon. Instead, it would sell licenses to its software and development environment as a custom, Do-It-Yourself solution for tens of thousands of dollars each, allowing the company to remake itself and survive. The downside was that, with no hardware, and marketshare continuing to slide, the packaged, "shrink-wrapped" NeXT software market contracted considerably. Most of the corporate firms such as Lotus, Wordperfect, and Adobe, exited the market. Many smaller, independent firms, like Glenn Reid's RightBrain, which had focused on desktop publishing, folded. Lighthouse Design sold itself to Sun.[17] Stone Design, which produced a drawing program and a database, contracted to a one-man operation. However, the corporate market for custom software was considerable. The reality was that Fortune 500 companies, and especially Wall Street banks, could not simply pick up NeXTSTEP 486 and sit down any old programmer to create a custom application for them. They needed developers experienced with NeXTSTEP, and contracted out to existing NeXTSTEP developers, which were few in number. The ad, which sells the NeXTSTEP operating system and development to be run on generic PC hardware, instead of proprietary NeXT machines, is targeted precisely at this small, high profit, low-volume market of large institutions.

Going back to the brochure, another feature marketed is the notion that applications, particularly ones with graphical user interfaces, can be written in a "fraction of the time." "And the interface you create, which may have taken 90% of your time previously, now takes less than 5%—a streamlining that could put a serious dent in your backlog of projects." An article in NeXTWorld Magazine, touting the advantages of small, independent, and fanatically devoted NeXT developers, note that "they're bringing out new applications five to ten times faster than they would be able to on any other platform." (Garfinkel 1992a, 86) Another

---

[17] Sun acquired Lighthouse to provide office applications for Sun's Java platform, but these never saw the light of day. However, Jonathan Schwartz eventually rose to the position of Sun CEO before overseeing its acquisition by Oracle.

article states, “Steve Jobs often says that NeXT can’t just be better than a competitive platform; it has to be five to ten times better… Happily for NeXT… [for] large companies in a few big vertical markets [that NeXT targets with its custom-apps message]… NeXT does offer order-of-magnitude advantages… customization… The software needs to be modular and extensible.” (Ruby 1992) In an interview by NeXTWorld editor Dan Ruby, in explaining his company’s new marketing strategy focusing on mission critical custom application development, Steve Jobs responds:

> Getting applications written is the number one problem in corporate American information technology. …the bottleneck is still getting them written.
>
> This is even more true when customers want to use computers for operational productivity as opposed to management productivity. You can’t buy shrinkwrapped software to do stock trading or run your hospital or do order processing. You’ve got to write custom apps. Now, in the past, these operational applications were written in COBOL… on a mainframe or minicomputer. Starting in the very late ‘80s, some companies started downsizing to client-server computing. They could buy a Sun and spend two years writing a good app, or as good as you could write on a Sun. Now, we roll in and say, look, you can write that custom app five to ten times faster on a NeXT. (Ruby and Jobs 1992, 30)

This metric, “five to ten times faster” long predates NeXT. The language of an “order of magnitude” increase in programmer productivity is one that goes back to the late 1960s, when software professionals first began to talk about a “software crisis” and began to propose solutions to it. Object-oriented programming gained steam in the 1970s and 1980s as one proposed solution to this crisis, culminating in the claim of Brad Cox, creator of Objective-C, that it was a “silver bullet” that would help usher in a “software industrial revolution.” NeXT’s operating system and development environment was created in the late 1980s in this context, and in the 1990s it would be marketed as another “silver bullet” with the language of “five to ten times” better programmer productivity.

The Software Crisis and Object-oriented programming

Recently, historians of computing have written a lot about the so-called "software crisis," first articulated in the 1960s, and the emergence of software engineering, structured programming, and formal verification of program correctness as possible solutions to the crisis (Abbate 2012; Ensmenger and Aspray 2002; Ensmenger 2010; Mahoney 2004; Slayton 2013a; Tomayko 2002). By 1968, a number of different problems seemed to converge, creating a sense in the computer industry that software was becoming a problem. For one, software costs were outstripping hardware costs; a widely plagiarized graph projected software to make up 80% of computer costs by 1978 (Slayton 2013a, 155–7). Moore's Law would seem to exacerbate this, with hardware improving by orders of magnitude over software. Secondly, a number of large, complex software projects had failed, most famously that of IBM's OS/360 effort, but also software on NASA's Mariner I spacecraft (Abbate 2012, 92) and MIT's timesharing MULTICS operating system, which inspired Unix at Bell Labs. (Slayton 2013a, 111–2) Thirdly, demand seemed to be causing a shortage of programmers and fourth, the programmers that were available seemed to be unmanageable, as programming had the reputation of a "black art" and a craft skill, rather than a quantifiable, scientific or engineering discipline (Abbate 2012; Ensmenger 2010).

These latter two were related. As Ensmenger notes, the labor shortage was not "an absolute shortage of programmers but rather a shortage of a *particular kind of programmer*. [emphasis in original]" (Ensmenger 2010, 18) "An early study at IBM suggested that exceptional programmers were ten times more efficient than their merely average colleagues. The alleged 10:1 performance ratio quickly became firmly embedded in the cultural wisdom of the industry." (Ensmenger and Aspray 2002, 6) Another IBM study declared "that a good programmer was at least twenty-five times more efficient than his or her merely average colleague. Whether the exact ratio of performance was precisely twenty-five to one (or a hundred to one—another commonly quoted figure) did not much matter. What did matter is that whatever its

deficiencies, this study and others seemed to confirm plentiful anecdotal evidence that good programmers appeared to have been 'born, not made.'" (Ensmenger 2010, 19) Fred Brooks, who had led the failed OS/360 effort and famously wrote *The Mythical Man-Month* to examine the software crisis and catalog possible solutions, cited a similar study: "Within just this group [of experienced programmers] the ratios between best and worst performances averaged about 10:1 on productivity measurements and an amazing 5:1 on program speed and space measurements! In short the $20,000/year programmer may well be 10 times as productive as the $10,000/year one." (F. P. Brooks 1995, 30) Clearly, there appeared to be an order of magnitude difference in programmer skill and productivity.

A number of different solutions were proposed over the years to address the software crisis. Some were explicitly managerial solutions. Brooks expanded on a top-down proposal by IBM's Harlan Mills, known variously as the Chief Programmer Team, Surgical Team, or Superprogrammer approach. As the title of Brooks' book indicates, Brooks argued that throwing more manpower onto a software project did not speed it up but rather slowed it down, as the organization got more complex. Programmer teams should be kept small, with 10 people or less. Moreover, because of the order of magnitude difference in programmer productivity, all of the high level design decisions should be done by the chief programmer a.k.a. head surgeon, with subordinate implementation tasks delegated to the supporting programming staff. Brooks argued that this top-down "aristocratic" division was necessary to maintain the conceptual integrity of the program's architecture. It did not deskill the work of the supporting staff, as implementation still required creativity and skill, though not as much as architecture. "The opportunity to be creative and inventive in implementation is not significantly diminished by working within a given external specification, and the order of creativity may even be enhanced by that discipline." (F. P. Brooks 1995, 48)

Another proposed solution, which came to be known as "structured programming" came from Dutch computer scientist Edsgar Dijkstra. "The main tools of structured programming were abstraction, modularity, and the use of conditional

loops (rather than "go to" statements) to model the logical structure of the process that was being automated." (Abbate 2012, 99) Dijkstra published a letter to the editor of *Communications of the ACM* in 1968 proclaiming "Go To Statement Considered Harmful." (Dijkstra 1968) Dijkstra's point was that the Go To statement, a command that allowed a program to arbitrarily jump around to any line of code, created a spaghetti-like maze that would be impossible for other programmers to follow. By restricting programmers to using only conditional branches, loops, and subroutine calls, programs would be much easier for human readers to follow. Structured programming also incorporated programming practices that called for making software more abstract (less machine-dependent), modular and reusable. More importantly for Dijkstra, however, is that these techniques would also make programs more amenable to formal, mathematical proof of correctness. Dijkstra promoted the formal verification of program correctness as one way to combat the reliability problems of software that had led to the sense of crisis (MacKenzie 2001, 37–61).

Both the chief programmer team approach and structured programming came to be seen under the larger umbrella of "software engineering," a term that began to pick up steam for its association with "systematic, disciplined, quantifiable" techniques. (Institute of Electrical and Electronics Engineers (IEEE), *IEEE Standard Glossary of Software Engineering Terminology: IEEE Standard 610.12-1990* (New York: IEEE, 1990), 67, quoted in Abbate, 2012, 99). 1968 saw the first NATO Conference on Software Engineering at Garmisch, Switzerland. Douglas Mclroy, a member of Bell Labs' Multics team at the NATO conference explicitly linked software engineering to mechanical engineering and the industrial revolution:

> We undoubtedly produce software by backward techniques. We undoubtedly get the short end of the stick in confrontations with hardware people because they are the industrialists and we are the crofters. Software production today appears in the scale of industrialization somewhere below the more backward construction industries. I think its proper place is considerably higher, and would like to investigate the prospects for mass-production techniques in software. (M.D. McIlroy, "Mass Produced Software Components," in Naur and Randell, pp. 138-150. Quoted in (Mahoney 2004, 11))

Much recent discussion of software engineering by historians sees in this industrial rhetoric a managerial drive to discipline and deskill unruly programmers. For example, Rebecca Slayton cites Department of Defense managers as having significant influence on driving the development and adoption of software engineering techniques. “While anticipating the rising costs of software, [DoD managers] also confronted shrinking budgets, perpetual scrutiny, and growing demands for flexibility, interoperability, and security. In this environment, defense department managers began to embrace a particular vision of software engineering, a means of quantifying software development and ensuring accountability.” (Slayton 2013a, 160) Under this influence, software engineering reemerged in the 1970s under a new community focused on the reduction of complexity through programming and design methodologies (Slayton 2013a, 162–4). Nathan Ensmenger, likewise, saw in much of the manufacturing rhetoric a call for Taylorist scientific management of programmers:

The solutions to the “software crisis” that [are] most frequently recommended… are not fundamentally different from the four principles of scientific management espoused by Frederick Taylor in an earlier era.” (Ensmenger and Aspray, 2002, 15-16)

Other historians, however, see software engineering as a rhetorical strategy used by programmers themselves to increase their status as a profession. “For the major organizations that were trying to define computing as a profession, embracing the term *software engineering* [emphasis in original] could signal a claim to professional turf rather than any commitment to engineering as such.” (Abbate 2012, 102) Abbate agrees with Ensmenger that this process of professionalization involved defining it as masculine, and relabeling programming as “software engineering” did precisely this. Nevertheless, Abbate notes that most of the proposed “engineering” solutions came from those who had deep knowledge of programming themselves, trying to improve programming practice from within. “But not everyone had aimed to deskill programmers in the first place. Software engineering advocates such as Dijkstra and Brooks identified as programmers themselves and had no desire to

downgrade their peers. Many programmers actively took up techniques such as structured programming as a way of easing their work and enhancing their own value in the job market. Rather than making programmers obsolete, software engineering methods became simply another skill that programmers could claim." (Abbate 2012, 108) Thomas Haigh goes further, making the case that the circle of people who directed the software engineering agenda at the NATO Conferences were not managers but the pillars of academic computer science, especially Edsgar Dijkstra. Dijkstra thought that the software crisis was really the fault of managers who thought programming was easy and hired hordes of programmers with low skill. Dijkstra sought to elevate programming as a formal, mathematical discipline that required intelligence (Haigh 2010).

Within this environment, programming languages emerged as one possible solution to the software crisis. Dijkstra's structured programming methodology promoted a number of techniques that could be applied, with self-discipline, to existing programming languages. Such practices included maintaining modularity of code. "Essentially, Dijkstra wanted programs to be made up of blocks of code, where each block did only one task." (Zepcevski 2012, 185) Nevertheless, soon afterwards, computer researchers began to devise programming languages that would enforce structured programming principles, by, for example, omitting the "Go To" statement. According to Thomas Haigh, the same circle of computer scientists who had participated in the NATO Conferences had previously been working on ALGOL, a language that many had hoped to promote good programming practice. This circle had dissented from the ultimate direction ALGOL had taken, but one among them, Niklaus Wirth, turned his passed-over ALGOL proposal into Pascal (Haigh 2010). Pascal was a language that would help produce programs amenable to formal proof (Zepcevski 2012, 261–2). The Department of Defense sponsored the development of Ada in order to standardize on a single language throughout the military (Slayton 2013a, 164–5; Zepcevski 2012, 262–3). Both Pascal and Ada were procedural languages developed to foster structured programming practices.

Object-oriented programming languages likewise enforced some of these same practices. Joline Zepcevski, in her recent dissertation, argues that object-oriented programming and structured programming arose in parallel. Modularity had long been practiced before either structured or object-oriented programming. "Segmentation, or creating modules of code, is one of the most frequently cited solutions to the problem of complexity. When working in teams, if programs are segmented or modularized, programmers can work on individual segments, with little interaction between segments. Modularity has long been perceived as a method of decreasing complexity and improving programming as an industrial activity. As early as the work being conducted on the SAGE project, techniques were being used to create modular code." (Zepcevski 2012, 199) Nevertheless, object-oriented programming made modularization of code mandatory for all programs. Objects are essentially modules of code that are completely separate from other objects. They can only exchange data with other objects by sending messages to them. Otherwise, the data contained in an individual object is hidden, black-boxed, or "encapsulated," in a way that prevents other code from accidentally or maliciously changing them, which enhances the reliability and security of programs. Zepcevski argues that object-oriented programming languages such as Smalltalk and C++ developed as a confluence of ideas from various sources; ideas pervasive in computer science became incorporated into these languages and eventually became part of the definition of object-orientation itself (Zepcevski 2012, 137–141). Alan Kay himself recounted that Smalltalk was influenced not only by Simula-67, the only previous object-oriented language, but also the functional language LISP, and ideas from Ivan Sutherland and Seymour Papert (Kay 1993; Zepcevski 2012, 237).

As a result, looking at such languages only as agglomerations of their components, structured programming advocates could claim that object-oriented programming was nothing new, just another language enforcing structured programming methodologies. "Donald Knuth, author of The Art of Computer Programming, has insisted that he has always thought of programming using the techniques that have been subsumed into the object oriented paradigm… 'but I

haven't used the languages that help enforce the discipline; I've always enforced the discipline myself in other languages.'" (Dan Doernberg, Interview with Donald Knuth for the Computer Literacy Bookshop, December 1993, http://tex.loria.fr/litte/knuth-interview, quoted in Zepcevski, 2012, p. 272) Zepcevski argues that, for older computer scientists as Niklaus Wirth and Donald Knuth, object-orientation did not add any new concepts but merely combined existing concepts and techniques into a single language that enforced those practices. However, other computer scientists such as Antony Hoare saw, correctly in Zepcevski's view, that object-orientation was in fact a paradigm shift in seeing programs not as processes but as collections of objects (Zepcevski 2012, 274–5).

Nevertheless, Zepcevski notes that Alan Kay created Smalltalk to directly address the issue of software complexity, which, along with the problem of program verification, was the key issue in the software crisis. This occurred in parallel with, and despite, advances made by structured programming.

> [The examples of the structured programming languages Pascal and Ada] illustrate that, while language designers enforced structured programming techniques in their languages, the problems of complexity and verification were still an important part of the discourse on programming theory… Constant discussion of the software crisis abounded… The object-oriented methodology was created in response to problems of complexity… Alan Kay often talks of a bet where he argued he could create "the most powerful programming language in the world" in a single page of code. The outcome of this bet was the creation of Smalltalk. The inference here is that other languages may have also been powerful, but the complexity of their long definitions was detrimental. (Zepcevski 2012, 263–5)

The enforcement of programmer discipline mechanically through language design was thus one solution to the software crisis that promised order of magnitude gains in programmer productivity. Fred Brooks notes at one point in *The Mythical Man-Month*, "Programming productivity may be increased as much as five times when a suitable high-level language is used." (F. P. Brooks 1995, 94) What Brooks meant, however, was that high-level languages could produce this gain when compared to programming in assembly languages, as it frees the programmer from

having to deal with the complexity of the low-level hardware, which is hidden by higher level abstractions. “To the extent that the high-level language embodies the constructs wanted in the abstract program and avoids all lower ones, it eliminates a whole level of complexity that was never inherent in the program at all.” (F. P. Brooks 1995, 186) This argument is commonly extended to object-oriented languages, which states that because object-oriented languages work at a higher level of abstraction than procedural languages, they will inevitably reduce complexity and improve productivity. This is often connected to a narrative of linear progress—the more abstract and higher-level a language is, the more advanced it is. Computer scientist Tony Hoare apparently thought of object-orientation in this way: “Hoare sees the change in programming methodologies as paralleling his own scientific progress moving from sequential code with hierarchical construction towards an object oriented worldview…” (Zepcevski 2012, 275)

By the 1980s, Brooks himself, however, was skeptical that object-oriented programming, or indeed anything, was truly a silver bullet that could solve the software crisis. In “No Silver Bullet—Essence and Accident in Software Engineering,” first published in 1987, he argued that software development was inherently hard because it had certain essential characteristics: complexity, conformity, changeability, and invisibility or unvisualizability (because it involves abstract concepts which are difficult to visualize). Following Aristotle, he categorized all other programming difficulties as “accidental,” or “those difficulties that today attend its production but that are not inherent.” (F. P. Brooks 1995, 182) Addressing both structured languages like Ada and object-oriented languages, he notes that, like all other high-level languages, they have produced incremental improvements through removing one or more of the accidental difficulties of programming, but that continuing advances in high-level languages will only produce diminishing returns, as the largest win had been gained from the initial move away from assembly in the first place (F. P. Brooks 1995, 188–9). He is also skeptical of attempts to realize graphical programming, of which NeXT’s Interface

Builder is an early example that came out after Brooks' article (F. P. Brooks 1995, 194).

Brad Cox disagreed. In a direct rejoinder to Brooks, Cox wrote an article in BYTE magazine in 1990 claiming that a silver bullet indeed existed, and further insisted that this solution could bring about a Software Industrial Revolution, echoing McIlroy's comment over twenty years prior (Cox 1990a). There were two components to this solution. The first was object-oriented programming, which required programs to be written in modular, even piece-meal fashion. Second was an economic arrangement that this technology would make possible: an open market of software objects that could be bought and sold off-the-shelf, and used by developers to build their own software without rewriting everything from scratch. Reusable objects would be like interchangeable parts, which made possible the first Industrial Revolution.

> I use a separate term—software industrial revolution—to mean what "object-oriented" has always meant to me: transforming programming from a solitary cut-to-fit craft into an organizational enterprise like manufacturing. This means letting consumers at every level of an organization solve their own software problems just as home owners solve plumbing problems: by assembling their own solutions from a robust commercial market in off-the-shelf subcomponents, which are in turn supplied by multiple lower level echelons of producers." (Cox 1990a, 212)

Indeed, Brooks had envisioned a similar economic arrangement where coding could be outsourced to the market, although he did not couple it with the technological solution of object-oriented programming:

> **Buy versus build.** The most radical possible solution for constructing software is not to construct it at all… While we software engineers have labored on production methodology, the personal computer revolution has created not one, but many, mass markets for software… Even software tools and environments can be bought off-the-shelf. I have elsewhere proposed a marketplace for individual modules.
>
> Any such product is cheaper to buy than to build afresh… The development of the mass market is, I believe, the most profound long-run trend in software engineering… The cost of software has always

> been development cost, not replication cost… (F. P. Brooks 1995, 197–199)

Why was object-oriented programming necessary to create this market? Cox argued such software components needed to be produced at a much higher level of abstraction than previously. Here Cox shifted from the metaphor of 18th century firearms production to modern computer hardware production. Computer hardware was made up of different levels of components, with each level made up of smaller components from the level just underneath it. Thus, a computer might be made up of several racks of printed circuit boards, or cards. Each card contained multiple integrated circuits, or chips. Each chip was designed modularly, as distinct combinations of logical "blocks." Each block was made up of even more fundamental building blocks, logic "gates" that represented AND, OR, and NOT. And these gates themselves were made up of individual transistors. A computer hardware manufacturer need not produce its own chips, but merely needed to buy the chips it needed from other companies. This, in fact, had made the personal computer revolution possible, when microprocessors from Intel, Motorola, and others had provided the conditions for people to make and sell computer boards using these chips. Assembly-level programming was like working at the gate level. Procedural programming using languages like C, while a dramatic improvement over assembly, was like working at the block or chip level. Working at these low levels still required the programmer to worry about hardware-level details, designing programs in terms of how the computer or operating system worked, rather than the abstract question of how best to represent a user's problem. What was necessary to increase programmer productivity was to increase the level of abstraction, and thus the level of code modularization and reuse, up to the "card" level or higher.

Alan Kay's object-oriented language Smalltalk had taken a huge step in this direction. For example, researchers using Smalltalk had created a graphical programming environment called "Fabrik" that seemed like it might bring programming to non-programmers (Cox 1990b, 29–30). However, Smalltalk was not widely used outside of the computer science research community. Most code in the industry was being written in the procedural language C, which Cox considered too

low-level. Cox created Objective-C to bridge this gap, to allow for "Programming Smalltalk 80 methods in a C Language." (Cox 1983) Objective-C would allow programmers to leverage the vast amount of existing C code, encapsulating it into higher level, Smalltalk-style objects. Chip-level code could thus be packaged into card-level objects that could then be sold to others. This would maximize code reuse, and help solve the software crisis: "By adding these, and probably other, architectural levels, each level can cater to the needs, skills, and interests of a particular constituency of the software-components market. The programmer shortage can be solved as the telephone-operator shortage was solved: by making every computer user a programmer." (Cox 1990b, 30)

Such talk of making everyone a programmer, and of making programming more like manufacturing, sounds like an argument for deskilling programming labor. Ensmenger has noted this, quoting Cox:

> When a prominent adherent of object-oriented programming [Cox] spoke of "transforming programming from a solitary cut-to-fit craft, like the cottage industries of colonial America, into an organizational enterprise like manufacturing is today," he was referring not so much to the adoption of a specific technology, but rather to the imposition of established and traditional forms of labor organization and workplace relationships. (Ensmenger and Aspray, 2002, 15-16)

Ensmenger takes this as evidence that Cox's vision was a managerial one, fitting into the Philip Kraft thesis that such advances as high-level languages and structured programming deskill and routinize workers for managerial benefit. Cox, however, was a computer scientist who created the Objective-C language and published in academic journals, despite starting his own company to pursue his vision. Like Dijkstra and Brooks, he identified as a programmer, not a manager, and sought to improve programming practice from within. Moreover, Cox insisted that building programs from off-the-shelf components, while easier than writing one from the ground up, was a skilled, build-to-order task like plumbing, not assembly-line manufacturing:

> [Software industrial revolution] means enabling software consumers, making it possible to solve your own specialized software problems

> the same way that homeowners solve plumbing problems: by assembling solutions from a robust market in off-the-shelf, reusable subcomponents, which are in turn supplied by multiple lower-level echelons of producers. (Cox 1990a, 3)
>
> But the principles of standardization and interchangeability pioneered for standard products apply directly to build- to-order industries like plumbing. They enabled the markets of today where all manner of specialized problems can be solved by binding standardized components into new and larger assemblies… The plumbing supply market lets plumbers solve only the complexities of a single level of the producer/consumer hierarchy without having to think about lower levels, for example, by reinventing pipes, faucets, thermostats, and water pumps from first principles. (Cox 1990b, 28–29)

The model for Cox is not Frederick Taylor, but Adam Smith. Software should be separated into layers of abstraction, with each layer corresponding to a division of labor governed by the market, not by managerial control. Cox's invocation of colonial gun manufacturing elsewhere in both these articles, reinforces this connection (Cox 1990b, 25–26; Cox 1990a, 5–7).

Connecting object-oriented programming to the Industrial Revolution allows Cox to marshal another historical argument to his side. Cox claims that the software industrial revolution is a Kuhnian paradigm shift. He reinterprets the software crisis within Kuhn's framework, seeing it as a scientific crisis produced through normal science:

> According to historian Thomas Kuhn, science… proceeds as a series of revolutionary upheavals. The discovery of unreconcilable shortcomings in an established paradigm produces a crisis that may lead to a revolution in which the established paradigm is overthrown and replaced.
>
> The software crisis is such a crisis, and the software industrial revolution is such a revolution. The familiar process-centric paradigm of software engineering, where progress is measured by advancement of the software-development process, entered the crisis stage 23 years ago when the term "software crisis" was first coined. (Cox 1990b, 25)

Having made this connection, Cox goes on to assert that his proclaimed software industrial revolution is as much a cultural, social, and economic paradigm shift as it is technological.

> The silver bullet is a cultural change rather than a technological change. It is a paradigm shift—a software industrial revolution based on reusable and interchangeable parts that will alter the software universe as surely as the industrial revolution changed manufacturing. (Cox 1990a, 2)
>
> The software industrial revolution involves a similar paradigm shift, with a similar assault on entrenched value systems, power structures, and sacred beliefs about the role of programmers in relation to consumers. It is also motivated by practical needs that an older paradigm has been unable to meet, resulting in a desperate feeling of crisis. (Cox 1990a, 5)

In citing changes in culture, value systems, and power structures, Cox is arguing that technological change cannot occur without corresponding changes to norms, institutions, and the identity of programmers. Kuhn has made Cox something of a social constructivist. Nevertheless, in Cox's worldview, he privileges the economic relationship between producers and consumers, and it is this relationship that object-oriented technology is disrupting.

> To get a grip on object-oriented means coming to the realization that it is an end, not a means—an objective rather than the technologies for achieving it. It means changing how we view software, shifting our emphasis to the objects we build rather than the processes we use to build them. It means using all available tools, from COBOL to Smalltalk and beyond, to make software as tangible—and as amenable to common-sense manipulation—as are the everyday objects in a department store. Object-oriented means abandoning the process-centric view of the software universe where the programmer-machine interaction is paramount in favor of a product-centered paradigm driven by the producer-consumer relationship. (Cox 1990a, 3)

Cox thus saw the term "object-oriented" as meaning much more than just a programming methodology, but a revolution in the social relations involved between the producers and consumers of software. In vogue with the neoliberalism of the 1980s, Cox felt that the old way of producing software, monolithically by a large bureaucratic organization, harkening back to the mainframe model of the 1960s, should be supplanted by a new model in which production was distributed to the market, and software consumers would be able to influence quality through their purchasing power. He seemed to be unique in this, however. Although later, some

NeXT programmers did try to sell collections of objects to other programmers, Cox's Objective-C language was taken up by NeXT largely divorced of the social philosophy behind it.

How did Objective-C come to be used by NeXT? In the 1980s and 1990s, object-oriented programming did come to be seen by programmers as a new paradigm, though for far more narrow technical reasons than Cox's. As discussed earlier, Joline Zepcevski argues that object-oriented programming did indeed involve a shift in worldview, from process-centered to object-centered. Such revolutionary language no doubt caught the attention of Steve Jobs after leaving Apple in 1985, already having presided over both the initial personal computer revolution and its graphical second wave. Always on the lookout for the next big thing, and having hired computer researchers such as Bertrand Serlet from Xerox PARC, Jobs directed his new company, NeXT, to make object-oriented programming central to its software technology. By bringing object-oriented software to consumers, and making it easier for them to program with graphical tools such as Interface Builder, NeXTSTEP would bring about an order of magnitude increase in programmer productivity, solve the software crisis, and usher in the next computing revolution, all courtesy of Steve Jobs.

Of course, Jobs' vision was not quite the same as Brad Cox's, although both imagined a future where everyday people might write their own programs. Both visions shared one thing in common: programmers' work building software was made easier because they could make use of reusable object modules that others had already written. A division of labor was implied: less skilled programmers, or even users, would use pre-built objects and arrange them together like building blocks, while more skilled programmers would write the code for these blocks. Cox felt that this division of labor was most naturally expressed in the market: coders would sell objects to the consumers who would use them to build their own custom software. Jobs, however, was not interested in selling software, at least not initially. Jobs wanted to sell a complete, vertically integrated computer system. NeXT was forced to become a software company only after its hardware business failed. Even

afterwards, NeXT would not sell individual objects or classes of objects, but its entire operating system and development environment together, later decoupling the development environment from the OS and selling it standalone, for competing Unix platforms such as Sun and HP. The point, however, was to sell an integrated system, whereby the objectware came for free, either with the hardware, or the operating system. Rather than an open market of objects, Jobs envisioned only one supplier: NeXT itself. As Jobs felt he had attracted the best and brightest computer researchers and engineers to work for him, he undoubtedly felt that no other vendor would be able to supply better objects.

This is borne out by the fact that NeXT licensed only the Objective-C language from Cox's company, Stepstone, but not its object libraries. Cox saw Objective-C as a means to an end: allowing him to sell libraries of objects to other developers. Drawing on Cox's computer hardware metaphor, Stepstone called its libraries "ICPaks," with the "IC" standing for "integrated circuit." According to Steve Naroff, the Stepstone employee who enhanced Objective-C to allow NeXT to create Interface Builder, and later went to NeXT, Jobs felt Stepstone's ICPaks were not very good. Steve Jobs personally took part in hiring, putting new engineers through a grueling process, and only hiring those whom he considered the best. Jobs thus trusted his staff to write much better object libraries than Stepstone could, and using Objective-C, they wrote their own low-level object library, Foundation, and a graphical user interface object library, AppKit, that was tailored to work with the graphical programming tool, Interface Builder.

> So, as far as the ICPaks go, I knew that the productivity of the library was huge. I also knew that the people working on it at Stepstone were not world-class; I knew that it would be really hard for world class companies like NeXT to buy into someone else's ICPaks; I also knew that, without a great IDE [Integrated Development Environment], the ICPak leaves you sort of empty. So, Stepstone did not have that person with the holistic vision of what it is they should do…
>
> One of the things… Steve [Jobs] is… good at, is, he knows it when he sees it… when he sees something good…

> He was very crafty. He really knew how to save a buck. He could've very easily bought Stepstone at the blink of an eye. And he said, "you know, I don't believe in 80 percent of what they're doing—yeah, I believe in ICPaks, but I don't believe in their ICPaks. And that's where our competitive edge is, so I like their language, I like this guy who seems to have a clue, let's get the language, let's get the guy with the clue and marry him with open source and see what happens. (Steve Naroff, Interview, December 22, 2012)

Despite NeXT's trouble finding a market for its hardware or its software, a few users-turned developers bought into Jobs' vision. In the experience of these developers, NeXT's object-oriented development environment really did deliver on its promise to make them five to ten times more productive, if coupled with a change in the way they approached programming.

Developers take up NeXTSTEP

NeXT may not have been successful selling its computers or software to the mass market, but by seeding universities with its systems, it managed to expose a number of computer science students and computer enthusiasts to its technology. These users began to play with the NeXT's development tools that came with its computers, and began to write their own programs. Many of these advanced users became fanatically devoted to the NeXT platform and would try to find ways to become NeXT developers. Those that did would form small, independent software companies of a handful of developers, who would go into business developing exclusively for NeXTSTEP, the platform that they enjoyed so much. These independents would make up the core of the NeXT developer community as the market contracted. "It was a really close-knit community and… there's a tiny group of engineers that had done the NeXTSTEP and there's a tiny group of engineers outside of [NeXT] who is [the] Indie community who had contributed to [the platform]." (Wil Shipley Interview, April 18, 2012)

Experienced developers interested in the latest developments in object-oriented programming were attracted to NeXT as well. One such developer was Bruce Webster, who wrote a *Byte Magazine* article in 1996 arguing that the "real

software crisis" was that "individual and team productivity to be the leading predictor in estimating software costs; it's twice as significant as product complexity" and that this productivity could only be achieved with virtuoso programmers, who "like excellent musicians and artists, are born, not made." Like Brooks, he was skeptical of a silver bullet, but he did acknowledge that "better tools, better on-the-job training, better methodologies" were essential "because they raise the quality of most developers." (Webster 1996, 218) This was because by 1996, Webster had become an expert in object-oriented design. If used correctly, he believed, it could improve productivity, but only if developers followed certain principles and practices and thought about object-oriented design completely differently from procedural programming. Such issues he had discussed in his book, *Pitfalls of Object-Oriented Development* (Webster 1995).

Webster had acquired this expertise as a NeXT user and developer, writing *The NeXT Book*, a description and catalog of NeXT hardware and software, in 1989. (Webster 1989) In the early 1990s, while Chief Technical Officer of a company called Pages Software, Webster was a contributer to *NeXTWORLD Magazine* and the president of the NeXT developer's group, the Association of NeXTSTEP Developers International (ANDI). (Webster 1992) In a 1992 NeXTWORLD editorial, Webster voiced a concern many NeXT developers were facing. There was widespread agreement that NeXT provided the best development environment, but the platform's small marketshare and proprietary nature, and the uniqueness of the programming language and tools it used, made it difficult for developers to see a reason to write for it.

> "Sitting somewhere on the sidelines is NeXT, offering one of the best development environments around—but on proprietary hardware running proprietary software and using development tools not readily available elsewhere…
>
> The irony is that one of NeXTSTEP's biggest advantages—use of Objective-C and supporting tools—also serves as a major roadblock. The rest of the world has gone chasing after C++ and has only slowly discovered its limitations for object-oriented design, particularly with regards to dynamic binding [a feature of Objective-C and Smalltalk

> that allows for more flexible design]. On the other hand, developers using Objective-C on the NeXT run into problems when they want to move to other environments. These barriers, real and perceived, have caused many developers to avoid NeXTSTEP development in the first place. (Webster 1992)

Webster, noting that the software world was moving to a heterogeneous one, in which developers needed to deploy "cross-platform" to multiple platforms, advocated that NeXT port some of its development tools, such as the Objective-C compiler, as well as a subset of its object libraries (the basic Foundation classes, but not the AppKit user interface classes) to other platforms, such as Windows NT. Despite NeXT's fears that this would cannibalize its own platform, Webster argued that this would attract and retain NeXTSTEP developers and users, as well as showcase the platform's advantages. Of this latter point, Webster noted:

> Even with these tools available, NeXTSTEP would still retain its key advantages for shrinkwrapped and mission-critical-application development. Thus, as developers and customers adopt or experiment with NeXTSTEP because of the cross-platform potential, it will quickly become the preferred development and deployment environment, subverting from within. (Webster 1992)

Whatever the disagreements with NeXT's marketing or business strategies, there was no disagreement about NeXT's advantages for developing software. Unfortunately, the business realities of NeXT's tiny market, and its proprietary strategy, was keeping these advantages hidden from the majority of developers. Those few who did develop for NeXT felt that it represented the way all development should be done in the future. That the best technology does not always win is no surprise to historians of technology, who have shown how technologies do not emerge in a vacuum, but are connected in complex socio-technical systems that affect the choices of which technological direction is taken (T. P. Hughes 1987). Ruth Schwartz Cowan has shown, for example, how quiet, gas absorption refrigerators might have been "better" than electric ones from a consumer's point of view, but lost to electric ones because the development of the latter was encouraged and financed by the powerful electric utilities (Cowan 1985).

Indeed, while large corporations were exiting the NeXT market, small, independent software developers were becoming NeXTSTEP fanatics. In an article titled "Getting Religion," Simson Garfinkel, frequent NeXTWORLD contributor and co-author of *NeXTSTEP Programming: Step One, Object-Oriented Applications*, which became the canonical NeXTSTEP programming book of the 1990s, writes:

> These days, there are two kinds of companies developing third-party programs for NeXTstep: small, energetic companies, and the big guys.
>
> The small companies are usually hungry start-ups.
>
> They're totally committed to the platform. They've gone bananas over Interface Builder and Objective-C. They worship the Application Kit and recite NeXT's technical documentation in their sleep. They're the lean-and-mean companies like Adamation, Stone Design, and Lighthouse Design, and they're bringing out new applications five to ten times faster than they would be able to on any other platform.

What differentiated these developers from most of the big corporate players is that they completely embraced the NeXTSTEP platform; instead of writing ports using cross-platform languages and libraries, which resulted in applications that looked no different on a NeXT than on a Mac or Windows PC, these small independents took full advantage of Objective-C, Interface Builder, and the AppKit object libraries to produce applications many times faster than the large firms. According to Garfinkel, corporate firms trying to cut corners were merely making their lives more difficult, and producing applications that were not as good.

> *Developer Tip #1:* Don't reinvent the wheel. Rather than trying to emulate the AppKit and make your program look like it was written using Interface Builder, take the time to learn Interface Builder and do it right. Send two people from your company to NeXT's Developer Camp… In the end you'll get a faster, easier-to-use, and simply better program.
>
> *Developer Tip #2:* Follow the interface guidelines. If you're porting an existing program to NeXTstep, keep the program's back end… but throw away the user interface and start over. NeXTstep makes user interfaces so easy that, even if it took you three years to develop your X Window [a UI library for Unix systems] interface, you'll be able to develop a better NeXTstep one from scratch in a few months…

> *Developer Tip #3:* If people can run the same program on their NeXT as they can on their Mac or PC, what's the point in buying a NeXT in the first place?
>
> *Developer Tip #4:* Dare to be different. Embrace NeXTstep—Use it all. A lot of programmers moving to NeXTstep are hesistant to use things like… [the] rich set of functions in the NeXTstep library. After all, a program that uses these functions is harder to port to another platform. NeXTstep works together as a reliable, integrated whole. Use it all and you'll bring your program to market that much faster.
>
> *Developer Tip #5:* Use Objective-C. Many NeXTstep programmers are afraid to use Objective-C for anything but the user interface. Some want to use C++ for their back ends; others are scared of object-oriented languages in general and want to use ANSI C. Don't be afraid. Objective-C is one of the reasons that NeXTstep's Application Kit is so good. Object-oriented programming is easy, once you get the hang of it, and, as an object-oriented language, Objective-C leaves C++ in the dust.
>
> C++ may be the "industry-standard object-oriented language" these days, but remember: Microsoft Windows is quickly becoming the industry "standard window system." If you want to use standards, get a PC.
>
> If you still need convincing, just look at Lotus. Rather than bringing out 1-2-3 for the NeXT, they went back to the drawing board and created Improv.
>
> 'Nuf said. (Garfinkel 1992a)

While Webster argued that NeXT port some of its development environment to other platforms to help committed NeXTSTEP developers deploy cross-platform, Garfinkel was addressing the other side of the equation, exhorting corporate developers, who had no love for the platform, to take the time to properly learn NeXTSTEP and write "native" applications for it, in other words, use the libraries (AppKit), developer tools (Interface Builder), and language (Objective-C) that were "native" to the platform, instead of "reinventing the wheel" with foreign libraries and languages. Not only would this kind of shortcut not realize the productivity advantages of developing in NeXTSTEP, but the result would be inferior; because all of NeXTSTEP "works together as a reliable, integrated whole," using foreign tools and code libraries would stick out like a sore thumb, creating mismatches in the

interface visible to the user, but also causing mismatches with the underlying system, which was a recipe for bugs.

In contrast, the small, "lean-and-mean" independent companies Garfinkel praised totally "got" the NeXT "religion." They "worshipped" the AppKit and went "bananas" over Interface Builder and Objective-C. For these developers, the NeXTSTEP development environment was the best programming environment they had ever experienced, and they believed that this was the future of programming. Despite NeXT's tiny marketshare, and maybe because of it, these developers felt like a special, chosen few. Precisely because the rest of the developer world seemed to be choosing C++, which they considered an inferior object-oriented language, they saw it as their mission to spread the gospel of NeXTSTEP and Objective-C. If only NeXT's own short-sighted business strategies had not gotten in the way, everyone might have seen the light. But as long as the platform was kept alive, there was still hope that some day, this evangelical vision would come pass.

In the midst of Microsoft's dominance over platforms, and C++, and later, Java's dominance in programming languages, only the die-hard fans of NeXT's development environment remained on the platform. The productivity advantages and consistency of the system made NeXTSTEP less frustrating and more pleasurable an environment to program in, and it was difficult to go back to Windows or C++ once having tasted the nectar of AppKit and Objective-C. But, as Webster noted, the economic reality was that making NeXTSTEP-only development, especially of shrink-wrapped applications, increasingly difficult.

> Imagine being in this industry for 24 years and for 12 of it, people are just like, you know your technology is on 50,000 [computers] worldwide. Who do you think you're fooling, do you think you're going to be Microsoft with your 50,000 computers? …It was hilarious, for the first 12 years of my career, everyone was like, 'what are you doing, why don't you work for Microsoft? I don't understand, you have no chance.' (Wil Shipley Interview, April 18, 2012)

Fortunately for NeXT and its developer community, NeXT's development advantages did make it attractive to certain niche markets for which speed of

development of custom solutions took precedence over platform compatibility or cost. NeXT's turn in 1992 to the "enterprise" market and to "mission critical custom applications" development, and its shift from a hardware company to a software and services company allowed it to survive, albeit a pale shadow of its former, world changing ambitions. Most of the customers that would pay any cost for rapid custom application development were Wall Street financial firms, and to a lesser extent, the federal government, in particular, the intelligence and security apparatus.

NeXT and Wall Street

As described by Randall Stross, NeXT in 1993 was a company struggling to survive. It had burned through millions of investor dollars, built a state of the art factory that ran idle, and eventually closed down its hardware business. NeXT's computers had been too expensive compared to Macintosh and IBM-compatible PCs, and its hardware was too slow compared to workstations from Sun Microsystems. NeXT computers had an identity crisis—they were trying to be personal computers, while competing in the higher end market for workstations. NeXT also had significant trouble deciding what its core market should be. Initially it targeted higher education, but this was not a large enough mass market, and the average college student simply could not afford a NeXT. NeXT then tried to market a vision of "interpersonal computing," a term that connoted networking PCs to create a collaborative working environment, but it was difficult to convey to potential customers NeXT's advantages compared to other PCs.[18] It appeared that NeXT would try any market it could sell to, without any coherent marketing strategy. Its

---

[18] In an interview with NeXTWORLD, Steve Jobs remarked, "You could call it groupware. You could call it interpersonal computing. You could call it collaborative computing… The problem is that people are not running around thinking they have a problem. In other words, I don't run into customers who are pulling their hair out saying my collaborative computing environment isn't good enough…" (Ruby and Jobs 1992)

first non-academic customer installations were eclectic, including the William Morris Agency, a Hollywood talent agency, Alain Pinel, a Bay Area realty, and the L.A. County Sheriff's department. (Karon 1992a; Stross 1993) "The interest of the William Morris Agency, Jobs told his staff, was validation of his laissez-faire marketing strategy, firing buckshot indiscriminately at the entire business world and seeing what customers stepped forward." (Stross 1993, 223) Jobs's attitude to marketing was that if he simply built the hottest, most state of the art personal computer, customers would automatically come to NeXT. "'If we build it, they will come.' This is, succinctly stated, the Steve Jobs Philosophy of Marketing." (Barlow 1992) This turned out to be a failed strategy for creating a new platform competing against established players in the market. A 1992 survey of the NeXT market by NeXTWORLD Magazine attempted to catalog who NeXT's customers were, and the results seemed to be all over the map. NeXT computers were being sold to the health care industry, the Department of Defense and the nation's intelligence apparatus,[19] desktop publishing, and office automation (DiNucci 1992; Karon 1992b; *NeXTWORLD* 1992a; *NeXTWORLD* 1992b).

Nevertheless, by 1992, a new picture was emerging, and NeXT's shift in strategy to pursue the "mission-critical custom applications" market by emphasizing the advantages of its object-oriented development environment was catching on in a few key markets where price was not a primary concern. National intelligence, for

---

[19] "Like shadows… government agencies have quietly become some of the largest NeXT installations, with up to 300 workstations. But for security reasons they hesitate to publicize their use of the platform.

Nevertheless it is known that NeXT has found a ready customer in the Department of Defense and its related intelligence agencies. A thriving industry of NeXT systems integrators and consultants has grown up in the suburbs around Washington D.C. to support this growth market.

…And several NeXT software suppliers have reported sales to 'secret customers.'" (Silverstone 1992)

one, found this to be useful. "'Intelligence and defense have a strong need for sophisticated software solutions that they cannot buy off the shelf,' says Gary Fuller, president of GWF Sierra Concepts, a Sonoma, California, consulting firm working with NeXT in the intelligence community. 'The NeXT facilitates the development of custom software on a quicker turnaround, resulting in higher productivity.'" (Silverstone 1992) However, a much larger market, and one that could be openly publicized, could be found on Wall Street. For financial companies, hardware and software costs were miniscule compared to the potential profits that technological advantages over competitors could bring.

> In the world of commodities trading, a few seconds can mean the difference between profitability and bankruptcy. For traders and Philbro Energy, the world's largest crude-oil trading company… to better compete in the 24-hour-a-day commodities trading arena of the 1990s, they've come up with a solution as black[20] as the product they sell… The solution they came up with—NeXTstations running custom in-house applications—has even allowed the energy giant to form a second company to resell the software to other players in the world of commodities trading. (Borsook 1992)

A similar logic was driving the use of NeXT at securities firms Lehman Brothers, UBS, and O'Connor and Associates, particularly in the emerging equity derivatives market.

> The margin for error on trades that typically reach hundreds of millions of dollars is nil. With many trades, decisions have to be made in seconds. And since competitors are watching similar screens, 'if they decide before you,' says [Hadar] Pedhazur [UBS vice president of Equities Technology], 'you lose.'
>
> …For UBS, it wasn't a question of choosing NeXT over another machine…
>
> NeXT's proven speedy development cycle was the main reason they felt the job couldn't be done on another workstation. 'You get a machine whose heart and soul is object oriented,' says Pedhazur. 'The

---

[20] NeXT computers were painted jet black and affectionately known as "black hardware."

> tools that come out of other companies don't stand up.' Companies that choose another hardware environment, says Pedhazur, are having to start from the ground up. 'Every single one is building their tools in-house,' he says. '*That's* a risk that isn't worth it to me.' [emphasis in original]
>
> …custom applications are driving the use of NeXT at UBS Securities…" (Littman 1991, 38)

For such firms, the technological risk of using a non-standard, niche platform with an uncertain future was vastly outweighed by the financial risk of losing out to competitors.

> "It's a major political battle to buy NeXT," says Hadar Pedhazur… "People are asking, 'What if NeXT goes out of business?'
>
> …We don't believe NeXT is going out of business, but it's a risk we're willing to underwrite for the benefit of using NeXT computers… We're in a business where responding in days or weeks [with a new custom derivatives trading program] instead of months is the difference between profit and loss," says Pedhazur.
>
> Besides, says Pedhazur, the machines will earn their keep quickly. "If NeXT allows the bank to play in a new financial arena, we might make all the money to justify the risk within a one- or two-month period." (Littman 1991, 38)

For Wall Street securities firms, NeXT's object-oriented productivity advantage for rapid development of custom software was not just a marketing slogan, or the blind faith of a Steve Jobs acolyte. It was real enough to put millions of dollars on the line, as the industry's newest instrument, derivatives, demanded the best technology available in order to maximize its profit potential.

Indeed, Wall Street would keep the NeXT development community alive, by providing developers with lucrative contracts to develop custom software. By the mid-1990s, packaged, "shrink-wrapped" software was a dying market for NeXT developers. In an article titled "Who Needs Shrink Wrap," NeXTWORLD editor Dan Ruby comments:

> Once in the vanguard of the crusade for the personal workstation, commercial NEXTSTEP [sic] developers now seem like an afterthought in a market focused on specialized custom applications.

> On Wall Street, it turns out, there isn't much call for great publishing software. Even when custom apps are deployed, it's not a sure thing that users will want general-purpose productivity applications… It's a pretty dreary picture for today's suffering NEXTSTEP developer. (Ruby 1993a)

Such companies such as AppSoft and Glenn Reid's Rightbrain had gone out of business (Ruby 1993b). Instead, contracting for custom in-house software made up the bulk of the available work. OmniGroup, now an independent Cocoa applications company in Seattle, got its start doing contracting development for Lighthouse Design, helping it with its desktop applications. For most of the 1990s, however, it maintained its income by contracting for firms like McCaw Cellular, (which later became AT&T Wireless) writing enterprise software. Wall Street banks, driven by the profit potential of the derivatives market, were exploring all manner of bleeding edge software technologies, including object-oriented technologies. "Banks have advanced technology groups that are into all kinds of things. You can find pockets of Lispers and Smalltalkers on Wall Street, too. Back before the Apple/NeXT merger, financial companies were where the bulk of the work was." (John C. Randolph, Personal communication with the author, instant message conversation, August 3, 2011) Indie Cocoa developer Bill Moorhead of Black Pixel was a Smalltalk developer in the 1990s, and had worked for a bank. "It may not be known, but banks [had] a lot of Smalltalk… For some reason it was part of the culture." (Interview with Daniel Pasco, Bill Moorhead, Chris Clark, George Dick, June 12, 2009)

Similarly, Aaron Hillegass, founder of Big Nerd Ranch in 2001 and had been a NeXT and later Apple employee in the 1990s, had one of his first jobs at a company selling technology based on the object-oriented language Eiffel. One of its customers was a Wall Street bank, and Hillegass, who was deep in debt, eventually took a contracting job with the bank in order to repay his debt. Although hired for his Eiffel experience, at the bank, he quickly shifted to writing Objective-C code for NeXT machines. The NeXT-based system he worked on was being used for mortgage-backed securities. (Aaron Hillegass Interviews, June 6, 2011, and July 7, 2011) John C. Randolph, a Cocoa Evangelist at Apple in the early 2000s, had also

contracted for Wall Street in the 1990s. This shift to enterprise consulting was so successful that the market eventually perceived a shortage of experienced NeXT programmers. "There ought to be a flyer in the NEXTSTEP [sic] Developer box that reads like this: 'Some Assembly Required. Skilled NEXTSTEP contractors not available for purchase, but may be rented for upwards of $100/hour—if you can find them.' The ever-increasing price of NEXTSTEP programmers is a symptom of a real problem: demand is outstripping supply." (Lavin 1993)

Steve Jobs had poured a considerable amount of his own personal fortune into keeping NeXT alive when other startup founders might have decided to fold and start over. Randall Stross's account of NeXT's early years (Stross 1993) suggests that Jobs had founded NeXT to redeem himself from his fall from Apple, and with his ego bound to NeXT's fate, he would do anything to avert its failure. This might suggest a reason why a self-proclaimed counterculturalist like Jobs might make the Faustian bargain of working with Wall Street.[21] Jobs had already become bedfellows with rich tycoons to finance NeXT. H. Ross Perot had been NeXT's largest investor, and Jobs, who later hosted the Clintons at his home, even endorsed Perot's presidential candidacy (Ruby and Jobs 1992). For some countercultural Jobs admirers, however, cozying up to Wall Street felt like a betrayal of the anti-corporate stance of his youth. John Perry Barlow, Grateful Dead lyricist and Electronic Frontier Foundation founder, wrote an article in NeXTWORLD worrying that NeXT might lose its soul.

> Well, now it appears [NeXT] knew how to sell itself after all. The Mystery Market is revealed at last and turns out to be none other than

---

[21] Another explanation is possible, however. Jobs's later deals with the music industry, Disney, and even Microsoft shows that he was not above working with large corporations if it suited him. He may have been an opportunist who would work with anyone if it furthered his own interests, and his binary tendency to label others "heroes" or "shitheads" could turn on a dime, differing from one day to the next. (*The Economist* 2007)

> MIS [Management Information Systems]! Corporations are suddenly using NeXTs to write custom applications they once wrote on mainframes and PCs… I have to admit that if my sole objective were to move black boxes [NeXT computers], this is probably the best way to do it. My optimism is restored. NeXT is going to make it. Nevertheless, I have several remaining concerns…
>
> I suppose, though, my greatest concern is for the soul of NeXT. It is invariably true, as Mitch Kapor learned at Lotus, that companies come to resemble their markets more than their makers.
>
> I can think of no exceptions to this rule. Consider the cultural transformation of Apple after it pegged its sights on corporate sales. Or the staid flavor of DEC. Or the tight resemblance between hackers and UNIX-weenies inside or outside Sun. Given that this rule exists, one should choose his markets with a measure of personal aspiration.
>
> I was already somewhat concerned on this account over large NeXT purchases by the Royal Canadian Mounted Police and the Los Angeles County Sheriff's Department. And I would have been alarmed to learn the CIA has sent over 40 employees to NeXT Developer Camp if I hadn't figured the cultural resemblance between The Company and NeXT to be striking enough already.
>
> But MIS?! Steve may find his company going down IBM's path in more ways than one. After so many years of corporate rejection, he may have acquired sufficient VP-ness Envy to welcome this prospect—but it's a grim realization for an unreconstructed hippie like myself.
>
> (Barlow 1992)

Barlow's jeremiad seemed to be one of the few that actively worried about this. Indeed, NeXTWORLD published enthusiastic articles celebrating the use of NeXT by any customer, whether it be law enforcement, the CIA, or Wall Street. NeXT's users believed that NeXT was the future of computing, and its disappointing sales were like the end of the world not arriving for an apocalyptic cult. "The Wave of the Future, the object-oriented operating system that both Microsoft and Big Pink [the Apple/IBM joint venture, Taligent] are millennia away from delivering, has been available for some time, already debugged and running like God's wristwatch—and it has not sold. Indeed, the world's most elegant computer, just about the greatest thing since group-sex, has only shipped 36,000 units since 1988." (Barlow 1992) In such a dismal environment, NeXT users might celebrate the use of NeXT computers

by anyone as justification that their preferred platform truly was the best. Wall Street and the CIA needed the best technology, and they gave NeXT their endorsement.

Moreover, *NeXTWORLD*, and possibly the NeXT community itself, seemed to be increasingly technolibertarian in ideology, which could support the idea of corporate use of NeXT. Jobs had already cozied up to Ross Perot. *NeXTWORLD Magazine* itself seemed to be a trial run for what would eventually become *Wired Magazine*. Its first few issues, though nominally about the NeXT platform, was actually advancing a technologically utopian view of society, featuring an editorial by Nicholas Negroponte, and an interview with futurists Alvin and Heidi Toffler. NeXT computers, which were supposed to be the most advanced, were a synecdoche for this vision of the future. Indeed, in this light, the presence of John Perry Barlow, who was a frequent NeXTWORLD contributor, is not that surprising. According to Paulina Borsook, Barlow himself represented one wing, the countercultural one, of the technolibertarianism that became fully articulated later by *Wired* (Borsook 2000). Borsook had become acquainted to this ideology from the inside; she herself had been a contributor not only to *Wired,* but also to *NeXTWORLD* before it, penning the article, "Striking it Rich: Oil trader leverages the future on NeXT." (Borsook 1992)

Nevertheless, NeXT users and developers who had experienced the NeXT did feel that it was the "world's most elegant computer" and "the Wave of the Future" as Barlow put it. However, the rest of the world did not seem to understand what they were missing. In this situation, NeXT and developers would take whatever validation they could, even if it came from Wall Street. The platform was in survival mode. However, ultimately, they were not satisfied with survival. They wanted to see NeXT and its superior technology make its way to the wider world, and empower individual consumers, not just Wall Street bankers and CIA spies. NeXT developers felt a sense of mission to not only keep the platform alive, but to ultimately evangelize it to the rest of the programmer world.

> I think a lot of us were, like, we need to keep NeXT alive until the world notices it. (Wil Shipley Interview, April, 18, 2012)

Third party developers such as Shipley, felt that NeXTSTEP represented the way all programming ought to be done in the future. Technological progress, however, was being inhibited by the monopoly power of Microsoft. But for true believers such as Wil Shipley, the time for NeXT's technology would come, if only the developer community could keep it alive.

NeXT's employees felt a similar sense of mission. Julie Zelinski, a former software engineer at NeXT, expressed the feeling that NeXT was special and exceptional, ignored and unappreciated by the mainstream of the computer industry. This was driven by their belief that they were creating the best the technology in itself, but also by the sense of purpose instilled in them by Steve Jobs' charismatic leadership.

> NeXT was really just toiling away in obscurity, I mean we were crusaders, I loved what I did… back in '92, '93, '94 we were selling like 1000 machines a year… no one cares, we get all this great press that says it's beautiful, it's awesome, it's well-designed, [but] no one buys it, right? And the thing is, it didn't bother us. We just kept doing it. We just kept going to work and we're excited and we'd talk ourselves into, like, well it doesn't matter whether we're getting appreciated by the world, we're doing what's right, our technology makes sense and it's worthwhile… We were oddballs. We weren't a mainstream company, we didn't have apps, you know. …It's unrequited. We work really hard to build these awesome things but they're not getting traction. […]
>
> I really believed in what we did… Steve [Jobs] had a way of… making you feel like you were doing something… important… worthwhile… You actually felt like it was kind of noble… I mean the technology was really great, but… Steve… he just really just infused that company with a sense of purpose. (Julie Zelinski Interview, April 24, 2012)

To NeXT employees, developers, and users, NeXTSTEP was another superior technology that, like Apple before it, had lost the marketshare battle to Microsoft. There was a feeling that NeXT technology, which supporters had once hoped would lead the way into the future for computing, might be lost to history if the company went out of business. Its use by Wall Street and a few other specialized markets vindicated their sense that NeXT technology was superior, and this use kept NeXT

alive, but only as a shadow of its former self, its world-changing ambitions reduced to scraping by as a forgotten, marginal player in the industry. NeXT developers like the independent startup OmniGroup, founded by Ken Case, Wil Shipley, and Tim Wood, kept the faith, believing that NeXT's advantages were self-evident, and hoping that someday, someone would notice, and give it its time in the sun. OmniGroup had coalesced as a group of friends who wanted to write NeXT software for fun and make a living out of it. While Tim Wood kept the company afloat working on the McCaw Cellular contract, Ken Case and Wil Shipley often worked on end-user oriented applications like OmniWeb (a web browser) that were often released for free. (Ken Case Interview, February 10, 2012) These were more fulfilling projects but were not a significant source of the company's revenue; somebody like Wood still had to take on the boring business contracts to keep them fed. Developers like Shipley longed for a day when they could make a sustainable living producing consumer software. But having experienced the advantages of NeXT's object-oriented development environment, they also refused to write software for any other platform. If NeXTSTEP would never reach the masses, they would have to live with being contractors in the enterprise market. Few would anticipate that NeXT technology would find its way to a consumer market sooner than they had anticipated. This would come from a most unlikely savior—the company that had scorned Steve Jobs before, Apple. Jobs had started NeXT to replace Apple in the marketplace. Now NeXT technology, its people, and its culture would remake Apple from within.

The Vindication of the NeXT community

When the news arrived that Apple was in talks to acquire NeXT, NeXT users and developers, having experienced a ghettoization of their platform, were extremely excited. Their stubbornness had paid off. In late 1996, then Apple CEO Gil Amelio and VP of software, Ellen Hancock, had concluded that Apple's project to modernize its venerable Macintosh operating system in-house had gone off the rails, and the company needed to acquire one externally to replace it. Initial talks had been conducted to acquire BeOS, created by Be, the company founded by Jean-Louis

Gassée, another former Apple executive. However, Gassée, sensing desperation, overplayed his hand, and soon Amelio was in talks with Steve Jobs. Apple's $429 million acquisition of NeXT in December of 1996 changed the fortunes of both NeXT and Apple. By 1997 Amelio and Hancock were out, Jobs had installed his lieutenants in positions of power in the company and assumed control as interim CEO (Deutschman 2000; Isaacson 2011).

With former NeXT executives fully in charge at Apple, work on replacing the classic Macintosh operating system with one based on NeXTSTEP could begin. In early 1997, Apple announced that its future NeXTSTEP-based Mac operating system, code-named "Rhapsody," would require developers to rewrite their applications using NeXTSTEP's Objective-C based AppKit and Foundation frameworks. This native Rhapsody development environment was called the "Yellow Box," to distinguish it from the "Blue Box," a compatibility environment that would run unmodified existing Macintosh applications in a virtual machine. This created a two-tier system in which existing Mac apps were now second class citizens, and sent a message to third party developers that if they wanted to write native apps, their existing C and C++ code bases that had been built up over a decade would have to be thrown out; moreover, they would all be forced to learn Objective-C, which was an obscure language that no other company in the industry used. Third party Mac developers revolted, including industry giants Microsoft and Adobe. Apple could not afford the loss of these two third party developers: Microsoft Office's dominance meant that any viable home or business personal computer had to be able to run it or risk irrelevance; most of Apple's profits in the late 1990s came from high-end Macs sold to content creators and publishers who used Adobe Photoshop or applications that Adobe had acquired, like Pagemaker or Dreamweaver. To lose the support of either Microsoft or Adobe at a time when Apple was at risk of going out of business would have been fatal. In response to this misstep, Apple came up with a compromise. In 1998, Apple announced Carbon, an updated, modernized but still procedural-C based version of the original Macintosh Toolbox APIs that would be fully native on the new NeXTSTEP-based operating system, which would itself

eventually be renamed “Mac OS X,” with the “X” being the roman numeral for ten, to differentiate it from the classic Mac OS, which had version numbers with Arabic numerals. The Objective-C based “Yellow Box,” composed of the AppKit and Foundation, was renamed “Cocoa” under this new scheme, and would co-exist in parallel with Carbon on Mac OS X. Development of Mac OS X would take close to five years, with its first full release in March of 2001. Until then, frequent updates of the classic Mac OS, versions 8.0 through 9.1, were released as stopgaps.

The announcement of NeXT’s acquisition by Apple had immediately reenergized the long-suffering NeXT developer community. Says Wil Shipley, “The day Apple bought NeXT, it was our Christmas… We were like, ‘Oh, my God, this is it.’ Like, we finally have our shot.” (Wil Shipley Interview, April 18, 2012) Another former OmniGroup employee recalled, “The sun [was] getting dimmer and dimmer and dimmer, and people are starting to figure out, are we going to have to become cannibals or what? And then all of a sudden it’s light again, it’s amazing! We had, like, customers with an actual opportunity for the technology to survive.” (Luke Adamson, Interview, February 22, 2012)

The merger of NeXT and Apple was the first major environmental shift for NeXT-turned Cocoa developers. NeXT developers had not had a viable market for end-user desktop applications since NeXT had closed its hardware business, and even before then, that market had never approached the numbers of the Apple Macintosh. Apple provided an already existing base of consumers who were staunchly loyal to its platform, and were willing to spend money on quality software. Moreover, the economics of independent software development were changing. In the days of physical floppy disk or CD-ROM software distribution, developers were separated into the large corporate firms that could afford to pay for expensive retail shelf space and independent shareware developers who distributed their wares for free at local user-groups, on BBS’s, or through the mail, and relied on users to send them money after the fact, often on the honor system with no guarantee of actual payment. The dot.com boom of the late 1990s, which created the e-commerce infrastructure for electronic distribution and payment over the Internet, allowed

independents to achieve distribution on the same scale as corporate firms without the risk or expense of physical inventories. Mac OS X and the Internet allowed a number of former NeXT developers to become "indies," which many Apple developers started to call themselves in the early 2000s.

One of the most prominent former NeXT developers to make the transition to indie consumer Mac application development was the OmniGroup. Omni released consumer apps such as OmniOutliner, an outlining tool, and OmniGraffle, a graphing tool written by Kevin Steele (formerly of Lighthouse) that was a ground-up rewrite of Lighthouse's "Diagram!" application. These apps enabled Omni to become independent of contracting by the mid-2000s and grow into a company of almost a hundred employees by 2012. Wil Shipley, who preferred working on his own projects rather than managing a company, eventually left OmniGroup to form his own company, producing a visually stunning book and media cataloging application, Delicious Library, which won Apple Design Awards in 2005 and 2007.

The Cocoa community did not remain solely composed of former NeXT developers, however. It expanded as a number of shareware Macintosh developers, seeing Cocoa as the future of Apple's platform, switched over to using Cocoa from Carbon. These included Brent Simmons, who wrote the popular NetNewsWire Usenet newsgroup and RSS feed reader, and Panic, which started with the MP3 player Audion on classic Mac OS, but shifted to making the Cocoa-based FTP client Transmit. Although such Carbon to Cocoa conversions were few compared to the large numbers of corporate Carbon developers who worked for Adobe or Microsoft, because of the small existing size of the old NeXT community, they made for a significant expansion of the Cocoa community. As OS X matured and its use spread, new programmers who wanted to write Mac software, such as Mike Lee, often decided to start with Cocoa. Mike Lee had been working for Alaska Airlines when he decided to apprentice himself to Wil Shipley for a year, ultimately joining Shipley's company, Delicious Monster.

For five years, Cocoa and Carbon were both officially supported by Apple, which told developers that both were equal peers that would continue to be developed in parallel for the foreseeable future. Apple's third party developer community, however, was initially divided between Carbon and Cocoa developers. Former NeXT developers, such as Shipley, did not hesitate to proclaim the superiority of object-oriented Cocoa over procedural Carbon. Although most third party Carbon apps were written in C++, an object-oriented language, NeXT devotees considered C++ an inferior language to Objective-C because it did not force developers to think in object-oriented ways. Carbon developers resented what they felt was arrogance on the part of the NeXT people, who they felt came into the Mac world and took over. Moreover, the NeXT executives in charge of Apple had initially sent them a similar message with the Rhapsody strategy. Despite retreating from this, Carbon developers continued to see slights every time Apple moved to favor a NeXT-based technology in OS X over a Mac-based one. A suspicious Carbon developer could reasonably think that Apple's long-term plan was to get rid of Carbon and eventually focus on Cocoa as the sole native OS X development environment, despite Apple's continued messages to the contrary. In 2006, their fears were finally realized. Apple was undergoing a major transition in its operating system from 32-bit to 64-bit, which required that application frameworks needed to be updated for applications to take advantage of 64-bit capabilities.[22] Apple announced that in the next version of Mac OS X, 10.5, Cocoa would be updated for 64-bit operation but not Carbon. This clearly signaled to developers that Carbon no longer be in active development in future versions of OS X, effectively relegating it to the dustbin.

---

[22] Such capabilities include being able to handle large contiguous chunks of memory. Applications such as Photoshop work more efficiently if they can map a large 2GB file directly into memory as a single unit, which requires 64-bits. 32-bit applications must rely on tricks to break up the memory into small chunks that have to be linked to each other, which requires additional overhead.

The second major environmental shift for Cocoa developers was the release of the iPhone in 2007 and the opening up of the iPhone App Store in 2008 to third party development. While some developers retrospectively believe that Apple intended to open up the iPhone for development all along, my assessment of Steve Jobs' orientation towards maintaining control, supported by evidence from Isaacson's biography of Jobs, suggests that it was third party Cocoa developers and iPhone users-turned-hackers that created the market for apps on the iPhone, forcing Apple to reverse its stance on sanctioned third party app development. On its release in 2007, the Cocoa developer community immediately understood the iPhone to be essentially a scaled down Macintosh, running a version of OS X. "[My coworker] Tristan calls his iPhone his trouser Mac. That's really, that's what it is. It is a Mac, in your pocket, that also makes phone calls and takes pictures. Which a lot of Macs do anyway." (Mike Lee, Interview, July 23, 2008.) The potential was obvious—one should be able to write fully native applications for the iPhone, that could be as functional and sophisticated as anything on the Macintosh, rather than the limited programs one could find on the cellular phones of that era. However, Steve Jobs announced initially that developers would not have access to an SDK to write native apps for the iPhone. They would have to make due by tailoring web applications for the iPhone instead. Walter Isaacson's biography reveals that Jobs was concerned with maintaining control over the quality of a user's experience with the iPhone, something that third party developers might mar with badly written, even malicious, programs, though some executives were actively lobbying to change his mind (Isaacson 2011). Hackers quickly discovered how to "jailbreak" the iPhone to allow customization. Developers such as Lucas Newman, at the time an employee of Wil Shipley at Delicious Monster, tinkered with jailbroken iPhones and discovered how to write programs for the Unix-based devices. Less than a year after the iPhone's release, an underground market of native applications written for jailbroken iPhones had exploded. The users had spoken. Apple could either try to stamp out this market against the enthusiasm of its users, or legitimize it and get ahead of it, providing their own app market under their control. They chose the latter, announcing the App

Store and the iPhone SDK in March of 2008, with the App Store scheduled to go live in July.

While creating apps for the underground jailbreak market had primarily been the work of hackers who searched through header source files to find hooks into the iPhone's functions, the iPhone SDK, which is based on the Cocoa toolkit on Mac OS X, immediately made existing Cocoa developers instant experts on iPhone development at a time when no other programmers had any comparable expertise. Apple signaled this by branding the iPhone SDK, "Cocoa Touch" to signal that it was simply the touch-input version of Cocoa on the desktop. Development for both platforms takes place in Apple's Integrated Development Environment (IDE), called Xcode, and use the same compiler and same programming language, Objective-C. Cocoa on Mac OS X is roughly divided into a lower level layer, Foundation, and a higher-level one, AppKit, which provides graphics routines and user interface controls. (Foundation and AppKit have retained their names from NeXT. The names of most object classes in these libraries start with prefix "NS," signaling their origin on NeXTSTEP.) On iOS, a new framework called UIKit, designed from the ground up for touchscreen interaction, replaced the AppKit, which had been tailored to mouse-based input. Although new, UIKit's design was based on similar principles as AppKit, but allowed the team to discard unneeded legacies and learn from its mistakes.

As noted in the previous chapter, the iPhone App Store made developing for an Apple platform attractive for the first time to thousands of developers who previously would have avoided it. Apple made efforts to make iPhone development more accessible to individuals. Unlike Microsoft, Apple distributed its developer tools for free on Mac OS X, and distribution on the App Store required only a $99 annual Apple Developer membership. For a 30% cut of revenue, Apple would provide a distribution platform free from having to host one's own website or sell expensive retail boxes. Developers of all sizes could compete with each other on a more level playing field. In this new environment, it seemed that anyone could write an app and sell it on the App Store. Apple seemed to be democratizing app

development. The press was awash with stories of programmers who quit their day jobs after making a fortune on the App Store. Other stories stressed the ease with which one could write an iPhone app—one story featured a Chinese boy. (Smykil 2009) To the indie Cocoa community, this was the culmination of the dream, and the fact that iPhone development used NeXT technology vindicated their faith all along that NeXT's object-oriented development kit was the future. Indeed, for some developers like Wil Shipley, Cocoa's triumph with the iPhone now allowed one to rewrite NeXT's history as not one of failure, but one of success, a tale of stymied technological progress that finally got to see its vindication. Likewise, for the formerly beleaguered NeXT developer community, the success of Cocoa Touch proved that they were right all along in their devotion to Cocoa. "Now we've crushed them. I install [on] like 125 million devices worldwide, or something. We've actually crushed everything else. Ha ha ha ha." (Wil Shipley April 18, 2012)

The market for small-scale iOS apps, which are smaller and quicker to write than large, feature-bloated desktop applications, also encouraged small-scale development by hobbyists and entrepreneurs. For the Cocoa community, this also validated the chosen economic and labor model, the model of the independent artisan-programmer, which had been the form of organization for them as NeXT developers during the 1990s. Looking back, Cocoa developers who had started on NeXT now tell a triumphalist narrative, in which their struggles in the wilderness during the 1990s would lead to NeXT technology powering Apple and iOS, and its loyal developers, to take over the world, creating in the process a technolibertarian utopia for independent, entrepreneurial software development of consumer applications.[23] They trace a straight line between their independent development on NeXT to their status as indies on OS X and now iOS.

---

[23] This sense of utopia may have been just an initial overreaction. In 2014, there has been much discussion on the Cocoa blogosphere that indie iOS development simply is not sustainable anymore. (Simmons 2014) In the iOS game

> "Yeah, so we instantly saw [NeXTSTEP in the 1990s] and we said… it's us versus [the hundreds of programmers writing Microsoft Word]… [That] is not the future we see, we don't want the automobile industry to be the software industry. We want it to be the individual artisan, and we won. We fucking won. There are still big software companies, but I don't think there's anyone that's going to sit here and tell you that innovation is coming out of Adobe. …Now, no one gives a shit. No one cares. It's not driven by them anymore. It's, what is the next Tiny Wings going to be."
>
> […]
>
> One guy is making Tiny Wings and making ten million dollars, I don't know, so much money, right? One guy. One guy makes this damn thing in fucking Newfoundland… over the ocean, and he's a dude. Remember the story of one guy making a car? No… One guy make a drug? No… It doesn't fucking happen. This is it! This is the industry, the only thing in the entire world, where one guy can [have] a hundred million customers. (Wil Shipley, Interview, April 18, 2012)
>
> "This is like my wildest dreams come true. Millions and millions of indies! And we're all working together in this way…"
>
> (Andrew Stone, Interview, June 7, 2011)

Longtime NeXT/Cocoa developers see Apple's iPhone and its app market as allowing individual developers to compete head-to-head with giant corporate software firms, dramatically democratizing software production, liberating it from corporate oligopoly control, and empowering millions of users by transforming them into app makers, free to customize their iPhones by making apps for their own use. Part of this they credit to the App Store itself, which removes the need for setting up one's own distribution and payment infrastructure. However, for veterans of the NeXT era, who have waited so long for the rest of the programming world to

---

development space, there has been increasing market consolidation, and independents have a difficult time competing with big name companies. It appears that the mobile market is no different from other markets—an initial period of entrepreneurial activity is followed by consolidation and eventually oligopoly of the big corporate players.

experience the pleasures and advantages of programming with Cocoa, it is also due to the qualities of Cocoa itself. As Brent Simmons, creator of NetNewsWire, remarks,

> A lot of the work that we had done for the past ten years [from roughly 1998 to 2008] at that point, was about trying to see this platform that we felt made us five to ten times more productive, succeed. So we could continue using this productive platform instead of having to go off into Windows programming or Java programming or whatever the alternatives were…
>
> [Cocoa gave us] power. The ability to do more with less code. (Brent Simmons Interview, February 17, 2012)

As developers like Brent Simmons see it, Cocoa's ability to magnify their power, to help them do more with less code, is a direct consequence of its design as an object-oriented development environment. As we have seen in this chapter, this rhetoric of magnifying productivity, repeated by NeXT itself and its developer community in the 1990s, was echoing arguments by Brad Cox and others that object-oriented programming could be the silver bullet to solve the software crisis. However, even in the late 1980s, when NeXTSTEP was first shipped with NeXT hardware, it was not the dominant object-oriented development environment. C++ was, and would continue to be until it was supplanted by Java. NeXT developers then, and Cocoa developers now, continue to argue that Cocoa is superior to both C++ and Java for making programmers more productive (though they might acknowledge that some newer environments, such as Python and Ruby, are equal to or better than Cocoa). Clearly, then, not all object-oriented development environments are equal in the eyes of Cocoa developers. In the next chapter, we will explore more fully the specifics of Cocoa technology and why Cocoa developers believe it is better than other object-oriented platforms.

Chapter 3: Why is Cocoa Better? Technical Design, Normative Practice, and Trust in Apple among Cocoa Developers

In chapter 1, we discussed the ideological, normative, and affective aspects of the cosmology of the Cocoa developer community, in particular, its celebration of the "indie." Cocoa developers spoke in particular of the pleasure they experienced working with Cocoa technology. However, this story might have been told about a number of programming subcultures and their affective commitments to particular tools and platforms. Do the specific qualities of Cocoa technology itself matter? If so, why and how do they matter to Cocoa devotees? In this chapter, we will examine the technical arguments Cocoa developers use to articulate why they believe Cocoa is a superior technology for their needs, and possibly for programmers in general.

In chapter 2, we looked at the historical origins of the discourses we will be examining here. Much of this argument for object-oriented programming in general, and NeXT specifically, focused on improving programmer "productivity." As we saw, NeXT marketed its object-oriented development environment as a way to magnify the productivity of programmers by 5 to 10 times. These productivity claims were realized in the experiences of many NeXT developers. After NeXT was acquired by Apple, this technology was renamed "Cocoa," but it was still composed of the same basic elements: two primary object-oriented code libraries, Foundation, which provided basic, general-purpose functionality necessary for all programs, and AppKit, which provided user interface objects for applications. In addition, these were tightly integrated with Interface Builder, a graphical tool for building an application's graphical user interface. What about Cocoa technology supported this increase in productivity? First, we will examine the three key elements of Cocoa that developers argue make them more productive. First, it is consistent. Second, it allows for flexibility. Third, less code needs to be written to perform the same tasks than with other development environments. Because Apple's Cocoa tools make them more productive, allowing them to write by themselves applications that can compete

with the products of corporations with hundreds of programmers, Cocoa developers credit Cocoa with giving them the ability to be indies, independent craftmen-entrepreneurs.

By examining these features of Cocoa, I will argue that Cocoa works for and is embraced by developers not because it deskills the tasks of programming, but because, while making many programming tasks easier by automating or black-boxing them, it raises the skill required to program in Cocoa because it focuses programming on the more abstract, conceptual level of design and architecture, rather than the low-level task of implementation and tinkering, although work at both levels is still required. This higher level of understanding is often a barrier for programmers new to Cocoa, responsible for what is often called, Cocoa's high "learning curve." In the latter half of this chapter, I will explain two of these concepts at the center of Cocoa that new developers have difficulty with, "model-view-controller," and "delegation." One of the reasons for this difficulty is that such concepts can only be grasped holistically, rather than piecemeal. The result is that until the whole is grasped, learners may be skeptical of Cocoa's purported benefits. Once this learning curve is surmounted, however, programmers report having experienced a "conversion," whereupon they can see the forest for the trees.

Consistency

The Cocoa developers I have interviewed have repeatedly expressed how consistency is a central design value at Apple. Cocoa developers say that Apple designs its graphical user interfaces with an element of consistency, so that a user who becomes used to using the mouse to drag and drop a file to move it from one place on a disk to another, later can expect to be able to drag and drop text in a document to move it as well. Certain gestures, idioms, or metaphors used in one place do something similar in another, and after the user becomes accustomed to this, he or she begins to be able to predict that the same gesture may work similarly for completely unknown tasks. Programmers using Apple's Cocoa libraries (which are often called "frameworks") also make use of interfaces in order to use them.

Programmers call "functions" that are defined in these libraries, which perform complex tasks for them, alleviating them of the need to write code to do these things themselves. The names of these functions are defined in textual code files known as "header files," which define the publicly available Application Programmer Interfaces (APIs) of the library. Just as with user interfaces, consistent programmer interfaces allow an experienced programmer to predict that a familiar idiom (such as how functions are named, or how certain problems are solved with particular patterns) will apply in similar future situations, and be correct that it does.

> "It became really obvious that there was no other framework I'd ever worked with that was that consistent… Now, when I approach a new problem… in the Cocoa space, it's very easy for me to anticipate how Apple is going to have thought about this problem. And so it makes the next thing much easier for me to deal with. And I can anticipate how they're going to handle errors, I can anticipate, oh, hey, they're probably going to [this] pattern [to solve this problem]… I come in with all of this… pre-knowledge without ever having seen whatever the new class is. And that's really powerful for me."
>
> (Hasan Edain, Interview, March 12, 2012)

One developer said Apple is driven to design both its user interfaces and its programmer interfaces to be consistent in order to make both simpler and easier to use, and thus, more "humane," a term which has been used in the title of a book on Human Computer Interaction, "The Humane Interface," by Jef Raskin, the original leader of the Macintosh team.

> "For me, it's about being humane… the general approach of most developers in this platform, is to recognize our users as human beings, worthy of respect, and to build things that treat them that way. On the other side of it, and this… gets at the ease of development [aspect], these are… humane frameworks and humane tools to develop with… And so they tend to treat the developer with respect …For the most part, the frameworks really are kind to us."
>
> (Curt Clifton, Interview, March 23, 2012)

Consistency can even help programmers predict what code to type. For programming, Apple provides what is called an Integrated Development Environment (IDE), a programming environment that integrates most of the tools one

needs for writing an application in a single program: a text editor, compiler (which translates human readable, high-level source code into the binary machine code to be run by the computer), and debugger. Apple's IDE is called Xcode, and by integrating a real-time compiler with the text editor, as the programmer types, Xcode can compare the first few characters being typed to its database of APIs in the Cocoa libraries, and suggest a list of possible APIs that it thinks the programmer is typing. Thus, by typing only one or two characters, and then selecting from a drop-down list, a Cocoa programmer can input a long function name, such as "NSAccessibilityPostNotificationWithUserInfo." This convenience is called "code completion." Because Apple has followed consistent rules with naming its APIs, an experienced Cocoa developer can often, without looking up the name of an API in Apple's documentation, simply begin to type what he or she thinks is the name of a function, and Xcode's code completion will bring up exactly what he or she is looking for. (Mark Dalrymple, Interview, April 11, 2012)

Such predictability was key to why one developer considered Cocoa the best of all development environments: "Well, at least it's predictable, right? You don't get that with the web… As far as I'm concerned, [Cocoa is] the best framework out there." (Gus Mueller, Interview, February 21, 2012)

NeXT and Apple software engineers spent considerable effort making sure consistency was a top priority. Becky Willrich, a former NeXT and Apple engineer who was on the Cocoa framework team in the early 2000s,[24] had this to say about how important consistency was. "I think the thing that gets lost by a lot of people is that when you think of the UI or the API, there's this enormous win from consistency." (Becky Willrich, Interview, April 15, 2012)

Another value that drives the design of APIs at Apple is simplicity. R.D Wilhoite, also a former NeXT and Apple engineer, noted:

---

[24] This author was a software quality assurance engineer on the Cocoa framework team in the 2000s, and was a colleague of Willrich at this time.

> "I think of what the NeXT folks are about, which is simplicity for simplicity's sake… I'm not saying every ex-NeXTer or Apple person is like this, but it's prevalent there in terms of the culture that there's almost an allergy [to things] that are unnecessarily complicated. It's like, you want to make it as simple as possible."
>
> (R.D. Willhoite, Interview, September 11, 2011)

Apple engineers pursued simplicity and convenience in order to make things easier for the majority of users, including users of its programmer interfaces, but achieving this could be difficult. Often, this meant that user or programmer choice needed to sacrificed, for the greater good:

> I certainly think that's true in the UI [of] iOS and Mac OS X… there are some configurations you don't get, period. Because it damages [consistency]… the only way to achieve consistency is to be absolutely draconic [sic] about options, and cutting them off, and refusing to allow things. …Steve [Jobs] has a very famous quote… about… the importance of saying no to most things. And the same notion has been applied to the APIs. (Becky Willrich, Interview, April 15, 2012)
>
> Because any time you added developer convenience to these simple APIs, they become less simple… You really… want kind of one-stop shopping in that simple API. You don't want them wondering which of three different flavors is going to best suit their needs. There should only be one choice. If they don't care, there should be one choice. (Becky Willrich, Interview, April 15, 2012)

Willrich pointed out that the lack of options in the simple API, which most developers would likely use, did not mean that a minority of developers who wanted to do something more sophisticated were left out in the cold. There were often two layers of APIs, a simple, high-level, object-oriented one, and a lower-level, procedural, C-language based API, which allowed for a lot more freedom and power but was significantly more complicated to understand and use. With the lower level interface, developers are given free reign, but with added responsibility. They must know what they are doing. "And then you provide them will a full toolkit, a full tool chest, if they want to look under the hood, it's all there, they can tinker with whatever they want, but now it's their responsibility." (Becky Willrich, Interview, April 15, 2012) The availability of the complex, knowledge intensive low-level API freed Apple's designers to strip out options in the higher-level interface. This

allowed it to be as simple as possible, in order to not unduly burden a developer who did not need all of the options with what Willrich called "cognitive load." As a result, the simpler, higher level APIs in Cocoa are significantly easier and more convenient to use. With one line of code, using a single call, one can accomplish what would take many lines of lower-level code to do.

Cocoa developers appreciated this level of consistency in the Cocoa APIs, especially compared to other platforms. Seattle developer Hasan Edain has written an app for Google's Android mobile operating system, and said of the experience:

> I had two different objects that I had to rotate on the screen… one of them took its rotation in radians and the other one in degrees. And it's like, wait a minute. You're Google. You can't even get it together to, like, coordinate on what your units of measurements are for your API? Seriously? …You [should] make that decision at a high level [in the company], you communicate it… and if it happened once and only once, it would have been, OK fine, here's this thing and somebody will sand off the rough edges at some point… And then there were like three or four other things, where I was like, wait a sec, here are two different APIs and they used two different design patterns to approach the same basic problems… oh, no. There's just not the level of quality in this API that makes me want to go and deliver lots of stuff in it. (Hasan Edain, Interview, March 12, 2012)

Willrich speculated that such inconsistencies in design mapped onto a lack of social coordination among teams.

> I think the people who were involved [with Cocoa] simply cared. … I wonder if at the other place there is simply a lack of this kind of core oversight or core vision to hold it all unified. Like, I suspect that other APIs are kind of developed with, one team develops this piece and then another team develops that piece, and then you take seven or eight of these and then mash them together and there isn't anyone who owns the—or who has a sense of ownership over the APIs as a whole set, to say look, no, it doesn't matter if these two ways are just as good as one another, we have to choose one. And that means there has to be one winner and one loser, and I'm sorry, you're going to be the loser. Nobody cared to apply that level of oversight.
>
> (Becky Willrich, Interview, April 15, 2012)

At NeXT and later Apple, a small team of people had a singular vision for how things should be designed, and the longevity of these people, who rose or remained in key positions at the company, made sure that such consistency was maintained over time:

> You will see that kind of consistency inside groups of five to eight engineers where it's small enough that it's easy to hash these issues out and to share the ownership. As what you're seeing is the incredible longevity of those people, right? So if you take those four core people who worked on the AppKit and then in the fullness of time, they each went on to their own piece of the Apple pie… Bertrand [Serlet] of course rose to VP, Senior VP [of Software at Apple]. Ali Ozer [manager of the Cocoa group] held the AppKit, which is sort of the core API piece… But if you think about Ali, he's been doing essentially the same job for twenty years? You don't see that at other companies. That's what's providing the longevity and singularity of vision. (Becky Willrich, Interview, April 15, 2012)

Cocoa developers mostly appreciated this singularity of vision and the simplicity and consistency it created, even if they occasionally chafed under their limitations. Apple's top-down level of control was necessary for the coherency of the Cocoa interfaces that they enjoyed, and showed a directed intentionality to the way Apple designed APIs and how it gradually introduced them over time.

> To me it's very natural… From the design of the APIs to the leadership that Apple shows in driving those APIs… This is a pleasant environment to program in, you don't feel like you're fighting it. (Adam Preble, Interview, August 8, 2011)

Why were they willing to live with this control by Apple? Many Cocoa developers mentioned that Cocoa's consistency, by reducing frustration, created in them an aesthetic, pleasurable response to its design, and to the experience of using it:

> NeXT technology was just so much—it was beautiful by comparison… I don't know, it is sort of an ephemeral quality. I mean… it's just consistent. (Luke Adamson, Interview, February 22, 2012)

Cocoa developers receive an affective, pleasurable response to using Cocoa APIs due to its consistency, simplicity, and intentionality, which they understand

comes from Apple's tight control over its design. They are willing to let Apple have this control because in return, they not only become significantly more productive, but they also get so much more enjoyment from programming in this environment, that they would rather not go back to coding for other platforms.

Flexibility

Cocoa developers have also spoken of the flexibility afforded to them by the technology. Depending on the context, they may mean two different things. Sometimes, they refer to Cocoa and Objective-C, the language it is written in, as "dynamic." Glossing over the details, this computer science term roughly refers to a category of object-oriented languages in which the types of objects and the kinds of operations they can perform are left undetermined until "runtime"—that is, until the program is actually run. Smalltalk was the first dynamic object-oriented language, and subsequent languages such as Objective-C, Python, and Ruby were patterned after Smalltalk, and are likewise classified as "dynamic." These are distinguished from "static" object-oriented languages, the exemplar being C++. In C++, object types and their operations are locked down when a program is compiled from high-level source code into binary machine code.

The debate between which approach was better has been going on since the 1980s, when C++ and Smalltalk were the two primary object-oriented languages in use in industry, each representing a different approach. C++'s main advantage over Smalltalk was its compatibility with C, the procedural language which was the industry standard. This meant that C code could be directly mixed in with C++ code, allowing existing C code bases to be reused in C++ programs. Bjarne Stroustrup at Bell Labs created the language for maximum compatibility with C, and for maximum performance. (Stroustrup 1993) The decision to make it "static," in other words, to make decisions about objects when the program is compiled, enabled C++ programs to run almost as fast as C programs. Smalltalk, on the other hand, relied on an interpreter, an interactive program that translated source code into machine code in real time. This made it significantly slower. Yet, because the interpreter worked

interactively, programs could be modified while they were still running, adding significant flexibility. Objective-C was created by Brad Cox as a compromise between these two approaches. Like C++, it would be compiled, and be fully compatible with C code. However, Cox wanted Objective-C to bring Smalltalk-style object-orientation to C, and thus made it more "dynamic," deferring key decisions until runtime.

This difference was touted by NeXT developers as a key advantage of NeXTSTEP and Objective-C over C++. For example, a sidebar in a NeXTWORLD article on NeXT's object-oriented advantages explained to readers the benefits of Objective-C's "runtime binding," also referred to as "dynamic binding," while taking a swipe at C++ through a quote from independent developer Andrew Stone.

> The most important feature of Objective-C, supporters say, is run-time binding. You don't need to know what an object is before you send it a message. Run-time binding different objects respond to the same message in different, but functionally similar, ways. More importantly, it frees the programmer from having to know all the details about the application program's run-time environment at the time the application is being written.
>
> For example, a headline and a photograph on the NeXT's screen are represented by different objects inside the computer's memory. Both of these objects, however, respond to the drawSelf: method [a function that only objects of a certain type can run] to make themselves appear. A drawing program doesn't need special-case code to handle all the different kinds of objects that a user can draw on the screen—it just sends every object a drawSelf: message, and the objects do the rest of the work for themselves.
>
> Closely coupled with run-time binding is NeXT's dynamic loading of run-time libraries. Instead of linking libraries into each application program, they are kept in a "shared library" on the disk that is loaded when an application program is run. This saves space on the disk. More importantly, it lets NeXT upgrade the shared library and have that change reflected automatically in existing application programs.
>
> […] What about C++, the object-oriented language that's become an industry standard outside the NeXT community? "There's nothing about C++ that invites you to write good object-oriented programs," says [Andrew] Stone [of Stone Design]. (Garfinkel 1992c)

A subsequent sidebar on NeXT's Interface Builder noted that Objective-C's run-time binding was the critical piece that made it possible:

> Interface Builder lets a programmer literally draw the interface that they want a program to have. Windows, text fields, sliders, buttons, and more are all on palettes ready to be dragged off and resized like some kind of high-tech drafting program… But Interface Builder doesn't just draw the interface—it actually constructs the Objective C [sic] objects that the application program will use and saves them into a NIB (NeXT Interface Builder) file. When the program runs, it loads those same objects out of the file and into the computer's memory.
>
> "You're not just prototyping," says [developer William] Adams. "You're actually developing what you need to develop…
>
> But the real boon of Interface Builder comes when changes have to be made. It's easy to add more buttons, move fields around, or even add whole new windows and panels to existing applications. That's something that no mere prototyping tool can do.
>
> The reason? Interface Builder works intimately with both Objective C and the App Kit [sic]. Without Objective C's run-time binding, it wouldn't be possible to simply draw an application's interface and have it work, because the application program would have to be compiled with the location and type of every widget in the program's windows. (Garfinkel 1992b)

Indeed, Steve Naroff, the Stepstone engineer who modified Objective-C for NeXT and later joined NeXT, said that key features were added to Objective-C for the express purpose of making Interface Builder, and its ability to make live changes to an already compiled application, possible. (Steve Naroff, Interview, December 22, 2011)

Cocoa developers today likewise express similar advantages of dynamic languages such as Objective-C and Ruby over static languages like C++. One developer said that the flexibility of being able to make changes to a program at runtime greatly improved his ability to adapt a program.

> And so in traditional, either C, or C++… I would spend a lot of time prior to implementing a class [of objects], really trying to understand, "OK, what happens if I have this change in the application"… really having to understand very, very deeply what the use cases on this were.

> And that hems you in a lot, right? Because then all of a sudden if one of your pieces of understanding gets shifted, which inevitably happens, then oh, crud, now I have to reexamine this. Whereas… having a higher level of flexibility, a lot of times it just means I take something from column A and I move it over to column B. And that kind of flexibility and/or change is really, really important, especially in an environment [like] the iPhone…You have the flexibility to handle design changes much more simply…
>
> (Hasan Edain Interview, March 12, 2012)

Another developer, who has written programs both with Cocoa, and with Rails, a popular object-oriented library for programs running on web servers, written in the dynamic language Ruby, implied that the order-of-magnitude improvement in productivity gained with dynamic languages allowed a small team of independent developers to write software with equivalent functionality as large teams of developers working for corporate firms such as Microsoft:

> We had huge armies of Microsoft developers [on the one hand] and [on the other hand, we] had all these little Rails guys who were all by themselves, two people at a time, just blowing out software. …Rails is a dynamic language, it's designed to take some of the best aspects of other things like SmallTalk, Python and Perl and make a language that is more expressive to the developer, and easier to manipulate…
>
> With Microsoft, it is a language and a platform that's designed to… put the proper restrictions around so people won't hurt themselves. …with Microsoft there's always a perspective or a mentality of wanting to give you the clear and safe path and keep you from doing things you shouldn't be doing. So… making the framework more static, and static typing and things that are really protective devices. They give you tremendous performance in the end, so you get this amazing performance out of the framework, so you can really leverage hardware, but what you get is *you get a lack of productivity in your team*. [Emphasis mine]
>
> And so the compensating fact for that is to scale out your team and have more developers and just bring down the level of expertise and have a larger team who can just pump out more code. *The bottom line is you just need to pump out more code to make the same thing happen.* [emphasis mine] You find yourself reaching an obstacle more frequently in a static language such as .Net… When you reach an obstacle you have to work around it or you have to work harder to overcome it, whereas with a dynamic language, you're basically given

> a gun and said, "Go hunt and make things happen. If you shoot yourself, it's your own fault.
>
> And there is definitely some risk… implicit in that… Certainly there's bugs and things like that in all software, but the nature of dynamic languages is you can insert and replace anything you want, change behavior.
>
> And the Microsoft framework, you can't do that. So that single piece of functionality, that single capability, it makes all the difference in the world in being productive.
>
> (Rusty Zarse Interview, September 25, 2012)

Zarse is thus associating several things together. The technical properties of languages and the library/frameworks based on those languages (dynamic versus static) maps onto programmer ability and skill. Static languages are designed for lower skill programmers, who need guard rails to prevent them from doing dangerous things. However, these protections also hamper more skilled programmers from being able to change things around at will, resulting in decreased productivity per programmer. Thus, technical properties of languages had organizational consequences. Static languages produce large, bloated armies of low-skilled programmers, the very kind that Fred Brooks argued in *The Mythical Man-Month* would cause a software project to become more complex and thus more likely to slip in its schedule. Dynamic languages, on the other hand, would facilitate small teams of highly skilled programmers—surgical strike teams, rather than massed hordes.

This notion of small, two-person teams of highly skilled developers competing against low-skilled armies of Microsoft developers fits directly into the utopian vision of indie Cocoa developers, who believe that all software should be developed by small teams of artisanal programmers. Recall Wil Shipley's quote:

> "Yeah, so we instantly saw [NeXTSTEP in the 1990s] and we said… it's us versus [the hundreds of programmers writing Microsoft Word]… [That] is not the future we see, we don't want the automobile industry to be the software industry. We want it to be the individual artisan, and we won. We fucking won. There are still big software companies, but …It's not driven by them anymore. (Wil Shipley Interview, April 18, 2012)

Dynamic languages such as Smalltalk, Python, and Ruby are also seen as significantly more abstract, that is, have significantly more layers between them and a computer's hardware, than static languages such as C++. Some programmers, such as the computer scientist Antony Hoare, "argued that object oriented programming was the natural scientific progression of the programming discipline." (Zepcevski 2012, 266)[25] Such a view is based on a teleological understanding of the history of programming languages, in which languages began very close to the hardware, and gradually rose up in levels of abstraction from machine language to assembly, through procedural languages, all the way up to object-oriented languages and functional languages such as LISP. In this teleology, procedural languages are more primitive than object-oriented languages, because their modeling of programs as processes is much closer to how the computer hardware sees them, whereas viewing programs as collections of objects is a conceptual abstraction that is layered on top of this process-oriented model. The logic then follows, if higher abstraction represents "progress" in the programming discipline, then more abstract object-oriented languages, i.e. dynamic ones, are thus more advanced than static ones.

Some Cocoa developers articulated a different reason that Cocoa makes them more flexible, one that is in some ways contradictory, in others complementary, to Zarse's argument that programming in higher-levels of abstraction makes developers more productive. That argument is that Cocoa and Objective-C allows both high and low-level programming, thus giving them the flexibility to choose the appropriate level at which to work to solve problems. Robert Walker, an Atlanta-area Cocoa developer, felt that Objective-C's ability to allow a programmer to write code at two levels, a high-level, abstract, object-oriented one, and a low-level, optimized, procedural one, made it *more flexible* than pure dynamic object-oriented languages

---

[25] Zepcevski's source is Charles Antony Richard Hoare, Oral history interview by Philip L. Frana, July 17, 2002, OH 357. Cambridge, England, U.K. Charles Babbage Institute, University of Minnesota, Minneapolis.

such as Ruby or Python, which could not produce code that rivaled C or C++ in performance. In addition, without a compiler making sure a programmer did not make type mistakes, such languages are more error-prone:

> The thing that I like about Objective-C is flexibility, from a programmer's perspective. …It's a compiled language, which when you go [to] fully dynamic, interpreted languages like Ruby or Python… you lose a lot of the… help from the compiler…
>
> An Objective-C application gives you most of the benefits of the fully dynamic language like Ruby, without the compromise in performance… Objective-C… sort of does fit in that middle [ground]—it's sort of a compiled language, but it feels very dynamic."
>
> (Robert Walker Interview, May 19, 2012)

Walker located object-oriented languages along a continuum from static to dynamic, with C++ on one end, Ruby on the other. Java, although dynamic in some ways, was static in others, and Walker considered it closer to C++. Objective-C lay somewhere between Java and Ruby. Walker described writing Java code like building Lego blocks, while writing Ruby code was like playing with Play-Doh. He characterized Objective-C as being more like clay, less malleable than Ruby but still significantly more pliable than Java.

If higher abstraction was better for programmer productivity, why work at a lower level of abstraction? The common answer to this is that working "closer to the machine" allows programmers to fine tune a program for performance and efficiency. This is a key engineering value for programmers in general. C++ programmers, in particular, argue that performance (as well as the added safety from added compiler checks) is the primary advantage of C++'s static style of object-oriented programming—the dynamic method dispatch of Objective-C and Ruby is an order of magnitude slower than C++'s. For Cocoa programmers specifically, performance is only one value among many, including usability. Nevertheless, Objective-C's compatibility with C gives them a way to have their cake and eat it too. Because Objective-C is a full superset of C, Cocoa programmers often argue that for key code that needs to be high performance, they can simply "drop down to C," write highly

optimized, low-level procedural code, circumventing Objective-C's normally slower dynamic method dispatch. This allows the programmer to design the program in high-level, object-oriented fashion, which is much more productive. Get the program working first. Once this is accomplished, measure the program's performance characteristics using tools provided by Apple's development environment to find where the program runs slowly or uses too many resources and optimize those later. Developer Wil Shipley summarizes this attitude:

> … People go, Objective-C is too slow… and I'm like really, I can prove you're wrong… What we'll do is we'll write a program in Objective-C and then we'll write one in C, and then you can say, well, this one is slower and I'll go, OK, I'll look at where it's slow here, Oh, it's slow in this part, I'm going to write that part in C and do a really good, tight job in that, and now mine is faster and I wrote it in Objective-C and it's six times shorter. So I win. And I can do this again, and again and again and again, all day.
>
> …Only the parts that need to be fast should be fast. And the parts that don't need to be fast shouldn't be fast.
>
> (Wil Shipley, Interview, April 18, 2012)

In this way, the programmer optimizes only the portions of the code that are performance sensitive. This attitude is summarized by an aphorism popular among Cocoa programmers: "Don't pre-optimize," which means that programmers should not try to anticipate performance issues and write all code in a highly optimized way, but rather should try to make a program fully functional first, then measure where performance bottlenecks actually exist, and then optimize those. Why not optimize everything first, according to Cocoa developers like Shipley? Why, as Shipley says, are there parts of the code that don't need to be fast? Partly this is because in applications that interact with users, most of the computer's time is actually idle, waiting many milliseconds between user input, which can represent thousands of the CPU's clock cycles. But more importantly, not wasting time optimizing code which does not need to be fast optimizes the programmer's own time, which could be spent fixing bugs, implementing additional features, working on other programs, etc. This is particularly important for independent developers like Shipley who may be

working alone, and thus may also need to spend time running their business, marketing their application, and interacting with customers.

In making this statement, Shipley and others are drawing on a normative programming discourse that has existed for some time. It is not clear who first articulated it, but the computer scientist Donald Knuth spoke of this pitfall in his 1974 Turing Award acceptance speech:

> …programmers in the past have tended to be so preoccupied with efficiency that they have produced needlessly complicated code; the result of this unnecessary complexity has been that net efficiency has gone down, due to difficulties of debugging and maintenance.
>
> The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming.
>
> We shouldn't be penny wise and pound foolish…
>
> (Knuth 1974, 671)

Shipley's former partner, OmniGroup co-founder Ken Case, similarly expressed his opinion that Objective-C's ability to let a programmer mix low and high level code in a single language makes it the most flexible and productive language for writing applications. "In my opinion, Objective-C is the language that makes us the most productive for the widest number of tasks, or at least for the type of software that we're developing… desktop application development." (Ken Case Interview, February 10, 2012) Case argues that working at the highest, most abstract level possible when designing a program's architecture increases a programmer's productivity because it is these abstractions that help the programmer manage complexity:

> One of the things I like about Objective-C is that you can dive down to straight C when you need, or even do some embedded assembly… But that didn't mean we had to give up the abstractions of Objective-C at the higher level. So you can tune the performance and you could write the overall conceptual framework in this nice object-oriented environment, which *helps you manage the complexity*. [Emphasis mine]

> And complexity is really one of the biggest barriers in programming. The more complex the system gets, the harder it is for any developer to wrap their head around the whole system, so that's often where you start to run into a wall in developing.
>
> And so Objective-C really does help you manage that complexity a lot. …Your abstractions didn't have to change, and all the other higher levels could still see it as an object the way they'd always seen it, but the parts that needed to go faster, you just started writing faster at a lower level, without ever having to leave the language. The language has a lot of scalability from the lowest level assembly to the highest-level abstractions. (Ken Case Interview, February 10, 2012)

As we saw earlier, Fred Brooks, in his article, "No Silver Bullet" (F. P. Brooks 1987) argued that complexity was one of the essential difficulties that makes software inherently difficult. Joline Zepcevski argues that the management of complexity was one of the primary drives behind the development of object-oriented programming itself, and especially of Smalltalk, which influenced all the dynamic object-oriented languages, Objective-C, Python, and Ruby, which followed it. (Zepcevski 2012, 263–5) Similarly, we saw that Knuth in 1974 railed against premature optimization because it created unnecessarily complicated code that was difficult to debug and maintain. Thus, Case's argument is that Objective-C's hybrid nature allows a programmer to have the best of both worlds. It allows the programmer to decide the proper tradeoff between optimizing his or her code (which optimizes the computer's time) and optimizing his or her own time, by allowing the programmer to switch between high and low-levels of abstraction. The more time spent at the higher levels, the more the programmer is able to manage the inherent complexity of the program, thus increasing his or her productivity by an order of magnitude.

Less code

Writing high-level, object-oriented code also has an additional benefit. It allows a programmer to use less code to express the same amount of functionality—in fact, often an order of magnitude less code. Calling certain Cocoa APIs often allow a programmer to trigger complex events with only a single line of code. As we

saw in Wil Shipley's earlier quote, compared to a programmer who writes completely in C, "I wrote it in Objective-C and it's six times shorter." (Wil Shipley April 18, 2012) Thus, a key reason using Cocoa makes a programmer an order of magnitude more productive is because she needs to do an order of magnitude less work for the same result. We also saw earlier that Brent Simmons equated this with power: "The ability to do more with less code." (Brent Simmons Interview, February 17, 2012)

Being able to "do more with less code" magnifies the capabilities of individual programmers, but at the same time, reduces the need for large teams to produce the same level of functionality. In this way, Cocoa levels the playing field for individual developers against large corporate software firms. The productivity benefits from being able to "do more with less code" makes the "indie" developer possible—it allows developers to retreat from corporately managed software organizations. For this reason, "less code" has become something of a virtue within the Cocoa community.

Within the Cocoa community, "more code" is a vice associated with corporate software practices that are the result of perverse incentives counterproductive to producing good programs. Developers sometimes spoke of older corporate compensation policies that paid by lines of code written, equating quantity of code with functionality produced. For Cocoa developers, however, more code does not mean better quality—it adds complexity, increasing the probability for errors, and thus decreasing software's maintainability over time. Cocoa developers feel that the perverse incentive to generate more code has contributed to the intractability of large, complex, buggy legacy software systems that programmers are afraid to touch for fear of breaking them. Developer Wil Shipley felt that the corporate software industry simply did not understand that less code translated directly to higher programmer productivity.

> The preponderance of evidence is, we've got all these [Cocoa] programmers who are making these programs in record time that don't seem very buggy and seem very performant… we've got all these

> things that we've made with tiny teams that are beautiful and they're using Cocoa and I think we've proven that having less code in an app is just better…
>
> As late as the '90's people were still saying… I want a language that lets me [run programs] really fast, I want a language that lets me get to the hardware—the goal wasn't see how little code you can write. And I think it's only in the last ten years that we started understanding… the only method that really matters is less code.
>
> It's very rare, it's the exception, do you say, 'Oh, no, this is too slow, so I'm going to make it a little more code,' but that's an exception. The only method that matters is how little code did you use, overall. That's the only thing. There's no other metric, because they all fall out of that. Is it fast? Well, generally less code is going to be faster. Not always, but generally. And also, if you use less code, you're [more likely to be] reusing things and you can optimize things that you're reusing and someday you'll think it's faster and magical, right? It falls out. (Wil Shipley, Interview, April 18, 2012)

Becky Willrich, a former software engineer on the Cocoa team, explained how, at Apple, Mac OS X's designers created two levels of APIs: Cocoa was the high-level, object-oriented Objective-C layer, which was built on top of a low-level, procedural, C-based layer. According to Willrich, Apple's engineers tried to make the Cocoa APIs as simple and convenient as possible, for the vast majority of the interface's users (who are programmers). However, if they needed access to functionality that the Cocoa APIs did not provide, they could drop down to the lower-level C layer, with access to the complete toolchest. This, however, meant having to manage considerably more complexity that the Cocoa APIs hid from the user, a complexity Willrich called "cognitive load."

> "Most developers may want to do this—what if they want to configure just one [thing]… Why don't we just add [this one option]? Well no. That adds to the cognitive load of the simple API. Right? Because now you have to learn a little more. …It was just a very high bar for adding essentially, developer convenience… Because any time you added developer convenience to these simple APIs, they become less simple…
>
> And then you provide them will a full toolkit, a full tool chest, if they want to look under the hood, it's all there, they can tinker with whatever they want, but now it's their responsibility…Once you reach

> that low level, you really have to… trust that the developer knows what they're doing…" (Becky Willrich Interview, April 15, 2012)

Willrich saw that precluding options in the high-level API was justifiable because if the programmer really needed those options, they could go to the lower-level API. This would keep the high-level API simple, to maximize simplicity and thus ease of use for the largest number of programming users. The unlucky few would have to take the extra time to learn the low-level API instead, but because they were few in number, this was a valid tradeoff:

> If we have satisfied 95+ percent of the developers we do not mind that the remaining three, four or five percent are going to have to work a lot harder. It is worth giving the 95% a simple one-line API at the expense of requiring substantial work on the part of this, three, four, five. Because… if you look at those 5% of developers for whom the simple API is not complex enough, [for] at least half of them, no API would have been sufficient shy of the full toolkit. …You're going to end up wanting all the bells, knobs and whistles anyway. So it's not like I could have provided a slightly more complex but still basically simple API and still satisfied you. So they don't count anymore. They were satisfied when I exposed the full toolkit, OK? So now you're only talking about 2% or so of people who might have genuinely been helped by a slightly more complex but still simple API. And the cost of telling them, "look, we're sorry but you're going to have to become at least passingly familiar with this bigger, fuller toolkit," that's fine. I have no problem with that. [Rather than burden the rest of the 95%] with even a slightly higher cognitive load…
>
> (Becky Willrich, Interview, April 15, 2012)

Thus, Apple's designs for Cocoa emphasize giving developers convenience functions to allow them to do common things simply and easily, with as little code as possible. Simultaneously, it also provides more specialized capabilities that would take developers hundreds of lines of code to implement for themselves, a task that likely would require collaboration with additional programmers to accomplish. Developers noted that Cocoa gave them capabilities they wouldn't have had otherwise:

> As somebody who is a creative person… tools like [the] CoreAnimation [framework] make it extremely easy to do things that

> on Windows, I would be like, I don't even know how to begin doing this. That's just an example of… just great APIs.
>
> (Adam Preble, Interview, August 8, 2011)

Prior to 2006, fancy animation effects had to be made using lower level, C-based graphics APIs such as OpenGL, taking a lot of extra work. In 2006, Apple introduced CoreAnimation, a framework exposing a set of Objective-C APIs that made doing animations relatively simple. CoreAnimation was layered on top of OpenGL, and provided a simple, abstract interface for developers that still allowed animations to be hardware accelerated. This is a good example of the kind of two-tiered API design Willrich discussed. Advanced programmers, who want significantly more options, control, and performance, will still want to use OpenGL, especially for games. However, for the kinds of simple, elegant, and common animations in the user interface, such as the smooth swiping effect that a user sees on an iPhone, CoreAnimation provides, in a few lines of code, what would take hundreds of lines to accomplish in the OpenGL APIs.

Cocoa code libraries such as CoreAnimation encapsulate complex functionality in simple interfaces. With only a few simple calls to these interfaces, a Cocoa programmer can implement what would take programmers using less functionally rich libraries hundreds or thousands of lines more code. In essence, by building more and more functionality into the high-level Cocoa libraries, Apple removes the need for developers to write such functionality themselves, using the low-level interfaces.

There is a division of labor implied in this. Remember Fred Brooks and Harlan Mills' chief programmer or "super-programmer" model of organizing software work, which Brooks discussed in *The Mythical Man Month* (F. P. Brooks 1995). Instead of throwing armies of programmers at a project, and understanding the vast skill difference between the best programmers and average ones, in Brooks' model, the chief programmer was responsible for the architecture and design, the big picture, while the lesser skilled staff merely implemented the components, conceived of as a more routinized job. The chief programmer, the architect, is thus "freed" from

the drudgery of routine programming tasks that need to be repeated. Brad Cox, creator of Objective-C, advocated a different division of labor. Object-oriented programming, in Cox's view, encouraged code reuse. Routine code could, and should, be encapsulated in objects and reused, reducing the need for such work to be performed repeatedly in the future. This was key to how object-oriented programming would improve productivity. The other piece of the puzzle was that such routinized labor could be outsourced not to one's team of subordinates, but to the market. Cox thought that software objects should be bought and sold "off the shelf." While some NeXT developers did indeed sell objects for other programmers to buy, this never really caught on, partly because there was a wide range in the quality of "objectware" in the market, and partly because such objects sometimes came with restrictive licensing agreements (Garfinkel 1993a; Garfinkel 1993b). It was not clear to programmers whether they should trust the objects that they bought from others.

NeXT, and subsequently Apple, offered a slightly different division of labor. Like Cox's solution, and using the language he created, in NeXT's model, routine code would also be encapsulated in reusable objects. In a way, they would also be purchased in the market. But rather than acquire them piecemeal from heterogeneous sources, NeXT bundled entire libraries of objects along with the operating system and development environment, and earlier in its lifetime, with the computer hardware as well. These objects thus were built into the system that the user received, and were part and parcel of that system; the system itself was built using these objects. The issue of trust is much ameliorated, because the customer was already buying NeXT's system, and thus was already putting a degree of trust in NeXT by doing so. And largely, NeXT, and later Apple, kept this trust. Because all of the objects were designed to fit together in a coherent, consistent system, with built-in support by all of NeXT's developer tools, in particular, Interface Builder, and these objects were used by NeXT itself to make its own applications, developers could trust that these were world-class libraries, because the system itself was built on it.

The division of labor promised by Cocoa is thus, from the point of view of the Cocoa developer, a democratizing one, but not a deskilling one. “Routine” tasks are automated by the Cocoa libraries, leaving to developers the creative, intellective work of solving their users’ problems with their applications, work that cannot be automated. Says a former NeXT and Apple engineer, “There’s just this beauty in mechanizing these things that don’t need to be the programmer’s job.” (Julie Zelinski April 24, 2012) Routinized work is thus delegated to Apple, reducing the amount of manpower necessary for programming. The result is that small teams, or even a single programmer, can write highly complex applications, which would require masses of routinized, less-skilled programmers in traditional corporate organizations. Because of the empowering effects of Cocoa, all Cocoa programmers are freed to do creative, rather than routine, work—in essence, everyone can be a “super-programmer.” And because it allows individuals or small groups to be as productive as corporate armies, Cocoa also has the potential to make all programmers “indies.”

This is not to imply that the work of the Apple engineers in creating the Cocoa libraries themselves is actually routine and unskilled. In fact, it is highly skilled, perhaps more so than application programming using Cocoa. As the variance in quality of the third party NeXT objectware market showed, writing objects to be reused by others is not an easy task, as a programmer may not anticipate all the ways that object will be used. Indeed, as Steve Naroff indicated in my interview with him, Steve Jobs did not buy Brad Cox’s object libraries because he felt that Cox’s employees were not “world class” but the engineers Jobs had hired at NeXT were. “I also knew that the people working on it at Stepstone were not world-class; I knew that it would be really hard for world class companies like NeXT to buy into someone else’s ICPaks.” (Steve Naroff, Interview, December 22, 2012) As Willrich mentioned, a lot of care at Apple went into designing object interfaces that were as simple as possible for the largest number of users. This involved some guessing, but with “world class” engineers, more often than not, they guessed correctly. When they didn’t, this could be corrected in subsequent releases by changing the API, but this was not to be done lightly, as such changes could make earlier software incompatible.

Moreover, designing the libraries to be consistent throughout took a significant amount of vision, intentionality, coordination, and the control of key managers in NeXT's, and later Apple's, software team. This meant that the work of crafting an object library to be reused long into the future, and anticipating all the use cases correctly so that the library would not have to change significantly over time, takes a lot of knowledge and experience in good object-oriented design, which is anything but routine work. A bad job can result in software that does not match most users' needs, or does not adapt to changing circumstances, thus needing to be thrown out and rewritten in only a few years. However, the AppKit's core architecture has remained largely unchanged for over twenty years, and decisions made in the late 1980s and early 1990s have "stood the test of time." Blaine Garst, a retired NeXT and Apple engineer, noted:

> "It's fascinating… looking back on design decisions you made 20 years ago, 18 years ago, 15 years ago that are still in play…
>
> And you look at the technologies and the deployment environments and all these other environmental things that change around it, and you say, well, good core architecture can go a long ways. And that's effectively what we did. We took the existing AppKit, we reworked it… and made a good foundation for a lot of other things.
>
> …Those names were still there last time I looked. You know? Stuff I did twenty years ago. So the naming conventions are still in place…
>
> [Those decisions] have stood the test of some time.
>
> (Blaine Garst April 13, 2012)

Another way that Apple makes it possible to write less code is that, using Interface Builder, it is sometimes possible to accomplish functionality without writing code at all. Over the years, Interface Builder has increasingly built in more ways to construct user interface components by manipulating them graphically using the mouse, and toggling a few options. The same thing can almost always be done "programmatically," by writing code, but this can often take substantial work, compared to the mere minutes involved in dragging a user interface object from a palette and configuring its options and connections. This convenience involves certain trade-offs, however. Not everything that can be done with the user interface

object is exposed in Interface Builder. A programmer may want to customize the way a button draws itself, for example. For these cases, a programmer has to write code; writing the code not only gives the programmer a lot more power and control, it also helps the programmer understand in a lot more depth what the AppKit framework is doing behind the scenes.

More recently, for iOS development, Apple has introduced a feature in Interface Builder that allows a developer to graphically design not just a single screen of a user interface, but to graphically lay out how all the screens in an iOS application are connected, such as how tapping a button on one screen transitions, or "segues," to another screen. Drawing on a familiar design practice in the film industry, Apple calls this feature, "Storyboards." This technology allows the user interface designers in a company, who may not have any coding experience, to participate much more closely in the development of an application, as the laying out of a storyboard does not simply build a mockup that needs to be coded later, but is actually creating the connections of the program's code itself.

Step Christopher, an iOS instructor at the training company, Big Nerd Ranch, said that Storyboards exemplified Apple's approach to technology: convenience, at a price:

> Storyboards is a very Apple technology. It's very powerful and simple to use, but you have to use it in the Apple way… you have to forget the rest of the world, and live in an Apple way, and then you can do powerful things easily, but easy in the Apple sense. (Step Christopher, Interview, May 22, 2014)

Christopher's explanation of Storyboards reveals the flip side to the draconian control and limiting of options that Willrich said was justified in Apple in the name of simplicity. Convenience is obtained at a price. While most of Cocoa is less limiting than Storyboards, Storyboards represents this impulse at Apple taken to its logical conclusion. We can make designing an application so simple that a developer almost does not need to write any code at all. The trade-off is that a developer must do it in the specific way prescribed by Apple. Part of learning to be a Cocoa developer, then, is in part learning to make this trade-off at some level, and learning

not only to accept it, but to love it, as the developer will gain productivity and pleasure in return. It is a bit like following a well-groomed ski slope—such predetermined paths make skiing easier, but are simultaneously constraining.

There is another side to Storyboards that is also revealing. While it is possible to build a simple application using no code with Storyboards, to construct anything remotely interesting, that a customer might pay money for, requires a developer to add custom functionality. Thus, developers do still need to write some code. However, because Storyboards hide from the programmer a lot of what is happening, understanding exactly where a programmer needs to plug code takes a deep understanding of how Storyboards work on an abstract level. This understanding is difficult to gain because so much of Storyboards is black boxed. In this sense, for a programmer, it may actually be easier to implement user interface screen transitions for an iOS program in the more traditional way, in code. By writing code, the programmer can see exactly where the transition occurs, and has direct control over the process. With Storyboards, Apple's code has control over the process, but provides certain hooks, known as "callbacks," that call a programmer's own code when certain events occur. iOS trainers such as Christopher recommend that novices learn how to construct an interface using code, not using Storyboards, because only by writing this code can the novice gain the mental model for how iOS programs are structured.

In other programming environments, programmers typically learn first to write a program that runs from start to finish, with loops and branches and subroutine calls. However, Cocoa programs start running in code owned by objects in the Cocoa libraries, and only call out to a developer's own custom code at specific times, in response to specific events. Only when built-in Cocoa objects need to "delegate" custom functionality will they call a programmer's own code, again, using specific interfaces known as "callbacks." A Cocoa developer thus needs to read Apple's documentation extensively to learn what callbacks to write code for and when, even to create simple programs. This involves learning extensively a mental model for the lifecycle that a Cocoa program goes through, what events various Cocoa objects

respond to, and when it is appropriate to customize those events by implementing a callback.

The result is that, by automating a lot of tasks for the programmer, Cocoa takes care of routine tasks and makes a lot of things more convenient, and thus, easier, for a programmer familiar with the environment. However, this benefit only comes after a programmer has spent considerable time learning how Cocoa works on a higher, more abstract level than he or she can see directly in code, because much of the code is hidden in objects written by Apple. Far from deskilling the work of the programmer, Cocoa actually requires the programmer to acquire significantly more knowledge and understanding of how Cocoa object libraries work in Apple's system, because it requires the programmer to work in a specific way in order to match the way Apple has designed its system. Although a Cocoa programmer writes less code, she must understand more. As we will see in the next section, this also means that a Cocoa programmer must trust that Apple's libraries will do what she wants better than code she could write herself.

Learning as a prerequisite for productivity

Aaron Hillegass, a former instructor at NeXT and Apple, founded his own company, the Big Nerd Ranch, in 2001, to train programmers how to write Cocoa, and now, iOS, applications. For Hillegass, a developer's willingness to learn Cocoa's APIs is directly related to how much Cocoa will be able to empower the developer. Like other former NeXT programmers, he asserts that it is true that "there is an order of magnitude higher productivity for a NeXT programmer," particularly because "it was big on Wall Street," as he himself, knew, having written NeXT programs for an investment bank. (Aaron Hillegass Interview, July 7, 2011) This promised productivity is partly a result of achieving more functionality with less code, because more functionality (and complexity) is handled by code in the Cocoa libraries themselves. This can only be achieved, however, if a programmer fully follows and embraces the design patterns the Cocoa libraries require, something that comes with significant learning. This changes the task from one of writing to one of learning.

> There's this trade-off between learning and coding. You learn more, but you code less. It's choosing to learn, rather than write…
>
> You have to choose to spend more time learning and experimenting and reading the documentation, and in return, you're writing fewer lines of code.
>
> It's not a platform for dummies… that's really what Cocoa is… it's a framework for smart people.
>
> (Aaron Hillegass Interview, July 7, 2011)

What does Hillegass mean, "Cocoa is a framework for smart people?" By "smart," Hillegass means a combination of humility and a willingness to learn, the opposite of the "Not Invented Here" syndrome that many programmers have, who are suspicious of other peoples' solutions and prefer to rewrite everything themselves. For Hillegass, a "smart" programmer is the one who is willing to trade off having full control over her code (by writing it herself), versus learning how to utilize what objects Cocoa's built-in libraries provide, thus reusing code Apple has already written. Cocoa requires more learning because it is more abstract, incorporating higher-level concepts and design patterns that help tame complexity. With these abstractions, accomplishing a task could be expressed in simpler, more elegant form. This is "smart" because the investment in learning now will greatly economize on the programmer's time in the long run, because she will be able to more easily leverage code written by Apple. As an example, Hillegass compared the way scrolling is implemented in Cocoa versus the way it was done on the original Macintosh operating system. This is done using a user interface object called a "ScrollView."

> That's always the trade-off with these object-oriented systems, is that you get the ScrollView for free, and the ScrollView will do anything that you need it to do, but you really have to understand it. Whereas back in the… [original] Mac… you had to write incredible amounts of code to make a ScrollView work. But since you had written all the code, you were in total control, so you didn't have to have a deep understanding of ScrollView because there weren't that many smarts there. (Aaron Hillegass Interview, July 7, 2011)

The return for getting a "deep understanding" of the ScrollView object, is that developers get scrolling "for free." The implication is that the more "smarts" that are

built into object-oriented code libraries, the more skilled and knowledgeable programmers need to be in order to fully utilize their functions. This runs counter to the traditional account of automation: that the smarter a piece of technology is, the more it deskills its user.

To illustrate the productivity difference this could make, Hillegass provided the example of Lighthouse Design's "Diagram!", a graphing application on NeXTSTEP. A competing company replicated the innovative user interface of Diagram!, but deployed their app on Windows, calling their competing product, Visio. Visio for Windows took twenty programmers to create an equivalent to Diagram! According to Hillegass, Diagram! was written by a single developer at Lighthouse, Kevin Steele, who, years later, rewrote the program from scratch for Mac OS X, now called OmniGraffle and sold by OmniGroup. Hillegass explains that, although the two applications look similar on the surface, they were not built in the same way.

> They are not the same sort of program. The guy who wrote Diagram! had a very deep knowledge of what was going on with the operating system… of everything that was happening in the standard classes of OpenStep,[26] and he leveraged those. So he wrote significantly less code, but in his mind he had a much deeper context that he had to understand. So the programmers [who] worked on Visio, wrote a lot more lines of code, [and it] took a lot more of them, but each one of them didn't have to understand how the operating system worked. Because they were writing the whole thing. (Aaron Hillegass Interview, July 7, 2011)

Hillegass's definition of "smart" is not universal among programmers. For others, a "smart" programmer might be one who writes everything from scratch, simply because one has the expertise and arcane knowledge—it is a proof of one's skill. Among Cocoa programmers, however, this is seen as a vice, not a virtue. It is

---

[26] OpenStep was a rebranding of the NeXTSTEP frameworks after they were ported to run on top of other Unix-based operating systems, as well as Windows, in the 1990s.

considered a waste of a programmer's time to "reinvent the wheel," spending time on problems that Apple has already solved, writing code Apple has already written, and in way that works consistently with the rest of the system. According to Hillegass, Cocoa programmers are smart in part because they are humble enough to be willing to learn, rather than assume that they can, and have to, do everything themselves. "Reinventing the wheel" is an aspect of "Not Invented Here" syndrome, a disdain of outside technologies that one did not personally invent, which Hillegass considers an indicator of hubris and thus stupidity, not intelligence.[27] By not falling prey to this hubris, Cocoa programmers are smart because they economize on their own time, and leverage the work of others. Smart developers spend time on what's important, solving the problems unique to their own application, which, after all, is what the user is paying for.

Hillegass thus contrasts these two social models of software production, implicitly connecting the design of software toolkits and how much "intelligence" they expect from their users, with the social organization of programmers. This

---

[27] Ironically, as a company that pushes its own proprietary standards, Apple itself has been frequently accused of "Not Invented Here" syndrome, especially prior to the NeXT acquisition. NeXTSTEP was based on Unix, an industry standard, and used the open-source GNU C compiler GCC, and use of open standards increased dramatically in the initial years of Steve Jobs' return. However, despite a nominal embrace of open-source, Apple maintains is own forks of these projects under its control, and in the years since, has gone back in the direction of inventing its own technologies rather than embracing outside ones. Apple has since moved off of GCC to a different open-source compiler, LLVM, whose less stringent license allows Apple to more easily modify it for its needs. Apple actively supported Java in the early 2000s, even creating a Java-bridge to allow developers to write Cocoa applications using it, but ran into conflicts with Sun's control over the language. In 2014, Apple created its own language, Swift (though using open-source tools like LLVM) to serve as a replacement for Objective-C.

social model matches rather neatly with that of indie developers like Wil Shipley, who argue that the future of the software industry lies with millions of small developer-entrepreneurs, not the large corporations with their armies of programmer-laborers. Indeed, Thomas Haigh argues that much of the discourse of "software engineering" arose from academic computer scientists like Edsgar Dijkstra, who felt that programming should be highly mathematical and thus reserved for the intelligent, and resented the corporations that seemed to hire hordes of unskilled programmers (Haigh 2010). In Hillegass's social model, Cocoa programmers, like Brooks' "super-programmer," are programmers of high skill, the one excellent programmer among the ten mediocre ones cited in early IBM studies during the software crisis. "An early study at IBM suggested that exceptional programmers were ten times more efficient than their merely average colleagues. The alleged 10:1 performance ratio quickly became firmly embedded in the cultural wisdom of the industry." (Ensmenger and Aspray 2002, 6) The 10:1 performance ratio understood in the 1960s as the ratio of average to elite programmers is the same 10:1 productivity improvement that NeXT claimed in the 1990s that its object-oriented environment could produce. It could be said that NeXT's alternative solution to the software crisis, then, is to remove the need for average programmers completely, by automating all tasks that their elite team leaders would have delegated to them. One way to interpret this is that, rather than being a deskilling technology, by automating routine work, Cocoa puts average programmers out of a job. Given the rapid boom in the population of Cocoa programmers writing iPhone apps, however, this is clearly not the case. Average programmers can write Cocoa apps, though not necessarily well. Rather, another way of looking at this, is that Cocoa requires even average programmers to improve their knowledge and skill in order to be competent at writing Cocoa programs. By requiring a higher level of knowledge, Cocoa enskills programmers, rather than deskills them.

Trust in Apple

Hillegass also notes, however, that being the "smart programmer" by delegating as much work to built-in code libraries as possible, requires a good deal of

trust in the provider of those libraries, which are often black-boxed and cannot be modified by the developer on closed-source, proprietary platforms. One of the benefits to "reinventing the wheel" is that the programmer has complete control over his code; he does not have to learn how it works because he wrote it himself, and can trust that it does exactly what he needs. As a result, novice Cocoa developers have to undergo a period of learning to trust that Apple's libraries (and thus, its engineers) have indeed provided the optimal solution to a problem. Of course, Apple's libraries do not provide the optimal solution all of the time, but it takes a fully knowledgeable expert to discern the exceptions to the rule.

> It involves a trust on Apple's part, which they're not always good about actually fulfilling. And the trust is, we are going to make stuff that you can leverage, and you have to be smart, and really understand it. (Aaron Hillegass, Interview, July 7, 2011)

Being "smart," then, is not simply a matter of intelligence, but also a matter of normative virtues: humility, and trust in Apple. The message is simple: sacrifice your control over your code by delegating much of your work to Apple's object libraries, trusting that Apple has hired the best and brightest software engineers to come up with the right solution. Learn how Apple's code and system works, and learn to leverage it. By letting Apple do most of the work, your application becomes faster, more stable across future operating system updates, and you spend less time on writing routine code to solve problems Apple has already solved for you. If you make this sacrifice, you will be rewarded with a ten-fold increase in productivity, allowing you to become a super-programmer who can compete against legions of lesser skilled corporate coders.

What happens when this trust breaks down? Hillegass, despite having once worked for Apple, and likely because of it, feels that he has the expertise to criticize when Apple makes mistakes in its designs. He gives the example of a feature called "Cocoa Bindings," introduced on Mac OS X in the mid-2000s, as such a misstep. Cocoa Bindings provided a way to synchronize the communications between objects automatically by configuring options in Interface Builder rather than writing such "glue code" by hand.

> "I think [Cocoa] Bindings are absolutely mis-designed because they hide too much away. And there's very little of it's public about how things are working and if you've ever tried to create a view that has Bindings, it's really, really hard. And they did all sorts of things to make it harder… it's a pain in the ass to do it right.
>
> So, when Apple screws up, it's often because they thought, "well we can give this crappy solution to programmers who are dumb," is what I really believe. They wanted to make it more accessible. And in making it more accessible they made it bad.
>
> …They said people found [an older technology which automated this same process] too confusing, we're going to go for a lower-grade programmer.
>
> But it was just bad—it's not good. And we have proof that it's bad because they didn't do it on the [i]Phone. (Aaron Hillegass Interview, July 7, 2011)

According to Hillegass, when Apple goes wrong is when it does not respect developers' intelligence and tries to make a programming task too easy, thereby deskilling it. Although Cocoa Bindings made it theoretically possible for a simple Cocoa application to be constructed without writing a single line of code, but rather "wired up" graphically in Interface Builder, by going too far in automating programming, it made the resulting app an opaque black-box. This, counter-intuitively, made programming more difficult for everyone, because even for skilled programmers, the program was harder to predict, debug, and fix. Although removing the need to code is seen as a good thing by Cocoa developers in general, doing so at the expense of understanding what the underlying system is doing is not. A Cocoa developer need not necessarily see the code that makes something work, but she does need to understand it in order to properly use it. For this reason, the introduction of the similar feature, Storyboards, has been rather poorly received by the instructors at the Big Nerd Ranch, who feel that it hides away too much and is not conducive to learning how iOS apps work. Hillegass's iOS book, which serves as his course's textbook, teaches programming iOS apps the traditional way through writing code, and in its fourth edition, only includes a short, inessential chapter on Storyboards towards the end (Keur, Hillegass, and Conway 2014).

Design Patterns and the Learning Curve

Hillegass is not the only Cocoa developer, or even Cocoa instructor, to argue that Cocoa requires significant learning. Julie Zelinski, a former NeXT and Apple employee who has taught Cocoa programming at Stanford, notes that the Cocoa frameworks, not the Objective-C language, have a high learning curve that a student must get over in order to become proficient:

> [The] Objective-C [language] takes like a week to learn. What takes you months to learn is the AppKit, the Foundation… the whole [Cocoa] toolkit. […] There's a huge learning curve and you have to get people to invest in the learning curve to get to the other side. (Julie Zelinski, Interview, April 24, 2012)

Moreover, developers will not initially be able to trust that Cocoa's way of doing things works better until they have gotten over this hump. Thus, not all programmers who try to learn Cocoa make it to the other side; until they do, it is not obvious to them how doing things the Cocoa way will benefit them. In order to explain how this is so, it is necessary to understand an important aspect of both the way Cocoa is designed and its implications for Cocoa programming practice.

One of the primary obstacles for students' learning is that Cocoa has been designed around a consistent set of "design patterns," recurring pattern-solutions for solving common programming problems in object-oriented environments. These run throughout the Cocoa frameworks, and extend far beyond any particular function or API. The benefit of these patterns can only be understood holistically, making Cocoa difficult to approach in piecemeal fashion:

> Just in the sheer depth of all the frameworks that are available to you, it's kind of like you're moving through this fog of war and you're seeing things appearing as you're discovering new things. But, because you didn't know about them before… you are… in the dark as to… how to approach a solution or… understand… the bigger picture. Because the bigger picture is so huge. (Chris Livdahl March 28, 2012)

For those who do manage to make it to the other side of the learning curve, however, a whole new world awaits, as if the developer has reached a moment of

understanding. Craig Hockenberry, the developer of the popular Twitter client, Twitterific, likened this to acquiring a new "mindset."

> There's a pretty steep learning curve for people coming from Windows and Java. I did Windows programming for a while, we still have a product that we sell that's for Windows and Photoshop, and *it's a totally different mindset, it's a totally different set of tools*, different—and I don't want to say one is better than the other, but *if you're going to develop iPhone applications, you gotta get the right mind-set*. That's a hard thing to do… there's some very [sophisticated] design patterns in the Cocoa framework. And part of the hard part of the learning curve is figuring out what those design patterns are. [emphasis mine]
>
> (Craig Hockenberry, Interview, January 7, 2009)

This discussion of "mindset" seems to subtly invoke Thomas Kuhn's notion of "paradigm shift." The term "paradigm" has moved from philosophy and history of science into common usage among computer scientists and practitioners, and indeed is used to describe different familial classifications of programming languages, including object-oriented programming. It is not clear how much of Kuhn's theory has moved along with the term. In common programmer discussions of the differences between procedural and object-oriented programming, or imperative and functional programming, the sense that one's "worldview" or "mindset" needs to be shifted when moving from one of these "paradigms" to another is implied. However, to what extent are these different language families seen as "incommensurable" with each other, a key entailment of Kuhn's theory? This is not clear, as elements of object-oriented programming, such as modularity, can be found in procedural languages that follow a structured programming methodology. Smalltalk, one of the earliest object-oriented languages, took significant inspiration from Lisp, a functional language, and indeed, more and more object-oriented languages today have been incorporating functional features. The fact that languages such as C++ and Objective-C, which are "hybrid" languages that straddle or incorporate multiple programming "paradigms" indicates that the component concepts of these languages are not incommensurable at all, but travel rather easily in academic computer science. Nevertheless, purist proponents of one or another of these approaches argue that such

hybridity dilutes and in fact destroys what makes working in these "paradigms" both pleasurable and productive. C++ programmers, they say, are not truly learning object-oriented programming, but a reduced, impoverished version of this. Object-oriented programming in C++, as opposed to the more pure Smalltalk, has been reduced to a kind of "pidgin" that lacks the full richness and meaning of its original. Similarly, despite the incorporation of functional features into newer languages such as Apple's Swift, some argue that these are not "true" functional languages like Haskell or Clojure. Clearly, the language of "paradigm" is fraught, and here I treat it as an actor's category which does discursive work for the actors, rather than an analytical category.

Nevertheless, Hockenberry notes that the key to understanding and learning the Cocoa frameworks lies in learning the design patterns built into them. What are design patterns? The canonical text on design patterns in computer science is Erich Gamma, et. al.'s *Design Patterns: Elements of Reusable Object-Oriented Software*, otherwise known as the "Gang of Four book for its four co-authors. The book is a catalog of "simple and elegant solutions to specific problems in object-oriented software design. Design patterns capture solutions that have developed over time… They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software. Design patterns capture these solutions in a succinct and easily applied form." (Gamma et al. 1995, 9) The intention of design patterns are to "make your designs more flexible, modular, reusable, and understandable—which is why you're interested in object-oriented technology in the first place, right?" (Gamma et al. 1995, 9) Making a catalog of known solutions to common problems is necessary because simply using an object-oriented language does not automatically generate the benefits promised by the methodology; one must learn how to think and design in object-oriented ways. The authors explain:

> Designing object-oriented software is hard, and designing *reusable* [emphasis in original] object-oriented software is even harder… Your design should be specific to the problem at hand but also general enough to address future problems and requirements… Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get "right" the first time…

> …expert designers… reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently, you'll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience…
>
> The purpose of this book is to record experience in designing object-oriented software as design patterns. Each design pattern systematically names, explains, and evaluates an important and recurring design in object-oriented systems. Our goal is the capture design experience in a form that people can use effectively. To this end we have documented some of the most important design patterns and present them as a catalog. […] Once you know a pattern, a lot of design decisions follow automatically.
>
> Design patterns make it easier to reuse successful designs and architectures. Expressing proven techniques as design patterns makes them more accessible to developers of new systems. Design patterns help you choose design alternatives that make a system reuseable and avoid alternatives that compromise reusability… Put simply, design patterns help a designer get a design "right" faster. (Gamma et al. 1995, 14–15)

As Gamma et. al. state, design patterns help make object-oriented programs more modular, flexible, and reusable—qualities that help make software more maintainable, and help improve a programmer's long-term productivity. Reusability in particular is difficult to achieve. Object-oriented languages by themselves are no silver bullet, and do not mechanically produce reusable code. Rather, only thinking conceptually about object-oriented design in a systematic way will lead to architectures that are flexible, general, and reusable, and thus maintainable. In other words, design patterns are one of a number of tools and techniques to address software engineering concerns. The accumulated knowledge of expert object-oriented designers has been, until now, relatively tacit, but in creating a catalog of design patterns, Gamma et. al. are putting it in an explicit form from which future programmers can draw.

The patterns described by Gamma et. al. have been widely adopted in object-oriented design, including at NeXT, although the names used to describe them may

differ from those used by the Gang of Four. NeXT made extensive, consistent, and recurring use of design patterns throughout what became the Cocoa frameworks. Today, these patterns remain deeply embedded in all of Apple's frameworks written in Objective-C, including the Cocoa Touch frameworks used to develop apps for iOS. (Apple Inc. 2013b) These patterns structure how programmers must think in order to write programs using Cocoa—both constraining them and channeling their problem solving into established directions. This has two consequences for learning. First, the deep integration of these abstract patterns in the frameworks means that learning Cocoa is not simply a matter of learning the Objective-C language and how to call the proper APIs to do what one needs. It also means learning abstract concepts that go by unfamiliar names: "delegation," "model-view-controller," "target-action," "key-value-coding/observing," "responder chain." These patterns are not generally taught in university programming courses, and sometimes the names they go by in Cocoa may differ from other programming environments or from their names in Gamma et. al. Cocoa developers recognize this as probably the predominant reason for Cocoa's high learning curve.

> You actually go into Cocoa proper and it's like, OK, you need to learn KVC [key-value coding], you need to learn KVO [key-value observing], you need to learn MVC [model-view-controller], you need to learn the patterns, you need to learn the Cocoa style guide for how code should be written. And writing an application in Cocoa is just like, having been a journalism major [myself], like writing an article for submission to the *New York Times*. There's a very specific style that you must follow to make sure you are compliant with the way it is done. (Mike Lee, Interview, July 15, 2008**)**

However, the second consequence is that once this initial investment has been made, subsequent learning becomes significantly easier. The recurrence of these patterns throughout the Cocoa frameworks creates a consistency that allows an experienced Cocoa developer to expect that an unfamiliar framework will work in the same manner as those she is already used to. This is the basis of the consistency that Cocoa developers praise.

This tracts with Edain's experience of learning Cocoa as a conversion: a period of confusion and struggle, following by a gradual awakening upon grasping the entire whole, accompanied by appreciation of the elegance of Cocoa's consistent design. This experience of holistic consistency is one of the key sources of the affective pleasure Cocoa developers' experience. Yet, this experience can only be attained after the high investment has been made in learning. This cognitive investment thus creates a barrier for outsiders and newcomers—without sufficient motivation or commitment, it is not obvious to an outsider why the experience of Cocoa development is supposed to be "better." While NeXT and the Macintosh were minority platforms, most programmers had no incentive to make this investment, but the lucrative allure of iOS mobile apps has pushed many newcomers over this hump.

Design patterns are thus not only important in improving programmer productivity and software maintainability, but are part and parcel of the techno-cultural frame of Cocoa development, the understanding of which marks insider versus outsider, expert versus novice, high versus low inclusion. Moreover, using the correct patterns, the ones preferred by Cocoa developers, is a vital aspect of learning normative "best practices."

In the next two sections, I will explain two such patterns, why Cocoa developers encourage their use, and why these make up elements of best practice in the community.

Design Patterns—Model-View-Controller

One major design pattern used in Cocoa applications is Model-View-Controller (MVC). Model-View-Controller is an architectural pattern for organizing the design of applications with graphical user interfaces that originated with Smalltalk at Xerox PARC. It is not only used throughout Cocoa, but has become prevalent in most object-oriented graphical frameworks, including many web frameworks. The pattern is a way of enforcing the principle of "separation of concerns" between different classes of objects in an application, to prevent objects from having too many dependencies on other objects, thus preserving both their

flexibility and their reusability. Both the AppKit and UIKit frameworks make such extensive use of this pattern that it is impossible to discuss specific object classes and APIs without implicit reference to it, as their names frequently bear their categorization into “view” or “controller” roles.

For example, desktop applications with graphical user interfaces contain windows, which contain sub-elements that are drawn on the screen. These are called “views.” Anything that is displayed in a graphical user interface is in the form of a view. Some different types of views can include table views, which display data in tabular form, outline views, which display data in outline form, or text views that allow the display and editing of text. Some views can be used to trigger actions, such as buttons or sliders. These views are called “controls.”

Applications also contain data that the user is interested in displaying, manipulating, or modifying. This data often models some quantity in the real world, and can be represented by characters, integers, real numbers, strings of text, or combinations or collections of these primitive elements. If the data is sufficiently complex it may be in the form of a database. This data is often stored on disk or retrieved from the network. Such data make up what is known as the “model” of the application.

Views often display data coming from the model. For example, a table view might display the contents of a database, and a text view might display the contents of a text file. The user may wish to alter this data in the model, by manipulating the view—say, typing into a text field to change some value. A naïve way to write a program would be to store the model data directly inside the view itself. However, this tight coupling creates problems if later on, either the model or the view needs to change, or, for example, two different views need to display the same model data—say, in outline form and in browser form. To reduce the tight coupling and dependencies that such view and model objects would have to have with each other, an object is introduced in between them, that handles the task of synchronizing the view with the model. This is known as the “controller.” Having a controller allows

both the model objects and the view objects to be reusable—in simple iOS apps, for instance, a developer may not even need to define her own custom views, but rather just use view objects provided by the UIKit framework. The controller is often the only custom object that cannot be easily reused, as it must contain the "glue code" that holds the application together.

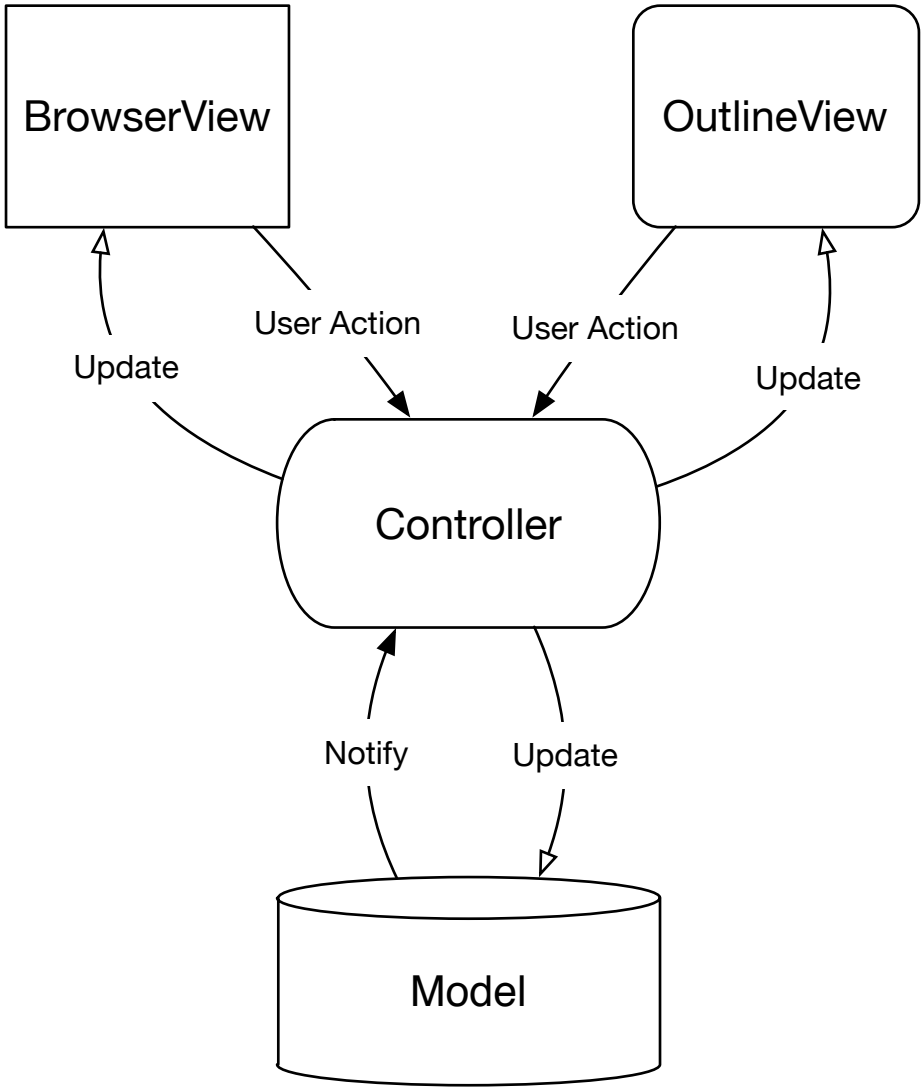


Figure 2: Model-View-Controller

The UIKit framework contains a kind of object known as a UIViewController, which is a controller for handling synchronization between a UIView object and whatever model data it represents on screen. It also manages screen updates and what

happens when it needs to transition to a different view. View controller objects are one of the primary objects that developers write code for in iOS applications.

Like other object-oriented design patterns, MVC is intended to increase maintainability and flexibility by keeping objects loosely coupled, and by maintaining clear divisions of labor, or "separations of concern" between different objects. Separating concerns among different objects increases usability because it allows the programmer to easily swap out one object for another when things change while keeping other objects the same. Edain explains how novices typically do not understand the reason to add this extra separation:

> So, if you don't think that model view controller is a good pattern, which a lot of people [in the online forum, Stack Overflow] [don't]… you're going to find a lot of people [are] like "hey, I do all of my UI in code. And like I think that's easier, I get more control." Well… what happens is… most of the people who do that confuse their view and controller objects and end up with a lot less maintainable code. That is a lot less… flexible under change. Because they didn't take the time to separate. (Hasan Edain, Interview, March 12, 2012)

What Edain hints at here is that although Cocoa requires the use of MVC due to the design of its object classes, it is still possible for developers to subvert this pattern, either deliberately, or through careless design. Creating clear MVC separation requires some degree of discipline and commitment that involves trusting that the pattern will yield long-term benefits despite extra short-term work, something novices tend to ignore. Edain explains that Apple has tried to encourage the practice of separating these concerns:

> Now, there are people who do in fact separate very well… but in general… when the whole system was being thought of, they [Apple] sat there and said, OK, how can we encourage people to do this? …We have this database abstraction layer, which is going to be our model. And we have this visual representation, which is going to be our view. And if you stick in those worlds, then your model and your rear end controller are going to be really well separated.
>
> (Hasan Edain, Interview, March 12, 2012)

Edain notes that Apple can go a long way towards encouraging the use of the pattern by building it into its system, requiring everything to conform to it and understand it. Programmers can still get away with doing the bare minimum, but at the very least, they must have this minimum separation if they are writing a Cocoa program. Novices may not understand the reason for the enforcement of such separation. For experienced web developers, this could a difficult thing to learn, because the most widely used web framework, PHP, did not have MVC separation:

> …I think within Cocoa there's even more of an emphasis on MVC. And it's a lot different than what you might see out of [a] PHP framework. So I've had to get used to a lot of that.
>
> (Chris Livdahl, Interview, March 28, 2012)

Of course, the pervasiveness of MVC in Cocoa has pedagogical consequences. A programmer cannot understand Cocoa without learning MVC. Thus, teaching Cocoa requires teaching MVC. Julie Zelinski, a Stanford lecturer who has taught the Cocoa class there, speaks to this:

> You can only operate in the Cocoa space with knowing some of these patterns, right? Learning MVC, it's just going to happen… you know you have to kind of get that… I don't think [the Stanford Cocoa course] stands up and says, "Now we're going to talk about [INAUDIBLE] design pattern," but it is going to say "this is what you're going to be seeing here and this is how this pattern operates and this is what you need to know about where your interactions are with this." (Julie Zelinski, Interview, April 24, 2012)

Similarly, Hal Mueller, who teaches an introductory Cocoa class at University of Washington's Professional Continuing Education program, emphasizes MVC separation in his course.

> Hal: I'm trying to tie concepts together more, so I really push the MVC separation. I talk about how you would move a project from the Mac to the [i]Phone. …So the graphics exercise we're going to do is more involved and again is going to reinforce MVC separation, so this project, I'm going to give them a skeleton project which has, it generates an array of random data that they have to plot and so now you've got the tableView of the data, you've got the graphical view of the data—

> Hansen: OK, right and so that is a very good example of why you'd need MVC separation, because you have two different views on the same data. So it's a very good way of getting across the need for this kind of thing.
>
> Hal: Yes.
>
> Hansen: Right. Whereas otherwise, they would go, why should I bother to set up this whole structure, I should just be able to access this data directly.
>
> Hal: Mmmhmm
>
> (Hal Mueller Interview, February 15, 2012)

My exchange with Hal here indicates that it can be temptingly easy to cut corners and create direct connections between objects in a program, even one written in Cocoa. My response to Hal implied that I had done this very thing for expediency. So teaching MVC in Cocoa had two purposes. Firstly, it was necessary simply to use Cocoa to know MVC, so teaching MVC is simply fundamental technical knowledge one needs to master. Secondly, however, MVC is also a normative practice that will improve a programmer's software the more she uses it in her designs, going above and beyond the minimum use of it enforced by Cocoa. By building MVC so heavily into Cocoa's design and making it a mandatory concept to learn, Apple conflates the "is" and the "ought" of learning MVC—conceptual knowledge and normative practice are learned simultaneously.

In summary, Model-View-Controller is built into the Cocoa frameworks and Apple and the community heavily encourage developers to maintain clean separations of concern between model, controller, and view objects when they write their applications. Learning this practice is critical to learning Cocoa, but following it in a disciplined and thoughtful manner still takes some effort. Conceptual knowledge and normative practice are thus conveyed together in the learning of this pattern. This duality is exhibited even more strongly by another design pattern heavily used in Cocoa and promoted by Apple and the community: "delegation." Even more so than MVC, delegation takes effort to learn, is necessary to fully grasp the way Cocoa works, and is a pattern Apple promotes over alternatives. This

difficulty is exhibited by the fact that delegation represents a very different solution to the common problem of customizing code provided by libraries such as Apple's Cocoa than the one most commonly employed in other object-oriented environments, as we will see in the next section. Getting over resistance to this new way of thinking requires even more trust than learning MVC.

Design Patterns—Delegation vs Subclassing

Mark, an instructor at Big Nerd Ranch, explained on the first day of his class, "delegation is one of the huge things in object-oriented programming that Apple does a lot." In my own experience, and in the experience of many Cocoa developers I interviewed, delegation was one of the concepts I found most difficult to understand when learning Cocoa. Delegation is second only to Model-View-Controller as a pattern central to the design of the Cocoa frameworks, and thus a critical concept to understand. Unlike MVC, which has become more prevalent recently in web frameworks, delegation is not commonly familiar to programmers with experience in other object-oriented languages, which means that learners do not have analogous concepts to mentally translate. Nevertheless, its pervasiveness within Cocoa makes it unavoidable. Mark told his class, "Even if you don't completely understand the idea, you will be using it." (Field notes, July 10, 2011)

In Cocoa, delegation is frequently used to solve problems that in other object-oriented environments are commonly solved using a different solution, subclassing, which is also available in Objective-C but whose overuse is discouraged by Apple and the Cocoa community. The use of delegation instead of subclassing is a central practice learners of Cocoa find difficult and must change their way of thinking to embrace. In learning to employ delegation as a design pattern, Cocoa programmers unlearn the familiar exemplar solutions of their original technical subculture and convert to a new normative order, in which more flexible solutions, which include delegation, should be used when possible.

To understand the techno-cultural difference between object-oriented programmers who favor delegation versus subclassing as a common design solution,

we need to understand what these concepts mean. I will explain subclassing first, as it is a standard technique available to the majority of object-oriented languages. Object-oriented programming languages are supposed to promote code reuse. One of the ways they do this is by providing mechanisms for programmers to extend and customize the functionality of code modules provided by others in libraries or frameworks, for which they do not necessarily have the source code. The standard way to do this in most object-oriented languages is through "subclassing," also known as "inheritance." Code modules in object-oriented languages are known as "objects." Objects have a type, known as its "class." The class definition defines what data or attributes (known as *instance variables*) these objects have, and what operations (known as *methods*) these objects can perform, or in other words, what messages they respond to.[28] Objects are particular "instances" of the class. For example, the objects "Fido" and "Spot" are both instances of the class "Dog." The class definition for Dog says that Dogs have certain instance variables that other objects may not have: color, breed, name, etc., and can perform actions only Dogs can do, such as "Bark," "Fetch," "RollOver," "PlayDead," etc.

Most object-oriented languages allow programmers to define new classes as subtypes, or *subclasses*, of another class, including those provided by the language or operating system's built-in library. Subclasses are usually more specific than their superclass. For example, a class "Cat" can have the subclasses "Lion," "Tiger," "Leopard," and "Cheetah." The subclass is said to *inherit* all of the instance variables and methods defined in its *superclass* (sometimes called its *parent* class). Subclasses

---

[28] Determining whether an object can respond to a particular message, and selecting and binding the proper code to run, can be determined either "statically," when the program is compiled, or "dynamically," when the user is running it live. "Static binding" creates faster code while "dynamic dispatch" allows for more flexibility; for instance, it might be desirable to switch out a completely different method implementation at runtime in a dynamic system, which would be impossible if it was statically compiled.

can define additional variables or methods beyond what its superclass defined, extending it. For example, while "Cat" may not have an instance variable defining a "Pattern," Tiger may have a "Pattern" instance variable set to "Stripes" and Leopard and Cheetah's Pattern may be "Spots." Subclasses can also *override* the definitions of methods defined in their superclasses, changing or customizing their behavior. For example, Cat may respond to a message telling it to "Talk" by playing a "Meow" sound. Lion and Tiger may override this behavior by playing a "Roar" sound instead. In the UIKit, UIView has many subclasses: UITableView, UITextView, etc. UIControl, which is itself a subclass of UIView, has its own subclasses: UIButton, UISlider, etc.

Subclassing is one of the standard techniques taught to undergraduates learning object-oriented programming in college, and in such languages as Java and C++, most programming tasks begin by making a custom subclass of something else.

However, as a design technique, subclassing can have disadvantages. The relationships between classes are defined by the tree-like structure of inheritance, and classes can only customize the behavior of classes that they inherit from, higher up the tree. Some languages, such as C++, support *multiple inheritance*, allowing a subclass to have more than one superclass; this solution, however, often results in certain complications, because if both superclasses define the same instance variable or method, which one wins? Objective-C gets around this using a feature known as a *protocol*. Subclasses in Objective-C can inherit from only one superclass but can *conform* to one or more *protocols*. (Java has this same concept but calls it an "*interface.*") A protocol is like an abstract class definition except that it cannot define any instance variables (which implies memory storage), only methods. All classes that conform to the protocol must implement the methods defined in the protocol. For example, many different kinds of objects that are not directly related to each other may need a way to encode their data in a way that can be written to a file on disk. In Cocoa, this is handled by the NSCoding protocol, part of the Foundation framework. Objective-C also contains a feature uncommon in other object-oriented languages: a developer can add methods to existing classes without having to create

a subclass. This is known as a *category* or a *class extension*. This is done without the developer having any access to the original class's source code, thus allowing developers to add methods to Apple's own classes. The advantage of this is that the extension is now available to all other developers on the same team automatically without them having to use a custom subclass. Any other subclasses written by other vendors will also get these extensions automatically. This gets around the restrictions of the inheritance hierarchy.

Subclassing, protocols, and class extensions are all techniques that are facilitated by direct support in programming languages. That is, Objective-C supports these practices natively by providing direct syntax for them. Similarly, Java provides direct syntax for subclassing and its version of protocols (Java's interfaces). Protocols and class extensions provide two different ways of designing a flexible object-oriented system without constraining it with the rigidity of an inheritance hierarchy. The delegation design pattern is another way to customize behavior outside of inheritance, which is not directly supported by the syntax of any particular object-oriented language, but is a higher level concept that exists independently of them.

Mark, the instructor at the Big Nerd Ranch, described delegation by saying, "I'm an object, I do a thing a lot, but I ask a helper to do this [instead of doing it myself]." In delegation, one object assigns certain tasks to another object, its "helper" or "delegate." This is often used to update some state in one or more objects in an application, such as whether a text field is editable, or whether a table should refresh its data. This helper object can be of any arbitrary class, not related to the delegating object's class in any way.[29] The delegate can also take on the tasks of not just one,

---

[29] In practice, this is done through the use of Objective-C's protocols feature. The delegating object merely needs to declare a protocol that the delegate must conform to. If the delegate implements the methods declared in the protocol, it can receive the proper messages from the delegating object and thus handle the tasks that

but multiple objects, allowing a single object to customize the behavior of any number of others. Apple's official documentation explains delegation thus:

> Delegation is a simple and powerful pattern in which one object in a program acts on behalf of, or in coordination with, another object… The main value of delegation is that it allows you to easily customize the behavior of several objects in one central object. (Apple Inc. 2013a)

Delegation also works well in conjunction with the Model-View-Controller pattern. As is often the case in Cocoa applications, controller objects frequently serve as the delegate of view objects. This is especially the case when a developer does not need to subclass any of the AppKit or UIKit framework's built-in view objects, but simply uses them as is. In this case, all custom behavior is delegated to the controller object, while the developer simply reuses a stock view from the library. In this way, we see that delegation allows the developer to avoid subclassing while still enabling customization of standard behavior; it also promotes code reuse, as the library view classes are being reused, avoiding the use of a custom subclass which is less likely to be reused by other developers. Again, we see that in the MVC pattern, controller classes are less likely to be reused—but the separation of concerns that delegates all of the views' custom behaviors to the controllers allows the views to remain as general as possible, promoting their reuse. An example of this pattern can be seen in TableView objects, a type of view that displays tables of data like a spreadsheet.

---

have been delegated to it. This use of protocols to create a delegate relationship requires the protocol methods to be optional, a feature that did not have direct support in Objective-C until Objective 2.0 was released in 2007. Prior to this, this relationship was created using an "informal protocol," which was actually a clever hack—a category on NSObject was created to extend the root object of the entire Cocoa inheritance hierarchy with the necessary methods. Thus, the delegation pattern was implemented using two different Objective-C features before and after 2007, both alternatives to subclassing.

TableViews usually delegate to their controllers the tasks of customizing their view display, as well as synchronization of the display with the data.[30]

Like class extensions, delegation also allows a programmer to customize the behavior of objects that the programmer may not have the source code to, such as objects provided by Apple's closed-source Cocoa libraries, which circumvents the fact that these are black boxes to third party developers. Thus, along with subclassing and class extensions, delegation is one of many ways that built-in Cocoa objects can be customized. Because these alternatives are available, Apple and the Cocoa community heavily discourage subclassing unless there is no other way to accomplish something. This leads to a crucial cultural and normative difference in technical practice between Cocoa and other object-oriented development environments.

This was revealed by a conversation I had one day with two developers at the Big Nerd Ranch, Andrew and Brian, who had experience with Android programming (which is done in Java) in addition to iOS (which uses Objective-C and Cocoa patterns), about the different ways the two platforms train people to think. Brian said that the biggest difference is that on iOS, Apple's online documentation discourages you from subclassing as much as possible, because they want you to use the available alternative design patterns, like delegation. Andrew said that on Android, subclassing is the only way to do most things. That meant that for him, the first thing he did when he got to work every morning was to write three different subclasses. Brian said that Apple and the Cocoa culture had discouraged him from subclassing to such an extent that when he had been working on iOS, he tried to avoid subclassing a

---

[30] Officially, in both AppKit and UIKit these two tasks are conceptually separated into two different delegate objects. One, which customizes display, is officially designated as the "delegate," while the other, which synchronizes display with data, is called the "datasource." In practice, these two are usually the same object, being the TableView's controller.

button even when it was actually correct solution to his problem. He read Apple's documentation that said that in order to customize a button to do the specific thing he wanted, he had to subclass it, and he said that he thought, "that can't be true! There has to be a way to customize this without subclassing!" While that is often true in Cocoa because Apple does provide various ways to customize classes without subclassing, in this case it was not true. This happened to Brian more than once. Whereas, after he spent some time on Android, he immediately would have hit upon subclassing as the solution to this particular problem. This little exchange shows how this cultural norm of avoiding subclassing in Cocoa programming had been so deeply internalized by Brian that he did not recognize when it was actually good for him to subclass. These different norms between Android and Cocoa deeply affect the kinds of solutions they typically reach for in their everyday practice.

Other Cocoa developers I have interviewed have also noted this fundamental difference in practice between Cocoa and other object-oriented environments, which rely almost exclusively on subclassing as a customization solution. Many noted how culturally difficult it was to learn delegation for programmers used to environments where subclassing was prevalent. Adam Preble, a Cocoa instructor at Big Nerd Ranch, said that he himself had trouble with this when he first learned Cocoa.

> For a long time my stumbling block with Cocoa was, just how on earth do I organize this application? Because it just doesn't really match up with the way that you do it with WXWidgets [a cross platform UI framework he used previously]. You know, with those kinds of frameworks it's all about the subclassing. And in Cocoa we try to keep that to an absolute minimum.
>
> (Adam Preble Interview, August 8, 2011)

The message that the use of subclassing should be minimized was reiterated by many Cocoa developers I interviewed. They all noted that this normative exhortation came primarily from Apple:

> Apple has come out and said, you know, try and not to subclass as much as possible. Try and use other mechanisms…
>
> (Robert Walker May 19, 2012)

> And [as Apple] we're going to attempt to encourage delegation over subclasses. Right?
>
> (Hasan Edain March 12, 2012)

Nevertheless, what is striking about this norm is that there is little dissension among Cocoa developers about the value of delegation. Their attitudes show that understanding and appreciating delegation as a superior practice over subclassing clearly marks their high inclusion in the techno-cultural frame of the community. For example, Brian and Andrew believed that the different design philosophies between Apple's iOS and Google's Android provided evidence for why iOS was superior to Android as a development platform. According to Andrew, because Android gives a programmer so little beyond the operating system, to create any user interface of decent aesthetic quality, the programmer had to "roll his own" functionality by subclassing. With iOS, the Cocoa Touch frameworks provided so much more from the start, with view objects that looked so much better in their default configurations, that customization was not often required. Another issue was that iOS had a well-designed class inheritance hierarchy with a tree structure. For example, Andrew explained that all drawing is done by UIViews and its subclasses. In Android, by contrast, the inheritance hierarchy is shallow, leaving few meaningful relationships between objects—almost everything is a direct subclass of the root Object class at the top of the tree. As an example, Andrew said that in Android, the Animator class and the Animation class are completely unrelated. All sorts of different, unrelated types of objects draw to the screen. Unlike iOS, Android had no consistency or higher organizing principle, making it difficult to apply patterns learned in one area to another. Brian intuited that this chaos and confusion was probably reflected in Google's internal organization—he said he felt like Android was written by five different teams that didn't talk to one another, which was why things didn't work consistently across the whole platform. Despite two things sharing similar names (like Animator and Animation), they could be completely unrelated.

Other Cocoa developers similarly spoke of the superiority of the Cocoa frameworks, in large part because the prevalent use of delegation in its design.

Delegation, they claimed, was a superior technique to subclassing for many problems because it allowed for more flexible, malleable code, which would help keep it maintainable over time:

> The objects there [in Objective-C] are just softer, you could add stuff to them very easily. You're not stuck with this, OK, here's how that class works and the only way I'm going to change this behavior is to make another object that might borrow from it, as a subclass. But you can't really change what that original object was. It's sort of set in its block and that's what it does. And you can use that block and extend, you know make another one that's similar to it that may be more refined, but you're always dealing with a separate object, whereas with Ruby or a dynamic language or Objective-C, you can start adding stuff onto that… original construct…
>
> (Robert Walker May 19, 2012)

Walker here speaks to Objective-C's ability to allow objects to be extended easily. Subclassing's primary disadvantage is that, by creating relationships between objects that are rigid, it locks in those decisions and make it difficult to modify in response to change. Similarly, in the following quote, Edain notes that being locked into using objects of a certain type creates structural constraints on one's code, which is a drawback of C++'s use of multiple inheritance:

> You have the flexibility to handle design changes much more simply without incurring the wrath of circular reference or other problems that get introduced in multiple inheritance [a pattern used in C++].
>
> So the real problem with multiple inheritance is, "oh wait a minute, I have car and truck and they both have tire." Right? Oh, whose tire wins?
>
> Well in the case of having [categories], or using a delegate, it's not actually important [which one wins]. OK, well, I'll just put the kind of tire I want on it, right? And then my ambiguity is resolved and I'm really flexible. I put whatever kind of tire I need on this vehicle. Right? And so that kind of solves a lot of structural problems very easily.
>
> (Hasan Edain March 12, 2012)

Edain then explicitly compares the design patterns used in C++ (multiple inheritance) and the ones used in Cocoa (delegation, categories):

> So in contrasting the two design patterns, like the design pattern that Apple encourages on its platform, versus the design patterns that you used on C++, is that deep inheritance pattern than you would use [in C++], because there was less flexibility meant that there was a lot more complexity and a lot more working out ahead of time what all of the issues might be, you know, are there multiple inheritance conflicts, etc., that with more flexible design patterns of delegation and categories, etc., you don't have to deal with those things.
>
> (Hasan Edain March 12, 2012)

Edain notes here that the more rigid designs created by C++'s multiple inheritance pattern means that to create software with equivalent longevity, a programmer would have to be able to predict and anticipate all the potential use cases of her code. Edain understands that in practice, this is impossible. No programmer can predict ahead of time what her code might be used for years into the future. Given this pragmatic software engineering reality, flexibility has become a much-pursued virtue in object-oriented programming practice more generally. The value of flexibility in the pursuit of making software more maintainable is not disputed among programmers. However, among other object-oriented programming communities, this has not led to a widespread discouragement of subclassing as a design pattern. Edain is implying that the Cocoa community, as opposed to others, has seen the light by heavily discouraging subclassing in favor of the more flexible patterns of delegation and class extension.

How does this affect Edain's experience of programming practice in Apple's Cocoa environment compared to other object-oriented environments? Edain finds the experience much improved, and would rather not go back.

> You look back and you go, I don't miss it [not subclassing]. That's fantastic. If I want to grab some piece of functionality, I just slap a category on something or I make myself a delegate of this other thing. And yeah, if you look at my code now, I mean of course there's some subclasses of things, I mean all my view controllers are in fact subclasses of UIViewController [a class from the UIKit framework]… But for the most part I don't have deep hierarchies. I don't subclass my view controller subclasses… I might subclass UITableViewCell to do something, but then it doesn't have a tree underneath it.

> …Before, I mean in [Java or C++], I clearly would have had some five or six class deep nesting pattern, and I would have had to think… or done a whiteboard to someone and talk about like, OK, well, where are we going to run into multiple inheritance problems and where do we need to split the class and where do we need not to split the class… And now I don't have nearly any of those troubles.
>
> (Hasan Edain March 12, 2012)

Again, Edain reiterates that in object-oriented environments that rely on subclassing, the rigidity of the designs made possible by that pattern meant that to be effective, he had to spend considerable time up front thinking very hard about design decisions, because once made, they would be difficult to change. This puts undue burden on the programmer to make good decisions up front, before any code has been written, and it also means that if requirements change midway through the development cycle, the developers are stuck with designs that might no longer be appropriate. Edain's embrace of the flexible patterns of delegation and extension ("categories") means that he has more agility in his everyday practice, allowing him to focus on the task at hand without having to think through every possible longterm consequence and pivot easily when requirements change.

This does not mean that subclassing should never be used, however. "Subclasses still make sense in some contexts. They just don't make sense in every context…" says Walker. He explains that a smart programmer should understand subclassing's strengths and limits and use it when appropriate, like any other technique. What he and others in the Cocoa community object to is the indiscriminate use of subclassing to solve problems that are better solved with different techniques. Subclassing should be reserved for the specific circumstance when one wants to make a class more specific than its parent class, rather than simply extend it in an arbitrary way:

> …It has a lot to do with responsibility, whether this object is really responsible for doing that, or not. Subclassing should always be taking something and making it more specific. If you start with a Shape object, the subclass should [be]… a Rectangle, or a Triangle. It's… extending some abstract object to make it more [concrete]. But a lot of

> times, we subclass just to be able to do more stuff to that original, [which we shouldn't]…
>
> [As an example of what not to do]: if you have a Shape object, but it doesn't have a fill color or something, you subclass it, and you make a Fillable Shape. But you don't really need to do that if you can just attach that behavior to the original object.
>
> …With a fully static type language [like C++], you get what you get and then the only way you can really extend it is to subclass it.
>
> So you end up with these crazy [designs] like… you've got three different objects where you really only needed one… [Instead of creating a] separate object that takes one of these [objects] and makes another one and sends it back to you… you can just add that behavior to the original.
>
> So if it makes sense to do that, [for example] if you want a UIImage [object] that's resizable… just add it to the original object, instead of creating a second object [that] is only subclassed just to do that one additional feature. [That's] not making it more specific, it's just extending its behavior.
>
> (Robert Walker May 19, 2012)

In this extended quote, Walker explains precisely the restricted use case he believes subclassing is for, and what common usage of it is inappropriate. What is the consequence of using subclassing for mere behavior extension that is not about increasing the specificity of the object's type? Walker lays it out clearly: increased complexity, the evil to be avoided. As we saw earlier, software engineering discourse lays out unnecessary complexity as the primary cause of missed schedules, higher costs, and unreliable, risky software. More complex software is inherently more difficult to understand and thus more difficult to maintain. Given two solutions to the same problem, software engineers should prefer the less complex one.

In addition to reducing complexity and increasing flexibility, patterns such as delegation and extension increase code reuse:

> And the design patterns where people started using that ability [to extend an existing class], really changed how you approach delivering a brand new product, from the most part, I mean, reusability has kind of been this mirage for a very long time. It keeps saying oh, we're going to make these reusable libraries, but inevitably it does 80% of

> what you need and you can't figure out how to tack on the last 20%. (Hasan Edain Interview, March 12, 2012)

Like flexibility, code reuse is one of the central goals that object-oriented languages are supposed to promote. Creating too many custom subclasses proliferates the number of object classes, which by itself reduces the likelihood that any one class will be reused. Moreover, libraries that provide no easy way to extend classes for which a programmer does not control the source code also reduce code reuse because programmers cannot simply use the stock class without subclassing it to do what they want. In Cocoa, this is kept to a minimum, as for most purposes, programmers can simply use stock Cocoa objects, and use delegation or extension to customize their behavior. In this case, programmers simply reuse code that Apple has already written. Here we see how code reuse directly magnifies a programmer's productivity. Every object class that is general enough and flexible enough to be repurposed is one less class that the programmer needs to write herself. Why write three slightly different classes if one will suffice? Better yet, why write any class at all if Apple has already provided one that works?

Design Patterns and Conversion to Cocoa

Design patterns like delegation are highly abstract and are not learned overnight. Nevertheless, because of how deeply embedded they are in the Cocoa frameworks, the frameworks themselves cannot be understood without grasping these patterns. And as we saw in many of the earlier quotations, learning these patterns is not just a cognitive exercise but a transformation in norms; it involves learning to love them, to believe that they are better than the alternative. Herein lies one of the great difficulties of learning Cocoa, a difficulty that is simultaneously conceptual, affective, and normative. Those unable to make the transition come away feeling that Cocoa is weird, ugly, unintuitive, and frustrating.

Partly this is because delegation feels inverted or backwards.

> Sometimes the code, the concepts can get inverted because the Cocoa frameworks themselves when you first get into it, feel like they're backwards.

> Because rather than an object taking care of some concern itself, it's delegating out to other objects all the time. And so that's not really that common of a technique, especially in the Java world… where it's much more difficult to delegate things out.
>
> (Robert Walker May 19, 2012)

With traditional programs, a programmer sees where code execution starts and can trace its flow. Delegates, however, reverse this, because they are a form of "callback," which we discussed earlier. Remember that with callbacks, developers write the code inside functions (or methods in this case) that get called by the application when an event occurs that is triggered by the system or the user. Programmers have no direct control over when these methods get called, and only when some event outside their control occurs, does control pass to their own code, making it difficult for them to follow their program's execution. Often these trigger points happen inside blacked-boxed Cocoa library objects that delegate to one of the programmer's own objects some action that she is able to customize. The programmer must read Apple's documentation to understand the proper points to "hook" into, and trust that Apple's objects will call her code at the appointed time. This feels foreign to many programmers initially.

Another developer experienced his own difficulty learning delegation as a process of assimilation into what had been a foreign culture:

> …It wasn't so much, oh, here is a language feature and I need to learn the language feature, it was more, you know, I step through a door and all of a sudden everyone's speaking Korean. It's not even the language, it's like they're speaking English but with Korean idioms. So it's just—I'm not in Kansas anymore… All of a sudden I've really had to just stop and back up and really go, "OK, let me assimilate culturally," what the expectations of this culture are.
>
> So early on, it's like, OK, model view controller, great, I understand, that's a good design pattern, I can cope with that. But it took me probably nine months to get what people were saying about "delegate" and why "delegate" was important…
>
> It should have been simple and it should have been easy, but… there wasn't a place that I had found that just talked to me culturally about design patterns, and hey, here is how this—I mean it's an ecosystem.

> It's not just a language; it's not just a framework. It is a language and a framework in this context of an approach to writing programs. And unless you sort of encompass all of those, you end up fighting some element of it.
>
> …My problem wasn't that I didn't understand the words, my problem was that once again I was in this room of people speaking English with Korean idioms. And I just didn't understand the culture, right? Oh, why is delegation important? When would I reach for it? Where does it fit in my tool chest? How does it change how I approach applications?
>
> (Hasan Edain Interview, March 12, 2012)

Edain sensed that Cocoa "culture" was like a foreign language, with its own "idioms," one that he felt was impenetrable at first because he could only grasp small pieces of it, not the entire system. This "culture" was fundamentally about technical practice. What was delegation and what was it useful for? What kind of tool was it, and would it fundamentally change his design practice?

Those who make it through the transition, however, come to feel that Cocoa is the most elegantly designed and pleasurable programming environment they have ever worked with. Brent Simmons, a well-known indie Cocoa developer, describes it as a kind of transcendent epiphany, a feeling of aesthetic sublimity:

> When you're still learning… every time you understand better… why they work the way they do… there's like a little flash of light in your head, and you go, 'Oh! That's totally fucking brilliant! Now I get it! Oh, God.' And you just can't help but just marvel at the elegance of it. …Cocoa certainly does [have a great elegant design]; and understanding that design and understanding its beauty at the same time is a really, really good feeling. And that goes beyond just knowing how to get something done, right? That's an actual… aesthetic response.
>
> (Brent Simmons Interview, February 17, 2012)

Similarly, Hasan Edain describes a kind of quasi-religious conversion experience, For him, this conversion was gradual, a process that took months, yet it was clear to him once it was complete that he was changed, because he could now see the whole whereas initially he could only grasp at parts.

> There is a day when you look back and realize that for the last month you've been reaching for this design pattern, as opposed to that design pattern… And it took all of these bits and pieces being in place.
>
> So it's not the one thing; it's the thing in its ecosystem… Fantastic. That makes sense now… It's like a religious conversion, right?
>
> (Hasan Edain Interview, March 12, 2012)

Edain made it clear that learning delegation was a central component of this process of conversion:

> The delegate pattern—I said it took me a long time to really internalize. But once I understood it, I mean it's kind of amazing, right? Because there's this consistent thing where we're like, oh, we want to be able to graft one of these on one of these. And they're dissimilar kinds of objects…
>
> Once you have the basics, it's easy enough to learn what it does, but understanding how that really comes to life in design patterns, that was sort of the moment of revelation. …The moment when I said, wow, this is where I want to be for a while, is when I realized how strongly the entire Cocoa framework had a design point of view… As that picture starts coming in your view, it became really obvious that there was no other framework I'd ever worked with that was that consistent…
>
> (Hasan Edain Interview, March 12, 2012)

However, because it took him so long to reach this holistic understanding and appreciation of Cocoa, it required a significant amount of commitment on his part while he was learning that everything would eventually make sense. Initially, it would have been easy for him to stick with what he knew and resist.

> Well, at some level, you have to have used the pattern to see that it's successful, right? But that means that you need to know the pattern. Well to know the pattern, means you need to have used the pattern… (Hasan Edain Interview, March 12, 2012)

Edain speaks to a Catch-22 for developers. The value of design patterns is not known in advance before it is used. But if its value is not known, how will the developer trust it enough to use it? According to Edain, this takes a leap of faith:

> Yeah, so it ends up taking a little while to build up the history of—I'm just going to step off the cliff, I'm going to trust that there's a reason for this pattern. Now I'm going to go do it and then I'm going to come back and I'm going to see if it worked…
>
> Yeah, it takes a little while to build that trust. I mean you've got a lot of experience with other languages and you know having completed lots of projects… you're good at this stuff… Why change it, right?
>
> And so that process… that's why I keep talking about culture and conversion… is because you don't step in with the knowledge that the experiment is going to be successful. You step in while trying to participate in something cool, right? And then there's all this crap in the way of the doing something cool.
>
> [But then,] Ooh, wow, it all makes sense… all of a sudden—and this is what I'm talking about, you sort of look back over nine months and you're like, oh, wow. OK, yeah! Now I really get it! And it all fits together and I can see the whole forest and I can look at individual trees and not get all distracted.
>
> (Hasan Edain Interview, March 12, 2012)

Edain's experience documents the Janus-face on either end of his conversion to Cocoa. Before the shift, he was resistant to changing practices that worked well for him in the past, not knowing how things could be improved. He felt lost, frustrated, experiencing "crap" that got in the way of him "doing something cool," making an app for the iPhone. It took a leap of faith for him to soldier on. Once through the other side, however, he now saw the forest for the trees, and a whole new world had opened up to him. He began to see his new self as different, enlightened, from his past self, who didn't know any better. He not only understood new concepts, but had acquired a completely new approach to solving problems.

Because delegation is among the most difficult concepts for newcomers to learn, it sometimes stands in for the entire way of thinking that they must acquire as they learn Cocoa programming. Edain saw delegation as a proxy for what he called the entire Cocoa ecosystem. Fully grasping this ecosystem was a gradual process that took him months, one which involved an enormous amount of trust on his part, trust that was ultimately rewarded with an experience of enlightenment at the end. What we can see in his example is that the process of learning Cocoa, while it involves the

learning of significant technical and conceptual knowledge, it also involves affective and normative commitments. It is not clear what motivated Edain to put his trust in Apple's libraries. Maybe, as he says, he "wanted to do something cool," to tinker with and create something on his consumer device of choice. Maybe he already liked Apple products. Or maybe he was motivated economically, to try to get in on the mobile app craze. Whatever motivated him to get over his initial frustration, he needed to suspend his resistance to how strange and foreign everything seemed to him at first, and trust that eventually he would see the light. This must have taken some determination. Gradually, as his hard work was rewarded, at some point, everything became clear and he was able to understand the entire picture. In hindsight, everything now made sense, as if it all had fitted together and had been working in conjunction all along. Moreover, he now saw the design of Cocoa and the normative practices it encouraged as connected. Not only was Cocoa designed with patterns such as Model-View-Controller and delegation built into them, thus requiring knowledge of them to program using Cocoa, but the Cocoa mentality also urged programmers to use these patterns in the design of their own applications and frameworks, even if it did not require them to. It was not enough to simply use them when required. One ought to use them because in the right circumstances, they provided for better designs, better solutions, because they increased the flexibility, reusability, and maintainability of code. Cocoa developers see the proper deployment of design patterns as a "best practice" in programming.

Delegation is interesting in another way. The design pattern of delegation, in microcosm, stands in for the general attitude a Cocoa programmer must have in relation to the code objects provided by Apple's Cocoa libraries. In the delegation pattern, a programmer asks a built-in Cocoa object to delegate certain responsibilities to one of her own objects whose code she writes herself. In order to work this way, however, she must already have mentally delegated many basic functions of her program to Cocoa library objects to handle for her. As we discussed earlier in the chapter, Cocoa programmers are taught to see mundane tasks that must be repeated for each and every application as tasks that should be delegated to Cocoa, while

custom tasks that are unique to an individual's application are the important ones that a programmer should write herself. This division of labor reinforces the notion that higher-level, more abstract, design-oriented tasks remain with the human while repetitive tasks should be machine automated. This requires a degree of trust in the provider of that machinery, Apple. Delegation cannot be understood without thinking about division of labor and trust. Object-oriented programs model, reflect, and reproduce these human social relationships in a programmer's mind and in running code.

Conclusion

If learning Cocoa is so difficult, involving learning new ways of thinking and solving problems, how is it that people manage to do it? Certainly, if the Cocoa community is to reproduce itself, it must acquire converts and train novices in the idioms, practices, and culture of Cocoa programming. How does this take place? How might one teach Cocoa?

Big Nerd Ranch, Aaron Hillegass's training company for iOS and Mac OS X programming, has become the leader in teaching Cocoa through its own particular pedagogical method. Like a religious monastery, the Big Nerd Ranch trains novices in a sequestered environment where expert instructors lead students in highly disciplinary exercises. These exercises do not explain concepts such as delegation up front, but ask students to practice writing and fixing code and trust that eventually, things will be made clear. Understanding gradually emerges through practice. The presence of experts who can answer questions or rescue students who are stuck creates a safe space for them, lessening the burden of having to trust that things will work out. Moreover, like a monastery, the Big Nerd Ranch provides normative lessons on proper practice. Adam, an instructor there, notes:

> I think that a lot of the advantage of what we do [at Big Nerd Ranch courses] is the experience and showing you the right way to do it. And then kind of trying to impart those design principles.
>
> (Adam Preble Interview, August 8, 2011)

Through disciplinary pedagogical techniques in an environment of intensive practice, the Big Nerd Ranch can jump start the process of learning Cocoa to help get students through at least a portion of the initial learning curve. In the next chapter, we take an in-depth look at one instance of this first stage of becoming a Cocoa programmer.

Chapter 4: The Pedagogy of Cocoa: Design Patterns, and Coding Style at Big Nerd Ranch

In the previous chapter, we saw that Cocoa developers sometimes described their experiences learning Cocoa as a "conversion," as a process where they gradually begin to see the forest for the trees and understand Cocoa programming holistically. Once this conversion has taken place, a Cocoa programmer becomes committed to the idioms, stylistic conventions, and norms of Cocoa programming practice. Having now experienced the conveniences and the pleasures brought by the Apple toolkits, a novice Cocoa developer is now more willing to trust in unfamiliar Apple tools to do the same. The developer becomes highly included in the techno-cultural frame of Cocoa development. In this chapter, we will see the beginnings of this process, the first stage of what anthropologist Sharon Traweek calls a community's "developmental cycle," or "how the group transmits to novices the skills, values, and knowledge that constitute a sensible, competent person…" (Traweek 1988, 7) For a generation of Cocoa developers, this first stage involved taking a class at the Big Nerd Ranch, a training company started by Aaron Hillegass, a former NeXT and Apple instructor. This chapter is an ethnographic study of how people learn to become Cocoa programmers at the Big Nerd Ranch. We will look at how instructors teach both technical knowledge and normative values in practice, and how one ought to think like a Cocoa programmer. We will examine how the affective experience of the Big Nerd Ranch course creates emotions of frustration, catharsis, and gradual awakening, through embodied learning-by-doing in a disciplinary pedagogical setting. The emotion of attaining new mastery and appreciation for elegant design becomes a powerful motivational force for developers' further commitment to the techno-cultural frame of Cocoa development.

Andrew Warwick and David Kaiser have put forward a theory of the pedagogy of techno-scientific disciplines synthesizing the work of Thomas Kuhn and Michel Foucault. Kuhn claimed that, rather than a disciplinary field's theories and concepts merely being illustrated by exemplary exercises, it is the repeated practice

of those exercises that generates in the student the meaning of those theories and concepts themselves. In this way, science is learned more as an embodied craft skill, rather than abstract propositions, and what constitutes what Kuhn calls a "paradigm" is rather not a theoretical worldview but shared experience in seeing new problems as variants of existing canonical ones with known solutions (Warwick and Kaiser 2005, 394–398).

Kuhn, however, does not say much about how such training occurs. Warwick and Kaiser use Foucault to fill in the blanks. In *Discipline and Punish,* Foucault outlined the numerous means that workers, students, inmates, patients, soldiers, and other objects of modern disciplinary apparatuses are cataloged, arranged, and subjected to subtle forms of micro-coercion to make them docile and productive. This was done through partitioning of their time and space and most importantly, pervasive surveillance. For students, not only were they themselves arranged into classes and ranks, but also the disciplinary field itself was similarly partitioned into gradations and levels. The ultimate surveillance tool of the school is the examination, which simultaneously punishes as it classifies the student under the teacher's gaze. This view fits well with Kuhn's observation that physicists' training is "regimented and authoritarian." (Warwick and Kaiser 2005, 398–400)

Taken together, this "Foukuhnian" perspective provides "a framework in which training is constitutive of professional practice" (Warwick and Kaiser 2005, 401), which explains "agreement within the scientific community in terms of shared skills, commitments, and value judgments," (402) "posits training as a general mechanism for the active production of knowing individuals that recognizes no natural distinction between the mind and the body, nor by implication, between theory and practice" and historicizes "the processes by which specialized technical competencies became the common preserve of widely extended communities of practitioners." (403)

The problem with Foucault's work is that his notion of discipline makes people into docile objects of study, rather than creative producers of knowledge, and

it places all agency on the teaching apparatus, not the students themselves. Warwick and Kaiser note that, even in a creative activity as jazz, improvisation is "pedagogically conditioned," depending on years of practicing basic elements which become available techniques ready at hand to draw on. Yet they also note that "the purpose of technical training is not just to manipulate the student's behavior for the purposes of the master, but to reproduce the master's skills in the student. For this process to work effectively the student must *want* to acquire the master's knowledge and be a willing and active participant in his or her own education." (404) Foucault's disciplinary model does not easily describe a student's own motivation and self-discipline. Suman Seth raises a similar critique: disciplinary mechanisms work well in producing obedient students, but not creative, innovative researchers. Yet in his examination of the Sommerfeld school of physics in Munich, Seth notes that disciplined creativity and training for original research was carefully fostered. To help explain this, Seth drawns on another Foucaudian notion, the distinction between "initiatory time" which is focused on individual apprenticeship, and "disciplinary time" with all of its rigorous exercises, grading, examinations. Unlike Foucault, who relegates "initiatory time" to a pre-modern era, Seth finds that a hybrid initiatory-disciplinary "pedagogical economy" was crucial to the production of innovative physics researchers at Sommerfeld's school. (Seth 2010, 64–70) The implication is that, in many modern disciplines involving creativity and craft skill, both initiatory and disciplinary techniques occur in their training methods.

As we have seen in previous chapters, software programmers are similarly engaged in creative and innovative tasks, often describing it in terms of "art" or "poetry." And despite attempts to routinize their work, it remains very much a craft skill. Yet, like jazz improvisation, it is a disciplined creativity. In order to produce working, maintainable code, developers have the motivation to self-discipline themselves, adhering to, and even advocating, "best practices," and admonish their peers who do not. Unlike a strict Foucauldian system, however, programmers have varying degrees of freedom in ignoring or flouting best practices, especially if they work individually or their clients never see the code, only the end product. The

problem of disciplining programmer practices is not purely a managerial problem, but also a problem for the community of practitioners, who often read each others' code; the community itself does a lot of self-disciplinary work, through pedagogy as well as advocacy on blogs and other forums.

Warwick and Kaiser's "Foukuhnian" perspective is useful in another way, as well. They locate Kuhn's often controversial notion of the "incommensurability" of paradigms, or more specifically, "disciplinary matrices," not in the incompatibility in the meanings of scientific terms or worldviews but in different interpretations of canonical texts and exemplar problem solutions at specific sites. Such differences in meaning are produced through different practices, which are themselves a product of different training regimes. "What is *incommensurable* on this showing is not the essential meaning of scientific theories themselves, but the particular skills, techniques, and assumptions that go into generation a working interpretation of them at different sites." (Warwick and Kaiser 2005, 405) The local training regimes are necessarily associated with the "shared skills, commitments, and value judgments" (402) of the particular communities of practice which constitute the disciplinary field. Incommensurability, however, may be too strong a claim to make for how different Cocoa programming is, or even how different object-oriented programming is from other methods, despite the computer science usage of the term "paradigm" to describe it, given the fact that many object-oriented practices, such as the use of modularity, are present in structured programming methodologies as well. It is not clear that even at training sites such as the Big Nerd Ranch, the skills, techniques and assumptions that generate a working interpretation of object-oriented concepts are at all incommensurable; in fact, such training often relies on shared understandings between Cocoa and other languages, such as Java or Python. It is more likely that the Big Nerd Ranch serves as a kind of "trading zone" in which concepts that straddle the boundary between different practices are translated or reduced into a simplified "pidgin" vocabulary, with their richer native meanings circumscribed for the purposes of communication. (Galison 1999) This pidgin suffices in the beginning,

but gradually, through practice, the student acquires enough experience to be able to understand the concepts natively, on a visceral level.

This perspective can help to explain the "conversion" experience associated with Cocoa's "learning curve." Most newcomers to Cocoa today do so out of economic or professional interest, because iOS is a prominent, lucrative mobile platform that has become a focus of high-technology investment and entrepreneurship. They start out with no particular loyalty to Apple or its way of doing things. Newcomers almost universally find Objective-C's syntax to be odd, and this syntax often, as we will see, becomes a site of affective resistance. Yet the syntax of the language itself is not the most difficult thing to learn. What is difficult are the ways that Cocoa programs are put together, the ways that Cocoa developers approach solving common programming problems. Like physics, programming is fundamentally a problem solving exercise, and what Kuhn calls different "disciplinary matrices" produce different canonical solutions to common problems. As we saw in the previous chapter, in object-oriented design such canonical solutions are known as "design patterns" from Gamma et. al.'s catalog of them. (Gamma et al. 1995) NeXT made extensive, consistent, and recurring use of design patterns in what became the Cocoa frameworks, and these patterns structure how programmers must think in order to write programs using Cocoa—it constrains them but also makes possible, by channeling their work through acknowledged patterns, disciplined creative production and problem solving. It is learning the design patterns that are pervasive throughout Cocoa that newcomers typically have the most trouble with. Despite this, once these design patterns are mastered, students often experience an epiphany in which everything suddenly makes sense. The productivity benefits of these patterns are felt emotionally, as a feeling that "now I am able to accomplish way more than I ever could before," because Apple's design makes it possible. At this stage of experience, developers acquire a new aesthetic appreciation for the way Apple has designed the Cocoa frameworks. This creates a sense of trust, in which developers, having seen the good technical decisions Apple has made in the past, begin to acquire faith that Apple will continue to make the right technical decisions

in the future, justifying Apple's technical control over their platforms. What may have begun as mere economic interest becomes a much deeper, affective commitment to Apple and its platforms. As we have seen in earlier chapters, it is this pleasurable, aesthetic experience of using Cocoa that third party developers, and in particular, indies, are tied to, not Apple as a corporate entity. In this way, they become ideologically bound to Apple's own interests.

Pedagogy at the Big Nerd Ranch

This chapter takes an in-depth look at how such learning occurs at a central site for Cocoa pedagogy. Over the course of my interviews, I found that a large number of the more experienced Cocoa developers, especially those that first learned Cocoa in the early 2000s, did so by taking classes at a place called the Big Nerd Ranch. Of the rest, almost universally, they had learned in whole or in part from a book on Cocoa or iOS programming released by the Big Nerd Ranch. What is this strangely named place, and why does it appear as an almost obligatory passage point for the learning of Cocoa programming?

The Big Nerd Ranch is privately held Atlanta-based company whose primary business is the teaching of Cocoa and iOS programming. It was founded in 2001 by Aaron Hillegass and Emily Herman, and until 2008 held steady at less than ten employees, but by 2013, had grown to between fifty and a hundred. It continues to be a privately held company, with Hillegass in majority ownership. Hillegass started the company the same year as the release of the first version of Mac OS X (10.0) in March 2001, anticipating a pool of Macintosh programmers wanting to switch from Carbon to Cocoa, as he saw Cocoa as the future for the platform. Today, the business has become enormously profitable, thanks to high demand for mobile programmers from companies including Facebook. However, the iPhone could not have been anticipated in 2001 when Hillegass started the Big Nerd Ranch, and until 2008 it remained a tiny operation centered around Hillegass's teaching, with some consulting work on the side to pay the bills. Besides his business partner, Herman, there were a few support staff, and two other permanent instructors who also did

contract programming, but many other courses taught under the Big Nerd Ranch name were taught by people Hillegass contracted with on an as-needed basis. Most of Big Nerd Ranch's business during this period came from desktop Mac OS X Cocoa classes and consulting, which provided Hillegass with steady income to keep the operation running. Hillegass was not going to get rich from teaching classes on programming for a PC platform with less than 10% marketshare, and during this time he could have made considerably more money doing web development, but chose not to. Nevertheless, Hillegass happened to be in the right place at the right time to capitalize on the enormous demand for Cocoa and Objective-C programmers after the iPhone App Store was announced in 2008, rapidly growing both the training and consulting side of his business. After this success, Hillegass has had the foresight to diversify and expand both his course offerings and his consulting services into development for the competing Android and Windows mobile platforms, front-end web development, as well as the backend web services that both mobile apps and web clients depend on. The Big Nerd Ranch is like a homestead started by humble farmers that happened to strike oil.

I had first heard of Aaron Hillegass when I was an employee of Apple in the early 2000s, who had joined the Cocoa framework group in the Mac OS X division at Apple as their Quality Assurance, or software testing, engineer. I had joined the team just in time to help test the first public release of Mac OS X, which, after four years of development, had finally brought the operating system and object-oriented development environment acquired from NeXT to Apple Macintosh personal computers. NeXT-based Cocoa technology was new, and there were almost no books on Cocoa programming. Ali Ozer, my boss and longtime manager of the Cocoa/AppKit group at Apple, suggested I purchase a new book by Aaron Hillegass, who had worked previously at NeXT and Apple teaching NeXT and Cocoa programming to employees and third party developers. Hillegass's *Cocoa Programming for Mac OS X* soon became the canonical text for learning Cocoa programming. In 2003, Ali introduced me to Hillegass at Apple's Worldwide Developer Conference, where I got my copy of his book signed. I remember at the

conference that Hillegass, a tall man over six feet in height, went everywhere wearing his trademark cowboy hat, which made him instantly stand out in the crowd. Another Apple coworker of mine explained to me that Hillegass ran a kind of dude ranch for programmers—you would travel to his facility, live on the ranch for a week, but instead of learning to ride horses or lasso cattle, you would learn to write Cocoa software.

Given this prior experience, when it came time to conduct my field research in 2011, knowing I wanted to study how people came to learn Cocoa, Hillegass's Big Nerd Ranch seemed to be the perfect place. Through my network of former co-workers at Apple, I was connected to Hillegass through the employment social networking site LinkedIn, and sent him a message expressing interest in volunteering to work for him as an intern in exchange for ethnographic access, and he was not only gracious, but welcomed me. Given that I had previous been an Apple employee, however, he offered to compensate me as well, at a modest intern's salary, which I accepted as I had not found any other sources of funding.

Upon my first meeting with Hillegass in the field, I was surprised to see him not wearing his cowboy hat. Hillegass explained that he only wore the hat as a marketing stunt. He wasn't actually a cowboy or a southerner, in actuality he had grown up in the Washington D.C. suburbs. A few days later, he agreed to tell me the story of the hat, and his company's unique name, over lunch. After leaving Apple, Hillegass wanted to continue teaching NeXT-style programming, and when Apple released Mac OS X, he decided to take a risk and start a business teaching Cocoa. Hillegass envisioned his Cocoa programming course as a monastic retreat for programmers. However, he realized that the image of a programming cloister would not be an effective marketing strategy to Americans, so he came up with an alternate metaphor—a dude ranch. It would be a dude ranch for nerds. Surprisingly enough, another company already had claims to the name "Nerd Ranch," so he named his company "The Big Nerd Ranch." Being a two-person operation in the beginning, with co-founder Emily Herman handling much of the business side of the company, Hillegass had little budget for marketing. He decided that he would personally brand

himself by attending Apple's WWDC and other Apple-oriented conferences always wearing a cowboy hat, which became his signature. This tactic was wildly successful. The cowboy hat was incorporated into the company's logo, adding a beanie's propeller spinning on top, combining "rancher" and "nerd" elements together.

When I arrived at the Big Nerd Ranch corporate headquarters, I was surprised to find that it was housed in an office park in a gentrifying neighborhood of Atlanta. Having heard stories of the "dude ranch for nerds," I had been expecting to go out into the countryside to some farm. Like Hillegass himself, the company's frontier identity was just marketing. It was only later that I discovered that the site where Big Nerd Ranch holds its classes, which is out in the countryside, is separate from its corporate headquarters, which gave me the impression of being a typical, hip, post-dot.com technology startup, like the kind I might find in Palo Alto.

The headquarters itself has moved four times between 2010 and 2013, in response to the exponential growth of the company. In 2010, they operated out of condo that Hillegass personally owned, which was sufficient as they had only a dozen or so employees. On my first visit in June 2011, they had just moved into an office complex in the Inman Park neighborhood of Atlanta, with up to 30 employees. By June of 2012, they had over 50 employees, and they were running out of space at the Krog Street complex. That August, they moved into a renovated ironworks building in the Kirkwood neighborhood of Atlanta that the company had purchased in cash with the previous year's profits. Plans were made for an expansion of this building in the years to come. However, by the time I left in December 2012, the company had just completed a merger with Highgroove, a company specializing in web server contract programming using the Ruby on Rails development environment. This necessitated yet another move to a larger office complex by 2013. The merger was also partly the result of Hillegass's duties as CEO growing to such an extent that he had to spend all his time managing the business instead of teaching or developing course materials. After the merger, he stepped down as CEO but took a role as head of instruction, allowing him to return to what he preferred to do.

On my initial arrival at Big Nerd Ranch in 2011, I was introduced to the management team. Aaron was still CEO at this time, but rounding out the management team were Jami and Jenn, who managed the courses, Jason, who handled the consulting side of the business, Stacey, who handled finances and human resources, and co-founder Emily, who had returned after several years away to run the company's efforts at developing apps under its own brand. Members of this executive team were middle-aged professionals, many with families, and four of them were women. By contrast, the rest of the company, which comprised the technical staff, almost all of whom did consulting work, with a few also taking on instructor duties, were male, except for a young intern who was just starting her freshman year at Georgia Tech. Despite the gender imbalance and her youth, she seemed to be enjoying her time at the company, got along well with her young team, and appreciated being given real programming responsibilities, as well as being well-mentored, on a large Android consulting job. Many of the men who were technical leads were in their 30s, but many of the recent hires were in their early twenties, with some hired straight out of college. Racially, the majority was Caucasian. Besides myself, there were a few other Asian-American men, including Brian, a graphic designer, and Bolot, a software engineer from Kazakhstan. Another intern was a young African American man, the only one in the company at the time.

As the company grew, it became more diverse as well. The graphic and user interface design group, initially composed of just Brian, added an African American man, and a Caucasian woman who became the group's manager. A newly created web team hired a female developer and was led by a Filipino. Another African American was hired as a Windows software developer. After I left the company, another African American joined the iOS developer team and became an instructor. The Highgroove merger, which occurred just as I was leaving the field, diversified the company even more, as it had a higher proportion of young female developers on its team, who even ran their own women Ruby developer's club. However, Highgroove's management team was predominantly male, and appeared a decade younger than Big Nerd Ranch's. As the merger involved much of Highgroove's

management team taking over operations of Big Nerd Ranch, some of the existing female managers unfortunately had their roles reduced.

The Krog Street office space, where Big Nerd Ranch had its headquarters for all of my first visit in 2011 and half of my second visit in 2012, was mostly composed of a large room with a high ceiling with skylights. It was surrounded on two sides by giant windows which looked out onto the parking lot and street. Aaron's office was on one side, looking out on the street, while the rest of the executive staff had offices against the opposite wall, with only Jason having a window. A large conference room was located at the main entrance, containing a large, rustic wooden table that evoked the company's ranch theme. The opposite corner contained a small kitchen with a nook for lunch. The rest of the main room was split up into numerous little cubicles, which comfortably sat two, but could seat three if necessary, after the company grew and space was at a premium. The cubicles were separated by wooden dividers that came up to only four feet or so on one side, allowing people to easily talk and socialize. Many employees used the tops of these dividers as display areas for little toys and knick-knacks. As a running joke, a life size cardboard cutout of Robert Pattinson's vampire character from Twilight was placed somewhere around the room, with people moving it around at random to scare their friends. A metal ladder in the kitchen led upstairs to a dark area that housed the company's server, and also contained some makeshift desks that were initially used by the instructors of the company's Android class to work on their course materials and book. By 2012, the growth of the company had forced newly hired employees to work up here full time, but being up here felt as if one had been exiled.

I was told that employees at the company were known as "Nerds," which was considered a badge of honor. In some sense, the term was just a way to identify as a member of the company, the Big Nerd Ranch. It was not something that one had to earn (unless one was an intern working on a trial basis) as the term could refer to any employee, though in practice it tended to mean the technically oriented people, the engineering staff, who wrote code or taught classes, and who were mostly men and could fit into the popular conception of nerds and geeks. A poster on one employee's

wall tried to differentiate between the terms “geek” and “nerd,” but this came from outside the company, as it gave “nerd” a negative connotation. In some sense, it felt as if the employees of Big Nerd Ranch were trying to reclaim the term “nerd” as a positive identity. Anybody who was passionate about technology could thus be a nerd, regardless of gender, race, or level of social ability. In fact, some of the young men who worked at the Ranch were rather hip, and seemed to have extensive social lives. Joe Conway, instructor and book author, was a track-and-field athlete who I considered more of a jock than a nerd. The notion of “nerd” as an inclusive and positive company identity was exemplified by the company employee of the month award, which was called the “Nerd of the Month” award. Typically this was given to an employee who had worked above and beyond the call of duty, impressed with their technical skill, or contributed in a particularly valuable or pivotal way to the business. Men in the engineering department usually won it, but in August 2012, Stacy, the CFO and HR executive, took the prize for managing the company’s move into its new headquarters.

Synergistic businesses

On my first full day, I went through an orientation, which involved a meeting with the three “Js,” Jami, Jenn, and Jason. I went through this with another new hire, Owen, who had recently been a high school computer science teacher, and a personal acquaintance of Aaron Hillegass through a book club. It turned out that Hillegass often found new employees through personal acquaintances, and Jami and Jenn were both family friends. The J’s explained that the Big Nerd Ranch had three different businesses. Teaching, of course, was the primary business. Jenn managed the bookings and scheduling for the weeklong bootcamp training courses, which she called “open enrollment” to signify that these courses were ones in which individuals enrolled themselves on the company’s website. These were for developers who either paid their own way (about $3500 for the weeklong experience), or got their employer to pay to send them to Big Nerd Ranch’s training center. Despite being at an information technology company, she used physical tools to do scheduling—a whiteboard was used to mark down tentative schedules, not only for which class was

being held during which week, but also which instructor was assigned to it. This was done in erasable marker, or with post-it notes, which gave her flexibility if an instructor's availability changed. This was highly important, as most of the instructors came from the company's own developers who, most of the time, were working on other projects. Her whiteboard gave her an overview of the entire year's bookings at a glance. Only when a schedule was more or less locked down did she enter a course into the computerized system on the web, which contained the "official" record.

My preconception upon arriving at the company was that the "real" Ranch was where the classes were held out in the countryside, the retreat where students stayed, not where the company was administered from or where its contract programming business was centered. The classes were what it was known for. Curious to find this "real" Ranch, I asked where the company's training facility was. Jenn explained that the "ranch" was at a place called Historic Banning Mills (which she and others referred to as "HBM" for short), which was a country resort about an hour and a half outside of Atlanta that the company rented out, though not exclusively—others could book the place for weddings, company meetings, or vacations. Thus, I discovered that no actual "Ranch" existed—the training facility was not owned by the company known as Big Nerd Ranch, whose headquarters seemed to be like that of any other technology startup. However, Jenn did mention that the company had plans to build its own training center, also out in the woods, to avoid having to rent HBM. Land had been already acquired, and architectural and landscaping plans were being drawn up, though no ground had been broken yet, due to other expenses taking priority. (As of 2014, work on this facility still had not started.)

Jami handled another type of training course, the "onsite" or "corporate training" course, which she also scheduled using a whiteboard. Jami explained to me the difference. Some companies want to train larger teams of developers, and paying $3500 a seat for more than three is not economical. Instead, Big Nerd Ranch flew one of its instructors to the client to run a weeklong course "onsite." This was an

extremely lucrative business, and allowed Big Nerd Ranch to have more than one course offered simultaneously, as Historic Banning Mills only held one course at a time. Many high profile companies have had their own mobile development teams trained in this manner, including Facebook, which in 2012 was reorienting its software engineering focus away from web apps towards native mobile development, (Chen 2012) and hired Big Nerd Ranch to train its developers in iOS and Android. Facebook's success since then has largely hinged on the success of this mobile transition.

A second business was its book publishing. Hillegass had published his first book, on Cocoa programming, before founding the company, so its first edition did not bear the company name or logo, but all of his subsequent books and later editions of the Cocoa book were published under the Big Nerd Ranch name.

The third business was contract software development. Thus, Big Nerd Ranch was in the business of actually writing applications, not under their own name, but for other companies and organizations, which included AT&T, Google, Spotify, GE, Coca-Cola, Whirlpool, Honeywell, and NASA.[31] The majority of the software developers in the building, therefore, worked on such consulting contracts. The majority of the company's instructors came from this pool of developers, and when they were not teaching, they were typically working on a project for a client. (The rest of the instructors, who taught courses held infrequently, were contracted on a per-course basis.) However, not all developers were instructors, or were interested in teaching. The company also had one full time user interface and graphic designer at this time, and would gradually hire more. Jason was the manager in charge of the consulting business, and he managed the contracts, the relationship with clients, as well as the business development of new clients. Most of the day-to-day work I

---

[31] "Our Work," Big Nerd Ranch LLC, accessed July 24, 2014, http://www.bignerdranch.com/we-develop/our-work.html

would witness and personally participate in at the corporate headquarters was related to contracting work, rather than the teaching side.

Aaron Hillegass saw all three of the company's primary businesses as tightly intertwined, and by the end of my first full week at Big Nerd Ranch, I began to understand that what these three businesses had in common was that they were all about selling expertise. Although consulting made up a significant chunk of the company's revenue, was its fastest growing business, and made up most of the company's new hires, Hillegass said he saw it as a critical component of the company's educational mission. Although Big Nerd Ranch contracted with a few instructors who were not permanent employees to teach a couple of its less popular classes, its primary classes in iOS, Android, and Mac OS X programming were all taught by full time employees who, when they were not teaching, worked on contract development. Because of the rapid pace of change in information technology and in Apple's platform in particular, programming knowledge and skill must be constantly updated to stay current. I experienced this myself—I had not used Apple's Integrated Development Environment (IDE), called Xcode, since version 2.0 in 2005, when I was an Apple employee. In 2011, I needed to learn Xcode 4.0, which presented a radically different interface than earlier versions, and took a while to get used to. While this constant change in the industry provides Big Nerd Ranch with continuing business, it also presents it with a dilemma—if it is in the business of selling its expertise, how does it keep its own expertise current? This is maintained in large part through continuing to practice software development in real world contexts, through contract development. Thus, expertise is sold in two different ways—to clients, who want the most skilled consultants to write their apps, and to students, who want the most knowledgeable instructors to teach them skills that come from experience writing real, shipping applications. I began to visualize programming expertise flowing from the consulting side of the business to the training side.

In turn, Hillegass told me that the courses explicitly inform the books. Although the book serves as the course text, it must be constantly in revision if it is to keep up with updates in Apple's operating systems. Over the next few months, I

got to know some of the instructors, who, besides their teaching and contracting work, would also spend time honing the course materials in response to feedback and experience from the courses. Most work occurred right after Apple's WWDC, which announced new versions of Apple's platforms, which inevitably meant new APIs to cover, changes to existing APIs, and sometimes, entirely new frameworks or Kits that provided new functionality or a new simplified layer over existing functionality. Other times, extensive changes in the design of Apple's operating systems, either architecturally or visually, required significant changes to code. These all meant that the Big Nerd Ranch's existing books were now obsolete and had to be updated. Students, of course, wanted to take a class that took such changes into account as soon as possible, but it would take several months to modify the materials comprehensively.[32] Instructors told me that questions asked in class to clarify lessons alerted them to things that were not clear in the course materials, and after the course ended, they would revise the book in response. Thus, every course taught constituted a chance to revise the book, and the course materials were treated as a beta version of the book, which after much refinement would become the next edition to be published, often six months or later after WWDC. According to the company, this approach gives its books a unique effectiveness most other programming books cannot match: they have been tested and iterated over constantly in actual classrooms before they ever make it into print. The Cocoa community seems to agree with this assessment, as the books are widely considered essential for learning Cocoa. The company's iOS, Objective-C, and Android books are all rated 4.5 stars on Amazon.com. Even other courses, such as those being taught by members of the Seattle Cocoa developer community through the University of Washington's Continuing Education program, use Big Nerd Ranch instruction manuals.

---

[32] Due to Apple's NDA restrictions on information presented at WWDC in June, the Big Nerd Ranch also could not actually teach any new material until the new version of the operating system was released to the public, usually around September.

In this way, the intertwining of the Big Nerd Ranch's three businesses, teaching, writing, and developing, contribute to what Hillegass informed me was the core mission of the company, learning. He said that he tried to create an environment in which his employees were encouraged, in their everyday work, to be constantly learning. As I see it, the interconnections between the three businesses generate expertise that can then be transmitted to others, through a recursive process. Contracting generates and maintains expertise, which then becomes the product sold to students in teaching. Teaching itself generates a different kind of expertise: that of pedagogy, which is then recycled into the company's books. Thus, learning by doing helps generate both programming and teaching expertise. The books make up only a fraction of the revenue of the training and contracting businesses, but they serve as a marketing tool, to spread the reputation and brand-name of the company and Aaron Hillegass himself, as the founder of the company. And though the consulting business contains far more employees than the training business, the company's primary reason for existence is the latter. Both consulting and books are intertwined with training, but are also in some way subordinate to this primary mission.

The Big Nerd Ranch Bootcamp

Since the Big Nerd Ranch's primary existence is its training business, upon my arrival I wanted to be able to observe the Big Nerd Ranch's courses at Historic Banning Mills. However, because of the limited availability of lodging at HBM, which the company would book for me, I could not go immediately. I signed up with Jenn to be a teaching assistant at the next available iOS programming bootcamp, justifying her booking me a room there. I did this partially because, with some previous Cocoa experience, I might be too knowledgeable to take the course, and in addition, being a TA would allow me to work and provide value for the Big Nerd Ranch, which was part of the terms of my access. Moreover, being a TA and not a student would provide me with time to observe and converse with the other students in the class. However, since I had never done iOS programming itself, in order to be a TA, I needed to be at least somewhat more knowledgeable than the students. In preparation for this, during my first month working at the Big Nerd Ranch's

corporate headquarters, I worked through the company's iOS programming book on my own. Although the book had been sent to the printers, the first run had significant misprints, and I was asked to help revise as I worked through it. I also helped revise the code solutions that would be passed out to students in class, which meant that many of the solutions used in my first iOS class contained my own code. Working at my own pace, I needed an entire two weeks to fully complete everything in the book. This is notable because the course is taught in only one week, so most of this material is covered in half the time it took me to do it on my own.

On July 8, 2011, it was time for me to go to Historic Banning Mills for my first iOS bootcamp. The iOS course ran in two versions. The seven-day version ran from Saturday through Friday, and included a two-day primer over the initial weekend on the C and Objective-C programming languages, which was necessary for many students because Objective-C is not used widely in the software industry outside of Apple. The five-day version simply ran from Monday through Friday and omitted the Objective-C language primer. Between 2011 and 2012, I would attend three different Big Nerd iOS Developer Bootcamps at Historic Banning Mills (all seven-day versions), the first time the whole way through, the second time, only through the weekend Objective-C primer, and the third time, only from Saturday through Tuesday, which allowed me to at least see how the students transitioned into iOS.[33] I volunteered to be a teacher's assistant in all three of these classes, which consisted largely of helping students debug their programs during the lab times, and helping to set up and tear down the classroom before and after. During my first time there, the instructor asked if I was willing to try taking over lecturing on one of the lessons, which I accepted, and during my subsequent trips I continued to expand on this, taking on a few additional lectures. Also on my first trip, I was joined by another TA who was in training to become a fully-fledged instructor. Giving TAs

---

[33] My engineering responsibilities at Big Nerd Ranch had increased enough by August 2012 that I could no longer afford to be gone an entire week at Historic Banning Mills.

portions of lectures to try, with the main instructor in the room to answer students' questions, was one of the ways that Big Nerd Ranch groomed new instructors. In addition to attending the iOS courses, I also attended two of the Big Nerd Ranch's other courses: OpenGL, the low-level industry standard 3D programming language, which I took because I thought it might help with the contracting work I was doing for Big Nerd Ranch, and the Android course, which I took in order to get a comparison between iOS and Android. In both of these courses, I met students who were already experienced iOS developers, having either taken Big Nerd Ranch's iOS course or one from another training company. While the following account of the Big Nerd Ranch courses draws primarily from the iOS Bootcamp, especially the first one where I attended it in its entirety, it is actually an amalgam of all of my trips, and is not intended to be a completely accurate description of any one single experience at the Ranch, but to portray an overall sense of the typical experience there. My attendance at the OpenGL and Android courses allowed me to experience the Big Nerd Ranch bootcamp as a student on the same level as the others, whereas in the iOS courses I had a very different role as an assistant to the instructor, which meant that I was somewhat of an expert, at least in comparison to the students in the course.

The result is that my participant observation at Big Nerd Ranch came with a unique vantage point. Although I had been working as an intern at the Ranch headquarters, because of my previous experience at Apple, Hillegass trusted me to be able to TA the course. Hillegass hired all of his employees himself, and trusted them to take on new challenges, and frequently encouraged members of his engineering staff who were interested in teaching to deliver guest lectures in preparation for becoming fully-fledged instructors. Hillegass extended this opportunity to me as well. I had enough expertise to help out novice students, yet not quite so much of an expert as to be the instructor myself. I was thus able to observe what techniques and concepts students had trouble with and how they felt, and my interaction with them caused me to be one of the people they relied on to get through the class. This meant that a number of the novices became very attached to me as I spent significant time helping them, and had a very positive opinion of my expertise. However, because I

was still not as knowledgeable as the instructor, there were things that I needed help with myself, and was learning as I went. Being a student in the Android and OpenGL bootcamp courses provided me with a different perspective and additional insights, especially into the emotional experience of the course.

On every trip to Historic Banning Mills, I would leave the Big Nerd Ranch's offices by car late Friday afternoon, and drive east outside of Atlanta for an hour on Interstate 20, exit, and drive for 30 minutes along a small, two-lane highway, through densely wooded countryside. Eventually I would arrive at HBM, a beautiful, and rustic, retreat and conservation center which preserved the site of a historic 19th Century mill. As I drove in towards the main lodge building, I would pass by small wooden bungalows, a swimming pool, and a couple of wooden towers and platforms for ziplining, which was the major attraction at Banning Mills.

Entering the lodge to register, I passed by a rustic-looking hallway with captioned photographs, as if the place were a kind of museum crossed with a hunting lodge. After emerging through the initial entryway, I found myself in the main room of the lodge, with a small waiting area to my left, where the registration desk was. I felt that the entire room, which included a gift shop on the left, had a kind of kitchy Western feel to it. What gave me this impression most were the stuffed small game decorating the lodge around the fireplace and chimney, and the mountain lion hanging prominently from one of the rafters. This was further accentuated by a horse-drawn carriage past the dining area on my right. Aaron Hillegass had definitely chosen a place that evoked the American West to host his Big Nerd Ranch courses. I was later told that this lodge was not the original building, but had been rebuilt after a 2006 fire burned down the original lodge. Thus, although the ruins of the historic mill elsewhere on the property were real, the lodge itself was a simulacrum of the 19th century American frontier experience.

This experience was reinforced after I registered with the young woman at the desk, who gave me my key and a map of the grounds allowing me to find my cabin in the woods. There was still enough daylight that I did not have to go fumbling

around in the dark (which I would have to do later that night on my way back). The standalone wood cabin had all the standard amenities—bathroom, kitchen, running water, bed, and an air conditioner, but everything had a very rustic feel, and the entire structure, and most of the furniture, was made of wood. I turned on the TV, and noted that it only showed a channel cycling through the resort's available activities, which included not only ziplining, but hiking and horseback riding. I checked my cellphone reception, which was not very good. I would essentially have no Internet or broadcast television, which successfully cut me off from the outside world and brought home the feeling that I was attending a retreat. This sense of seclusion was an intentional part of the design of the Big Nerd Ranch bootcamps. Aaron Hillegass felt that what made his courses work was the fact that students, like religious people on retreat, left their daily lives (including, for most, their families and significant others, though occasionally one showed up with a partner) with their attendant distractions, in order to focus all of their attention on learning programming in a kind of deep immersion program, like learning a natural language. The intensity of the courses, betrayed by the military allusion to "bootcamp," was hit home by the regimented nature of the week's schedule. However, as there was also a "dude ranch" and vacation element to the class, there were times allowed for hiking, ziplining, and other recreation on the grounds. The fact that all of the students experienced this together, both the intense classes and the occasional recreation, created a bond of camaraderie between them that often lasted after the course was over.

After locating my cabin, I went back to the main lodge for dinner. When I got back, the shuttle from the airport had arrived with the majority of the students. Dinner was being served at the long table, and plates of salad had already been placed at each seat. Dinner came later, in large, extremely hearty plates of rich southern food. For drinks, there were ample pitchers of lemonade and sweet tea, and a table nearby had bottles of wine that a few would try. People were trickling in and taking their places, and as we ate, we would strike up conversations and introduce ourselves to the people seated immediately near us.

It was at dinner that I would have my first opportunity to meet the students who I would be spending the rest of the week with. The demographic makeup of the classes usually corresponded with the general makeup of programmers in general, though with a small sample of size of about 30 per class, these must be taken to be more my impressions than reliable statistics. Most of the students were white men, who were professional software developers, ranging in ages from early thirties to late forties. About ten percent of the iOS and Android classes were women, and this was fairly consistent across all the classes I attended, which I feel means that there is a significant level of interest in mobile development among women programmers. While Asians (primarily South Asian and East Asian) made up probably a third of the men, they often made up half or more of the women; in one class, the only woman was Indian. Latinos seemed to be present more often than African Americans, but both groups were very underrepresented. Students came from all over the United States, though because of Big Nerd Ranch's Atlanta location, a substantially higher proportion of students than one might expect from a technology class were southerners from Georgia, Tennessee or Alabama. Sometimes there would be international students—in the first iOS course, a group of three came from Amsterdam, where interest in mobile development and technology entrepreneurship was being stirred by a non-profit called "Appsterdam" started by Mike Lee, an American Cocoa developer and transplant to the Netherlands. One student later turned out to be Argentinian.

Historians, sociologists, and anthropologists of computing have written much about the gender disparity in computing and especially software programming. It is now well known that women were the first programmers of the ENIAC, (Light 1999) many having switched over from the Moore School's human computing department. Both Ada Lovelace and Grace Hopper are justly celebrated as pioneers of software. "Coding" was initially seen as lesser skilled than working on hardware, and working the card punch, like typing, was a feminized profession. Historian Nathan Ensmenger shows that in the 1960s, women were recruited to programming jobs through ads in Cosmopolitan Magazine (Ensmenger 2009). Gradually, however, as programmers

began to professionalize, computer science became an academic, mathematically-oriented discipline, and the masculine discourse of "software engineering" became mainstream with its male, manufacturing associations, the proportion of women in programming dropped sharply (Ensmenger 2010). Thomas Misa finds that the percentage of women graduating with bachelor's degrees in computer science dropped from a peak of 37% in the mid-1980s to between 11% and 15% around 2007 (Misa 2009, 6). Masculine association with technology, particularly the affective pleasures of tinkering, and the achievement of mastery, both at work and at play, is not new in Western culture, especially when it comes to electronics and radio (Haring 2003; Faulkner 2000a; Kleif and Faulkner 2003; Dunbar-Hester 2008). The rise of "nerd," "geek" and "hacker" identities, culturally stereotyped as a lonely, anti-social young male, however, have exacerbated these trends. More recently, a different kind of masculinity, the competition and crass-ness associated with the college-age young men filling the startups of Silicon Valley, the "brogrammer," has been highlighted for creating hostile work environments for women in the tech industry (Hicks 2012; Parish 2014). Given this enormous gender imbalance, it is not surprising to find that the proportion of women in the Cocoa community is low, and the qualitative impressions of some of my actors and myself suggest that the ratio of men to women in the Cocoa community might be worse than average in the industry. Nevertheless, the consistency of the fact that all three of the iOS classes I attended averaged 10% female students might indicate an upswing in women's participation in Apple development. The more advanced OpenGL class (a technology not specific to Apple) contained only 15 students, none of whom were women. The boom in mobile app development could be diversifying the population of Cocoa developers.

In my first iOS class, I got to know three students especially well, Walt, Victoria, and Anna, in part because I spent more time helping them than the other students. Walt was a half Scottish, half Jewish retiree from East Hampton, Long Island, who had paid for the class out of his own pocket, and looking to write an app during his retirement. He had been an engineer, but had not written programs since the 1970s or 1980s. Victoria was a Chinese-American woman in her late 50s, who

reminded me of the aunties at my Chinese church back home. Victoria worked for Smith College, doing SQL database work on their backend servers. Anna was one of the three from the Netherlands, two out of three of whom were women (the other Dutch woman also happened to be of East Asian descent.) Anna and her husband were running and independent app business, with her husband doing most of the coding, and she handling the user support. While she does have a programming background, she took time off to have a child, and was unsatisfied with not being involved in the technical work, so she decided to take this class. She has also been very active in trying to build relationships, both with the other Dutch developers sponsored by Mike Lee's Appsterdam, but also with the people she met at Big Nerd Ranch, not just her fellow students, but also Aaron Hillegass, who she met when he traveled to support Mike Lee's initiative and recruit him to teach a Big Nerd Ranch bootcamp in Amsterdam. These three students needed the most help in the class because they were all new to object-oriented programming, having only previously written procedural programs. This meant that they had a lot of difficulty matching the way they thought about programs (as processes) with the way that Cocoa was designed, which treated programs as collections of objects sending messages to each other. Victoria also had never used a Mac before, and was having trouble simply getting used to the user interface. Another relative novice was Nathan, a businessman who was trying to create a new social networking app for inviting friends (real friends, not social media "friends") to parties. He was taking the class not so he could write the code himself, but so that he could better evaluate the work of people he was hiring to write the code. He asked a lot of questions during lectures, but did not ask for as much help during lab times, probably because he was not as invested in actually getting his apps working or learning everything. In a later class, one of the novices was Lincoln, a young Caucasian who had been in finance but wanted to do something different after the financial crisis.

I got to know a few of the more advanced students in conversations over meals or during breaks in instruction, when we took hikes out on the trails behind the lodge. Also, on Thursday, about half of the class took several hours off to take the

zipline tour, which became a big team-building experience for those of us on it. One of these was Chris, an Asian-American in his late 30s or early 40s who worked for the FBI. He and Andrew, a Caucasian developer, were both used to using Java and a development environment called Flex, while another, Sean, was used to Microsoft's C# language. These were both object-oriented languages, which meant that for these three developers, learning Objective-C and Cocoa was more a matter of translating similar concepts that they were already familiar with. Another was Sunil, an Indian who had written COBOL for mainframes for the Indian agriculture department, and worked to develop local villages, sometimes working with cows. Tomas, the Argentinian, wore a T-shirt with a giant robot from Japanese cartoons on it. Both Sunil and Tomas used and were familiar with Macs. In a different class, one of the most advanced students was Parvati, an Indian woman who worked for Microsoft but wanted to learn something new and move back to India. There was also a Southern woman, Julie, who was a web developer and knew all the markup languages, and said she was used to "memorizing languages." She called herself a rabble-rouser, and liked to ask the instructor trick questions. Another notable student was John, a Caucasian contractor for the Air Force, who proudly displayed a sniper rifle as his desktop picture.

As I attended the iOS bootcamp multiple times, I also got to experience the course being taught by different instructors. Every time, there would be a different instructor for the weekend portion of the class, the Objective-C primer, from the main 5-day iOS bootcamp. The first time I attended the class, the Objective-C instructor was Mark, a freelance web developer who ran his own web design consulting business and helped write the Big Nerd Ranch's website backend. He was not a permanent Big Nerd Ranch employee but was contracted to teach as needed. Mark had a wry and edgy sense of humor, and because he was not a full-time employee, often proclaimed opinions different from the other instructors. The main part of the course was taught by Joe, the co-author of the Big Nerd Ranch iOS book and course materials, who had been the original designer of the course. Joe was atypical for a programmer. He was an athlete who took several months off in the

beginning of each year to train track and field for the Olympics. He had been hired straight out of college by Aaron Hillegass, and with a hypercompetitive and masculine personality, fit more of a jock archetype than a geek. He was very opinionated and did not hide his disdain for what he considered to be stupid practices. Aside from myself, there was also another TA, Brian, a recent hire who was in training to become a full instructor himself. Brian had been a religious studies major in college, an Apple fanboy, and had worked at the Apple Retail store before being hired at BNR. The second and third times I attended the course, the weekend primer was taught by Step, an employee who had joined the company between my 2011 and 2012 field visits. In 2012, I worked very closely with Step on Big Nerd Ranch internal app projects. Step was an older, more experienced engineer than Joe or Brian, a family man and a devout Christian. Step had spent a significant portion of his engineering career in quality assurance (testing) and thus was especially interested in promoting and using disciplined software engineering practices and methodologies to help improve software quality. The main instructor for the course that week was Christian, a young Georgia Tech graduate who was also a former Apple Retail store employee and Apple fan. After Joe left Big Nerd Ranch in 2013, Christian would take over the job of revising the iOS book and became the third co-author.

Classes took place downstairs. At the foot of the stairs were located two restrooms, as well as doors to two bedrooms. Typically, the instructor stayed in one of these two rooms, though occasionally a student would be booked there. To the right of these rooms, two more doors opened up onto the main classroom. This was a large, wide presentation room. On Friday evening, I helped the instructor set up the classroom. In the middle, on the far wall of the room, we set up a projection screen, which would be used for slides or live demonstrations on the computer. Large windows on either side looked out into a small grassy slope, which led down to a forested stream. During the next few days, we would be able to see, and to hear, zipliners through these windows. To the left of the screen was a small desk for the instructor to sit and place the laptop that he was projecting from. To the right we set up a small whiteboard on a stand for drawing diagrams.

Facing the projector were three rows of long tables, which we covered with a green tablecloth. The projector itself we placed between the two front-most tables. In the middle, the rows were parallel with the wall, but on the sides, past the support pillars holding up the ceiling, the tables angled towards the wall, so that overall, each row had a trapezoidal shape. Each individual table would seat two. From a closet near the door, we retrieved the items we would need to set up the classroom. On the floor in front of the tables we placed power strips and extension cords that we taped down with duct tape to prevent the students from tripping. Most students would come with their laptops. However, a few needed a computer set up for them, so I helped set up three all-in-one flat-panel iMac desktop computers. The closet also contained extra course materials and T-shirts for students.

As with the room upstairs, displays and decorations around this classroom added to the frontier ambiance. At the back of the room, just next to the door, was a stone mantle. Below the mantle, at the lip of the fireplace, was a rug made from the skin of a big cat, and two softly cushioned lounge chairs sat on either side. At some point during the week, this cat rug would become host to Big Nerd Ranch branded goodie bags, consisting of a coffee mug, T-shirt, a pen, and bumper sticker.

Atop the mantle was a portrait of Chief McIntosh, a local half-white Cherokee chief who was killed by his fellows after selling their land to his white relatives. The room was named the McIntosh room in commemoration of him, a striking coincidence to me, as this was an Apple programming class. One afternoon I found myself reading the chief's story, or rather, two slightly differing accounts of it, one portraying him as a traitor to his people, and another more nuanced, on two historical wall displays. Two other displays around the room told the story of the Trail of Tears of the Cherokee and other Native American tribes of this part of Georgia. Upon reading these, I felt a sense of irony; here we were at a place where mostly white American knowledge workers, privileged members of the ruling class, were perpetuating themselves, and the violence of the triumph of whites over the indigenous population was commemorated on its walls, with the natives mostly

invisible and relegated to history, aside from a chief who sold them out, for which he was remembered by whites. I never did meet a Native American student in the class.

At 10 pm, the instructor and I closed up and headed back to our cabins. The next morning, a hearty breakfast was served, buffet style, in the dining room. This consisted usually of scrambled eggs and bacon or sausage, but sometimes, chicken and waffles. There was also cereal, yogurt, orange and apple juice, and lots of fresh fruit, which I took advantage of. I was not able to finish my fruit before class was supposed to start at 9 am, so I took my plate downstairs.

Students had chosen their places for the week, and had hooked up their laptops to the power strips, and placed their nametags, which they had received, on the table in front of them. I choose an empty seat on the right side.

Mark, the instructor, began his lesson with an introduction, saying that he was going to teach the class the idioms of programming in the Objective-C language. He was going to teach them what they needed to know, but more than that, he was going to teach them the jargon on the first day, and then the foundations on the second. This language of "idiom" reflected language in Aaron Hillegass's books and course materials, and seemed to indicate a particular style or way of thinking with Cocoa that differed from other programming environments.

Because the course was hands on, the first thing to do was to get all of the students familiar with Apple's programming environment. This meant that they needed to launch a program called Xcode, provided free by Apple, in which they would do all of their programming and debugging. It was important to make sure that everyone was running the latest version of Xcode, however, so that everybody could follow the precise instructions in the course materials. Because most people brought their own Mac laptops, and did not necessarily have the latest version, as the teaching assistant, I went around sharing a copy of Xcode on a thumb drive. The WiFi network was slow and unreliable, and we did not want 30 students to be downloading 5 GB from Apple all at the same time, which would take hours. This was important because as we will see, time was precious and could not be wasted.

Xcode is Apple's "Integrated Development Environment" or IDE, an industry term that denotes a program that collects a code editor, compiler, debugger, and other programming tools together into a single, integrated application, so that the programmer rarely has to switch to another program to accomplish a task. IDEs such as Microsoft's Visual Studio, and Eclipse for the Java platform, have become the dominant method of programming on graphical operating systems. This is in contrast to the practices of Unix programmers, who typically use dedicated programs for each purpose: text editing, compiling, and debugging on Unix are all separate programs invoked from the command line.

Each day's class was typically divided up into approximately 6-8 lessons, each lesson lasting anywhere from two to three hours. The first twenty to forty minutes of each lesson was usually a lecture. In the main iOS course, the lecture was usually given on slides. However in the Objective-C primer, the lecture usually involved a live demonstration in Xcode. In the demo, instead of presenting an abstract idea, the instructor typed in some code in the IDE, and clicked a button called "Build and Run" which compiled the program and ran it. At the bottom of the Xcode window was a console that would output the result of the program, typically some text. For the Objective-C primer, which was the topic for the weekend, all of the programs we would be writing would be the simplest kind possible—text-only programs, often only a few lines of code, that ran on the command line, in Mac OS X itself. After this lecture period came the hands-on lab time, in which students were asked to try out an exercise from the course materials, either the official book, or if the materials incorporated new or revised content, were printed booklets in spiral plastic binding. (Unsatisfied with the Objective-C books on the market, which were more reference manuals than tutorials, Hillegass wrote his own Objective-C book to serve as course materials, which was not ready for 2011 but was in use in 2012.) The first thing students did was to create a new project in Xcode, using a blank template. In some later lessons, which built on finished work from a previous lesson, they could continue from their existing project. Then they could start typing in code that was printed in the course packet, build it, and run the program for themselves, and

see the results. If they typed in something incorrectly, the compiler would give them an error, with a cryptic message, and highlight which line of code was the source of the error. During this lab period, the instructor and teaching assistants, myself included, roamed around the room looking for people with their hands up, indicating that they had run into such an error. I spent most of these lab times helping students debug and fix their programs and answering other questions. Most of my interactions with students were either in these lab times or in conversations during meals.

The instructor kept track of time, and around the two hour mark would call for an end to the lab time, whether the students had finished their exercises or not, in order to begin the next lesson. As the schedule called for the course to get through a certain number of lessons every day, the instructor could not afford to give the students too much time, and had to keep things moving. This created a relentless pace in the course, felt especially during the main iOS portion of the course, which, I will discuss later, is a central part of the Big Nerd Ranch bootcamp experience.

Dinner was scheduled at 6 pm, though sometimes we ran late and started at 6:30. After dinner, additional lab time was scheduled in the classroom for students who needed more time to get their programs working. These after-dinner lab times were purely optional, and students often took this time to go hiking or simply rest back at their cabins. However, given the pace of the course, generally at least half of the class, sometimes more, would come in to work after dinner. During these times, more advanced students could also use this time to complete special “Challenge” exercises at the end of each chapter. The instructor also encouraged advanced students to work on their own iOS projects, for either work or their own hobby, in order to apply what they had learned to something they really cared about. All of the instructors made it a point to say that it was this application of learning from the course to a student’s own project that really cemented the learning. The weeklong bootcamp by itself went by very quickly and if a student did not quickly apply it to his or her own work soon, in the weeks after the class, the hard-won knowledge might not fully sink in. This exhortation had the additional effect of reinforcing the moral message that good programmers should want to spend their free time

constantly programming and learning, that programming is not just a profession but ought to be a vocation and a passion.

Learning through Typing

Instructors at the Big Nerd Ranch bootcamps used a number of pedagogical techniques for teaching programming. The most important was to have students manually type in code out of the book into their own programs. During lab times, students worked on exercises specified in the book, which asked them to implement some new functionality for the program they were constructing. Rather than ask them to figure out for themselves how to express an algorithm in Objective-C, the book simply provided them with the answer. The code to do what they needed was printed right in the book. The intention of the class was not to teach people how to program, but mainly get students familiar with the feel of programming in Objective-C using the Cocoa APIs. What the instructor asked them to do was to type this code into their program manually, rather than copy and paste it out of the solutions we provided on a flash drive. This was crucial because the repeated typing of certain codes, such as the names of objects, functions, and APIs from the Cocoa frameworks, such as "NSArray," "NSString," "NSDictionary," "alloc," "init," "retain," "release," built up a vocabulary that could eventually be immediately recalled, and the meaning of which, understood. Because many Cocoa APIs follow consistent, grammatically-based naming conventions, such as "objectAtIndex," "valueForKey," "setNeedsDisplay" or "selectionShouldChangeInTableView," repeated typing of such code also familiarized students with these conventions, which meant that using previously unknown APIs gradually became more familiar and less foreign. I began to understand that it was partly in this sense that Aaron Hillegass described Cocoa to me as an "idiom," with its own "style." We will discuss the importance of coding style later.

What happens if a student tries to take a shortcut through typing and cut and paste instead? At one point, Sunil attempted to do this, which I discovered when I helped him find a bug in his code. He had taken a bit of code he had previously

written for an earlier lesson, copied it into the current program and modified it to make it work. Unfortunately, he had not caught every detail that needed to be modified. One of the objects he was using was the wrong type, something that he might have caught if he had retyped all the code out from scratch. The result was that the object did not respond to the message he was sending it. This was a "view" object. In Cocoa applications, windows contain a hierarchy of "views" which display their contents to the screen. A window can be thought of as a collection of these views, which can contain other views. The result of Sunil's mistake was that his view was not getting added as a subview of another view, but was being added directly as a subordinate of the window itself, causing it to not be able to scroll.

Manual typing is a highly disciplinary pedagogical method. The repetition of manual typing builds not only a visual, conceptual, and linguistic familiarity with the code, but also a material one, of muscle memory, through typing the same code over and over again. A repertoire of codes is built up, not only in the programmer's mind, but also in his or her body, in the muscles of the fingers. Later on, the programmer is then able to draw on this "vocabulary," supplemented by documentation, to more quickly write code for an original program. This recalls Warwick and Kaiser's example of jazz improvisation in their "Foukuhnian" model of pedagogy. Jazz improvisation, a creative activity, is "pedagogically conditioned" by a repertoire of embodied skills produced by disciplined, repetitive practice, which the musician draws on during improvisation. What is similar about both programming and jazz improvisation is that both are creative activities conditioned by embodied practice, in accordance with Kuhn's view that conceptual knowledge arises materially through practice (Warwick and Kaiser 2005, 401). Rachel Prentice's work has similarly shown the interconnection of embodied practice with knowledge acquisition in the training of surgeons (Prentice 2013).

Pacing

Another aspect of the course that tied together with manual typing was the relentless pace of the lessons. While the two days of the Objective-C primer had

been at a relatively leisurely pace, to introduce students to the language gradually, the iOS course intended to cover almost the entire 300+ page book in five days, which required each day to cover six to eight lessons. This was a grueling pace. The course began at 9 am, right after breakfast, and covered at least two lessons in the morning, with lunch starting at 12:30. Class recommenced at 1:30. After two more lessons, in the mid-afternoon around 3:30, the instructor usually led everyone outside into the hot, humid Atlanta summer for a 30 minute break, to go hiking on the trail down to the stream, through the woods, to see the historic mill. At 4 we resumed for another two lessons, until 6, when dinner was served. After dinner, from 8-10, the instructor and the teaching assistants went back downstairs to help students who wanted to continue working during the free lab time. Many did, because they were not able to get their programs running correctly during the day.

Although he first lesson was standalone, in most subsequent lessons, the program written in each exercise would be continued in the following lesson, so that over the course of four or five lessons, what had begun as a relatively simple program had acquired a large and sophisticated set of features. This helped to build confidence in the students, teaching them that they too could build sophisticated apps in gradual, piecemeal fashion. This also resembled real-world practice, as an existing program is gradually added to over time. The drawback, however, was that students who could not complete an earlier lesson, usually due to not being able to find and fix all of the bugs in time before the next lesson had to begin, would be left behind. And indeed, most students did eventually fall behind; the novices would be struggling by Tuesday, while the advanced students would gradually be overwhelmed by Thursday. During my first time at the bootcamp, most students on Thursday were still working a chapter behind, trying to fix their programs from the previous lesson despite the instructor having moved on. I was helping one novice student, who was lagging back three chapters behind. The instructor and I did provide solutions for each chapter, distributed on a flash drive on Monday at the beginning of the week, and also available on the company website, that contained snapshots of the program at each stage. Students who fell behind could begin the

next exercise from a completed and working version of the previous exercise. The students thus had all the "answers" available to them right from the start. The point was not getting the right answer, but going through the process, and learning from the experience.

Victoria, one of the novice students, remarked at one point to Joe, the instructor and co-author of the book, that she was happy to get to the Challenge at the end of each lesson, because that meant that she was done, not that she wanted to tackle the challenge. Some lessons also contained additional technical or historical information on older practices that had been made obsolete by new changes made by Apple to the Cocoa frameworks. These went under the heading, "For the Curious," to signal that they were not necessary. Another student, Phil, piped in, "same with the 'For the Curious section!' I'm not that curious, I'll just skip that!" Joe protested, "That's some of my best work!" We all laughed.

Challenge exercises, which came in three grades of Bronze, Silver, and Gold in progressively higher difficulty, allowed for the course to address the wide range of skill and experience of students taking the course, offering more advanced students a chance to exercise some creativity beyond copying code out of a book. These asked students to implement or extend a feature in the program they were working on for the regular exercise, but this time without providing them the code to do so. As we saw in the previous example, these were purely supplements, with no requirement or pressure to do them. However, I noticed that for many of the more advanced students, it was a point of pride for them to do as many Challenge exercises as possible, at least on the first few days when they could still keep up. Some of them might compare with each other, in the spirit of friendly competition, how many they had managed to accomplish. All of these advanced students were men.

More than one student remarked that the class felt like "drinking from a firehose." I got to experience this "firehose" myself when I took the Android programming five-day bootcamp, which I attended along with another Big Nerd Ranch employee, Mikey, who was an iOS instructor. Like many of the novices in the

iOS class, I felt as if my brain could not keep up with all the information, and I was simply typing in code that I could barely understand. The other students I spoke with at dinner said they were in a similar state. One student next to me said he was just typing and couldn't tell what was stuff he needed to understand and what he could safely ignore. Another student similarly noted that he would need to reread the book after the class was over. Mikey said that actually, we should all do that. Drawing from his own experience as the iOS instructor, he said that "the first time is more about learning the vocabulary, while the second time is when you start to make connections."

In my field notes from Tuesday night of the class, I wrote:

> I think that a lot of the bootcamp is simply learning through immersion. By forcing us to type so much code in ourselves, and just keep the information coming, eventually the material from previous days, even if we didn't fully understand it then, begins to look familiar by day 3 and 4, just from familiarity, even if we aren't completely sure what it means, we no longer feel lost reading the syntax.
>
> I definitely feel as if I'm learning a new foreign language, or at least a new "idiom" if that is the correct word.

My own attempt to catch up after dinner Tuesday night failed, when a woman working for the resort came at 11:40 pm to close the room up. I had been typing non-stop for the last two hours, and I was not alone. Even though I had, unlike the previous day, gotten enough sleep and had been alert most of the day, the concepts still went by too fast to sink in. I had been treading water but could not keep it up, as the sheer volume of code I had to type was enormous. I wasn't even bothering to understand what I was typing, I was just typing as fast as I could, and my wrists were getting sore. I felt bad, wondering if I was the only one who was so behind, but at dinner in conversation with the other students, we all commiserated over being in the same situation. Later, the Android instructor, Brian, reassured me that I was actually still ahead of some other people. It seemed as if one of the effects of the course was to make everybody feel dumb, inadequate, and behind. Only one student, I learned the next day, was actually ahead—he had gotten up to code at 5 am. It was in this relentless pace that the course lived up to its name as a "bootcamp." This shared

experience also had the effect of bonding all the students in it, who felt as if they had survived something tough. Indeed, at the end of the week, the instructors handed out T-shirts and bumper stickers and mugs, some of which proclaimed, “I survived!” Big Nerd Ranch swag was much prized among many Cocoa developers as proof that one had gone through this rite of passage.

And indeed, I observed that the students in the iOS bootcamp went through much the same experience I did in the Android class. Phil remarked Wednesday evening that so much material was getting crammed in his head so fast, that he probably wouldn’t remember half of it after the class was over. Jenn, the Big Nerd Ranch course manager who had come in to visit the class, remarked that that was OK, that was what the Big Nerd Ranch alumni forum on its website was for. This allowed students who had gone through the bootcamp to keep in touch with each other afterwards, continue to ask each other and the instructors questions, including answers to the Challenge exercises. Phil then said that he probably wouldn’t have any feedback on the course until three weeks after the class, after it all sank in. Anything he said before that would just be gibberish, as if he was speaking in tongues.

Thursday morning Phil and Nathan were joking with each other over breakfast about how behind they all were. Phil had finally had to throw in the towel and resort to starting chapter 11’s exercise from the chapter 10 solution, something that he had resisted doing for as long as possible, probably out of pride. By Thursday, most students were no longer attempting to do Challenge exercises; it was enough to simply to be caught up and not have to use the solutions. A student named Mark had been managing to finish every lesson on time through Wednesday, but gave up on Thursday. Nathan and Phil were talking about all the parts in the book that they would have to go back to and reread after the week was over. Phil said he wished he had color-coded bookmarks—he would put green for stuff he knows he will use immediately, red for things he knows he will need to review. Nathan was also writing lots of little notes in his book, often adding extra annotations in separate

places to point back to the book. Phil said he wished he'd taken a picture of the book before the class started, when it was clean, because his copy was now all marked up.

In summary, no matter the level of the student, novice or relatively advanced, the relentless pace eventually caught up to them. This had a number of emotional effects. Students felt as if they were constantly behind or struggling to keep up, generating feelings of inadequacy in many. For the most advanced students, it hit home the message that Cocoa programming was hard, and that there was a more to learn than they could possibly do in a week. This difficulty also motivated students to help each other, even though their exercises were individual, highlighting the communal aspect of learning. Moreover, the shared experience of going through this difficult class together, in this remote place, created communal bonds between the students in the course, and their dependence on instructors for help created bonds with them as well. Additionally, in the iOS class, Aaron Hillegass himself often visited sometime during the week to chat with students, hear their concerns, and sign their books. Although the Big Nerd Ranch did not provide certification recognized by any professional body, "alumni" still felt proud to receive their little printed certificate from instructors at the end of the week. Students came away from the class feeling a sense of community with both each other and with the people of the Big Nerd Ranch.

Documentation

The classroom environment, with the instructor and teaching assistants, provided a safe environment for students to fail gracefully, as experts were available to answer questions and help them when they got stuck. However, they would not have this resource forever. With thousands of APIs, how was a student to discover them all? Even the instructors did not have this encyclopedia of knowledge memorized. Therefore, another crucial skill that the instructors wanted to teach was how to look up information on your own, in Apple's documentation. These could be accessed directly in Xcode itself, or accessed free on the Web. Both Mark and Step spent a section of a lesson early on the first day explicitly showing the students how

to get to Apple's documentation, and how to navigate it to find information about the object classes and interfaces that a student might want to use in their programs.

Looking for documentation to find an API was necessary, but if a student had already used an API previously but simply did not remember the exact code to type, looking it up in the documentation could be tedious. For this purpose, the instructors also showed students how to use Xcode's "autocomplete" feature. Because the Xcode environment is integrated, its compiler can search through the Cocoa libraries that a programmer links to in his or her program in real time. The result is that, to type "NSString," a programmer only needs to type the first three letters, "NSS" and a list pops up of all the object classes that begin with those three letters, and the programmer can then select NSString from the list with the mouse or arrow keys. This facility is always on while a programmer is typing code. Thus, another benefit of the exercise of repeatedly typing in code is that programmers begin to learn how to use autocomplete as a shortcut to typing in long codes.

Debugging

Both typing out code, and the relentless pace of the course, contributed to the experience of debugging and fixing their programs, where students actually learned why it was important to follow the practices and conventions they were being told to use, and where the abstract concepts of the course became transformed into practical knowledge. Theoretically, because students were merely copying code out of the book into their programs, the lab exercises ought to be trivial. Practically speaking however, students would mistype something, and the program would either not compile, or worse, it would compile, but would crash into the debugger. Typing out code was not itself the primary vehicle through which learning was to take place; rather it was a means to create situations that would develop the critical skill of diagnosing and fixing errors. After making the same mistake repeatedly and seeing the same error messages printed out, students gradually began to learn what the errors meant. Of course, upon seeing an error for the first time, how would they know what to do? In that situation, novice programmers often feel completely

helpless. The novices in the class frequently asked me for help in the first three days, as if they felt that they could not make any progress without aid. Some students in the class mentioned that they had bought Hillegass's book and tried to work through it previously on their own, but had struggled. For some, it was a constraint of time and priorities—with other obligations to attend to, they simply could not focus on working through the book. Others, however, became frustrated because they ran into difficulties and did not know how to get past them without help. The classroom setting of the iOS bootcamp addressed both of these problems. First, being sequestered for a week out in the woods with spotty internet access took students outside of their regular lives and allowed them to focus their entire time on learning Cocoa. Secondly, because we had an experienced instructor and teaching assistant around who could help them identify and fix their bugs, we provided them a safe place to fail. This safety net allowed them to feel that bugs were not scary but fixable, because we could explain why a bug happened and how to fix it again the next time. Mark, the instructor, noted "That's the advantage of a class—if you're on your own, you stare at something for days, can't figure it out, and say, 'Apple sucks, I hate them!'" (Field note, July 9, 2011) This quote reveals that Mark understood the crucial affective work that the Big Nerd Ranch course provided, which was that by helping students get over the difficult learning curve of Cocoa, it would also help transform them from frustrated Apple haters to Apple lovers. And indeed, this also explains why the Big Nerd Ranch does not see its books, which cost between forty to sixty dollars, as cannibalizing sales of the courses, which cost around $3500. Hillegass understood that the books, as good as they are, are simply not a substitute for the kind of intensive, immersive, communal, and affective learning that the course provides. He saw the books, which were not a large share of the company's revenue, as more of a marketing tool to spread the word about Big Nerd Ranch, increase his reputation and that of his company in the community, and create awareness of the value of his courses. The availability of instructors (as well as other students) to help students debug their programs and overcome frustration is a key component of the class that a person working through the book alone is missing. In

what follows, I will present a few of examples of such instances in which I helped (or failed to help) a student debug a problem in their program.

During one lab session, I spent twenty minutes trying to help Tomas figure out why his application was crashing. In this application, we had a tableView, which if you remember, is a view that displays a table of data, like an Excel spreadsheet. Recall from the previous chapter that tableViews follow the Model-View-Controller design pattern, in which programs are split into model objects that represent the data, view objects which display this data to the user or provide ways for users to modify the data, and controller objects which coordinate user actions and data updates between views and models. In this particular application, the tableView displayed a table of "Possessions," which was represented in the model by a linear array of Possession objects. The Possession class of objects was defined by Tomas as a custom class, but rather than write his own tableView class, he simply used the stock UITableView class provided by the built-in UIKit framework in Cocoa Touch. UITableView provided a way for users to easily add a new item to the table. Tomas connected this mechanism to his own method in his custom controller class, "addPossessions:" which would add a new Possession to his array of model Possession objects.

The trouble Tomas ran into was that the name of his method, "addPossessions:" wasn't being recognized as a valid method name.[34] In the debugger, I typed a command, "bt" that printed out the "backtrace" or "stack trace," which was the chain of method or function calls that had led to where the program currently was executing, in order to figure out in what code the crash was occurring. Unfortunately, I did not have the experience to read the results. I had to wait until the

---

[34] In Objective-C terminology, the "selector," a text string representing the name of the method, was not being recognized as valid. This check is made dynamically at runtime.

instructor, Joe, was free to ask him to take a look. Joe immediately saw what was wrong, and pinpointed the problem as a memory management issue.

To understand the bug, I need to explain Objective-C's method of memory management, which involves a concept known as "reference counting." Objects that need to remain in the computer's memory are ones that are referenced, or "owned" by another object. When object A wants to make such a reference to keep another object B in memory, it sends object B the "retain" message, thereby incrementing B's reference count by one. Once A no longer needs B, it sends it the "release" message, decrementing B's reference count by one. Objects with a reference count of one or higher are kept in memory, but once their reference count drops to zero, the system frees up the memory to be used by other objects. This system of "retains" and "releases" made up the manual memory management scheme that NeXT had created for Objective-C in the early 1990s, and had continued in Cocoa at Apple until 2011, when Apple introduced an "Automatic Reference Counting" scheme that made much of this work unnecessary.

Manual reference counting was not only tedious, but error-prone, as the students in the class discovered. The trick was that retains and releases needed to be balanced. If there were more releases than retains, the program would crash, the program would be trying to access an object that no longer existed. If there were more releases than retains, nothing catastrophic would occur, but there would be a "leak," an object that sat around in memory that did not need to be there, causing excess memory usage. (The latter was something that students did not need to fix immediately.) In Tomas's program, one of his objects was somehow being over-released. Joe told us to use the debugger to step through the code, which meant using the debugger to execute one line of code at a time, while counting the exact number of retains and releases to make sure that they matched up. It turned out that Tomas had left an extra release in his code from the previous version of his project from the last lesson. I learned along with Tomas that if a program crashed and the message EXC_BAD_ACCESS was printed by the debugger, this signaled that the program had tried to access a deallocated object, which almost always was the result of the

object being released more times than it was retained. The next time I saw this same crash with another student, Eric, I immediately suspected an over-release. I placed a breakpoint in the method "dealloc:" that gets called when an object is deallocated. A breakpoint causes the program to stop running at specific line of code. If Eric was not over-releasing his object, this method wouldn't get called, and the program would simply continue running. Eric then ran his program again, and lo and behold, the program broke in dealloc. Eric then examined his code and found that he was releasing one of his instance variables twice when he only needed to do it once. Soon afterwards, Walt had a similar problem as Tomas, except that he wasn't even getting to the crash. His navigation bar wasn't showing up, because he had also left in too much code from a previous exercise. A different object needed to be released in the new version, but the old code still released the old one, causing it to be released twice. After seeing this message repeatedly, I would become adept at identifying it almost immediately the same way Joe did. Similarly, the students would also begin to pick up the same patterns I was learning myself. This repetition was crucial to my own learning as well as that of the students, but interestingly, there was no need for the instructors to design exercises to produce this explicitly. They understood that the normal practice of programming and debugging itself generated such repetition, and thus, asking them to debug their programs would suffice.

Another bug that students frequently ran into was often the result of a misspelling. On another day, I spent more than half an hour helping Tomas figure out the source of another crash. Again, a tableView was involved. Remember that in Cocoa, tableViews follow not only the Model-View-Controller design pattern, but also the delegation pattern. Because the UITableView class is provided by Apple's UIKit framework, it cannot be directly modified to customize its functionality, and is used unchanged. However, it contains an instance variable that can be set to point to a helper object, or a "delegate." It then relies on this delegate to handle custom tasks that often depend on state that the program can only know when it is running, for instance, user input. UITableViews have two such helpers, one officially named the "delegate" to handle user interface behaviors, and a second one, the "datasource," to

handle data model coordination. In practice, many Cocoa programmers often designate the tableView's controller object to be both its delegate and datasource.

In Tomas's program, his datasource was not returning the tableView cell he had asked for, even though he had correctly implemented the cellForRowAtIndex:Path: method. We both scanned every line of his code but could not see what the problem was. Again, I had to call in Joe for help, and he managed to pinpoint that Tomas had miscapitalized the 'v' in tableView, spelling it "tableview," which meant that his variable names did not match. I was so close to catching it, but neither I nor Tomas had seen it. Tomas said, "I knew it had to be some obvious thing." Later Eric would run into a similar problem. He had missed the capitalization of a single letter in a tableView datasource method as well. The program had compiled correctly and run, but instead of crashing, it simply did not work as intended—one of his views just did not appear. Because the method was misspelled, it was never getting called, and nothing was happening.

One of the reasons such errors could be tricky to find was because in Cocoa, a lot of things happen dynamically, that is at runtime. In some other object-oriented languages, like C++, things like whether variable names match are locked down when the program is compiled, which allows the compiler to check if they match. This means that such simple errors are caught early in the development process. A dynamic system like Cocoa defers many such things to runtime, because this allows the program to be significantly more flexible. However, it also means that bugs like this can slip through. Programs can compile and even appear to run perfectly fine for a while, until suddenly, one tries to send a message to an object, and the debugger crashes the program with a message saying that the receiving object does not respond to that "selector," in other words, the name of the message. This is often an indicator that the selector, the method name, was misspelled. Selectors can be any arbitrary text string that the programmer has chosen, and thus the compiler will not check to make sure if the selector names used in different places in the source code are consistent. If they do not match, the program will still compile and the programmer will run into the mysterious "does not respond to selector" errors. Such crashes are

actually a good thing, because they signal immediately to the programmer that something wrong is occurring. It is much more difficult when a message is simply sent to "nil," which represents the idea of "no object." It is perfectly valid to send a message to nil in Objective-C; the result is simply that nothing will happen. This allows for some interesting designs, but it can be frustrating to debug. Once again, this was an error that took me, and the students, many painful experiences to recognize. Such knowledge, even if explicitly stated in the course book, had to be personally experienced to be fully understood. Dynamic systems like Cocoa allow for more freedom and flexibility for programmers, both to do powerful things but also to make mistakes, manifesting in weird, difficult to trace bugs that might be the result of a mere typo. Rather than make such errors impossible mechanically through the compiler, the response by Apple and the Cocoa community was to codify conventional naming practices to make such errors less likely, and for Apple to regularly use such conventions in its own APIs so that programmers got used to them in an idiomatic way.

Misspellings were just as likely to occur to the more advanced students, as Tomas and Eric were, as to the novices, like Walt and Victoria, because the advanced students typed so much faster and were less meticulous, so typos often slipped through. Similarly, advanced students were often more likely to take the shortcut of copying and pasting code from a previous lesson than retype everything. This, as we have seen, was a frequent source of bugs, as mismatches between what objects the old code referenced and what the new code needed would remain even if the code successfully compiled, leading to many instances where an object of the wrong type was being messaged.

Sometimes students had problems not with the code that they typed, but with the graphical elements of Cocoa programming. One of the features of constructing applications with Cocoa is that the relationships between user interface objects (views) and other objects, such as their controllers and delegates, can be made graphically in the Interface Builder program (which used to be a separate program but is now built into the Xcode IDE). When viewing the app's user interface in

Xcode, the programmer can make a connection between objects by dragging with the mouse from one object to another while holding down the Control key. In this way, the programmer can relate what the user sees (the interface) with their representations in code. To be more specific, view objects in the interface, such as tableViews or buttons, are represented in code as instance variables owned by a controller object. These variables are given a special meaning to Interface Builder, called an "Outlet," and are tagged in the code with a special label, "IBOutlet" that signals to Interface Builder that they are special. This allows Interface Builder to allow the programmer to shift-drag from a controller object to the view object they wish, and select from a menu of outlets that have been defined in the code to associate with that view. (See Figures 3 and 4.) Developers can also create the view objects completely visually in Interface Builder and ask it to generate the correct code for them.

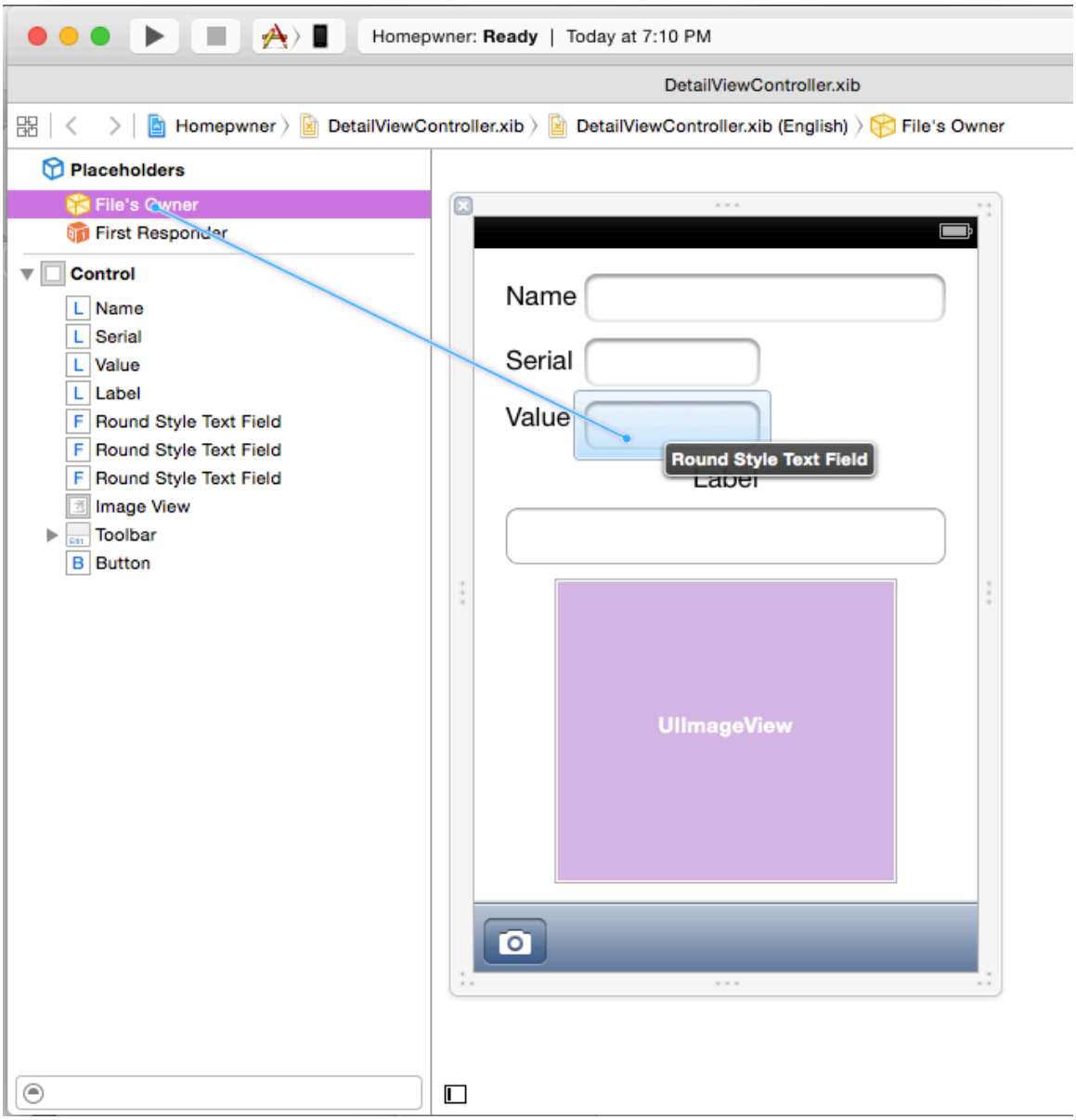


Figure 3: Wiring up in Interface Builder 1.

Control-dragging from a placeholder object named "File's Owner," representing the programmer's custom controller object, to a text field to in order to "wire up" the text field so that it responds to user actions, such as typing in it. This will result in the popup menu seen in Figure 4.

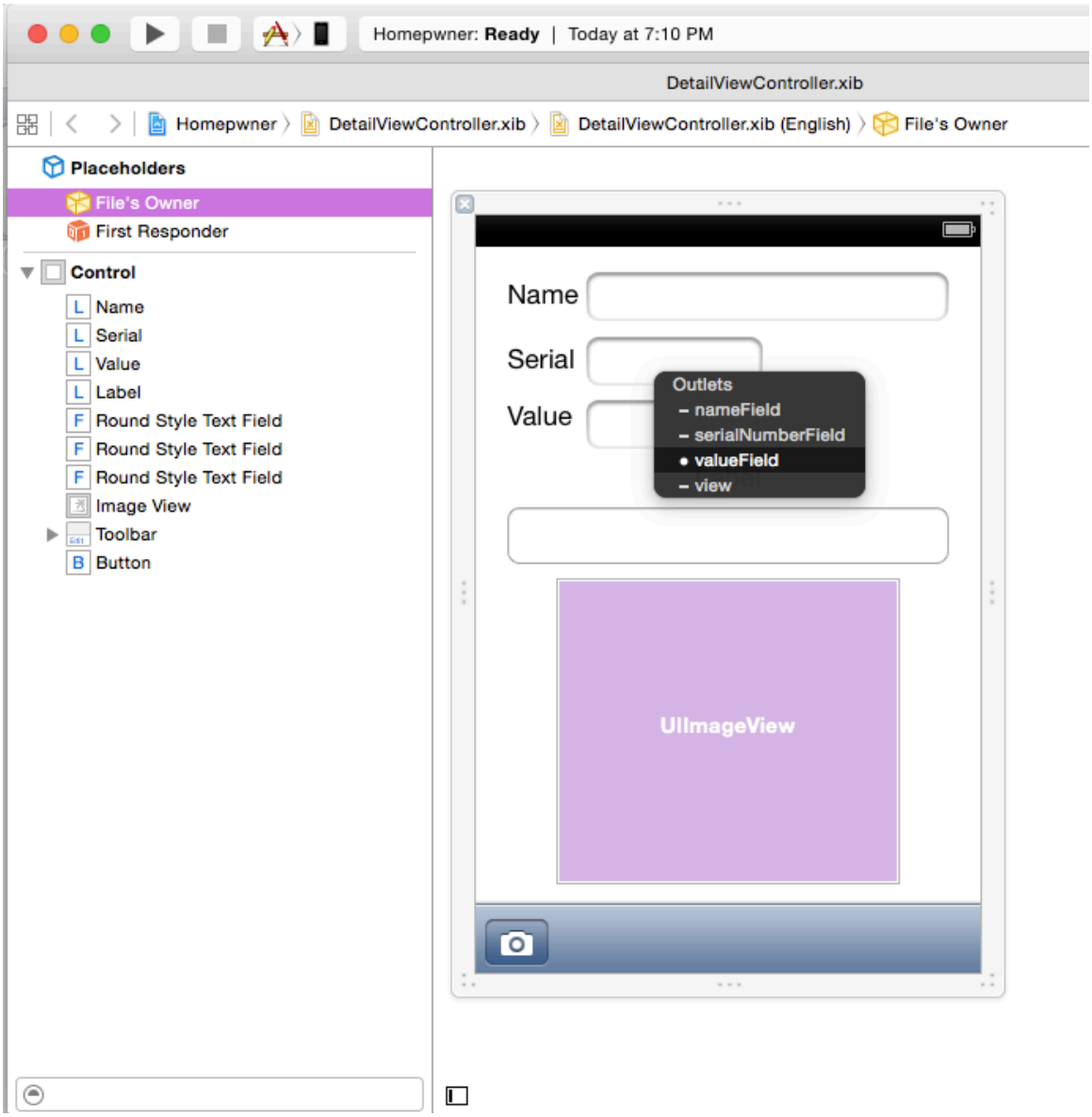


Figure 4: Wiring up in Interface Builder 2.

What the programmer sees on releasing the Control-drag. This gesture designates the text field next to the label “Value” in the graphical display to correspond to the valueField variable, or “Outlet,” in the code of her controller object, represented by the “File’s Owner” placeholder. The valueField Outlet in code is now wired up to the text field shown in the graphical display.

It matters what direction the developer drags. If dragging from a controller object to a view object, the drag means that the developer wants to associate that view with an outlet defined in the controller's code. What does dragging in the opposite direction mean? This is how developers define what happens when a user manipulates a view object in the interface, such as clicking a button, or entering text into a text field or table. When, say, a user clicks a button, the button signals to the controller object that it wants to run an "Action" associated with the button. An "Action" is just a special kind of method defined in the controller object; the code in it will run when the button is pressed. In the Model-View-Controller pattern, views are not supposed do anything on their own aside from displaying themselves or responding to user input; all behavior triggered by views should actually be implemented in "Action" methods in controller objects. "Action" methods must be labeled with the "IBAction" label, signaling to Interface Builder that they allow graphical connections to be made to them. Thus, when the developer drags from a view object to a controller object, he or she can select from a list of possible "Actions" for the view object to trigger. (See Figures 5 and 6.) However, dragging in this direction could also mean something else, if the view object itself contains "Outlet" variables. The developer could want to designate the view's delegate Outlet to be the controller object dragged to, by selecting the controller's delegate Outlet in the popup menu. Because both of these are possible, both "Outlets" and "Actions" will appear in the popup menu when the drag is released on the controller. (See Figure 6. The "Actions" appear under the label, "Sent Events," and one can tell that they are names of methods because their names all end in colons.)

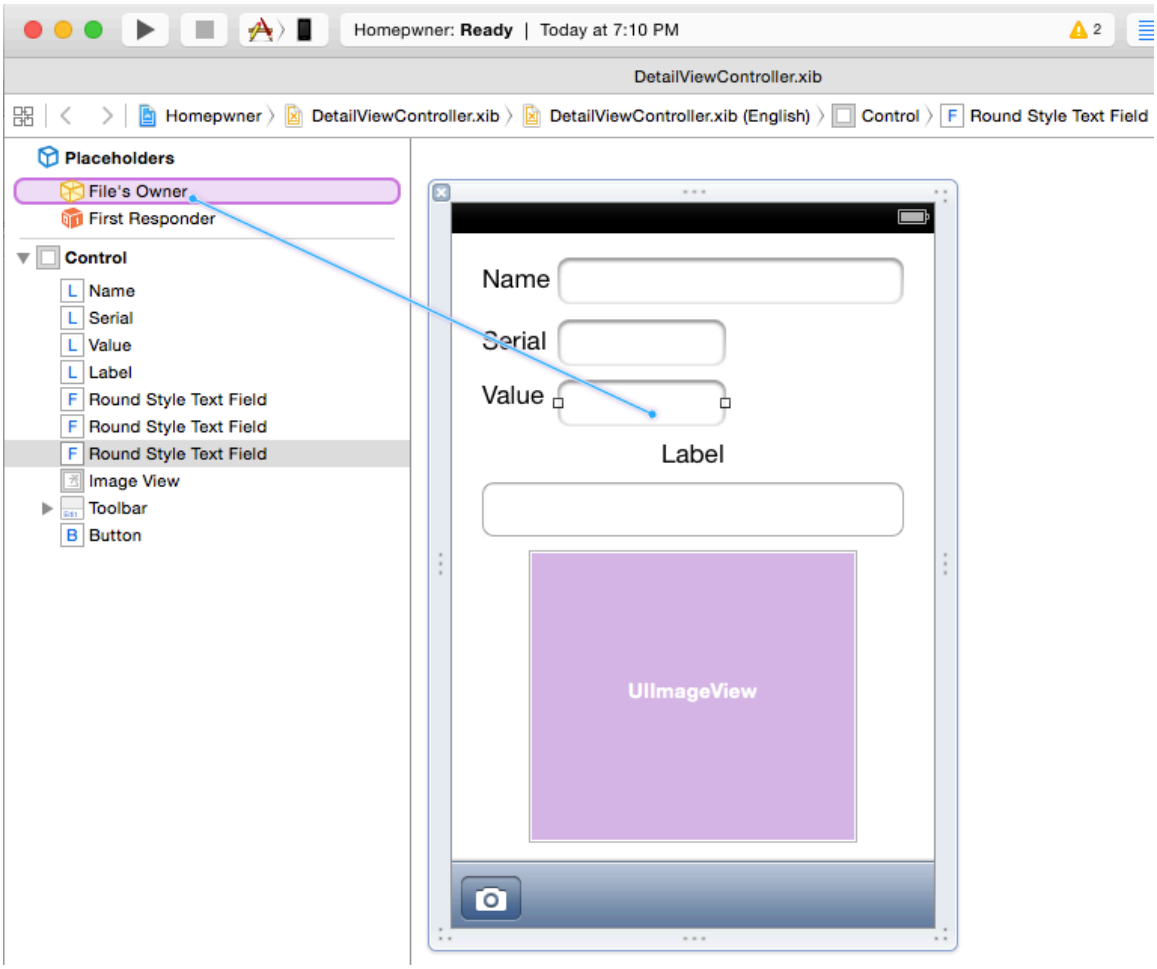


Figure 5: Wiring up in Interface Builder 3.

Control-dragging in the opposite direction, from the text field to the “File’s Owner” placeholder, which is the proxy for the programmer’s controller object. This will result in the popup menu seen in Figure 6.

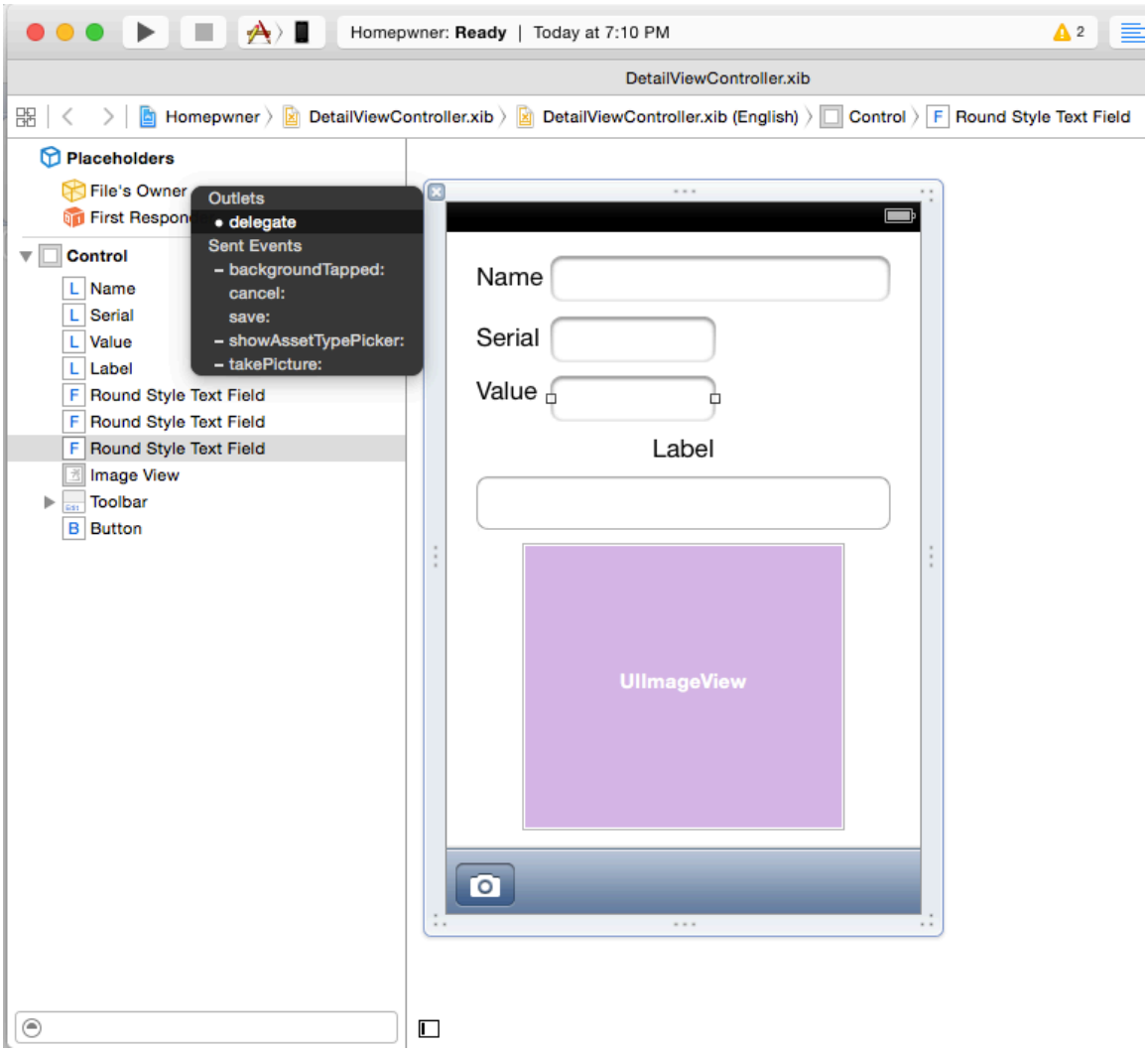


Figure 6: Wiring up in Interface Builder 4.

What the programmer sees on releasing the reverse Control-drag. The programmer has two options. First, the programmer can designate the "File's Owner," which is a proxy for the controller, to be the text field's delegate outlet. "Delegate" is the only Outlet available. Alternately, the programmer can select an Action method from the list of Actions in the "Sent Events" section of the menu to trigger when the user types the Return or Tab keys into the text field. Note that all the Actions have names ending in a colon, signaling that they are method names.

This was a problem that Anna ran into. She needed to designate that a text field in her application had a delegate, which was her controller object. Text fields conform to a protocol defined in the UIKit framework, that allows them to automatically handle user input, such as what happens when a return key is pressed, that require developers to implement specifically named methods specified by the protocol. If the correct methods were coded in the controller object, and that object was hooked up as the text field's delegate, then the user input would be automatically handled. Anna was finding that her text field was not doing anything when she hit return. It turned out that she had not correctly made the connection between the text field's delegate outlet and her controller object. The instructions in the book for doing this were not clear; it told students to hook up the delegate outlets significantly before the students actually wrote the protocol method code to implement what happens when the return key is pressed. It took me a while to find the text in the book that said, "That's why we told you to hook up those delegate outlets earlier." The actual instruction to do that was even more difficult to find. It turned out that Anna had dragged from her controller to the text field, associating the text field object with the Outlet variable specified in the controller, but not the other way around, which was necessary to associate the controller with the delegate Outlet variable of the text field. The graphic in the book further confused her, which showed only the Outlet connections between the text field and what they were in the controller's code, but not the connection for setting the delegate outlet. Further confusing the issue, the text field outlet connection had actually been made automatically earlier.

I became frustrated attempting to explain to Anna why it was necessary to drag from one direction but not the other, because I felt that the lab time was running out, and Anna seemed to be more intent on explaining to me her misinterpretation, or going on tangents in conversation, rather than fixing the problem and moving forward. In writing my field notes later that night, reflecting on this, I wondered if my own gender role had played a part in this. Because Anna had been the most behind in the class, my concern had been to help her catch up, on improving her

efficiency, but her conversation seemed to be interfering in achieving this goal. Possibly, she had been embarrassed at the mistake and wanted me to understand why she had made it, but I felt that I already understood, and there was no need to explain; she was only wasting precious time that she couldn't get back. Her concern may have been that building our social relationship, and improving my opinion of her skills, was more important than catching up in the class, which she may have decided was not going to happen.

It is extremely difficult to explain this process of dragging in prose without actually showing what to do on a computer screen. The Big Nerd Ranch can show this process of dragging in their demos during the lecture, but during the lab exercises, students are still expected to follow textual instructions, or even static graphics, in the book. This means that the dynamic nature of graphical programming can be lost in a text-based tutorial. Like you, the reader, students in the class such as Anna often find the process of "wiring up" connections in Interface Builder to be easy to do once shown, but find the meaning of what exactly they are doing hard to understand. Code can be read and parsed, but discerning the difference in meaning between dragging in one direction versus its opposite requires significantly deeper knowledge of two different design patterns (delegate, and target/action), knowledge which takes time to seep in. Students are simply told, do it this way, trust us, you will understand later.

What is important to recognize about this example is that students in the class are told what to do first, without being told the reasons why. The philosophy of asking them to type in code without asking exactly what it does fits into this. They are told not to question it at first, to just do it and make it work, because that is the way Apple has designed it. This is because a full conceptual understanding of what makes all of this work would be too overwhelming at the beginning. We will see in the next section that such fuller understanding comes over time, after the repetition of many of these concrete examples. Through debugging a program, students get a feel for what a working program, versus a non-working program, looks like, and what it takes to convert one into the other.

My experiences helping the students debug such issues began with me struggling along with them to read their code, trying to figure out what was wrong. At first, I was only marginally better than them at diagnosing the source of errors, and as we have seen, I often had to ask Joe for help. I wrote about my feelings of inadequacy as a TA in a fieldnote during Thursday night of my first bootcamp:

> I wonder if the students are better served when Joe helps them (he can find a problem usually almost instantly, or at least very quickly) or when I do. Because I'm only just slightly more experienced with this stuff than them, I'm going through all the steps of debugging it too. Sometimes this means I don't actually figure out the problem—but more than half the time, I do, but with much effort, after a long time and lots of searching. I wonder if it is more useful for them to be working with me to figure it out, so we work through the process together, and then they learn more about how to debug and track down such problems. Certainly, it is helpful for me to learn what issues to take note of.
>
> A number of things I've already seen recur—over release errors, how to debug those. Objects that are uninitialized and pointing to nil—that should be very easy to fix, you just didn't set the object somewhere.

I got much better towards the end of the week, but even on Thursday, there were still instances where I was stumped. Given the pace of the course, with lab time restricted to around 30 minutes, this often was not enough time to fully diagnose a problem, and with around 30 students in the class, I might have more than one person to help. Nevertheless, I could see myself getting better as the week went along. By the time I returned as a TA to the iOS course in 2012, I had significantly fewer problems helping students find their bugs, although in my subsequent trips, I left the class after Sunday and Tuesday, respectively.

Debugging and fixing programs is central to learning how to program. This is true everywhere, not just at the Big Nerd Ranch. However, at the Ranch, instructors recognize how important this is by devoting at least two-thirds of the day's class time to lab time. They understand that it is in learning by doing, not passively listening to lectures or even watching demos, that programmers build up their skills. Repetition is key. Many of the bugs students run into come up repeatedly, and it is through this repetition that they learn how to recognize them in the future.

Significantly, this also helps the instructors learn what pitfalls students run into and find difficult, which helps them call out such pitfalls in subsequent classes, and even modify the course materials to better explain them in later editions of the book. Students are not given specifically designed bug exercises to figure out, but are simply asked to debug code that they copied out of the course materials. Inevitably, someone makes a typo or copy-paste error that results in a common bug, so specific exercises are not needed—students will inevitably run into bugs as they go. While repetition is not built into individual exercises, we will see in the next section, over the course of the entire class, concepts such as design patterns are repeated over multiple exercises. This means that students gradually develop a sense that problems are solved in a consistent way. It is only after they have experienced this a number of times that students are then introduced to the abstract concept behind what they had been experiencing viscerally.

Practical to Abstract, Specific to General

Another pedagogical element in Big Nerd Ranch lessons was the illustration of course concepts with practical examples, often from actual jobs. For example, Mark presented a rock collecting application that he had written, showing his actual code to the students on the projector, to hit home the idea that what they were learning was not academic, but was used for real shipping code. Similarly, Joe showed the students who had stayed after dinner a trick he used, using actual code he was writing for a client. This illustrated the difference between Big Nerd Ranch's approach to teaching programming compared to academic computer science courses; it was fundamentally practical, applied knowledge. Students were paying Big Nerd Ranch a lot of money to teach them skills directly applicable to the programs they wanted to write, for either their jobs or their hobby. Given the high learning curve of Cocoa and the unfamiliar concepts they were being exposed to, students needed to be able to trust that the frustration and strangeness they felt while wrestling with Objective-C would be worth it. Consistently showing them real-world examples let students know that what might seem like esoteric knowledge would be of immediate practical use.

As we have seen from my examples helping students with debugging, Cocoa programming involves learning highly abstract concepts such as the delegate design pattern. While this pattern is powerful, it is not easy to grasp or visualize. Mark mentioned that it had been a difficult concept for him: “I didn’t understand delegation for a long time until I wrote my own framework.” However, he acknowledged that a full understanding of this concept was not necessary, that it could be acquired over time by simply using it. “Even if you don’t completely understand the idea, you will be using it.” The Big Nerd Ranch approach to teaching such concepts was based on this notion that complete abstract understanding was not necessary to begin using it in practice. Students merely needed to be shown examples of how such concepts could be used in solving particular programming problems, and then exposed to numerous such examples repeatedly to hit home the general usefulness of the concept.

For example, the first time the delegation pattern was used in the course was fairly early on, in Lesson 4 (out of 29) on the afternoon of the first day. In designing the course, Joe had front-loaded it with a key piece of functionality that mobile developers were interested in using—the GPS sensor for determining a phone’s location. In Lesson 4, Joe introduced Apple’s CoreLocation framework, a set of Objective-C APIs that provided developers with access to the device’s location, which the hardware determined using a combination of GPS signals, and triangulating cell phone towers and WiFi hotspots. Not only was this of enormous practical interest to developers, but because CoreLocation also used the delegation pattern, it also served to introduce the concept through the practical example of a feature that developers were highly motivated to learn. Students were told to use a stock “CLLocationManager” object from the CoreLocation framework. How does the student customize what happens when the phone’s location changes? This is done by designating another object to be the location manager’s helper, which receives location updates from it. When these updates occur, custom code in the helper object, which the student will write, executes. The delegation pattern thus allows the student to customize the behavior of a black-boxed CLLocationManager object whose code

he or she cannot modify. As we have seen, this pattern is later applied to several other types of object interactions. TableViews delegate to their helpers tasks related to updating themselves when their underlying data changes, or mirroring changes made by users to the data. Similarly, text fields also use delegation to allow helper objects to cause user input in a text field to commit those changes when the user hits return, tab, or simply clicks outside of the text field. These actions can even trigger additional code to be run. The common pattern between all of these situations is that developers use stock versions of the objects provided by the Cocoa libraries. CLLocationManager, UITableView, and UITextField are not customized or subclassed in any way. These are used as provided by Apple, and although their functionality is fully documented, their code is not available for modification. All custom behavior is delegated to a helper object that is under the programmer's complete control. This allows Apple's stock objects to be reused as much as possible in a general way, while still allowing for customization to the particular circumstances the programmer needs. Customizing the stock objects directly would reduce their generality and thus reduce their reusability, which would make programs much less flexible and maintainable in the future.

The effect of this approach of introducing abstract concepts through the repetition of specific examples is that students are much more likely to understand the concept and agree with the need for it, because they have already seen it applied successfully to a number of general problems. The abstract concept already has a referent to a pattern they have experienced—it now simply has a convenient name they can refer to. Moreover, repetition not only reinforces the concept in their minds, but also hits home how consistent Cocoa is in utilizing the same solutions throughout its design. This also cuts down on resistance to the use of the pattern.

Humor

Part of the presentation style of the instructors at Big Nerd Ranch is approachability and casualness. Instructors make a point of portraying a persona that is not very different from the students. We are programmers, just like you, working

on real-world projects, they seem to say. One of the ways they create this rapport with students is through the use of humor, not only lectures in but also in answering student questions. Additionally, humor is useful for defusing possible tensions around students being hostile or resistant to Cocoa or Objective-C idioms and practices. It is also used when the instructor is presenting an opinion about a practice that could be construed as controversial, in order to qualify it.

The use of humor by instructors was not one-size fits all, but was something each individual instructor developed as part of his particular persona. This often took on gendered forms. For instance, Joe used masculine geek humor in some of his presentations. For example, he created a class and gave it the name "Megaman" after the Nintendo video game character. In the Megaman class, he created an instance variable named "girlfriend." He then presented a command that will return how much space a variable takes in memory, the "sizeof" command. Using this command, he said, you could get the "sizeof Megaman's girlfriend." Joe chose to name the class after a male video game character, not a female one, and made the character's girlfriend a property of that character, not the other way around. The audience did not seem much bothered by the gendered nature of the joke, treating it as typical geek humor.

In a subsequent lesson on the accelerometer, Joe's irreverence was highlighted in another joke. He was discussing the use of high and low pass filter algorithms to smooth out the signals coming from the accelerometer. The low pass filter highlighted large changes to how gravity affected the sensor, making it useful for detecting changes in the phone's orientation. This detected changes in the 10-20 Hz range. The high-pass filter was useful for detecting a user shaking the phone as a means of input, with oscillations in the 70-100 Hz range. Joe now used humor to signal that he was not an expert in signal processing. "For those of you in EE [Electrical Engineering] who do this stuff, I apologize for butchering your craft." However, just afterwards, having qualified this, he signaled how he really felt about the authority of Electrical Engineering. "Otherwise, who cares about that stuff anyway?" Later, in describing one possible use for the high pass filter, he said,

"maybe it's a game—you shake it for 10 seconds, you get a prize… that's not good." It appeared as if he had unconsciously made a joke about masturbation, and only realized it after the fact. It is possible that the joke may have been deliberate, however, and used the qualification at the end to apologize for the frat boy humor.

Another time, Joe warned students that the Big Nerd Ranch forums had spam on it, and thus they might see some unsavory images. He joked that he put those there especially for them.

In other cases, Joe used humor to highlight practices that he wanted students to avoid or pitfalls he wanted them to watch out for, but knew full well from experience that they would run into. Sometimes he would violate these practices in his demonstrations for expediency due to lack of time, but make sure to tell the students not to do what he was doing. For example, in one lesson, he said, "you'll write three lines of code… I'll just copy and paste and forget that [the variable] valueInDollars is an integer, and it WILL crash. I'm looking forward to seeing you make this mistake!" This got a big laugh from the class. Joe used this tactic more than once to highlight potential errors he knew students would make.

Other times, Joe used humor to defuse potential dissent over his recommendations for avoiding a controversial Objective-C coding practice, known as "dot notation." We will discuss this controversy in more detail in chapter 6. Suffice to say that Joe first made technical arguments against using it, particularly pedagogical arguments, which justified why he insisted that no one use dot notation in the class. However, he said that ultimately it was up to each student to decide for him or herself. He also called the practice, "an abomination," and joked "you can use dot notation after this week, but if you do, you can't come to my parties."

In any case, despite Joe's cantankerous, often cynical and opinionated persona, the mostly male students seemed to appreciate Joe more for his authenticity and candor. This made him seem more personable, as someone the students would like to hang out with after the class was over.

This kind of relatability for Big Nerd Ranch instructors is notably different than university courses where the professor takes on an aura of authority. Big Nerd Ranch instructors derive their authority in part from the Big Nerd Ranch name and from Aaron Hillegass's own authority as a famous instructor and quasi-celebrity, but also from their real-world expertise in programming, acquired from daily work on actual client contracts. Thus, their authority depends in large part from being just like the students, except that they have real-world experience in what the students are learning. Big Nerd Ranch instructors, while insisting on the disciplinary practice of typing out all the code in the exercises, and setting the pace of the course through the sheer quantity of material needed to cover, do not otherwise use surveillance to make sure students remain on task. They did not take on the persona of taskmasters. There are no exams or any mechanisms for assessing students' mastery of the concepts. Students are expected to be self-motivated in wanting to learn iOS, Cocoa, or Android programming and get as much out of the course as possible; after all, either they or their employer paid $3500 for them to be here, and it is their decision if they want to make the most of it. The after dinner lab hours are treated more like office hours, for students to catch up and finish their programs because there was not sufficient time during the day. Although the instructors encourage advanced students to work on their own projects during these hours, only a few actually do this. What is important, however, is that this exhortation signaled to students the moral quality expected of would-be professional developers—a willingness to learn, and a passion for writing code, whether on the job or in one's off time, in order to continue to improve one's craft. Developers are seen as independent, entrepreneurial self-starters and self-learners, who need little disciplinary prodding to work, and rather view arbitrary authority, bureaucracy, and surveillance with suspicion and hostility.

However, with a hint towards fostering a more healthy work-life balance, instructors also encourage students to go out and enjoy the grounds, whether after hours, or during the official breaks when the instructors lead them out on hikes, in order to get exercise and refresh both their bodies and their minds. (Despite this, some students chose to stay in the class, either to try to catch up, or simply to avoid

the summer heat and humidity.) Despite the relentless pace, the instructors also encourage students to take the zipline tour, even though it takes at least two hours out of the day, in part because of the benefits of building a team spirit among the attendees, and probably also because it strengthens the Big Nerd Ranch's relationship with Historic Banning Mills, which is a separate business. They are, after all, at a beautiful country resort.

Humor is also important because it helps keep the class light, enjoyable and fun, when it might otherwise be experienced as painful and stressful. This is crucial because the instructors want students to enjoy the experience of the class, despite the hard work they are doing, reinforcing the message that Cocoa programming itself should be pleasurable and fun. And by eschewing a persona of authority in favor of that of a male colleague, instructors reinforce the notion that programmers are an egalitarian, meritocratic, and primarily masculine brotherhood.

Style and "Stylishness"

Big Nerd Ranch instructors do not only teach the technical aspects of Cocoa programming, but also what the Cocoa community considers proper coding style, an aspect of normative practice that has concerned programmers generally, not just Cocoa programmers. Coding style often involves choices that do not make any difference to the compiler; the resulting code is functionally the same no matter what choice is made. Rather, such choices often reflect idiomatic conventions built up over time among a community of programmers. Although they may not have meaning to the computer, meaning is ascribed to such choices by the human programmers who read and write the code.

In his 1974 Turing Award acceptance lecture, Donald Knuth, author of *The Art of Computer Programming* series, pointed out that programming style was then becoming a topic of interest among computer scientists (Kernighan and Plauger 1974; Dijkstra 1971), and this was a component of being able to treat programs as aesthetic works of art, and programming as a pleasurable activity (Knuth 1974, 670). Knuth argued both that style was in part personal preference ("there is no one 'best'

style") and thus people should not force their own styles on others, and yet style could still be judged "good" or "bad." Utility and efficiency were important, but an excessive focus on efficiency might be counterproductive. Other measures of "quality" were important too, including correctness, flexibility, and usability. "In the first place, it's especially good to have a program that works correctly. Secondly, it is often good to have a program that won't be hard to change, when the time for adaptation arises. Both of these goals are achieved when the program is easily readable and understandable to a person who knows the appropriate language. Another important way for a production program to be good is for it to interact gracefully with its users… It's a real art to compose meaningful error messages or to design flexible input formats which are not error-prone." (Knuth 1974, 670) Readability is a key to the maintainability of code, in terms of both getting it to work in the first place and modifying it in the future. In corporate organizations, the engineering concerns for quality seem to have outweighed laissez-faire personal preferences, however, as places such as Google have standardized on common styles and have even made them public for contributors to their open source projects (Google Inc. 2015; Seibel 2009, 72). As we will see, in practice, the idea that style is purely an individual preference runs into conflict with communal and organizational pressures to conform, highlighting the social nature of programming work.

Mateas and Montfort have noted that in some cases, programmers deliberately subvert clarity and beauty in the creation of code, adhering to an alternative aesthetic of obfuscation to create unnecessarily complex and tortured Rube Goldberg-type code. Although such code must work (it is crucial to the aesthetic that it does, often cleverly), the creation of such code is made primarily for the literary purpose of parody or ironic commentary, in circumstances in which the code is not intended for production use, such as during the International Obfuscated C Code Contest, or academic research into "weird" languages. The target of this commentary is to focus attention on the fact that code needs to be read by humans as well as machines. "Obfuscated code and weird languages highlight the importance of the human reading of code in software development. If some code is only to be read by a

machine, it can be neither obfuscated nor clear: it can only function properly or not." (Mateas and Montfort 2005, 1) By explicitly creating "bad" code, hackers point out what is "good" code, code readable by humans. This is often accomplished through "double-coding," the creation of code that has meaning in two different registers, such as the sentence "Jean put dire comment on tape" which has meaning in both French and English (Mateas and Montfort 2005, 5). Mateas and Montfort point out that because code has meaning to both humans and machines, all source code is double-coded. Perl poetry, and the programming language Shakespeare, in which all programs are written in the form of a Shakespearean play, highlight this duality.

Case and Piñeiro's study of attitudes on code aesthetics on Slashdot find that debates over code aesthetics explicitly involve performance of identity (as "artist," "hacker," or "geek"), espousal of values (the "hacker ethic," especially disdain of monetary motives) and group membership: boundary work over who is 'in' and 'out.' Frequently, the 'out' group consists of managers whose impositions programmers are explicitly resisting. Style mandates are thus seen as authoritarian encroachments on programmers' professional expertise and autonomy (Case and Piñeiro 2006). This aspect of coding style is explicit in the Cocoa community.

As we saw earlier in chapter 1, Cocoa programmers care deeply about aesthetics, not only the visual look of their applications but also to the code itself. Cocoa coding style generally follows conventional programmer conceptions of good style, emphasizing readability and clarity, though the practical details may be interpreted differently. The most notable difference in comparison with other programming aesthetics is that "good" Cocoa code tends to be verbose, rather than terse or brief, in order to emphasize clarity of the programmer's intention rather than "cleverness," or mathematical elegance. This aesthetic difference is stark, giving many newcomers a sense of distaste upon their first exposure to Objective-C code. Attitude towards verbosity versus brevity is a clear marker of inclusion or exclusion in the Cocoa community. For this reason, instructors at the Big Nerd Ranch make significant effort to teach "good" Cocoa coding style, in order to show students how to be a "proper" Cocoa programmer. Normative pressure is brought to bear on this

message. Big Nerd Ranch founder Aaron Hillegass explicitly connected coding style to social acceptance:

> The Objective-C compiler… would let you do any sort of stupid thing you wanted to. And so to create good designs, we as a community developed a set of idioms that everybody would follow… there is nothing in the compiler that enforced that, it's just something that as a member of the community you were expected to follow.
>
> (Aaron Hillegass Interview, July 7, 2011)

In the following quote, Hillegass highlights the social nature of coding through normative entreaty while simultaneously highlighting practical, utilitarian reasons to conform.

> A lot of what I have to do is teach these idioms that have been handed down basically by word of mouth from the very first Cocoa programmers, the first Objective-C programmers. And so when I teach, I don't just talk about, "you have to do this," I say, "all stylish Cocoa programmers do this." And that idea that you would be too embarrassed not to follow this idiom.
>
> …The next person who's going to take over ownership of your code, is going to… look down upon you if your code doesn't follow these stylish conventions…
>
> If someone came in and saw your code did not follow those things it would be embarrassing to you…
>
> (Aaron Hillegass Interview, July 7, 2011)

Hillegass makes several points. First, is the idea that style is a sort of tacit knowledge, handed down by "word of mouth." Hillegass referred to this as a kind of "folklore," as if it were part of some oral tradition among the tribe of Cocoa programmers. Secondly, Hillegass insists that he does not teach style dogmatically, relying on his authority as an expert. Such a tactic would simply not work, as programmers, valuing independence, tend to chafe at arbitrary authority and reserve the right to make such decisions for themselves. He must still acknowledge the view that style choices are personal preferences, but provide a compelling reason why someone should choose to follow his style recommendations. Rather, Hillegass tries to persuade students to accept these choices as necessary for their social integration

into the community. Part of this social argument is instrumental and pragmatic—existing Cocoa programmers follow such conventions, and because it is rare for successful software to remain under the authorship of a single programmer over the course of its lifetime, for maintenance reasons one should expect that one's code will be read by others and thus write code with this in mind. However, a significant component of the argument is emotional—you will be "embarrassed" if others look at your code and it does not follow the proper conventions. Worse, people will "look down upon you." Thus, maintaining proper style conventions is important to maintain and improve one's status in the community of practice. Improper style is the first sign of peripheral membership. Hillegass signals the importance of being accepted by the community with the specific turn of phrase he uses. Instead of "style," he talks about being "stylish." By alluding to an alternate meaning of "style," as in "fashion" and "trendiness," Hillegass implies that to be one of the cool kids, to be accepted by the popular crowd, a programmer ought to follow "stylish" coding practices. Being "stylish" conveys social status, and is a marker of inclusion and group membership.

This meaning of "stylish" relies on feminine gender connotations. This can cause some pedagogical problems. Hillegass acknowledges that among his audience, who are mostly men, talk of "stylishness" often causes a deal of scoffing at first: "it's funny, because I often talk about stylish code for programmers in the class, and people giggle the first time." In one sense, the term works precisely because it is unexpected from a man, and thus somewhat humorous as well as memorable. However, in order to counter this tension, Hillegass goes to some lengths to explain the technical basis for coding style choices in the class and in the text of the book. He insists, "by the end of the week, it becomes clear [to students in the class] how important in the community it is to be stylish." Such technical arguments for following proper style choices are gendered masculine, while social reasons are gendered feminine.

We can see how Hillegass qualifies his talk of "stylishness" with a technical argument using an example in the introduction of the third edition of *iOS*

Programming: The Big Nerd Ranch Guide, in a short section on the "Style Choices" that were made in the book's code. Conway and Hillegass explain:

> We have attempted to make that code and the designs behind it exemplary. We have done our best to follow the idioms of the community, but at times we have wandered from what you might see in Apple's sample code or code you might find in other books. You may not understand these points now, but it is best that we spell them out before you commit to reading this book… We believe that following these rules makes our code easier to understand and easier to maintain. After you have worked through this book (where you *will* do it our way), you should try breaking the rules to see if we're wrong." (Conway and Hillegass 2012, xvii)

Conway and Hillegass repeat that the primary reason for their recommendation of a particular coding style in the class is pedagogical. Once the class is over, such style issues are now a matter of personal preference, so that independent-minded programmers can make their own choices. In the class itself, one instructor, Step, explained to students that "Objective-C is a language that has a lot of patterns and conventions that are not necessarily enforced [by the compiler]. The style I am using… is frequently used by Objective-C programmers, but you can use what you want." The message was that this and other style issues were a personal preference, and that the book was not in the business of dictating a particular practice to people who had different preferences. Nevertheless, the book still exhorts them to make the "right" decision, by giving rational reasons for why such issues of style are not inconsequential or trivial. It encourages readers, once they have worked through the book, to "break the rules" later and evaluate for themselves, assuming that all rational readers will come to the same conclusion. The text implies that despite the seeming arbitrariness of style, objectivity will lead people to the "right" practice.

How does this manifest itself in the class? Instructors, following the pedagogical strategy laid down by Hillegass, did refer to coding conventions and other practices as "stylish." For example, at one point, Joe explained to a student the reason for a coding convention in Cocoa. Before I recount this interaction, however, I need to explain the technical underpinnings of their conversation.

In Objective-C code, objects are declared like this:

```
UITableView *tableView;
```

The first term, UITableView, is the class (or type) of the object. The second term, tableView, is the name of the variable being declared, or in other words, the name of the object itself. What is the '*' character? That refers to the fact that the variable tableView is in actuality not the object itself, but a reference, or "pointer," to that object in memory. This is unlike most other object-oriented languages, and has to do with the implementation detail that Objective-C is fully compatible with, and largely implemented using the constructs of, the low-level procedural language C.

When I first took C programming in the 1990s, I was taught to write variables that were pointers like this:

```
int* aNumber;
```

The '*' character is written flush against the type on the left, not the variable name on the right. This emphasizes the concept that pointers to specific types of data are themselves a type. Int and int* are not the same type, and the compiler enforces this difference. Int* is a type that is a pointer to an integer, not an integer itself. The compiler, however, does not care whether there are any spaces between the '*' character and the type preceding it or the variable name after it. C programmer convention, however, is to write the '*' next to the type on the left.

One student in the course, who had a C background, asked why Cocoa programmers write the '*' flush against the variable name on the right, in contrast to C convention. The student said that he puts it next to the type because he sees the pointer as a type, which makes sense in the procedural C context. Joe responded by saying that in the case of Objective-C code, the variable is a pointer to an object, so the '*' should go with the variable. The book written by Aaron Hillegass as course materials for the Objective-C primer provides additional reasons for why a Cocoa programmer should follow this practice, which it labels as "stylish." On page 59 of the first edition, in a section titled "Stylish pointer declarations," the text explains:

> Because the type is a pointer to a float [floating point (real) number], you may be tempted to write it like this:
>
> ```
> float* powerPtr;
> ```
>
> This is fine, and the compiler will let you do it. However, stylish programmers don't.
>
> Why? You can declare multiple variables in a single line. For example, if I wanted to declare variables `x`, `y`, and `z`, I could do it like this:
>
> ```
> float* x, y, z;
> ```
>
> Each one is a float.
>
> What do you think these are?
>
> ```
> float* b, c;
> ```
>
> Surprise! `B` is a pointer to a `float`, but c is just a `float`. If you want them both to be pointers, you must put a * in front of each one:
>
> ```
> float *b, *c;
> ```
>
> Putting the * directly next to the variable makes this clearer.
>
> (Hillegass 2011, 59)

This text presents a technical, instrumental, and practical reason for this choice of coding style: to make the meaning of the code more easily legible and readable to human readers who will be responsible for maintaining the code in the future. Thus, community norms and conventions are presented as not purely arbitrary and an outcome of social pressure, but as "rational" choices which benefit practitioners in writing more maintainable code. Nevertheless, although the reasoning is supposedly objective, the exhortation is moral and social: "stylish programmers don't."

"Stylishness" was brought up in the course again when a student asked why, on page 178 of the iOS programming book,[35] the example code used "NULL" instead of "nil." Both symbols represent the concept of "nothing," which is not the same as a

[35] The student was referring to the second edition of *iOS Programming: The Big Nerd Ranch Guide*. (Conway and Hillegass 2011, 178)

numerical value of zero. Joe replied that it didn't matter which one was used, they're the same—but NULL is not as stylish. In this instance, he did not explain further, and it was not clear why. Possibly, there was not enough time in the class and he needed to move on, and simply replying that one was the preferred, more "stylish" choice sufficed for the moment. However, Hillegass's Objective-C book, the text for the primer, does give an explanation. On page 58 of the first edition, it explains that "NULL" represents a pointer, or a reference, to nothing. This is implemented as a memory address with a zero value. Remember we discussed earlier that in Objective-C, objects are referenced by pointers also. Thus, "nil" means a pointer to no object, or in Hillegass's words, "the address where no object lives." (Hillegass 2011, 58) The book does not explain that "NULL" is a keyword in the procedural C language, while nil is a keyword in Objective-C, which includes everything in C, including the "NULL" keyword. Therefore, use of "NULL" indicates that one is programming in procedural C, using that language's procedural idioms. "Nil" is used only in Objective-C to refer to objects, or rather, the lack thereof. The reason that Joe said that "nil" is more stylish is because to write idiomatic Objective-C code, it is more proper to use "nil." Although to the compiler, "NULL" and "nil" are interchangeable, they have slightly different meanings to the programmer. What the student caught on page 178 of the iOS book, the use of NULL, works correctly, but may have been a stylistic mistake on the part of Conway and Hillegass, because "nil" is used properly later on the same page to refer to the same bit of code. This mistake was corrected on page 193 of the third edition of the iOS book. (Conway and Hillegass 2012, 193)

Discussions about style can also involve a test and display of membership in the community. For example, this occurred between Aaron Hillegass and myself during an interview, when he asked me about my own Objective-C coding practices.

> Aaron: Did anybody teach you that when you were learning this?
>
> Hansen: I don't know if I was ever taught it explicitly, I just sort of learned that these were the ways that it was done, but never exactly why, or why those decisions had been made…
>
> Aaron:A lot of it has to do with internal consistency, right? Is that you start with—OK we're just going to accept that A and D—we're going

to do that. And then B, C, F and Q all fall out of that… To be consistent with what decisions you've made early on.

Hansen: Right. I mean, even little things like, you know, whether or not the spacing between the minus and the parenthesis, you know, and where—

Aaron: I put a space in there, do you put a space there?

Hansen: I think I do. I think I do.

Aaron: All the Cocoa [headers have them] so I would imagine that you would have to.

Hansen: I think I do. I just kind of do what the code I see does.

Aaron: Yeah. What about after colons? […] Usually they have a method name that has three parts for each colon.

Hansen: Right, OK. I usually put—no, I don't put a space after the colon.

Aaron: I didn't think you would. Some people do.

(Aaron Hillegass Interview, July 7, 2011)

In this exchange, Aaron Hillegass was asking me about my preferences for adding spaces between syntactic elements in code. For example, an Objective-C method from one of my own programs is written like this:

```
- (void)moveItemAtIndex:(int)from toIndex:(int)to
{
      [Contents omitted for clarity…]
}
```

Following the convention, I have added a space after the initial '-', but no spaces after the colons, as I described to Hillegass. However, the code could have been written like this with no change in functionality:

```
 -(void) moveItemAtIndex: (int) from toIndex: (int) to {
      […]
}
```

None of these choices have any functional impact; Apple's Objective-C compiler ignores differences in what is known as "whitespace," which means spaces, tabs, and carriage returns. The language parser of the compiler uses the other characters (the minus sign, the parentheses, colons, and braces) to determine where

some elements begin and others end, without needing to separate all of them with spaces. However, the code written by the Cocoa community and Apple's own code follow the conventions used in the first example. The reasoning behind such choices is that the lack of spaces between certain elements groups them together more logically so that the reader can more easily understand that they go together. This improves the readability and understandability of the code. Moreover, it is important that even if the programmer chose to write the second, that she remain consistent with her choices—mixing the first and second styles arbitrarily would be considered even worse, as a sign of sloppiness or lack of care.

Another element of my exchange with Hillegass was that he was inquiring whether the stylistic elements that he was promoting in his classes were widely followed by other programmers. In this way, he was respecting me as a fellow Cocoa programmer. However, my response, that I did indeed follow the prevailing practice, by maintaining consistency with Apple's own code, also signaled to Hillegass that I, like him, followed the proper style conventions, which meant that I was somehow part of the in-crowd. By giving the "right" answer, I gained credibility in his eyes as an oldtimer, which established a trust that I "got it," I understood—I was a highly included member of the Cocoa community.

To a large extent, writing code using Apple's Xcode IDE greatly helps developers follow conventions. When hitting return to break up the line, as long as labels are not too long (as in my latter example) Xcode's text editor will automatically indent the line to line up colons in the names of methods so that they are easier to read. However, a programmer could still change the indentation if he so preferred, although if he does, Xcode's automatic indentation behavior will get in his way. In this way, Apple's own tools go a long way towards fostering these conventions; although the compiler does not force developers to follow them, the text editor makes many of them the default. Code inserted by Xcode's autocomplete feature also follows the conventions for whitespace. These can be completely avoided, however, if the developer prefers to use a different text editor for writing code.

Thomas Hughes' notion of "technological style" offers a way to understand how choices made in the design of technologies are shaped by particular cultures and histories. "There is no one best way to paint the Virgin; nor is there one best way to build a dynamo." (T. P. Hughes 1987, 68) "The concept of style applied to technology counters the false notion that technology is simply applied science and economics" and that the shape of technology follows logically from them. Hughes illustrates this using the contrasting styles of British and German electric grids of the 1920s, and the values motivating each. "Berlin possessed about a half dozen large power plants, whereas London had more than fifty small ones… In the London and Berlin regulatory legislation that expressed fundamental political values rests the principal explanation for the contrasting styles. The Londoners were protecting the traditional power of local government… and the Berliners were enhancing centralized authority…" (T. P. Hughes 1987, 69–70) Other technological style differences include that between American and European automobiles, and American and Soviet spacecraft (T. P. Hughes 1987, 70); each presented different ways of solving similar problems motivated by different contextual factors. What the notion of technological style illuminates is that technological choices are not always constrained by material, economic, scientific, or mechanistic factors deriving from the internal logic of the technology itself—there are many choices that are open and are shaped by social, cultural, political, or aesthetic influences.

Style in programming can be seen as one particular version of this. On a macro-level, the different choices made by the designers of different platforms, operating systems, programming languages, and even text editors shapes the styles of the way people work with these technologies in such a way that engenders strong affective responses, either positive or negative (whether it be Mac versus PC, iPhone versus Android, emacs versus vi, Objective-C versus C++). In this sense, Hughes' notion of technological style shares much in common with Bijker's notion of technological frame. Within a particular style, choices made by designers can constrain users from working in a way contrary to that style. Nevertheless, not all elements of the style are governed mechanistically or technologically; some are

governed normatively. The actors understand that programming style is, by definition, not constrained by the machine, but possible only because there is choice. Given the libertine and meritocratic elements of programmer identity, this choice is often interpreted as "freedom" and "preference," and thus impositions of style are often resented as authoritarian dogma. Good, intelligent programmers ought to be able to figure out the best practices for themselves, not be disciplined or chastised by others. Despite this, programmers frequently do normatively advocate for what they consider to be correct or at least "better" choices. Yet this advocacy is controversial precisely because norms are seen as "soft" and thus open to renegotiation, and in the absence of a disciplinary practice such as code review, the norms of "good" style are often broken in practice.

For the actors, the term "style" has as much a literary connotation as technological. Programmers understand that source code is a text, which, although constrained by their programming language's compiler, nonetheless allows for a wide range of choices in the naming of entities such as objects and functions defined by the programmer. Such choices are not governed mechanistically by software but normatively by the community, outlined in "conventions," informal guidelines that most programmers who are highly included in the community follow. As we saw earlier, Hillegass in his book explicitly calls Cocoa an "idiom," which encompasses both ways of thinking as well as specific coding practices, which includes style choices for code. For example, in criticizing the code examples in an Objective-C book written by Stephen Kochan (Kochan 2012), Hillegass said, "If you look at the Kochan book, it's actually not very stylish. The spacing is not consistent with the Apple standards and that's big. It's because he's new. He's been living in his own bubble too long. Really. I've been doing it for 17 years now." (Aaron Hillegass Interview, July 7, 2011) Certainly, this comment is in some ways self-interested, as Kochan's book could be seen as a competitor to Hillegass's. However, Hillegass's criticism is motivated by the fact that he had tried to use Kochan's book for his Objective-C primer course, but found it unsuitable for his purposes, as it was more of a reference than a tutorial. Hillegass wrote his book because he needed a text tailored

to the way his Objective-C primer was taught; this included promoting the stylistic practices he felt students needed to at least be aware of. In his comment, Hillegass implies that his own years of experience, many of them at NeXT and Apple where the stylistic conventions were established, and his continued engagement with Apple and the Cocoa developer community, gives him the authority to recommend coding styles that Kochan does not. Kochan's isolation from the community means that he has not properly internalized its norms, which is reflected in his non-standard style. This remark was made right after the exchange in which Hillegass had asked me about my own style preferences. I understood this as an instance of boundary work: Hillegass was including me as an insider, but excluding Kochan, due to his non-conformity to convention.

More recently, however, some style elements in Cocoa code that were previously social convention have become to be relied on by newer versions of Apple's compiler. Over time, Apple's engineers have begun to depend on the fact that a majority of the community follows these coding conventions, and will do special tricks taking advantage of them, offering conveniences to programmers who do so. Thus, not following such conventions will put programmers at a disadvantage because Apple's libraries have now begun to count on them. For example, prior to Apple's creation of its Automatic Reference Counting (ARC) scheme, there was a lot more freedom in stylistic choices:

> The Objective-C compiler, at least until ARC came along, would let you do any sort of stupid thing you wanted to…
>
> But… a lot of things that were social conventions in terms of memory management are now codified in there. So you can't name any property starting with the word "New" [any longer]. Because we [the community] always said, well, "New" returns objects that were not retained. So ARC just said, nope you're not allowed to do it.
>
> (Aaron Hillegass, Interview, July 7, 2011)

Apple's Automatic Reference Counting scheme could only make memory management automatic if it made certain assumptions about programmers' code, and disallowed certain practices that used to be possible. Because most programmers in

the Cocoa community did follow convention, Apple could safely assume that the new technology would not put a huge burden on them to make their code compatible with it; only a few would be affected. In implementing ARC, Apple made a deliberate tradeoff that is emblematic of their approach to improving their tools. It provided developers with a tool that, by automating a previously repetitive, error-prone, and burdensome task, made their lives significantly easier while black-boxing a process that previously they had direct control over. It was enabled in this task by the Cocoa community's adherence to convention, which allowed Apple to build the idiom directly into the compiler itself without breaking the majority of people's code. What was previously social fashion is now a material part of the technology itself. One might think that Cocoa programmers might react harshly to having direct control over memory management taken away from them. However, the reaction to ARC has been overwhelmingly positive, as the community is mostly grateful for not having to perform this mundane task that other object-oriented languages got rid of using a different technology, garbage collection. Thus, ARC is seen as a sign of progress towards a more "modern" programming language that keeps Objective-C on par with the competition. For example, the iOS developer who I met in the Android class, Charlotte, told me a friend of hers called tedious coding tasks such as manual retain/release memory management, "donkey work," and was glad that she no longer had to do this.

A slightly different use of "stylish" appears in Chapter 12 of *iOS Programming: The Big Nerd Ranch Guide*. Here, it is used to signal that implementing an additional bit of functionality would be better for the user interface, as it would make things more convenient for users. "It would be *stylish* [emphasis mine] to also dismiss the keyboard if the user taps anywhere else…" (Conway and Hillegass 2012, 256) The programmer does not need to do this, and will have a perfectly functioning app, but without this bit of behavior, the interface will be annoying for users. To improve the interface, the programmer needs to do a little bit more work. The use of "style" here retreats from a purely textual or idiomatic meaning pertaining to the code, and refers to larger issues of the application's design,

including that of the functionality of its user interface, which here is understood not purely as an aesthetic issue but one of added function and convenience, which requires extra programming work. Here, Conway and Hillegass are saying that in the Apple developer world, it is "cooler" to take the extra effort to make the user interface more convenient for users. It is a social norm among Apple developers that user interface is highly important, and not paying sufficient attention to this will earn disapproval among one's peers in the community.

Student Resistance

As we have seen, many of the more advanced students in the iOS class came in with knowledge and experience with other object-oriented languages and environments. This gave them a significant leg up on students such as Victoria, Walt, and Anna who had only procedural programming experience, as they did not need to learn the fundamental concepts involved in object-oriented programming. However, although Objective-C shares many concepts and features with other languages, sometimes it uses different words for the same concepts.[36] Students often find themselves translating the relevant term from the language they are familiar with into the Objective-C equivalent. Some concepts are analogous, but not exactly equivalent, and a simple translation then may cause confusion. Such students also often have very ingrained preferences for how they write code, based on the prevailing practices and conventions of their native programming environment. This can cause some resistance to the way things are done in Cocoa and Objective-C.

Some of these advanced students came in with a hostile attitude towards Objective-C, treating it as something of a necessary evil, because at first Apple would not let other languages be used for iPhone programming. One student in one

---

[36] For example, the feature called "interfaces" in Java, which allows multiple inheritance of methods but not instance variables, is known as "protocols" in Objective-C. Similarly, methods and instance variables in Objective-C are called "member functions" and "data members" in C++.

of the Objective-C primers was John, was familiar with Python, a popular object-oriented language used for web servers. Step identified him jokingly as a potential "rabble-rouser" and "troublemaker" in the class, somebody who would ask tough questions and give him a hard time. John quickly lived up to this. He announced on the first day of class that he hated Objective-C syntax. As Step began to explain the benefit of argument labels in Objective-C with an example, John loudly announced that he "threw up a bit in [his] mouth." Other students also said that they found the syntax confusing. Step reassured them that when he first learned Objective-C, he found it confusing too, but that now he sees the benefit in it.

Novices and outsiders with significant experience in other programming environments often have a visceral negative aesthetic reaction to Objective-C, especially its "weird" bracket syntax. This initial feeling begins to stand in for a whole host of other things that are unfamiliar to them. They start out unwilling to see the benefit of this foreign way of thinking. However, because they want to write apps for iOS, and Apple has provided them with little alternative, and because the instructors at the Big Nerd Ranch insist that learning Objective-C, because of its tight conceptual coupling with the Cocoa frameworks, is the best way to learn Cocoa itself, the students in the class have little choice but to accept it.

Later during the week, Chris remarked explicitly that he felt that coming into this new Apple environment was like learning a whole different culture, which included different ways of doing things from what he was used to. Chris was used to a Java development environment called Flex, which he said had amenities that Cocoa and the Xcode environment lacked, although Cocoa provided new conveniences that he was beginning to learn. Flex had easy and extensible ways to bundle up and package custom controls (such as buttons) for other people to use, and he did not know if Cocoa had similar functionality, which was important to him. He asked me during one of the lab periods if there was any way in Interface Builder to create your own control classes and reuse them. I told him that in earlier versions of Interface Builder, Apple's graphical user interface development program, it supported the creation of something called an IBPalette, which allowed this kind of reusability. An

IBPalette, once made, could be shared with others (or even sold for money), and could then be dropped in and treated like any built-in Apple control. Nowadays, this functionality was slightly different, called an IBPlugin, but people did not seem to use it much anymore. The student next to Chris, Andrew, asked me if there was a way to create "application modules" using Cocoa. They explained to me that in Flex, a module was a full blown application that could be plugged into another program, swapped in or out, and interface with other modules according to a predetermined protocol, and all sharing the same front end interface. I remarked that Cocoa did not have this kind of functionality for large-scale code reuse. Chris remarked that in his area of work, enterprise software development, packaging up code for reuse or resale was frequently done, so those development environments contained facilities to support this.

Chris asked if I had ever programmed Java, Flex, or C# professionally, to which I responded that I hadn't. He said that because I had never worked in enterprise software, I had no concept of these facilities of other environments, and never knew what I was missing. He was implicitly critiquing the insularity of the Apple developer culture, and the tendency of many of them to assert the superiority of their technology without having any experience of other environments to compare to. In a later conversation over lunch, Chris told me he felt that the Cocoa community was very narrow. They never did anything else, outside of Cocoa, they stayed inside the nice little garden Apple provided them "where everything is beautiful." He felt that people ought to have a range of exposure to different programming languages and environments, in order to more objectively see their pros and cons. He held up the example of Rob Williams, a luminary in the Java world, who had learned Apple's iOS and Xcode and thus used that experience to criticize the experience of developing with the Eclipse IDE for Java. Chris was probably implying that there needed to be this kind of critique and cross-pollination inside the Cocoa community as well, that it could only benefit from exposure to the outside world. Chris felt capable of making this critique because he had been using Macs since a Mac Plus back in the 1980s, which he still owned, and also had first

learned to program for the Apple Newton handheld. This meant he was a bonafide Apple fan, and not some outsider taking shots because he hated Apple.

Another philosophical difference between Cocoa and Java/Flex that Chris expressed was in their respective memory management strategies. Java used an automatic scheme known as garbage collection, which removed completely from the programmer the need to worry about freeing up the memory for objects they had created. A program called the garbage collector would run periodically to look for unused objects and free them up. Objective-C, being based on the low-level C language, had used a manual memory management scheme called reference counting up to 2006, when Apple introduced garbage collection. However, this never took off on Mac OS X for compatibility reasons, and was never used on the iPhone, because garbage collectors run at unpredictable times. On phones which used Java and thus garbage collection (which included all Android phones), when the collector decided to run, the whole phone would freeze up for a second, and this could happen at random. Apple decided this was an unacceptable user experience, and thus to keep the iPhone's performance characteristics predictable, all iOS programming up till the release of iOS 5 in 2011 used the older manual reference counting scheme. During the first iOS bootcamp I visited, iOS 5 had been announced but had not yet been released, so the Ranch instructors were still teaching the old method, which, as we will see, was a prime source of trouble for many of the students. This had been a case in which Apple had prioritized the end-user experience over programmers' convenience, something that it often did. However, this did not mean Apple did not care about working to improve the programmer's experience. iOS 5 introduced the new Automatic Reference Counting (ARC) method of memory management. It largely (though not completely) removed much of the hassle of the old manual scheme, without incurring the indeterminate overhead of Java's garbage collection scheme. Chris said that Apple's use of ARC represented a completely different approach to memory management compared to most of the industry. Because garbage collection was slow, and because Apple owned its entire stack of software and hardware, it could attack the problem at the level of the compiler, which it had

control over, something Google, which does not control the Java language, can do. I seemed as if Chris, in a technical sense, admired that Apple, by controlling its platform vertically, could then solve a problem at the correct level it ought to be solved, rather than be constrained by the artificial boundaries of firms and markets, even if he did not necessarily like the fact that Apple had this kind of control.

Learning this new language made a lot of the students explicitly compare the merits of the various programming languages and environments that they knew. A number of developers in both iOS and Android classes had web experience, and compared Perl, PHP, Python, and Ruby on Rails. Others had enterprise software experience, and compared Objective-C with Java and C#. In the OpenGL class, which is a low-level technology and thus taught in C, one of the students was critical of contemporary computer science education being completely in Java. He said that the real world, most real code was still in C, but he told a story about a networking class he had taken in which the only person in it who knew C had a master's degree. If he were hiring for CIS (Computer Information Systems), not knowing C might be fine, but for a Computer Engineer, he needed people who knew C. It was because so many people only knew Java that there were simply not enough engineers who were experienced at manual memory management. If you knew C, then Objective-C's manual reference counting scheme was actually pretty simple in comparison.

Conversions

Although many students expressed resistance and hostility to the way things are done in Objective-C and Cocoa initially, over the course of the week, once they began to build working applications through the exercises, some began to see the merit behind the way Cocoa works.

A student named Mike, for example, said he was "blown away" by how little work he had to do to get his iPhone app's user interface to switch between portrait and landscape orientations when the phone is rotated. He noted that he didn't have to write any annoying code to get the views in his user interface to automatically lay themselves out again.

Victoria, who was struggling in the class initially, was by Wednesday starting to understand core concepts in the course through sheer immersion. She was no longer making basic mistakes and did not constantly need my help anymore. She was asking intelligent, probing questions in her quest to understand more, signaling that she knew enough to ask the right questions. Victoria, echoing others, told me that she felt like she had been drinking from a firehose. Despite feeling like she had been drowning and struggling to stay afloat, by now, things that she didn't understand from earlier chapters were finally starting to sink in. She was finally starting to "get it," to acquire a feel for Cocoa programming. Through typing out code and fixing her programs, she acquired concepts by experiencing them viscerally as almost infrastructural to the way she had to interact with the Cocoa libraries.

Later, when I returned to Historic Banning Mills as a student to take the OpenGL and Android classes, I met some iOS developers who had previously gone through either the Big Nerd Ranch bootcamp or an iOS class from some other training company. These developers exhibited more complete signs that they had fully converted to the way things are done in Cocoa and agreed with the merits of its approach. One notable developer in the Android class was Charlotte, a woman from Chattanooga, who ran her own company doing mobile development. She and another independent had considered forming a Limited Liability Company (LLC) to work together, and possibly hire additional contractors if business picked up further. She also writes applications for her husband's business, doing internal enterprise apps. Originally a PHP web developer, she had learned iOS the previous year, in 2010. She had wanted to take a Big Nerd Ranch course but her schedule precluded it, so she had taken a class from a company called Practical Development in Chicago.

Charlotte said that it had taken a while to wrap her head around the syntax of Objective-C, but now she's gotten used to it, to the point that she now feels it is "intuitive." She also noted that she really liked how Objective-C method names have labels for arguments so that they are self-documenting. Charlotte said that since she started programming iOS, her life had gotten easier. When she first learned iOS, she had felt that the Interface Builder program black-boxed the process of designing the

user interface too much. In iOS, designing the user interface is done largely graphically, using Interface Builder, in contrast to Android's development tool, which generates XML code for the interface, which is made available for developers to hand-tweak, and which, in practice, requires them to hand-tweak, in order to get their interface to look correctly. Interface Builder also generates output in XML format, but access to this is restricted and developers are warned not to touch it, because the system is fully automated, and their tweaks could cause problems. Charlotte said that most developers she knew wished they could go in and "futz" with the XML, partly because they didn't trust "the machine" to do the right thing, as they were used to having to go in and fix things on their own platforms. This, however, was the wrong thing to do on Apple's platform, as Apple's tools go to great lengths to generate good looking interfaces, and prevent developers from tinkering with it beyond laying it out graphically. Charlotte noted that this was one of the cultural things to learn about "doing Apple," the importance of trusting Apple's tools to "do the right thing." However, Charlotte noted that she may be biased in this assessment, as she confessed to being something of an Apple fangirl.

This attitude contrasted sharply with Damian, another student in the Android class, who had never taken iOS and did not plan to. Damian was openly hostile to Apple, and said that his instinct was to never trust Apple to do anything. He especially hated Apple fanboys. He worked for a company that needed corporate applications to talk to each other and integrate together. Apple's vertically integrated platform made it difficult to flexibly deal with the heterogeneous systems that needed to work together in the enterprise. Damian wrote mostly Java code for backend servers, though he had started out using Microsoft Visual Basic, also a black-boxed user interface editor. Damian was taking the Android class because he was bored with his job and was learning to program for it on the side. I felt that Damian was equally partisan as Charlotte was about his choice of platform. Damian argued that it no longer made sense (in 2011 when the class took place) to do only an iOS product and not Android, because over half the market was using it, and he did not think it should be ignored. While this argument has merit, statistics show that

even today, iOS applications make significantly more revenue than Android applications (Krakow 2014), so from my biased perspective this sounded like a rationalization for his predetermined preference; Damian certainly never intended to learn iOS and thus he himself would be ignoring the more profitable half of the market. However, he justified this by saying that he wasn't interested in consumer facing apps, for which it makes sense to learn iOS, he is only interested in corporate software. Android was a no brainer to him because he already knows Java, the language Android is programmed in, and he prefers its flexibility, as he hates Apple and its closed system. The corporate world needs flexibility, he argued. Damian noted that some of his friends had tried programming iOS but didn't like it, simply because it wasn't like Java, which was what he was used to. It seemed to be purely a matter of preference.

Charlotte and Damian thus represented two very different poles. Charlotte was a fully converted iOS developer, had bought into Apple's programming environment and approach, trusted that Apple's tools did the right things to help her develop, and felt that it had some real advantages for her, which more than justified giving up some flexibility and tying herself to its closed platform. Being an Apple partisan myself, I felt that she was "one of us." In contrast, my reaction to Damian was that he was an Apple hater and thus would be considered by the Cocoa community as an outsider, somebody who "didn't get it," who was hung up on the wrong thing. Since he was primarily interested in enterprise and not consumer software development, there was no practical reason for him to be interested in Apple at all. Yet to me his hostility and his preference for Java and Android felt equally as dogmatic as that of the most rabid Apple fan. Damian would be the kind of programmer Cocoa developers define themselves against. At least the students in the iOS class, having self-selected to take it, were at least open to learning it, as even those who might be initially hostile to its culture and ways of doing had an economic or professional incentive to learn.

Conclusion

This chapter examines the beginning stage of the social reproduction of Cocoa programmers, which for many is a class at the Big Nerd Ranch. Instructors there use a number of pedagogical techniques. Students copying existing code out of a book and typing it in builds up embodied muscle memory and familiarity with a repertoire of Objective-C idioms and Cocoa APIs. Subsequently debugging their program and trying to make it work is where students acquire much of the tacit knowledge of how and why Cocoa programs are constructed the way they are. This knowledge is simultaneously conceptual as well as material—it is through practical real-world examples, doing specific, concrete exercises that such abstract concepts as design patterns are presented to students. Their benefits are not initially explained, but through repetition over many chapters, learned tacitly as a sense of, "this is how and why I want to use delegation." The intense pacing of the course adds to this. Students feel as if they are constantly behind and inadequate, and learn to rely on each other to keep up. As the students feel as if they are drowning in a firehose of information, gradually the immersion results in concepts seeping in through osmosis. Eventually, once they get their programs working, the initial feelings of despair turn to elation as they realize they really can master what appeared so difficult and foreign at the beginning. These positive emotions motivate the student to forge ahead, opening them up to accepting both more knowledge and more of the cultural and normative attitudes that go with these ways of doing. Conversely, students attempting to learn Cocoa on their own, even with a Big Nerd Ranch book, often give up due to the initial frustration. This is mitigated in the context of the Big Nerd Ranch class, where the ever-present instructor provides answers to difficult questions and help when a student can't fix a bug. Use of humor, and the cultivation of a friendly, accessible and "I'm just like you" persona contribute to this. This is crucial to get students over the initial hump of Cocoa's steep learning curve. Once over the hump, students have felt the exhilaration of getting a Cocoa program working, and moreover, experienced how little code it took for them to get it working. The amount of code they had to write (or rather, copy) obscures the real work that went into what they accomplished, which was acquiring the tacit knowledge to understand what that code does, how it works, and why Apple designed it to work that way. Being able to

write a little code to accomplish a lot gives students a feeling of power, as well as aesthetic appreciation for Apple's designs. This leads to a greater feeling of trust in Apple's tools—since Apple has empowered me in this instance, I am more open to seeing what else Apple's tools can offer me. The result is a progressive buy-in to Apple's ecosystem and way of doing things. The work that is done in the Big Nerd Ranch class is as much emotional as cognitive. This affective work is probably the real reason for the Big Nerd Ranch's success in training Cocoa developers: it converts feelings of hostility, apathy, or frustration with Apple software into feelings of accomplishment, mastery, pleasure, aesthetic appreciation, and trust.

This affective work comes with powerful normative messages for how one ought to practice Cocoa programming, and thus be a proper Cocoa programmer. The message is reinforced that Cocoa programmers do not exist in a vacuum, but live in a community with established norms and conventions for practice that one should respect. At the Big Nerd Ranch, this comes in the form of the exhortation to "be stylish" and write "stylish" code. Write code this way, and you will be accepted into the community as one of us; you're cool. Do your own thing, and we'll look at you funny, criticize your code, or simply treat you as a newbie or outsider. You should do it this way, because everybody who's anybody in the community does it this way, including Apple, who's the coolest around. This has some practical benefits, because now everybody will be on the same page, understand the meaning of each other's code, which will make for more readable and maintainable software. Moreover, since everybody does it this way, Apple can make some assumptions and adjust its tools to take advantage of these conventions to better support what we do. This means that the more you conform to convention, the more likely it will be that you'll get benefits for free from Apple over time.

Thus, despite the discursive focus on the "independent" individual in the ideology of Cocoa, a significant amount of its moral order is based around the notion that programmers exist in a community of practice, which regulates such practice and normative behavior surrounding it. It is this social organization of the Cocoa community that we turn to in the next chapter.

Chapter 5: The Cocoa Community

Theories of collective practice

As we saw in chapter 1, the ideology of Apple and of indie Cocoa developers emphasizes the empowerment and pleasure in the creative making of individuals. However, indie developers are not merely a collection of atomistic entrepreneurs, each doing their own thing in isolation. Rather, such ideology is collective; it is because third party Cocoa developers, most particularly the indies, share a coherent body of knowledge, practices, skills, values, and identity—in short, a techno-cultural frame surrounding Cocoa technology—as members of a larger Cocoa developer community. This community is maintained through both face-to-face meetings and online discourse, through which knowledge, practices, and values are shared and spread. This chapter is concerned with describing the structure of the Cocoa community, what Sharon Traweek calls its "social organization." (Traweek 1988, 7)

The "Cocoa developer community," as I use it, is an actor's category, the way Cocoa developers themselves describe their collectivity. A number of different analytical categories can be mapped onto it, with various degrees of success. For one, the "Cocoa community" can be called a "community of practice," in the sense of Jean Lave and Etienne Wenger (1991): it is a community of practitioners of a craft skill, powerfully concerned with transmitting knowledge and normative practices to new generations of practitioners. Central to Lave and Wenger's articulation of a community of practice is that it is a community whose membership is based on "participation in an activity system about which participants share understandings concerning what they are doing and what that means for their lives and for their communities." (Lave and Wenger 1991, 98) This concept comes from their attempt to theorize learning as an activity situated in practice and in social relations with other practitioners, rather than simply a cognitive transfer of information. Learning is described as "legitimate peripheral participation" in practice with other newcomers, under the supervision of old-timers. In participating in practice, newcomers gradually become fuller members, and acquire the identity of a practitioner. The

movement from peripheral to full membership *is* learning, which is not simply about the acquisition of knowledge or skills but of identity. "…learning and a sense of identity are inseparable: They are aspects of the same phenomenon." (Lave and Wenger 1991, 115) This theory of learning is modeled to a large extent on ethnographic studies of apprenticeship, enlarged to encompass not only midwifery, butchery, tailoring, spirit mediation, but also Navy quartermastering and becoming a member of Alcoholics Anonymous. Because socialization and enculturation in a community of practice is central to Lave and Wenger's theory of learning, apprenticeship is seen to be the model for all learning, and formal classroom instruction becomes rather a deformation of this, focusing more on talk *about* practice rather than talk *within* practice (109), sometimes restricting access to actual participation where learning actually takes place (104), and in some cases producing not future practitioners but simply informed adults (100). For communities of practice, however, learning must on some level be about the social reproduction of the community itself, of training apprentices to be future masters. This reveals an inherent tension, in that as newcomers become full participants they necessarily change the community and knowledge of practice through their own ideas; this is the mechanism for generational shifts in knowledge and practice.

Brown and Duguid (1991, 2001) have expanded on this notion of communities of practice. In Lave and Wenger, learning is simultaneously learning to be as learning to do, and to know. Brown and Duguid point out that the reason learning is communal is because knowledge acquisition occurs only with the transfer of practice, which depends significantly on tacit knowledge, such as that held by Xerox repair technicians (Orr 1990). "…tacit knowledge is required to make explicit knowledge usefully tradable or mobile. Only by first spreading the practice in relation to which the explicit makes sense is the circulation of explicit knowledge worthwhile… Knowledge, in short, runs on rails laid by practice." (Brown and Duguid 2001, 204) They also point out the consequence of the fact that communities of practice transcend the boundaries of individual firms or organizations. Knowledge which might be "sticky," in other words, have a hard time traveling between parts of

an organization, may simultaneously be "leaky," easily traveling outside it to other members of the community of practice. Brown and Duguid cite the example of the Graphical User Interface, which, although created at Xerox PARC, found an easier time traveling to other practitioners at Apple and Microsoft than to the rest of Xerox.

The word "community" can imply locality, as well as constant face-to-face interaction. Brown and Duguid note that such communities as professions and academic disciplines are more like communities of communities, where sub-communities of tightly bound practitioners interact on a constant basis, but meet up with the larger global community only periodically at conferences, or over print or online media. Such larger "communities" are more akin to Knorr-Cetina's notion of "epistemic cultures," Strauss's notion of "social worlds, or Ziman's notion of a "public." (Knorr-Cetina 1999; Strauss 1978; Strauss 1982; Strauss 1984; Ziman 1968) Brown and Duguid, however, prefer the term, "network of practice." "Reflecting what binds these networks together and enables knowledge to flow within them, we call these extended epistemic networks 'networks of practice.' Practice creates the common substrate. With the term *network*, we also want to suggest that relations among network members are significantly looser than those within a community of practice. …unlike in communities of practice, most of the people within such a network will never know, know of, or come across one another. And yet they are capable of sharing a great deal of knowledge." (Brown and Duguid 2001, 205)

Yuri Takhteyev has noted that for Brown and Duguid, local communities of practice thus serve as nodes in a wider network of practice, through which individuals (and thus knowledge) may freely travel. However, he notes that this language of network places the focus back on the individuals traveling the network, downplaying the collective nature of shared practices, norms, and meanings. Instead, Takhteyev prefers Strauss's language of "social worlds." "This term has often been used to denote loose collections of people united by interests, outlook, or activities. Social worlds can be quite large in their spatial dispersion and (unlike most notions of 'community') do not carry the implication that the members know each other or

interact on a regular basis." (Takhteyev 2012, 26) Instead, Takhteyev describes the global collective of software developers as a "world of practice." Takhteyev uses the term "to refer to systems of activities comprised of people, ideas, and material objects, linked (and defined) simultaneously by shared meanings and joint actions," both material practices and discursive values. Such worlds can be global, but are simultaneously rooted in local contexts. Takhteyev uses Giddens' notion of disembedding and reembedding (Giddens 1991) to describe how practices rooted in local context are made to travel to other local contexts: they must be disembedded from particular local elements to become mobile, then re-assembled, in combination with elements from the new local context, in order to successfully move.

Takhteyev's use of the term "world" may also evoke Gabriella Coleman's use of the term "lifeworld," drawn from the phenomenology of Merleau-Ponty and Alfred Schutz. Coleman uses "lifeworld" to denote intersubjective experience, which involves "sites, practices, events, and technical architectures" as well as affective senses of "excitement, humor, and sensuality." (Coleman 2013, 28)

Is the global collective of Cocoa developers a "community," a "network," or a "world" of practice? All three terms have advantages and disadvantages. Takhteyev largely uses the term "world of practice" to refer to the global world of software developers, writ large. Going with that usage, then, is the subculture of Cocoa developers its own sub-world? Brown and Duguid's term, "network of practice," can be used to describe this meso-level of organization, with "community of practice" referring specifically to tight local communities with frequent face-to-face interaction. However, Lave and Wenger's original definition of communities of practice did not limit it to small, purely local communities, but referred to whole crafts and professions as well. Moreover, restricting the term "community" to local, physical communities excludes communities that interact virtually, online. In addition, like Takhteyev, I feel that the term "network" takes focus away from the collective nature of what these groups share, and places focus back on the individuals who travel between the network's nodes. My actors, Cocoa developers themselves, refer to having a "community," because they feel a sense of collectivity,

a sense that they are a group different from other groups of programmers. As a group, Cocoa developers have "a shared past, hope to have a shared future, have some means of acquiring new members, and have some means of recognizing and maintaining differences between themselves and other communities," which is for Sharon Traweek, a sufficient definition of what constitutes a community (Traweek 1988, 6). I wish to highlight this sense of "groupness" that Cocoa developers feel, rather than their distributedness. Certainly, they have social connections to each other in the form of a network, but what matters is whether or not these connections are felt to belong inside the boundaries of the group or fall outside of it. For these reasons, I refer to the collectivity of Cocoa developers as a *community* of practice, although I take Brown and Duguid's notion of knowledge traveling along paths laid by material practice.

Another term that could be used to describe the Cocoa community is as a "public." Much of the community's interactions are mediated through the online discourse of web sites, blogs, mailing lists, IRC, Twitter, and podcasts. A "public" better describes a collectivity mediated through discourse. Michael Warner defines a public as "the social space created by the reflexive circulation of discourse" among strangers[37] (Warner 2002, 90). This definition is circular: the space of discourse that is a public exists only because it is being addressed by discourse, and is thus to some

---

[37] Although many Cocoa developers are not strangers to each other, I would argue that a "public" does not necessarily have to be composed strictly of strangers, only that it may consist of a continuum in which some are strangers to each other at one end, while others are not. Publics formed through broadcast media are more likely to be composed exclusively of strangers, but online networked publics, because members have the ability to speak directly to each other, can create relationships between members, so gradually they no longer remain complete strangers. Certainly many Cocoa developers who follow or read the blogs of the big names remain strangers from them, but sometimes they attend conferences and meet them in person.

extent “virtual” or “imagined” (much like Benedict Anderson’s notion of “imagined communities,” in which a public reading the national newspaper imagines their nation as a single community despite not having personal contact with the rest of the polity) (Anderson 1991), but also real, in that members of the discursive public are actual persons who actively participate or at least are attentive to the discourse. Attention to the discourse is all that is necessary to become part of the public; no other institutional requirements exist. Although publics are composed of strangers, each member reads the discourse as being at once personal and intimate, and yet simultaneously addressed to myriad other strangers. The texts that address a public are not isolated texts but must circulate through time, and exist in social relation to other circulating texts through dialogue and inter-textual citation. Publics create a world of understanding for its members, and can have agency (Warner 2002).

Publics constituted by online discourse have been called “networked publics” or a “networked public sphere.” (Benkler 2006; Tierney 2013) Thérèse Tierney defines a “networked public” as “a community that forms among some set of members of a social media site. They are defined as publics that are restructured by networked technologies as spaces and audiences bound together through technological networks.” (Tierney 2013, 32) For Tierney, networked publics are to be differentiated from both “spatial” publics that gather in physical spaces and broadcast media publics. Yochai Benkler similarly differentiates the networked public sphere from the public sphere as constituted by mass media. Mass media is broadcast one-to-many, and the public sphere constituted by it, the “audience,” does not easily communicate back to the broadcaster. Because the technologies involved in mass media are capital intensive, mass media follows an industrial model that easily leads to monopoly, and the concentrations of wealth required for mass media create a passive public that can be easily manipulated by money. According to Benkler, the networked public sphere, on the other hand, is many-to-many, and fundamentally participatory in nature, because the means of production and communication are cheap and available to individuals. Networked publics, unlike undifferentiated mass mediated publics, tend to be self-organized affinity or interest

groups that are deeply engaged in shared concerns (Benkler 2006, 242). Benkler says that "The networked public sphere is not made of tools, but of social production practices that these tools enable." (Benkler 2006, 219) By these practices, Benkler means the user-driven peer production of content that often comes under the rubric of "Web 2.0," in which information and content are produced as much as consumed by users. Benkler considers networked publics to be much more conducive to participatory and democratic governance than mass media publics, because they are harder, though not impossible, to control, with many fewer single points of control, without giving up on their economic advantages (Benkler 2006, 271).

Chris Kelty uses the term "recursive public" to refer to the community of free software and open source hackers and programmers who have developed much of the infrastructure of the Internet. For Kelty, this public is "recursive" in a further way: this public's discourse is made possible by infrastructures, such as the Internet, whose development is the very topic of the discourse itself. In other words, this public is "recursive" because their very ends are also the very means, the conditions of possibility, of their own existence. "A recursive public is *a public that is vitally concerned with the material and practical maintenance and modification of the technical, legal, practical, and conceptual means of its own existence as a public; it is a collective independent of other forms of constituted power and is capable of speaking to existing forms of power through the production of actually existing alternatives."* (Kelty 2008, 3) "Recursive publics are publics concerned with the ability to build, control, modify, and maintain the infrastructure that allows them to come into being in the first place and which, in turn, constitutes their everyday practical commitments and the identities of the participants as creative and autonomous individuals." (Kelty 2008, 7)

The Cocoa community is *not* a recursive public in Kelty's sense, although it is a public constituted through the circulation of discourse, much of it online. It is not recursive because Cocoa developers are not primarily interested in producing the infrastructural technologies that make their own production and their own discourse possible. These technologies are either produced by other developer communities

(such as open source hackers), or by corporations such as Apple, which places restrictions on the ability of outsiders to contribute to the technologies; Cocoa developers remain mere users, not producers, of these technologies, and are in most cases content to remain users. Cocoa developers, while involved in sharing knowledge and even code, are also more interested in making a profit from their work, and are not generally ideologically committed to the project of free software. Rather, as we explored in chapter 1, they are committed to a different moral order, that of the "indie" developer: a social vision in which software is developed, and useful tools produced, by millions of independent developer-entrepreneurs. Nevertheless, these indies are members of a collective, a public whose discussion of shared concerns, values, and practices serves to unify otherwise isolated individuals.

The "Cocoa public" is not only a space of strangers brought into being by online discourse, however. Although in some sense it is true that, like a profession or academic discipline, not all of its members will ever meet all the others, a surprising amount of them actually do have face-to-face contact. Conferences and conventions are one space where this occurs, transforming the nature of relationships from reputational and textual to fully embodied. Of free software hackers, Coleman says that although "Public discourse is a vehicle through which hackers' immediate experiences with technology along with their virtual and nonvirtual interactions with one another are culturally generalized… In-person interaction is also a pervasive feature of their lifeworld, working to confirm the validity of circulating discourse." (Coleman 2013, 45) Hacker conferences or "cons" "reconfigure the relationship between time, space, and persons, allow for a series of personal transformations; and perhaps most significantly, reinforce group solidarity. All of these aspects of conferences make them ritual-like affairs." (Coleman 2013, 47) "As if making up for the normal lack of collective copresence, physical contiguity reaches a high-pitched point. For a brief moment in time, the ordinary character of the hackers' social world is ritually encased, engendering a profound appreciation as well as awareness of their labor, friendships, events, and objects that often go unnoticed due to their piecemeal, quotidian nature." (Coleman 2013, 47) As we will see, Cocoa developer conferences

are not dissimilar from hacker conferences, although more focus is spent on presentations rather than hacking. However, like hacker cons, Cocoa conferences provide a place where older generations of developers share stories with younger ones, providing a sense of being involved in a shared, collective enterprise. Similarly, at conferences, Cocoa developers, like hackers, can take on "the awareness of a shared social commonwealth" with a "decidedly moral character…" (Coleman 2013, 53) At hacker conferences, Coleman also notes that the semiotic signs of sameness are pervasive: "Most people are attached to their computers, and share a common language of code, servers, protocols, computer languages, architectures… wear geeky T-shirts. With each passing day, the semiotics of sameness are enlivened, brought to a boiling point, as participants increasingly become aware of the importance of these personal relations, this form of labor, and F/OSS [Free/Open Source Software]—in short, the totality of this technical lifeworld." (Coleman 2013, 55) Described thus, the experiential social world of the conference creates, maintains and strengthens, in a ritualized and affectively heightened situation, identity and collective belonging. Coleman explains that theorists of publics such as Michael Warner, Charles Taylor and Jurgen Habermas have not paid sufficient attention to the "ways that physical copresence might sustain and expand discursive forms of mediation. Perhaps the circulation of discourse captivates people so strongly, and across time and space, in part because of rare but socially profound and ritualistic occasions, such as conferences, when members of some publics meet and interact" and highlight the importance of "social enchantment and moral solidarity" generated by the ritualized characteristics of conferences to modern publics (Coleman 2013, 59). "The cultural ethos and class of a group is inscribed in where they are willing to meet, what they are willing to do with their bodies, what they are willing to do with each other, and what they are willing to express during and after these conferences. Despite the differences in their moral economy, conferences tend to be the basis for intense social solidarity that sustain relationships among people who are otherwise scattered across vast distances." (Coleman 2013, 60)

As we will see soon, Cocoa developers also meet face-to-face more frequently with their fellows at local clubs and meetups, in addition to meeting occasionally at conferences. These local communities are at once tied in to others through online discourse on the Internet. It is because of the prevalence of these physical, local relationships, and as we will explore, the exportation of the norms of some of these local communities to the larger online public, that I prefer the term "community" over "public" in describing the collectivity of Cocoa developers. And although the larger "community" is really a "community of communities" or maybe even a "network of communities," it is still a group that relies powerfully on the strength of affective relationships built face-to-face, collectively, rather than solely on the networking between individuals traveling between localities, though the latter does play a role. For this reason, I prefer the term "community" over "network."

This Cocoa developer community is tied together by core values that delineate identity, group membership, common practices, a sense of collective mission, and a particular understanding of the world, the role of technology in it, and their role in developing that technology. Like free software hackers, Cocoa developers have a sense of a particular moral, social, and technical order in mind, an "ideology," in Geertz's non-pejorative sense of a system of cultural and symbolic values that structures the way members interpret their world (Geertz 1973). Chris Kelty prefers Charles Taylor's notion of "social imaginaries" to describe "ideas of order that are both moral *and* technical—ideas of order that do indeed mix up 'operating systems and social systems'" (Kelty 2008, 43), tying together ideas and material practices. But what is the analytical place of the technologies that software developers are interested in developing and using? Sociologists of technology associated with the Social Construction of Technology (SCOT) method have proposed an analytical concept that highlights the interplay between technology and society. This is Wiebe Bijker's notion of "technological frame." (Bijker 1995)

As I discussed in the introduction, technological frames encompass the commitments of technology makers to their ideas about the meaning of a technology (theorized as a "sociotechnical ensemble"), its purpose, and direction for future

development. What I find useful in Bijker's concept is his corollary that social groups involved in the use or development of a technology have varying degrees of "inclusion" in the frame (Bijker 1995, 283–5). Remember that for actors with high inclusion, the key meanings of a technology are taken as fundamental assumptions. For example, for longtime Cocoa Mac developers, the benefits of Cocoa's design for programmer productivity and software maintainability are unquestioned. At the same time, however, these expert insiders are able to develop and modify the technology and thus see it in its full differentiated and malleable complexity. For example, Cocoa may debate the relative merits of various components of Cocoa and whether or not they live up to the intended ideal that Cocoa's design strives for as a whole. For actors with low inclusion, technologies appear as a monolithic black box, and their acceptance or rejection of it must be total. Because they are peripheral to the frame, they are able to question its fundamental precepts. For example, relative newcomers to iOS are frequently skeptical or critical of the way Cocoa as a whole operates because they are less familiar with it.

This notion of social groups with high and low inclusion in a technological frame is useful particularly because it can be mapped onto the division between "oldtimers" versus the "newcomers" in Lave and Wenger's notion of a community of practice (Lave and Wenger 1991). Of course, community involves what Traweek calls "cosmology," which includes a group's knowledge, skills, beliefs, and values (Traweek 1988, 7). This is "ideology" in Geertz's non-pejorative sense, a system of meanings that constitutes a group's culture. Culture also incorporates identity and feeling, which are largely missing from Bijker's notion of technological frame, which tends to deal with utilitarian or instrumental orientations to technology. I need a term that incorporates aspects of all of these analytical concepts, ideology, culture, and technological frame, in which material practice, material artifacts, technological meanings, community membership, professional identity, affect, learning, and worldviews are all encapsulated together. I call this a "techno-cultural frame," extending Bijker's term to incorporate these additional aspects.

The concepts of communities of practice and inclusion resonate with an ethnographic study of Washington D.C. area iOS developers by Qiu, Gopal, and Horn (2011). Qiu et. al. do not draw on STS scholarship but on the literatures on professional identity and institutional logics. Their analysis of different motivations of iOS developers groups them into two categories: those who follow the "professional logic" of indie app developers, and those who follow "market logic." They note that among those developers who follow the professional logic, an identity of being a "builder," someone who reflexively wishes to make an app for one's own use, is central, as is a "craftsman" identity, a concern with software quality and following professional norms and practices associated with good software engineering, as well as concerns with aesthetic beauty. Drawing on Van Meeteren (2008), Qiu et. al. show that these developers also follow a marketing strategy relying on an app's quality and on peer recognition among the community.

Developers who follow "market logic" take a different approach to coming up with app ideas, and implementing and marketing those apps. Because for many new iPhone developers, it was not clear what would make a successful app, the strategy was to try a lot of different things, implement them as quickly as possible without regard to quality, and essentially throwing them at the wall to see what would stick. The initial gold rush of the App Store and its rather lax policies encouraged such a market approach, as a key worry during this period was that the first app to do something would capture the market and make subsequent competition difficult. As the mobile app market matured, this was shown to be a fallacy, as apps that had previously been at the top of the charts could easily fall off—nothing had a permanent lock on the market.

Qiu et. al. describe a process of synthesis of the professional and market logics among iOS indie entrepreneurs, whereby some who follow professional logics have to, by virtue of the economics of the App Store, incorporate some market strategies for boosting their sales. On the other hand, developers who followed the market logic of writing a bunch of low-quality apps, as they became more involved

in the community, gradually moved more towards the professional logic of focusing on one app and making it high quality.

While Qiu, et. al. capture the difference between developers who follow "professional logic" versus "market logic," but by attributing these to the "logics" of markets and professions, they miss the cultural and technological commitments of the community of practice being studied. The "professional logic" that they describe is in fact very specific to third party developers of Apple software. While sharing some elements with software developers in general, other aspects of it are specific to the technological frame of programming with Cocoa for Apple devices. Moreover, the followers of "professional logic" are more centrally located within the Cocoa community, dominate the online discourse, and are recognized as knowledgeable experts, while those following "market logic" tend to be newcomers. Those who follow "professional logic" are highly included in the Cocoa technological frame, oldtimers within the Cocoa community of practice. Those following "market logic" have low inclusion in the technological frame, and are peripheral in the community of practice. The extent to which these newcomers begin to incorporate more of the "professional logic," i.e. the norms and values associated with the identity of a Cocoa developer, marks the extent to which they are moving from lower to higher inclusion and to which they become less peripheral members of the community of practice. How then to explain why those who began by following "professional logic" began to incorporate elements of "market logic?" I would contend that the oldtimers' rejection of market logic came out of the period when they were primarily indie Mac developers, knew their competitors by name and considered them friends. The market for indie Mac apps was small and collegial. The iPhone App Store, however, drastically expanded the competitive environment, and relying purely on word of mouth simply no longer sufficed. At least some effort needed to be expended on more traditional marketing in the crowded App Store. The synthesis of the two logics thus represents both gradual assimilation of newcomers to the norms of the existing professional community, but also a more pragmatic attitude towards the market from oldtimers in a highly competitive new environment.

For Cocoa developers, their technological frame encompasses user interfaces, aesthetic design, and a coherent vision for a singular end product of their labor, the "app," which ought to be both useful and pleasurable to use. At the same time, the frame also emphasizes both pleasure and productivity in the process of making the app itself, as well as coding practices that promote sharing of knowledge and collaboration and the quality of an app as an end in itself, as a work of a master craftsperson. We will now take a closer look at some moments of crisis and transition in the Cocoa community, and what this reveals about the technological frame.

The Developer Community from NeXT to Cocoa

What came to be called the "Cocoa community" in the early to mid-2000s was in fact a mixture of two previously separate groups, the NeXT developer community, and converts from the original Macintosh developer community. As we saw in chapters 1 and 2, because of of NeXT's limited marketshare, it never had a large developer community in the first place, and when the company changed strategies away from hardware, many of those developing end-user applications went out of business, while the rest turned to contracting. This limited market had several consequences. First, developers who exclusively programmed for NeXT, such as Wil Shipley, claim they did so out of their pleasure in programming with NeXTSTEP and a belief that it was a better way to program, and stuck with NeXT during the dot.com boom, a time when they might have been able to make millions more at a startup. As we saw in chapter 1, being loyal to a platform that might go out of business was a self-conscious decision made not for rational reasons of income, but because it was part of their identity and because of their affective pleasure using NeXT. Second, such self-selected people were few in number, and everybody seemed to know everybody else. The entirety of the community was on the NeXT developer mailing list.

> The NeXT community, by that point, was probably pretty small… the community as it existed probably consisted of people who were on the NeXT programmer's mailing list. And I can't imagine it being more

> than a few hundred. And most of that community was consulting, and so on.
>
> (Ken Case Interview, March 23, 2012)

Third, in part because of this mailing list, which was maintained by NeXT and frequented by NeXT employees, relationships formed between third party developers and NeXT engineers. These were strengthened considerably at NeXT conferences and user groups, where developers and users were able to meet NeXT engineers, because both the community and NeXT itself were small enough that one-on-one, personal connections were possible. Developers already knew who the engineers were because their names were included in the source code header files of APIs that they were using, and at conferences they were finally able to meet with those very engineers. The reason that NeXT engineers were so accessible to third party developers was partly because NeXT itself was a relative startup, and thus was informal, but also because its developer community was so small that NeXT had to cherish every single developer. The small size and importance of individual developers meant that personal relationships could be formed between them and NeXT engineers. The intimacy of the NeXT developer community had the feel of a small town, where everyone was on a first name basis, even with key engineers and managers at NeXT.

> Like, we used to be the platform. The NeXT community—that was this one little insular group in a way.
>
> (Luke Adamson Interview, February 22, 2012)

Apple's purchase of NeXT in 1997 did not significantly change this among those NeXT developers who had already formed these relationships. With the rebranding of NeXTSTEP's AppKit and Foundation libraries together as "Cocoa," the NeXT developer community simply renamed itself the "Cocoa" developer community. However, Cocoa technology and its developer community faced resistance from the existing community of Macintosh developers. Recall that initially, Apple's plan for Mac OS X was for Cocoa (or "Yellow Box" as it was then called) to be the only native API, but this met with resistance from corporate Mac developers

with legacy code bases. Apple created the Carbon APIs, based on the original Mac Toolbox APIs but fully OS X native, to appease these developers.

During this transition, the former NeXT community, now calling itself the Cocoa community, was uneasily becoming integrated into the larger Mac developer community. Within Apple itself, NeXT technologies and managers were ascendant, and engineers working on old Mac and Carbon technologies felt that they were being shunted aside, something I personally witnessed as an engineer at Apple. This tension spilled out into the developer community, as existing Carbon developers began to feel slighted by Apple's decisions, while Cocoa developers, who long felt beleaguered during the NeXT era, were feeling newly invigorated and empowered. Partisan rhetoric in the blogosphere enflamed passions on both sides.

> I'm willing to just be arrogant about it—I'm willing to just say, if you program in Carbon you should be fired from Apple. Which I said a little while ago and, you know, some Apple engineers wrote me and they were really mad. "Fuck you, I'm a great engineer, I wrote this, you don't know me, I shouldn't be fired." I'm like, "Whoa, whoa, whoa. I'm just saying Carbon sucks. I'm not saying you personally, Bob Smith, should be fired." I'm sure you're great, I'm sorry. So I'm a little bit of a firebrand.
>
> (Wil Shipley Interview, April 18, 2012)

Nonetheless, from 2001 through 2006, among small Mac shops composed of less than a handful of developers, the new Cocoa technology began to be embraced, and such independents saw much in common with their peers who had been using NeXT technology prior to NeXT's acquisition by Apple. A new generation of Mac programmers began to learn Cocoa as Apple's preferred method to write Mac applications. More importantly, small Mac developers and small NeXT developers began to see themselves as engaging in a common enterprise: "indie" development, independent of large corporate firms, where developers who considered programming their vocation would maintain creative control over their work. As we discussed in chapter 1, the attitude of such indies as Wil Shipley and Brent Simmons was to try to achieve the standards of quality, usability, and aesthetics of Apple's own applications, and by winning Apple Design Awards, they were showing that it

was not only possible, but good for business. Among the Cocoa community, Shipley's Delicious Library became an exemplar for the heights an indie application could achieve, and Shipley motivated many young programmers, such as Mike Lee, to become Cocoa developers.

> With apps that are well respected… I was shocked when I saw how well Wil did with Delicious Library. It blew my mind, it changed my whole perception of, Wow, really, a little Mac program, you can make that much money doing this? …Rogue Amoeba is another example, I always saw Rogue Amoeba's stuff and I'd be, wow I wonder if it's just two guys and this is what they do in their spare time—Rogue Amoeba, what have they got, like, ten people? Twelve people? …They are really charging ahead, showing the way for independent Mac software.
>
> (Chris Parrish Interview, March 2, 2012)

In the period from Mac OS X's release through 2006, the year before the iPhone's release, the Cocoa community, especially as it was constituted online through blogs, was dominated by the indies, a great many of whom were concentrated in the Seattle area. The Macintosh remained a minority platform, and and thus the community remained relatively small and tight knit:

> I feel like in the PC community there really aren't that many, maybe that's just because the Mac community is smaller and closer-knit. That you tend to get more of the community feel.
>
> (Adam Preble Interview, August 8, 2011)
>
> When your programming community is not that big to start out with, then you tend to encounter more individuals. And for the longest time, the community has kind of been in underdog mode, so there is no the whole Microsoft and Windows thing, the Mac has infinitesimal market share, but we're still working on our apps and still having fun programming, so there's kind of like yeah, we're sticking it to the man, or we're unique little sunflowers because we're using this technology that not everybody else is doing.
>
> (Mark Dalrymple Interview, April 11, 2012)

Dalrymple and Preble both point to the fact that the Mac was a minority platform for its coherency. Since the 1980s, Steve Jobs and Apple marketing cultivated a rebel image for the Mac and its users, and feeling different,

misunderstood, superior to others, militantly fanatical, and perennial underdogs became part of many Mac users' identities (Kahney 2004). This became more pronounced as Windows eroded the Mac's marketshare and threatened to put Apple out of business in the 1990s. Jobs' return revitalized the company with the Think Different marketing campaign, which further elaborated on the idea that Mac users were iconoclastic. If Mac users thought of themselves as a marginalized but special elite, then Mac programmers and former NeXT programmers must have felt that they were part of an even more select group. Even including Carbon converts, the Cocoa community was still small enough that members felt they could still name every one of them personally.

> I joke with people, I'm like, five years ago I think I knew every Objective-C programmer. Of course I didn't but it feels that way. Felt like you knew everybody.
>
> (Chris Parrish Interview, March 2, 2012)
>
> In 2005, I think it was still a small town, basically. It was distributed, I would call it one large community, but with a number of hot spots, obviously Seattle, San Francisco, Boston.
>
> … If you did something good, if you had an app out there that was pretty good, people are going to know who you are.
>
> If you had a blog, obviously that would help a ton too. Yeah, it was a piece of cake. Well, not that writing a good app is easy, but still. If you just did that—step one.
>
> Yeah, all twenty people knew all twenty people.
>
> (Brent Simmons Interview, February 17, 2012)

This small-town feeling, where everyone knew everyone else, was facilitated by physical meetings such as conferences, especially conferences organized by the community itself, such as C4, where indies were often among the featured speakers.

> What was great about C4 was, it was a small group of the people that were probably your role models, right? That was who was at that conference. People who were brilliant, or people who you really respected, [or] brilliant that you hadn't known… and so it's full of all of the superstars of the Indie, Mac at the time, community, at C4.

> Yeah, so Shipley and Brent, and Gus, and all the Rogue Amoeba guys, and you know, Rich Siegel from BBEdit, and BareBones and from Tidbits, just people in the community, developers or not… All those people, at that time you could count all of them, you could keep track of all of the well-known Indie Mac developers.
>
> (Chris Parrish Interview, March 2, 2012)

The prevalence of indie speakers at conferences, the dominance of their apps at the Apple Design Awards, their frequent appearances in Mac podcasts and press articles, and their social ties with Apple employees, has made indie Cocoa developers quasi-celebrities in the Mac world. The fact that the community was so small meant that everybody was on a first name basis. Online, it also meant that most developers followed each others' blogs, generating conversations and debates with each other.

> …back a long time ago before iOS, the development community was so small that even if you were a nobody you were still a somebody.
>
> And your name meant something, right? And you were attached to your name so I found a lot of helpful people along the way.
>
> (Joe Conway Interview, July 15, 2011)

However, as Conway notes, the release of the iPhone in 2007, and the App Store in 2008, changed the makeup of the community. Existing Cocoa developers on the Mac now faced an influx of newcomers. As I will show in the next section, although the Cocoa Mac developers became the core of this enlarged community, many felt that it was losing its earlier small-town feel. Worse, they worried that the community would lose its values.

The iPhone Gold Rush: A Community in Transition

As discussed in earlier chapters, the opening up of the iPhone to third party software development through the App Store sparked a new "gold rush" amongst both software developers and investors to cash in on the mobile "revolution." (Wortham 2009) Soon, large corporations began to see the need for their own iPhone apps, just as they had seen a need for a website during the dot.com boom. Developers

who had previously written software for Windows or the web now flocked to Apple's platform, eager to learn Objective-C and Cocoa so that they could "get in on the ground floor" of mobile and possibly get rich.

When the iPhone was first released in 2007, Apple had not initially allowed third parties to develop applications software for the device. Only Apple could write true "native" software for the iPhone, that is, software written using the platform's native programmer interfaces (APIs), rather than with a translation layer on top, which not only would make programs slower but would likely make its user interface look different from that of native apps. In the original iPhone, third party developers were relegated to writing web applications that would run in iPhone's Safari web browser, which not only made them slower than Apple's built-in apps, but also precluded them from taking advantage of many of the iPhone's hardware capabilities, such as its GPS and sensors. Soon after the announcement, the Cocoa developer community on the Macintosh began to clamor for this policy to be changed. They understood that the technical underpinnings of the iPhone were based on Mac OS X, with a Unix operating system layer underneath, and Objective-C based, Cocoa-like APIs on top. This meant that, even though Apple had put in place security measures preventing users from programming and installing their own software on the iPhone, in theory, this was perfectly possible, since they understood the iPhone to be not really a phone, but a pocket-sized Macintosh computer. Technically proficient early users of the iPhone discovered that they could hack the phone to get around Apple's security measures, opening up the phone for third party applications, customizations of the iPhone's user interface, and allowing the phone to be used with carriers other than AT&T, Apple's initial exclusive partner. This process became known as "jailbreaking." Soon, an underground market of applications written by hackers for jailbroken phones had sprung up, including games and Twitter clients written by prominent Cocoa developers such as Craig Hockenberry. Clearly, Apple's users were telling the company that there was an enormous interest and potential in allowing legitimate third party development for the iPhone.

In March of 2008, Apple did an about-face and announced an official, legitimate App Store for the iPhone, which would go live in July of the same year. Many developers today suspect that this was Apple's plan all along, but Walter Isaacson's biography of Steve Jobs suggests that Jobs' penchant for absolute control over the platform and users' experience of it overruled the opinions of voices at Apple who clamored for a more open platform (Isaacson 2011). It is possible that after the iPhone's successful initial launch, the spontaneous appearance of an underground jailbreak app market proved there was enormous demand for third party apps, convincing Jobs to change his mind and open up the iPhone as long as measures could be put in place for Apple to retain control over the market. This decision turned out to be extremely beneficial for Apple, as the creation of this ecosystem of third party apps would help lock in users to the platform, generate additional revenue for the company, and over time, become one of iOS's primary advantages over later challengers such as WebOS and Windows Phone, which, despite having excellent technologies and user interfaces, languished from lack of apps. Moreover, it allowed iOS to benefit from the creative energies of third parties, rather than relying on its own employees to determine, and thus limit, what functions the iPhone should have.

To facilitate third party software development, Apple created a Software Development Kit (SDK) for the iPhone, a set of APIs and libraries that developers could use to build their applications. This SDK, named "Cocoa Touch" by Apple, was based on similar design principles as the Cocoa libraries on Mac OS X, and used the same Objective-C programming language. This made app development much easier for developers than was possible in "jailbreak" app development. Jailbreak programming consisted of hacking, using unauthorized tools to discover the function calls Apple's own apps were using and writing code that called those functions. This meant that the process was much like fumbling in the dark, without an overall picture of how the iPhone's system was designed. The benefit of jailbreak, however, was that with the iPhone hacked open, all of the iPhone's native functionality was available. With officially sanctioned app development through the SDK, app

functionality was limited to only that which Apple exposed via its interfaces. For example, third parties are still not allowed to write software that customizes the iPhone user interface's entire look and feel (known as "skinning"), a facility available on Android and on iPhone only through jailbreaking. Cocoa developers argue, however, that going through Apple's official Cocoa Touch SDK has a significant benefit. Cocoa Touch provides a high-level, object-oriented way to write applications that is consistent and conceptually coherent. Individual APIs undergo rigorous consideration at Apple for inclusion or exclusion in the SDK. Cocoa developers such as Mike Lee argue that such thoughtful design put into the APIs helps developers write in a more disciplined fashion to consistently produce higher quality, less buggy, apps.[38]

---

[38] "There are a couple of differences [between programming in jailbreak and using the official SDK]. One is that, when you're working in jailbreak, you're working in, basically whatever the phone can do, you can do. So you're limited by the phone itself and your imagination. Obviously with the SDK what you can do is dictated by what Apple is willing to make public. And because of the App Store model, you don't even have the option of using private frameworks and cheating a little bit like you do on the Mac side… [To figure out what you could do on the iPhone, people were] Decompiling [Apple libraries], dumping, and writing headers. And a lot of that work originated actually at Delicious Monster. Because obviously Lucas was really interested in the phone… Yeah, so a lot of hacking, and so that's kind of the major difference, is that ultimately anybody, any sort of programming in jailbreak is at best [a] hack. So that reflects itself not only in terms of the quality of apps, the stability of apps, but also just in terms of the type of programming that you do, you don't really have the same sort of discipline that you do when you're approaching the [official] frameworks and the language… It's whatever's there… When you're programming in jailbreak, it's just you and a bunch of headers." (Mike Lee, Interview, July 15, 2008)

A legitimate App Store would also let third parties create businesses around their applications without the threat of legal action from Apple. The App Store provided a central digital distribution point for iPhone software, so that developers would not have to set up their own websites, servers, and digital payment systems. It also provided a central place for consumers to find applications, making it possible for an individual developer with no marketing budget to theoretically compete against corporate offerings. In return for this legitimacy and distribution mechanism, Apple instituted policies whereby it would review and approve all applications to be sold on the Store, and also take a 30% cut of earnings from the App Store. While some large software firms balked at these terms, for many small-scale developers, including many hobbyists, the App Store infrastructure and the SDK made it possible for them to become entrepreneurs for the first time, as they no longer needed the capital and the knowledge to create online distribution and payment mechanisms. Apple had dramatically lowered the barriers to entry for software development with the iPhone App Store, and for the first two years of the Store, media stories abounded of iPhone users trying their hand at becoming app developers and striking it rich (Smykil 2009; Wortham 2009). Apple's developer base had suddenly expanded by several orders of magnitude. Apple claims that over 380,000 developers have joined its paid developer program, and that its "app economy" has created over 627,000 American jobs (Apple Inc. 2015a). In a January 8, 2015 release, Apple boasts that it offers 1.4 million iOS apps in the App Store, and that its app developers have earned $25 billion cumulatively since the App Store opened in 2008, with $10 billion of that in 2014 alone. A new report on mobile development states that iOS remains the primary platform of 37% of mobile developers, and this is skewed strongly towards "the West," with 42% of developers prioritizing it in North America and Europe compared to the rest of the world (VisionMobile Ltd 2015). Another indication of the explosive increase in Apple's developers has the attendance at Apple's Worldwide Developer Conference (WWDC). Prior to 2008, it was not difficult to get a ticket to WWDC, and a developer could count on being able to see the same community faces at the conference year in and year out, helping to foster that sense of community. Since then, because the venue, Moscone West convention

center, has not changed despite increased demand, WWDC has sold out in ever decreasing amounts of time, from a week in 2009, to less than twelve hours in 2010, under two hours in 2011, and under *two minutes* in 2012.

Existing Cocoa Mac developers, some of whom had been involved in the initial wave of apps on the App Store, were worried, however. The quality of a vast majority of the applications on the store did not meet their standards for what good apps should be. Some applications seemed to be barely more than modified versions of Apple's own sample code. Others were gimmicks, such as applications that made flatulence sounds. Some were bald-faced attempts to cheat customers of their money, such as one application priced at $1000 that did absolutely nothing. Despite widespread developer discontent at Apple's review policies for approval of apps on the Store, some Cocoa developers wished that Apple actually rejected more, that it should be an arbiter of taste. "They [Apple] have complete control over the market, but they don't want to exercise that control in a really bad way. They *have* control over it, but they don't want to *take* control over it… It's like, we don't want them to be arbiters of taste, but they pretty much are. They're one of the most tasteful companies in our industry, maybe they should be holding third party developers to their standards? And [then] everything on the App Store would be *wonderful!*" (Chris Clark, Interview, June 12, 2009)

At this juncture, some Cocoa developers, such as Mike Lee, felt that their community and its values were under siege by "unwashed masses" of newcomers whose only concern was not making quality products that users would love and enjoy, but only for making a quick buck:

> So, I think that there's a cultural battle for the soul of the iPhone. Are iPhone developers and iPhone customers ultimately going to be Mac developers and Mac customers? Or are they going to be Windows developers and Windows customers? Or are they going to be something else entirely? And I am very, very vocal and very involved with saying, it should be, must be, as it is on the Mac. Not just because that's kind of like what I'm used to, but… I mean the reason I'm a Mac developer is because it's a wonderful place to be, you know? It's a wonderful world, where we make wonderful things, and that's where

> I want to be. I don't want to see it to be turned into this sort of crappy environment. …To work in some kind of a crap farm where people make crap all the time.
>
> (Mike Lee Interview, July 15, 2008)

At stake, as Lee mentioned, was nothing other than "the soul of the iPhone." Lee believes that not only is Apple's Mac platform superior to Microsoft Windows, but that the third party applications made for the Mac are higher quality compared to most Windows applications, because Apple's users expect and demand such quality. This also means that Mac developers aspire to produce apps of a higher level of quality than Windows developers, and that the culture of the Mac developer community promotes and rewards excellence, while in Lee's view, the culture of the Windows developer community is to simply put out something profitable. Herein lies the danger in rapidly expanding the Apple developer community. Would new iPhone developers be assimilated into the existing cultural norms of the Mac development world? In other words, would they learn that quality engineering, careful craftsmanship, and tasteful, aesthetically pleasing user interfaces were the standards they should be striving for? Or, in Lee's words, would they be like developers for Microsoft Windows, relying on mass markets and cutthroat pricing to make as much profit as possible? Already, the signs were not encouraging. Apps on the App Store overwhelmingly priced themselves at the minimum price, 99 cents, or were free and relied on ads or in-app purchases. This created a race-to-the-bottom whereby apps competed not on quality but on price.

Gabriella Coleman has noted that moments of crisis in a collective group offer opportunities to examine when the ethical norms of the group are made visible because the stakes of the crisis are the possibility that these norms may change. A moment of rapid expansion, with the possibility of the loss or dilution of the existing group's values, can be such a moment of crisis. Coleman relates a similar instance in the community of open source developers of the Linux distribution, Debian: "As Debian grew quickly, the project found itself in the midst of a crisis that peaked beteween 1998 and 1999. New members were being admitted at rates faster than the project's ad hoc social systems could integrate them. Some longtime developers grew

skeptical of the quality of incoming developers… The populism of open membership began to come under attack… the 'growing pains' Debian had been experiencing were not merely technical but also ethical. A small group of developers had ben clamoring to loosen the commitment to free software… the spirit of free software was seemingly losing its potency with the addition of each new wave of developers." (Coleman 2013, 141–142) At such moments, oldtimers may engage in boundary work (Gieryn 1983) to separate out those who "don't belong" to the community because they violate its norms.

As the community grew, this original core came to be regarded by newer developers as "superstars" of the Mac development world. Once the iPhone gold rush hit, this original core remained intact, leading the way as the established experts in Cocoa technology, which translated directly to iPhone development. However, it was also this group that began to worry about the newcomers. Despite their relatively small numbers, this core of older Cocoa Mac developers continues to be among the most vocal in the community, trying to maintain norms and values that predate the iPhone and the App Store. As we will see, they form the core of a community that is growing like an onion, with each successive outer layer less connected to the core.

Boundary work during the iPhone gold rush

With the rapid expansion of the Apple developer base with the iPhone, the existing community was worried that the normative commitments that had obtained before would be lost. As Lave and Wenger noted about communities of practice, a central concern among oldtimers is the social reproduction of the community, and the transmission of values and practices to the next generation. This transmission to a new generation is a critical component of what Traweek calls a community's "developmental cycle" (Traweek 1988, 7). The concern for the reproduction of the Cocoa community can be located in the form of statements by oldtimers that constitute "boundary work," explicit drawing of the boundaries of community membership (Gieryn 1983). Brent Simmons, a Cocoa convert from classic Mac development who we've encountered before, understood that reproducing the values

of the pre-iPhone Cocoa community was important, and advocated preaching by example:

> So we had a structure and a set of values that pre-dates all the people coming in, right? And I remember discussing that early on, how to retransmit those values to the future. And my attitude was, well we'll just keep doing what we're doing; it'll work or it won't.
>
> (Brent Simmons Interview, February 17, 2012)

Cocoa developers on the Mac repeatedly expressed concern that newcomers did not care about the craft and quality of the applications they released on the App Store. As we discussed earlier, Qiu, Gopal, and Horn (2011) found many iOS developers following a "market logic" for app development and marketing, in contrast to developers who followed "professional logic" focusing on quality and peer recognition. "Market logic" driven developers chose a strategy of quantity over quality, putting as many apps into the store as possible in the hope that one might catch on. These two groups can be mapped onto newcomers and oldtimers in the Cocoa community, with oldtimers feeling that this new "market" driven mentality was causing a precipitous decline in the quality of apps on the App Store. Many cited the fact that the App Store in its first few years became littered with gimmicks, joke apps, and tasteless quasi-pornography, until Apple changed its review policies to emphasize apps with actual utility, as well as controversially banning quasi-porn, but making an exception for Playboy (Foresman 2010a; Foresman 2010b). While much criticism was directed at Apple's inconsistent and seemingly arbitrary review policies, others directed their ire at the low morals of newcomers, who seemed only concerned with making profit through low quality, low-cost apps.

> I feel like there's a part of the community that feels like it's been overrun by all these newcomers. With no morals.
>
> …[Somebody tweeted to the effect that] this newcomer to our community has no taste and no decorum and how dare they.
>
> (Adam Preble Interview, August 8, 2011)
>
> …So I definitely, things like that give me a hint that there are these sort of unwashed masses of programmers out there who are out there

> to make a quick buck on an iOS app or something like that. And even just the sheer number of iOS apps out there…
>
> all these people out there who are just out to make a quick buck, they're probably not going to go to WWDC or the big companies are like, maybe we'll send someone, send us a requisition and in three weeks we'll approve it. And it's like, too late.…
>
> And the people out there who are writing $0.99 iOS apps that are barely functional, they're not part of that same community
>
> (Dan Wood Interview, April 9, 2012)

Again, this ran against the prevailing norm in the Cocoa community, that developers were vocational craftspeople who cared about quality workmanship first, instead of profit at all costs. Mercenaries and "carpetbaggers" were abhorred.

> So it'll be interesting to see what happens to the community over the long haul now that its The Big and The Popular platform, at least for devices. So, because we have seen more kind of like, trash apps in the store, things that are just thrown together or, you know that the programmer who did this really did not care about the user, the platform, or the program. Whether that's just like we've got the carpet baggers in with everybody else, or if that's a fundamental shift in the community because we are larger, more distributed, fewer people can go to the same conferences, [I don't know].
>
> (Mark Dalrymple Interview, April 11, 2012)

Much of the feeling that the oldtimers were losing their community came from the change in the population attending Apple's Worldwide Developer Conference (WWDC). In the Ante-iPhone era, WWDC had been a place where the Cocoa community could count on seeing each other, strengthening social bonds that, during the year, were maintained purely online. With the rapid influx of new developers, not only did oldtimers feel that they couldn't recognize most of the new people, but because the conference was selling out in minutes, they could no longer even guarantee a ticket.

> Well, I think that yeah, the community is so big, that WWDC is—it's hard. It used to be, I would go and could count on seeing certain people and see them. Now, there's so many people doing so many things and so many groups that I could spend the whole week there and at the end run into someone who I feel like I know pretty well, and

> be, like, 'Ah! I haven't seen you all week!' And that never used to happen. Like, it's so big and there's so many people now.
>
> (Chris Parrish Interview, March 2, 2012)

This engenders a sense of loss of the community's feeling that it was tightly knit and built on personal relationships:

> I guess if you just want to say the demographics of an Apple developer has to be radically different from what it was. Like … five years ago… feel like you knew everybody. And now it's like I can't run [without running into an iPhone developer]… just telling people I write iPhone apps, and it's like "And who doesn't?" They're all, "yes, yes, and you're an actor too, right?"
>
> (Chris Parrish Interview, March 2, 2012)

Related to this was the feeling these newcomers were not really part of the community because they did not take the time to get to know the Who's Who, the Important People, in the community.

> For one thing, it's so large, unfathomably large now, compared to what it was ten or even five years ago. The introduction of iPhone has changed things dramatically…
>
> Nowadays, I'm not sure, because there are people out there who've been writing Cocoa code now for years who have no idea who the hell I am, or may not know who Wil Shipley is, may never have heard of Omni and are still Cocoa community members in some way. It's harder to know, now, it's so damn big.
>
> (Brent Simmons Interview, February 17, 2012)

Part of the reason for this was simply that the newcomers, ironically, were independent of the Indie community. They were off on their own, writing their own code, without participating in any of the community's local events, traveling conferences, or online discourse.

> [These are people] working at companies funded by ex-Microsoft managers. Not universally, but I mean there are some of those. And they don't reach out. I think part of it is they may not be aware of it. The other part is, they may feel like they've got it figured out. Whereas… you will find companies in Seattle that, more commonly than not, are like started by some ex-Microsoft person. And they're

> doing iOS stuff. And they talk to the press, they do stuff, but they don't talk to anybody else. And they don't go to any of the other things, in my experience.
>
> Yeah. And they don't function in our ecosystem. They just do stuff. And so it's not necessarily—some of it's OK, some of it's just kind of like, wow, you should interact with the group more.
>
> (Daniel Pasco Interview, March 28, 2012)

Lack of participation in the community, whether in-person or online, meant that these newcomers were not being exposed to the community's norms, nor could they contribute back to the community anything of their own.

> But I think most of those kind of developers, I don't think they're necessarily transmitting any values to anybody; I don't think they're taking part in the community in any big way, they aren't necessarily blogging or tweeting or going to Xcoders [the local Seattle Cocoa developer club] or the conferences or anything. They're not active; it's a day job. When they're not coding, they're not thinking about it, probably…
>
> The only barrier to participation is deciding you want to. Pretty simple, I think.
>
> (Brent Simmons Interview, February 17, 2012)

Similarly, others drew boundaries against developers for whom programming was not a vocation, but just a job. This, they correlated with lack of participation in the community:

> And then there's the other group, which comprises probably just about everyone else that I work with in my, at my actual job. And those are the people who don't take the time to go out to the community groups, or spend extra time on their own outside of work programming, or coming up with ideas of start-ups or whatever, the people that just go to work and they do their job and they go home and I think you can really tell the difference in just attitude. It's like, at least when you go to one of these communities, you're with people who are willing to take your own personal time to build their craft.
>
> (Robert Walker Interview, May 19, 2012)

For others, participation itself defined community membership:

> This guy isn't *one of us*. Nobody's ever heard of him, he doesn't participate in the community
>
> There are a lot more people who can code Obj-C [Objective-C] and write for the Mac and now iPhone than are in the club of indie Mac developers.
>
> There's more to being a Mac developer than simply writing the code.
>
> (Mike Lee Interview, July 15, 2008)

What makes for a "true" Mac, iPhone, or Cocoa developer, according to the community? A commitment to bettering the Mac and/or iOS platforms by contributing useful, easy and fun-to-use, aesthetically pleasing, well-crafted, designed, and engineered apps, over pure profiteering. In other words, love—of the app, and of the Apple platform.

> I think where I have trouble relating is when they're in it, and they don't love it. I mean that was sort of what the Windows community was about, from my perspective. [Laughing] And I feel like, now that we have gone mainstream, there's probably more of that happening here as well, maybe the publishing community is like that, I don't have contact with how they're doing, I'm not clear whether they love the iPad or they just think they have to be on the iPad to keep their magazines going, or something. There's now a lot of people who might just be in it because they think it's going to be a good place to make money. It's business area.
>
> And so that, it doesn't mean that I dislike them, but I have to worry, like in 2000 you had to be in it because you loved what you were doing, because there was no other reason to be there! [Laughing]
>
> (Ken Case, Omni Group Interview, March 23, 2012)

Another community norm that was violated was that of collegiality among members. When the NeXT community and later the Cocoa community had seemed to be a small town where people knew each other personally, there was a sense of collegiality and camaraderie between them. The metaphor of the small town used by many NeXT and Cocoa developers explicitly evokes the Main Street U.S.A. image of the white American country town, in which everyone is a neighbor with everyone else and knows them by name. Community members are conceived of as small business owners, who wouldn't hesitate to point a stranger in the direction of a

competing store because the storeowner is a friend. Indie developers who congregated in the Seattle area, in particular, saw their local community in this way. Seattle indie developers had developed close friendships with each other, and followed an honor code whereby they voluntarily did not compete with each other:

> In the NeXTSTEP world, it was so small, we had this very strong sense that if you're doing an app, I'm not going to do that app. There is no point is us both doing it, because we're going to split the market, we're both going to suffer, it's a classic loser's dilemma. So if someone is like, "I'm making a word processor," it's like, "OK… I'll come up with a different idea."
>
> And that was a very strong thing in that whole community. I mean it was good for all of us, and we're so small, and we had a much larger goal.
>
> (Wil Shipley Interview, April 18, 2012)

Part of this, especially during the NeXT era, when NeXT developers were so scarce, and the platform itself might have died out, was that everyone was in it because they loved NeXT technology and was developing for it in order to help the platform succeed against its competition. Even after NeXT was acquired by Apple, the Cocoa indie community, though bolstered by converts from the classic Mac OS, was still very tight knit. The Mac was still a minority platform for desktop computers, with Windows still dominant. Cocoa developers felt that as a collective, they were fighting the good fight against Microsoft to make the Mac platform a better place for users. If by doing good quality work, they would help Apple and the Mac, and inspire other Cocoa developers, and a rising tide would raise all boats. There was no need to succeed at each other's expense. Seattle-area Cocoa developers, working in the shadow of Microsoft, probably felt this most keenly.

> We're not trying to cutthroat each other, we're all in this together. which maybe comes from that same sort of, like, "Hey we're the crazy [ones]"—Maybe it is that pirate mentality.
>
> (Chris Parrish Interview, March 2, 2012)

In this statement, Parrish explicitly makes two references instantly recognizable to Apple fans. "We're the crazy ones" is explicit reference to Apple's

Think Different advertisement, which begins, “Here’s to the crazy ones, the misfits, the rebels, the troublemakers, the square pegs in the round holes.” The text praises iconoclasts for being the “geniuses” that invent new things, shake up the status quo, and change the world. The ad campaign singled out historical individuals who defied the status quo as heroes to be emulated. For many Cocoa developers, it stands as an ideological statement for how they can make a contribution to society through technological development. The “pirate” mentality refers to a pirate flag that Steve Jobs had raised over the building where the original Macintosh development team worked (Hertzfeld 2013b). It signified that the Mac team, led by Jobs, was a rogue group within the rapidly corporatizing Apple of the early 1980s, and embodied the mentality that by bucking the trend of the rest of the company and doing something new and revolutionary, they would change the world again. By evoking both of these cultural touchstones in Apple lore, Parrish is saying that what brings Apple developers together is a shared sense that they are all iconoclasts, who share a unified mission to improve the world through their work as technologists. Their goal is to overthrow the bureaucratic and corporate world of Microsoft with their art. Beating each other in the market with cutthroat tactics is not consistent with this ideological vision.

Cocoa developers saw themselves in opposition not only to corporate Windows developers, but also to corporate Carbon developers on the Mac platform itself. Although Apple was sending signals that it favored Cocoa over Carbon for future development, a large contingent of classic Mac developers, especially those that worked for Adobe, one of Apple’s most critical corporate third party developers, continued to be committed to Carbon. Chris Parrish, who was a Cocoa indie convert in the 2000s, had originally worked on Adobe InDesign in the Seattle area. Carbon seemed to be the tool of these corporate firms; Cocoa, on the other hand, was embraced by the indies, in part because its productivity advantages over Carbon allowed indies to compete feature for feature with corporate software offerings. There was an association of Cocoa with indie development against the corporate use

of Carbon. Thus, Cocoa indies, especially in Seattle, needed to stick together and help each other out against the corporate behemoths in their backyard.

> I like developers that help other developers. You know, I think we're all in the same boat in a lot of ways, and the person who is your competitor today may be your ally tomorrow. So it's a good idea to treat everybody well, and I like to see… developers like that. I can name only a very few that I don't think quite share that same outlook.
>
> (Brent Simmons Interview, February 17, 2012)

There was a distinct concern that this attitude was being lost with the influx of many corporations as well as venture capital or angel investor-funded startups. Recall Mike Lee, the former Seattle indie who was working for such a startup in 2008. Lee was upset with his COO for publicly denigrating a Twitter client made by a well-regarded indie developer simply because it competed with their own. Lee felt that this was in bad taste, as it violated the norm of collegiality that Cocoa developers, so far, had lived by. Lee felt that Cocoa developers were artists, and artists did not need to tear each other down, but simply represented different visions, which could coexist in the world:

> You may have a successful product, from a company that's following a vision, and are dedicated to the platform, you may have two successful products that are following separate visions, that are solving the same problem, that are respectful of the platform, and vice versa.
>
> …This is one of the things that makes the Mac a different platform to develop for than Windows, or even Linux. Is that, we're engineers, but we're also artists, in a certain way. …We're solving problems, but the reason why we're solving these problems is to express a vision.
>
> […]
>
> I like being able to recommend my competitors to people, and I've done so… being able to recommend other visions on the same thing, when somebody is obviously not getting along with my product and I want them to be happy, I want to be able to tell them to go where they're going to be happy.
>
> (Mike Lee, Interview, July 23, 2008)

Lee's statement about "vision" is also an ideological pronouncement that he has a higher, more transcendent sense of purpose for himself, his work, and his

company than his COO, whose only concern is base and material, for money and market power. Lee's concern was to build a company rooted in the indie Cocoa community and its values and which could make a contribution to that community. This put him at considerable odds with the other two founders and the company's board.

Many of these boundary work statements were made in the first few years of the App Store, during the initial "gold rush" period. Some have been made slightly later. However, in the meantime, some developers have begun to participate in the community and have acquired its values, and there has been significant new membership in local-area Cocoa developer clubs. Gus Mueller is one of the Seattle old guard, who released his first Cocoa Mac app in 2003 and went fully independent around 2005. Mueller feels that Windows programmers who only joined to write iOS apps, not Mac apps, still managed to absorb the right values.

> No, they [the former Windows guys] assimilated. Yeah, there's definitely iOS-only guys that show up [people who don't write for the Mac]. You know, they're good guys. And a lot of them are good guys.
>
> (Gus Mueller Interview, February 21, 2012)

What is interesting is that many of the more recent developers who have since assimilated into the Cocoa culture have begun to do boundary work of their own, against developers who have not assimilated. Nevertheless, these developers are also more likely to be open to new ideas and contributions of others:

> It's a good thing to have all these new people coming into the community and it's good to have this focus all on building a Cocoa community.
>
> (Robert Walker Interview, May 19, 2012)

This new reality, where the Cocoa developer world has greatly expanded beyond its previous "small town" status, has diluted, but not eliminated, the influence of the indie Mac developers. Indeed, given the continued influence of the oldtimers on the Cocoa developer blogosphere and Twitterverse, they have now become the core of the wider community.

Core and Periphery in the Cocoa community

With the iPhone App Store's expansion of the Cocoa community post-2008, a number of developers have begun to describe the community as no longer a small town but a layered onion, in which the previous "small town" grew to become the center of a larger metropolis. For example, Chris Parrish felt that the core of the Cocoa community is the Mac developer community that predated the iPhone: "I think the core is the Mac developers. I think that really is where it started." (Chris Parrish Interview, March 2, 2012)

Mark Dalrymple described the Cocoa community like an onion, with an inner core of old hands and experts, and an outer periphery of newcomers. The inner core is characterized by participation in the community, and this participation involves active sharing of their knowledge about practice to others. A key marker of core membership is that, because they are constantly sharing and communicating to the community on the Internet, through blogs, Twitter, or podcasts, everyone who's "in" knows who they are. Moreover, a key marker of non-membership is not knowing who the "big names" are. Adherence to the "indie" ethos, of not being in app development as a "get-rich-quick scheme" is another key marker.

> It's in layers. So you have kind of the core that everybody knows about. Everybody knows Wil Shipley and Delicious Monster. Everybody knows Mike Ashe. And I think that those individuals, the Jalkuts and the Atomic Birds and those folks—as well as the folks who write the books—so Dave Marks, Jeff LaMarche, Jack Nutting… so it's folks who are fairly well known, [these people who have] a core of deep knowledge, and the willingness to share it.
>
> …
>
> And then you have kind of the immediate hangers-on; I want to better myself, I'll be like you; I will emulate you and then kind of, on out to the, yeah, I do .Net for my day job but Cocoa is kind of fun at night, so there may be then, an outer shell. Those are the folks who come to CocoaHeads [a local developer club] to meet once or twice and then don't come back for seven or eight months and then come back again. It's like oh, yeah, I have some free time again—I'm going to go back to work on my Cocoa app and do stuff.

> And then of course, as you get farther and farther out from the core, you know, kind of the colder and colder the temperature gets, that's kind of where the carpet baggers and whatnot, on the out ends of the scale.
>
> (Mark Dalrymple Interview, April 11, 2012)

Dalrymple's onion-like model of the Cocoa community is not static, but in transition and flux:

> So it will be interesting to see kind of over the long haul if the community changes, because there are a lot of folks who are… kind of the core community, who share a lot. The Wil Shipleys, the Mike Lees, Tom Harringtons, Dan Jalkuts, Mike Ashe. Those folks who are very knowledgeable, and they share deeply their information. If that kind of ethos, can withstand the get-rich-quick onslaught, then it will continue to be a very, very nice, very friendly community…
>
> (Mark Dalrymple Interview, April 11, 2012)

Dalrymple is thus showing his concern with how the community is transmitting its values and reproducing itself. Will the values of the core expand to include the outer layers of the onion? Or will the outer layers gradually overwhelm the core?

Another feature of the core members is that they are very interested in community building and knowledge sharing activities, such as podcasts and organizing conferences. They also have to be good developers, working hard to put apps out there. These core members, however, are more interested in the community than in apps as a means to build an acquisition target:

> And so far it seems that that kind of inner core has been sticking around and has been prospering, so that Scott… I always forget his first name—originally started out with the late-night Cocoa podcast, which I had done a couple of. And it was just a podcast, an hour interview, with somebody who's done stuff in Cocoa or to explain some technology and through an amazing amount of hard work and charisma has built it into the I-Developer Network. Now [he] puts on conferences and does video stuff.
>
> But fundamentally, it is because he is an awesome individual, doing awesome stuff. So it's nice seeing folks like Aaron [Hillegass], being able to build larger company. So it's nice seeing that same core in

> there, sharing and getting rewarded for that sharing. And people can see, oh, they're sharing, they're awesome people, and they're getting rewarded. They have jobs; they're selling apps. Unfortunately they're not the folks who get a billion dollars from Facebook for Instagram, or Angry Birds or something like that…
>
> (Mark Dalrymple Interview, April 11, 2012)

For core members, participation in the public discourse of the community, especially knowledge sharing, either online or physically, is what signals membership. This has allowed a significant number of young newcomers to partake in what Lave and Wenger call "legitimate peripheral participation," a key feature of communities of practice. The oldtimers serve as exemplars of full community membership to the newcomers through their knowledge sharing and participation.

> But for the most part, it's like, those people make good livings, and they have impact from the community and they're fun to hang out with. And so I think that's… inspiring the next shell of people around it. So if you can kind of keep that strong core and make that core bigger, because more and more people are writing books these days, which I love—I'm a bookworm … So as this core of people who are fundamentally sharing stuff, I think that will be good for the community overall. If the platform collapses, then that's a different story… But I think as long as the platform can at least maintain its popularity, then it will be good for us in the long run.
>
> (Mark Dalrymple Interview, April 11, 2012)

Thus, full membership is based on community participation (which means knowledge sharing, for the most part), not merely on intelligence or even making a good app.

> I'm trying to think of, like, I do know some big huge brain programmers, that they are very concentrated on their jobs and they don't blog or anything like that. So I don't consider them to be kind of like, the core of the community because as much as I love them, if they were run under a bus then… it's like their immediate family and work folks will be very sad, but the rest of the community won't know about it… If you don't have some way to get in front of the community, whether it's through sharing…
>
> (Mark Dalrymple Interview, April 11, 2012)

Dalrymple implies that if a developer is not communicating, participating, or sharing with the community, they are not really members at all.

Another marker of increasingly central membership is knowing the who's who in the community, having a dense Twitter network of them, or following their blogs. Dan Wood notes that there is a relationship between participation in the public discourse and the relative "fame" and thus relative centrality of a Cocoa developer, noting that there are many Cocoa developers who do not participate and thus remain peripheral:

> But, you know, there's a lot of people out there that are developers and they probably are writing Cocoa all day long and yet, nobody knows them…
>
> But I bet if someone wanted to, they could probably get themselves into the circle, if they just wanted to participate and they got to know these people, and… then just some of the developers have been around for a long time and some people are just getting started. But generally, if I meet someone in person at a conference and I think they're cool and I like them, or whatever, then I'll end up following them. Whereas, I'm not likely to follow someone that I've never met. Personally I don't follow very many celebrities.
>
> (Dan Wood Interview, April 9, 2012)

Even in local communities such as Atlanta, participation in club meetings, sociality, and the knowledge sharing that goes on there has become a key measure of centrality in the local community, even if those individuals do not have a large national presence online. For example, Robert Walker sees a hierarchical tier of membership even among the local Atlanta members. The core members are those who participate fully, give talks, and share their knowledge and experience with the others. According to Walker, the peripheral members are simply people who program for their jobs, but aren't committed to developing their programming skills as a craft and a vocation, and who only come to the meetings to find information useful for their profession, not to make friends.

> And it's usually the same people, that same core group there, so you sort of meet in with that core group and you have these—like there's two tiers. That core group, and I think that's probably common

> throughout these communities. There's a core group that—even going back to, here's the core experts, these are the guys you really go to, and there's a lot of people that are at that second or third tier.
>
> And then there's the other group, which comprises probably just about everyone else that I work with in my, at my actual job. And those are the people who don't take the time to go out to the community groups, or spend extra time on their own outside of work programming, or coming up with ideas of start-ups or whatever, the people that just go to work and they do their job and they go home and I think you can really tell the difference in just attitude. It's like at least when you go to one of these communities, you're with people who are willing to take your own personal time to build their craft.
>
> (Robert Walker Interview, May 19, 2012)

These descriptions of other Cocoa developers as central versus peripheral members of the community and their varying commitment to the core values of the community fits well with Wiebe Bijker's notion of "technological frames," which I have expanded to include affect, ideology, identity, and normative values into a concept of "techno-cultural frames." Walker's statement that the core group comprises those experts who participate in local community meetings, and spends their free time programming because it is a passion and a calling for them to work on their craft skill is one of the central normative and ideological components of the "techno-cultural frame" that Cocoa developers ascribe to. We can see that in their view, a developer's social distance from the core is related to their normative difference from its values. The social centrality or periphery of various Cocoa developers is related to their levels of inclusion in the techno-cultural frame (Bijker 1995). Although this does not mean that peripheral members have less agency over the technology that they produce, it can mean that they have less power to control the dominant ideological and normative discourse because they are not participating in the public discourse. The famous names in the old Mac Cocoa developer community continues to dominate much of the online public discourse about Cocoa programming. These oldtimers disapprove of newcomers with low inclusion in the Cocoa techno-cultural frame, especially those from either the giant corporate developer world or venture-funded startup world, whose focus is less on app quality, user experience and craft and more on profit, revenue, or growth. A key marker of

this was how much resistance or criticism the newcomer had to the Objective-C language, and more specifically to certain syntactic peculiarities of the language (which we will examine in more detail in the next chapter.) Acceptance, and even enjoyment, of Objective-C and its syntax marked the transition of a developer to high inclusion and to full membership within the community, whereas continued hostility marked low inclusion and outsider status. Such differences in inclusion between members are revealed much more starkly in moments of great change in the community, such as following the iOS gold rush of 2008-9, and similarly following the integration of the NeXT developer community into the Mac developer community post-2001.

One striking, and disconcerting, fact about the core of the Cocoa community, however, is that it is almost exclusively male. As discussed in earlier chapters, female participation in software programming in general has steadily declined since the 1980s (Misa 2009), resulting in masculine "geek" or "brogrammer" cultures at tech companies that women often find hostile or exclusionary. (Hicks 2012; Parish 2014; Raja 2012) Nevertheless, the Apple developer community may be even less diverse than the rest of the industry. Ge Wang, a Stanford professor who co-founded the music app company Smule, said to me in 2009 that his qualitative impression from Apple's developer conference (WWDC) was that male to female ratio was less than 30 to 1. I do not have any statistics to back up this claim. However, my impressions from my own visits to WWDC are similar—the lines waiting for the men's restroom are actually longer than the women's. Having worked at Apple, I know that the gender balance among software engineers there is significantly better than that of its developer community, as my group in the AppKit had three women out of 13 engineers total (though they were not there all at the same time), and I met many women engineers from other departments. I can only speculate as to the reason why the core of the third party Apple developer community includes so few women, or at least well known ones. For one thing, the minority status of the Mac platform made it a highly risky venture to program for it, especially as an independent. Since much of the core of the community is dominated by indies, who started their own

small businesses, such programmers also have to be entrepreneurs. A recent report on entrepreneurship among women by the Ewing Marion Kauffman Foundation reveals that among employer firms, women-owned businesses account for only 16%, and among high growth-firms, less than 10%. The survey highlights three challenges for women entrepreneurs: lack of mentors, a view that emphasizes past failures compared to men, and lack of access to financing (Alicia Robb, Susan Coleman, and Dane Stangler 2014, 3–4). This means that women tech entrepreneurs are doubly disadvantaged compared to men, as significant barriers to their participation exist in both technology and entrepreneurship. This means that the ideological focus of the Cocoa community on indie development, which places working for oneself on a higher plane than working for a paycheck, overwhelmingly favors men. Women programmers may be more likely to take the stability and the regular hours of a corporate job. As Christina Dunbar-Hester has shown, even when technology people hold progressive values and strive to work towards inclusion, existing cultural biases with regard to gender and technology can be difficult to overcome in practice, and women in technology may find themselves forced to downplay their feminine identity in favor of masculine-associated technology identity. This in turn turns off other women who might otherwise be inclined to take up technology, because they are not willing to make such compromises. This leads to a situation in which even when one is seeking to recruit women with "good tech skills," one keeps finding "kick-ass men." (Dunbar-Hester 2008, 218)

Since the iPhone gold rush, however, I have noticed that this gender balance is improving somewhat. As I discussed in chapter 4, Big Nerd Ranch courses frequently had women students, and at another course, I met Charlotte, a woman indie iPhone app developer from Chattanooga. In Seattle, I attended a weekly "NSCoder Night" meetup of Cocoa programmers seeking to help each other with code at a local coffee shop. At least a good one fourth to one third of the attendees were women who were taking the University of Seattle's extension course on Cocoa programming, and one attendee was an indie developer who had released her own app on the App Store. Similarly, I saw many women in the line to get into Apple's

WWDC Keynote in 2011, and interviewed a few of them, finding that most of them worked for the mobile divisions of large corporations. However, few of these women are core members of the Cocoa community with high inclusion in the Cocoa techno-cultural frame. Most of them have only recently begun to program for iOS, and almost none of them program for the Mac. As we have seen in this chapter, most of the core of the Cocoa community programmed for the Mac before the iPhone was ever released. The minority status of the platform meant that only the most die-hard fanatics stuck with it and refused to program for other platforms, with the result that most of these were men. Another factor in the movement of peripheral members to the core, as we will see in the next few sections, is participation, both online, and offline. Core Cocoa members have large Twitter followings and frequently write blogs about programming, and often write controversial opinions that engender acrimonious debate. Women programmers may be much less inclined to present themselves as experts on technology publicly on blogs lest they receive online harassment or dismissal of their expertise or views.

The Cocoa Online Public

Coleman notes that "Public discourse is a vehicle through which hackers' immediate experiences with technology along with their virtual and nonvirtual interactions with one another are culturally generalized." (Coleman 2013, 45) This "growing unification of technical experience and its representation became notable on project news Web sites, mailing lists, blogs, books, and articles; these texts provided developers with a rich set of ideas about creativity, expression, and individuality." (Coleman 2013, 44) Coleman, quoting journalist Scott Rosenberg, notes that programmers have begun to write "personally, intently, voluminously […] pouring out their inspirations and frustrations, their insights and tips and fears and dreams on Web sites and in blogs. It has been the basis of… an informal literature around the day-to-day practice of programming." (Scott Rosenberg 2007, 301, quoted in Coleman 2013, 41) This fits Michael Warner's definition of a "public," as a group of strangers organized into a social group by the circulating discourse which addresses it and in which it participates (Warner 2002). Blogospheres are a form of

networked public (Benkler 2006; boyd 2006; Bruns and Jacobs 2006; Notaro 2006; Schmidt 2007). Yochai Benkler notes that the affordances of blog technologies "made the Web writable," facilitating the writing of pages in periodic, journal form, in "journalistic" time of days, weeks, or months, with older posts chronologically archived. Secondly, blogs also are often open to comments from readers, generating forums for conversations rather than simply being one-way communication (Benkler 2006, 216–217). Indeed, many programmers since the 2000s, both those devoted to open source as well as to proprietary platforms, have used the affordances of blogs to publish their thoughts and opinions to like-minded peers, constructing dense networked publics on the blogosphere.

Cocoa developers were no exception when it came to blogging. One factor in the increased activity of Cocoa developers on blogs was a Newsgroup and RSS Feed reader application, NetNewsWire, written by indie Cocoa developer Brent Simmons for Mac OS X. NetNewsWire included by default a number of Simmons' personal favorite developer blogs, including John Gruber's Daring Fireball, which has since become the opinion-leader for Apple developers and power users. Blogs allowed indie developers to get their applications, expertise, and opinions on technical matters known to peers. A few bloggers, such as Wil Shipley and Marcus Zarra became infamous for outspoken or controversial opinions among the community.

> There are definitely better programmers than me, I think I lucked out because I have a reputation for not giving a shit what people care about me which is not at all true, but it can seem that way because I just say whatever I think… And I'm willing to just be arrogant about it… So I'm a little bit of a firebrand.
>
> …You have to be very careful about it, because it becomes addictive to be known and then you realize you could be one of these people who just pushes buttons and continues to get more and more [followers?] but you know, those people always turn into jokes. For a little while they're like, oh, rally the community, and then after a while people are like, God, it's him again with another stupid article just poking people's buttons.
>
> I don't want to turn into that. I really want to not be famous for famous sakes [sic], to the extent that I am famous.

(Wil Shipley Interview, April 18, 2012)

Yochai Benkler, citing studies of Internet topology, finds that on the Web, the distribution links to sites follows a "power law distribution" with a very long tail—meaning that certain sites have significantly more connections and thus more visits than the vast majority of websites. At "a macrolevel, the Web and blogosphere have giant, strongly connected cores—'areas' where 20-30 percent of all sites are highly and redundantly interlinked… That pattern repeats itself in smaller subclusters as well." (Benkler 2006, 247–8) This means that certain websites tend to become central nodes to which others link. When such sites are blogs, the bloggers have a tendency to become "superstars" among their community of readers. In the Cocoa world, John Gruber's blog, Daring Fireball, has become one such central node. Gruber's blog, which provides commentary on both technical topics (often linking to other developers' blogs) and Apple news from the perspective of an experienced Mac developer, has become a must-read among Apple developers and well-informed users for speaking from a point of view that developers recognize as their own. Some developers who frequently blog, such as Shipley and Zarra, generate views and traffic due to opinionated posts, and not only become "famous" in the blogosphere but can seem to dominate the discourse. This notoriety on the blogosphere complements and magnifies community reputations achieved through other means, such as developing apps respected by peer developers, or publishing educational books. As we will see, blogging and other social media activities are important for indie developers, who have limited marketing resources, to get their names, and their work, known.

Blogs provide a means for a developer, who may be working in isolation, to share one's knowledge and expertise with others. Although a few experts such as Zarra and Jeff Lamarche have managed to publish programming books, which may reach a much broader audience of newcomers, blogs allow developers to self-publish their writing, and discuss more in-depth topics and practical experiences that may not easily fit into a pedagogical book, and do so much more rapidly than in print. In one case, a subset of Mike Ash's blog has actually been self-published as a print book

that a reader can order. Some developers have taken to podcasts, another means of self-publishing one's knowledge or opinion, usually centering around discussion between two or more developers or an interview with a developer. Developer-centric podcasts with names such as iDeveloperLive, OnMacDevelopment, cocoaFusion:, CoreIntuition, NSBrief, Edge Cases, Build and Analyze, focus on topics of concern specifically to developers, which can include Apple's App Store policies, the merits of various frameworks, APIs, and tools, life as an indie entrepreneur, and sometimes pedagogical pieces on using a particular Apple framework. In some cases, because they often take the form of discussions about happenings in the Apple technology world, developer blogs and podcasts take a turn away from pedagogy and towards punditry about Apple and the computer technology industry in general. John Gruber's Daring Fireball has shifted in this direction over the years. Developers who appear on user-oriented podcasts such as MacBreak Weekly or ArsTechnica's Hypercritical often fit this role of pundit as well; their identity as a Cocoa developer, usually of a popular app among users, affords them the status of an expert in all things Apple amongst the Apple users who listen to such podcasts. Podcasts involving Apple-centric media, such as Macworld, often involve roundtables with members of both the press and prominent members of the developer community as peer experts on Apple's platform, and can significantly raise the profile of the developers who appear on them among the wider public of Apple users.

Developer blogs and podcasts are often intended to be educational—the developer is frequently teaching the tips and tricks of the trade. However, as we will see in the next chapter, this information is laden with normative import—developers are constantly advocating particular coding and/or design practices, often practices which are not universally agreed upon and which, for some, seem to be purely personal preferences. This is the reason why some developers come across as opinionated and controversial—they advocate for the objective benefit of practices that others see as merely personal preference, and often take a dogmatic tone when doing so.

Blogs have since become supplemented by Twitter as a central medium for online sociality. There has been an explosion of academic literature on Twitter recently, with various takes on Twitter as an imagined community (Gruzd, Wellman, and Takhteyev 2011), its facility for self-branding and the creation of micro-celebrities (Page 2012), and the role of opinion leaders on Twitter (Xu et al. 2014). Twitter is not really a replacement for blogs—the character limit prevents the kind of long form, in-depth writing that typically occurs on blogs. However, for Cocoa developers, Twitter allows conversations between developers to form that are simultaneously broadcast to the developers' network of followers. Twitter also allows developers who write blogs to broadcast links to their latest posts, generating a larger audience. Twitter, a short-form medium, can thus actually increase engagement with the long-form literature of blogs (McNely 2011). Once again, the adoption of Twitter among Cocoa developers was facilitated by a Macintosh application written in Cocoa by an indie developer: Craig Hockenberry of the Iconfactory wrote one of the first desktop Twitter clients with a much more refined and aesthetically pleasing interface than Twitter's web portal, driving up Twitter adoption among fellow indie Cocoa developers eager to support Hockenberry's quality work. The early days of the iPhone App Store became awash in Twitter clients, competing on features, usability, and aesthetics. As a result, Twitter interconnection among indie Cocoa developers is very dense.

The prolific output of discourse about Cocoa programming through media such as blogs, Twitter, podcasts, and print books has broadcast the expertise and opinions of a relatively small cadre of individual developers to a larger audience of users and new iOS developers, extending far beyond the initially tight-knit community of Cocoa developers on the Mac. This is supplemented by the frequent conference talks that Cocoa developers give at community-run conferences, which we will discuss later. For these reasons, many Cocoa developers have become famous micro-celebrities, with large Twitter followings. Recent work on social media, which include blogs, podcasts, wikis, YouTube, and social networking platforms such as Twitter, Facebook, and MySpace, has highlighted how these

technologies are allowing their users to engage in literary production, blurring consumer and producer dichotomies. In this socio-technical environment, such 'produsers' (Bruns and Jacobs 2006) must compete with others for readers. Affordances built into social media platforms facilitate self-branding and self-promotion in what Page calls an "attention economy" (Page 2012). Successful self-branders are able to generate significant online followings, making them "micro-celebrities" (Marwick 2013; Senft 2008). Similar phenomena on Amazon user reviews have been described by David and Pinch as a "reputation economy" (David and Pinch 2005). The prolific, and intertwining, use of Twitter, blogs, and podcasts has made many Cocoa developers into such micro-celebrities, at least within the social network of Cocoa developers.

> That's kind of another cool effect of the community is that, it's not really well defined. I mean there's probably—you could probably name like twenty people that everybody in the Mac community eventually follows on Twitter. And you know then there's probably another fifty or so people that some of the people follow, and then there's another couple hundred, or couple thousand or whatever, that not very many people follow at all, except for that maybe they just have some sort of connection. I wouldn't put myself in that core group, I'm guessing. I may be in the second group out.
>
> (Dan Wood Interview, April 9, 2012)

Wood here sees the Cocoa community similarly as Dalrymple expressed earlier: like an onion, with a core. Wood however, sees the community explicitly structured according to networked online connections and communications. It is the density of Twitter connections, according to Wood, that connotes whether someone is in the core or periphery. This could mean that the Twitter network simply reflects some other offline social reality, or it could mean that the Twitter network itself is instrumental in constituting the community. It is not clear which Wood really means. Being a central node in the Twitter network of Cocoa developers could also take on a different meaning, that of "celebrity," or "superstardom."

> It is interesting just to notice how certain people are superstars, like Wil Shipley or Cabel Sasser, they're like the superstars in the Mac community… or John Gruber or whatever. If you get followed by any

> of these guys, it's like oh, consider yourself lucky, it's like getting to meet a famous person or something like that.
>
> (Dan Wood Interview, April 9, 2012)

For Brent Simmons, reputation or "fame" equated to having a large audience for one's blog or Twitter account. For indie developers, such fame is vitally important because they do not have large budgets for marketing or advertising; as noted by Qiu, Gopal, and Hann (2011), many of them rely on the quality of their apps and word of mouth among peer developers. In other words, they rely on the reputation economy (David and Pinch 2005) within their community of practice for marketing. Blogs and Twitter are a low-cost means of advertising—readers are also potential users and customers:

> [Fame,] that's the shorthand for you want a big audience, right? You hope you have thousands of users and thousands of people listening to you, right?…
>
> It does for anybody, I mean, any—just think of any bigger company; they want to be well-known. They want to have influence, they want big audience and a lot of users, and just because we're smaller shops, we're no different. We want the same things.
>
> (Brent Simmons Interview, February 17, 2012)

To a large extent, for indies, because the individuals themselves are their companies (and to a large extent, their brands), their products are intimately associated with them, and the more popular an app becomes, the more famous its developer becomes. Conversely, becoming more well-known through blogs and social media can result in higher name and brand recognition and thus higher sales.

This is particularly true in the case of Aaron Hillegass, who does not sell apps, but the knowledge for making apps. As the founder of Big Nerd Ranch, Hillegass created a company expressly devoted to teaching developers Cocoa programming on the Mac, and later on the iPhone and iPad. Taking on the persona of a cowboy at a dude ranch, Hillegass came up with an inexpensive marketing idea to don a cowboy hat whenever he appeared in public at developer conferences; he understood that he was the company brand, and as his own celebrity grew, his company's reputation and

business would grow. Hillegass makes frequent appearances at Cocoa conferences and on podcasts such as MacBreak Weekly, in addition to writing on the Big Nerd Ranch's blog and Twitter account.

For others, participation in the reputation economy is critical for employment:

> As a professional, you have to go on your reputation. And your reputation is very important… the only thing that decides whether I get a job or a project that I'm interest in, what decides how much money I'm able to charge, what decides so many things about my career, is entirely on my reputation.
>
> (Mike Lee Interview, July 23, 2008)

Pinch and David note that in the reputation economy of Amazon user reviews, "notions of quality, reputation, and expertise are tightly bound and… often conflated." (David and Pinch 2005, 8) In the Cocoa community, increased sharing of knowledge by publishing blogs and podcasts and broadcasting links to them through Twitter helps build their reputations in the reputation economy. Page notes that "The discourse contexts used for self-branding in social media genres are aptly described by Bourdieu's (1977) metaphor of the linguistic marketplace… where linguistic competence (in Bourdieu's terms (1977: 648), using language appropriately in order to command a listener) results in linguistic capital, which can, in some contexts, be converted into actual economic value." (Page 2012, 182–8) Page notes that "celebrity" or reputation can be converted into monetary capital, making it a form of cultural capital. (Bourdieu 1986). Thus, the reputation economy creates an economic incentive to share knowledge with others in the community in order to bolster one's reputation for expertise. For example, Luke Adamson, who left the NeXT/Cocoa community for a number of years to work on financial enterprise software and recently returned to do iOS work as well as teach iOS programming for a certificate program at the University of Washington, felt this incentive keenly as a way to revive his presence within the community:

> I keep thinking that I really need to do some blogging. If for no other reason, to increase my business's profile.

> I mean, I have a lot of experience in the technology, but I don't have any sort of presence on the 'net…
>
> I think it would be—I mean certainly it would be good to have more of a presence from a business perspective.
>
> I mean my little business hind-brain knows that. I think to the extent that I like—as an instructor, like teaching people—I think that it'd be sort of fun to share some of what I know. Through, like, a blogging format.
>
> (Luke Adamson Interview, February 22, 2012)

Adamson clearly understands that his social standing in the community, his professional reputation, and his business success are all connected, and these would all benefit if he increased his online networked presence. The virtual existence of the Cocoa community as a networked public greatly facilitates the reputation economy that indie Cocoa developers in particular must depend on for their livelihood. Without corporate or institutional resources, reputation becomes a much more important form of capital for indies. Indeed, Fred Turner has argued that the individualizing focus of technolibertarianism coincides with the movement towards a networked economy, in which individuals become disconnected from social institutions and instead become network nodes (Turner 2006). The emergence of famous Cocoa "superstars" can be seen as a symptom of this move toward a more individualized but networked form.

What we have seen, then, is that the Cocoa community's sense of who is in the "core" is highly correlated with high participation in networked public discourse over blogs, Twitter, and podcasts, and that a significant amount of this discourse involves knowledge sharing. Those in the "core" are densely connected through these online networks, and have become "famous."

Nevertheless, the Cocoa community is not only a networked public or virtual community. It has significant roots in relationships built offline, in the physical world. Cocoa developers have cited the need for face-to-face interaction as the source for, or supplement to, online relationships:

> Twitter for me works when you actually have some personal connection with the people involved.
>
> (Dan Wood Interview, April 9, 2012)

For Wood, online and offline interactions critically feed back and supplement each other, but the two groups are not necessarily co-extensive. It is possible to know of, follow, and even communicate with someone online but not have an offline relationship with the person. Wood, however, feels that something is missing without "personal connection."

> I think that there's probably some overlap [between the online community and offline], but there's some disconnect. Like for instance, there's someone like Matt Gemell… But I've never met him in person, and yet he's definitely, I would call him a 'core' member of the Mac development community, he's a prolific blogger, and he's opinionated, and that probably kind of helps, it generates interesting content and it generates interesting opinions, and I'm guessing he's probably been to a lot of the UK conferences, I've just never met him in person, but I still totally feel like he's part of the community, even though I've never met him in person and I'm trying to think if there's the opposite where there's someone I see at WWDC all the time but I don't see them on Twitter. There are some developers and they just don't do Twitter…
>
> (Dan Wood Interview, April 9, 2012)

The networked Cocoa public that exists online interacts with, supplements, and magnifies relationships offline, even though the online and offline relations exist somewhat orthogonally to each other. We will see in the next section that offline connections foster deep social bonds, even friendships, among members of the community.

Local Cocoa Developer Clubs

Virtual, online, textual interactions between relative strangers can foster the development of a networked public, but the Cocoa community is not just a networked public of strangers. Especially among core members, much of their sense of community is based on bonds of friendship that require significant and prolonged physical interaction. Coleman notes that for hacker communities, "In-person

interaction is also a pervasive feature of their lifeworld, working to confirm the validity of circulating discourse." (Coleman 2013, 45) This face-to-face interaction takes place at two kinds of sites, according to Coleman—local, quotidian interactions, and conferences. As we will see, the kinds of face-to-face interactions favored by Cocoa developers are geared toward male bonding, reinforcing the Cocoa community's gendered nature.

Among Cocoa developers, local interactions, the "ordinary stuff of work and friendships" (Coleman 2013, 45) are particularly important. Coleman points out that hackers often "live in a location with a high density of geeks, generally big cities with a thriving technology sector" (Coleman 2013, 46). Cocoa developers, and in particular, indies, similarly congregate in large numbers in these urban areas: Seattle, Portland, Boston, Denver, Amsterdam, Washington DC, Pittsburgh, Atlanta, Raleigh, Salt Lake City. It is in these cities where Cocoa developer clubs form. CocoaHeads is the umbrella organization of a loose confederation of local Cocoa developer clubs, co-founded in Pittsburgh by Mark Dalrymple, with additional chapters in Atlanta, Salt Lake City, Australia, and even Dubai. No less important is Xcoders, the Cocoa developer club of Seattle, home to many of the most prominent indies and indie companies in the community, including OmniGroup, Black Pixel, and the original base of Wil Shipley's Delicious Monster (himself a co-founder of OmniGroup). Atlanta has three Apple developer clubs. Besides the Atlanta chapter of CocoaHeads, founded by Big Nerd Ranch instructor Mikey Ward, there are two separate iOS Developer Meetups, one meeting outside the Atlanta beltway perimeter, in the suburbs, and the other meeting in the city. (The suburban meetup is also more peripheral to the community socially.) The downtown iOS Developer Meetup and CocoaHeads have some overlapping membership, but the iOS Meetup has much larger meetings, though the vast majority of members only listen to presentations, never participate in giving presentations, and rarely go to the social afterparty. CocoaHeads is a much smaller, more tightly knit-group, almost the entire group attends the social afterparty, and friendships have formed among members. The group is also much harder to join, as the website has not been updated to allow

people to easily find the group, which is partly due to lack of time but also partly due to the preferences of core members who prefer a more exclusive club. Keeping "newbies" out allows them to focus on more advanced topics, but it also has the unintended side-effect of keeping women out. The iOS Developer Meetup, meanwhile, attempts to be more open: each meeting, it contains two presentations, with both a beginner and an advanced topic. This group is also more diverse in gender and ethnicity, although most of this diversity is located in peripheral members who never give talks or go to the afterparty. The iOS Meetup also welcomes recruiters, although the organizer, Rusty, is careful not to let them take up too much time pitching at the meeting; although this necessary nod is made to the reality of business considerations, especially for independent iOS contractors, Rusty is careful to make sure that the heart of the club remains technical. CocoaHeads, which is explicitly a club for lovers of programming using Apple's Cocoa frameworks, does not welcome recruiters, and members prefer to keep it that way.

Local club meetings like CocoaHeads usually involve a presentation on a technical topic given by one of the members. For presenters, it is an opportunity to share their knowledge and perform their expertise. Sometimes presenters will have slides, but more often the presentation is a live demo, where the presenter shows his project source code, possibly making changes and adding new code live, and then compiling it and running the app to show the results. This type of presentation has similarities and differences from the kinds of presentations given by Apple engineers at Apple's official Worldwide Developer Conference (WWDC). Like Apple's presentations, they are often meant to teach the audience some concept or technique in Cocoa programming. Apple presentations, however, often provide information about new functionality available to developers in the next version of the operating system to be released, and how to use this new functionality. Apple's presentations also often explicitly contain official normative messages about what practices Apple recommends to its developers, especially in light of changes Apple has made to the OS, which often require change in practice. For example, Apple may need to update certain APIs and thus signal to developers to cease using the older interfaces. It

marks these APIs with the label, “deprecated,” to signal that they will go away in a future release, so developers need to stop using them. Unlike Apple’s presentations, which have an aura of officiality and authority about them, local demos are given by peers to peers, often with a touch of humility as presenters are showing their own code, which may not be polished, to others. Local presentations are much more about people sharing their own knowledge, practices, methods, code, and opinions with each other. For audience members, it is an opportunity to learn from the actual practice and experience of peers, which in some ways is more valuable as peers, unlike Apple’s engineers, are trusted to present both their frustrations and their pleasures with Apple’s tools or APIs unfiltered and unbiased. The normative content of Apple presentations is sometimes viewed with a grain of salt, as developers understand that Apple will tell them what is in Apple’s interest, not necessarily what is in developers’ interest.

In the STS literature on demonstrations, demonstrations differ from experiments in that while the outcome of experiments is uncertain, and can only be interpreted by experts, demonstrations are carefully rehearsed and intended to have certain outcomes, because their purpose is not to produce new knowledge but simply to communicate to or persuade new audiences (often students, but also laypeople in the general public) of knowledge that has already achieved closure among experts. A successful demonstration is a bad experiment, and a failed demonstration can sometimes become a good experiment, if it leads to new insights (Collins and Pinch 1998a, 62–65). The presentations at both local clubs and at Apple’s official conference are intended to be demonstrations; they are rehearsed affairs that communicate settled knowledge. However, local club presentations are much less polished and rehearsed than Apple ones. Both presentations can contain live code demonstrations, which contain the possibility that something could go awry. In Apple presentations, when things go awry, because of limited time the presenter often has no choice but to move on. In local club meetings, there is often more time for the presenter to try to debug the program live, and more direct interaction with

the audience can allow the exercise to become an interactive group activity. In this way, such demonstrations can shade somewhat into experiment.

It is at these local clubs that the closest bonds between Cocoa developers are formed. Although the official monthly Seattle Xcoders meetings involve a formal technical presentation, for many of the oldtimers, the "real" meeting is the afterparty that takes place at a local downtown Seattle bar, currently the Cyclops (though in years past it was held at the Luau). The afterparty is primarily a social event, where fellow nerds can get together over beer and talk about coding topics, debate best practices, and discuss the latest news about Apple, its products, the merits of its newest frameworks and initiatives, and policies that directly affect the work and livelihoods of developers. This afterparty culture is predominantly masculine. Almost all of the fully independent entrepreneurs I met at these afterparties were male, as were most of the senior engineers of the two larger indies that frequented Xcoders, OmniGroup and Black Pixel, although occasionally a few female employees of OmniGroup would show up. A typical Xcoders afterparty consisted of roughly twenty people, less than a fourth of whom were women. It is possible that the social bonding over beer created a feeling of an old-boy's club that, while not explicitly exclusive, tended to welcome sociable men while unintentionally discouraging women and men who preferred to go home to their families after hours.

What is clear is that, more so even than the regular meeting, the afterparty was a chance for the men who were the core of the club to hang out with their buddies once or twice a month. Some core members, especially of Seattle Xcoders, did not even bother to attend the regular meeting, but only went to the afterparty for social purposes, in part because, as experts, they felt no need to attend the regular meeting:

> We would always go drinking after Xcoders and that's really where the fun part happened. Eventually we stopped going to the meetings and just [went] drinking. The meetings still happened, but the Omni guys and Brent, Brent will sometimes go to the meetings, and Joe will go to the meetings, but I would skip the meetings, just because parking's hell downtown, and then it's just much easier to directly go

> to the part where, you know… so, we usually go there first, and that's where the meetings happen now. And we do talk about technical stuff.
>
> (Gus Mueller Interview, February 21, 2012)

Mueller credits going drinking after the formal meeting as the primary reason the Seattle community is so close, socially:

> When he was an [Apple] evangelist, they came up to Seattle, to talk with us, and he went to a CocoaHeads meeting and then he gave a little presentation and we're like, let's go afterwards, and they came along and they were kind of amazed at what we had and how tight it was. And he asked us the exact question you just did, "How did this happen?" And our response was, "beer." And that's what he did; he laughed! And we were dead serious. It's just what it is. It's just like, going out and drinking together.
>
> […] As long as you're not an asshole, you can come out and have fun, and we'll get to know you. We've had people drive up from Portland to come up and hang out with us, for this. But I think it's just, you show up and 99% of the time you're gonna have a good time. But yeah, it's… beer.
>
> (Gus Mueller Interview, February 21, 2012)

Mueller describes getting together for beer as the primary reason that the Seattle developers have bonded. What I observed of the Xcoders afterparty was that it was more or less a group of guys hanging out at a pub with their friends, a typical male bonding activity, except that they talked about technology rather than sports. Moreover, it was at this afterparty that I was introduced to most of the big names in the Seattle developer community, who were regulars. Indies such as Gus Mueller, Hal Mueller (no relation to Gus), Brent Simmons, and Chris Parrish were mainstays, as well as many of the leaders and senior engineers of the larger indie Cocoa companies, Black Pixel and OmniGroup. I discovered that Wil Shipley, who has since moved from Seattle to San Francisco to be closer to the center of the tech world in Silicon Valley, had been a kind of ringleader for this afterparty group in terms of leading the charge to go drinking and partying. Shipley felt that the group provided him with a safe place where he not only felt like he belonged, but where everyone was united in a sense of cause and mission about making great apps for the Mac.

> I really miss it; I miss drinking with those guys and you know there was a closeness and a camaraderie that—it's an incredibly powerful thing. Everyone needs a tribe… And yeah, I very much felt like they were my tribe and it was really hard to leave them.

(Wil Shipley Interview, April 18, 2012)

Since Shipley's departure, the party ringleader position seems to have been filled by Brent Simmons, who, at the end of the afterparty, led a subset of people to an *after*-after party at another bar after the Cyclops closed, lasting far past midnight. Thus, there was a successive winnowing from the club to the afterparty to the after-afterparty, where the inner core members of Xcoders formed the closest friendships with each other. Needless to say, very few of the women who attended the afterparty attended the after-afterparty. The men who attended were either single, or had wives who understood that this was their male-bonding time. Also, fewer of the men who worked for the larger companies, Black Pixel and OmniGroup, attended the after-afterparty, which consisted mostly of indies who kept their own schedules.

The friendships that developed between these men have done so largely outside of work, especially as many of them work alone. For indies who are social isolated, the Xcoders group presented them with a place to hang out with the like-minded and share knowledge. Chris Parrish, for instance, felt grateful that he had found in the Mac developer community a group of buddies to which he could belong:

> I can't remember how many times I've sat down with Brent Simmons and our toast has been, "Here's to Mac Developers!" Because it's like there's this group of people that we are all so lucky to have met and found each other. And I would never have guessed that this is what it was like, that the typical Mac developer was this person that I can become really good friends with and rely on and enjoy so much… Across the board, [I] just keep meeting more and more people over the years now, as I started ramping up, like starting here in Seattle and meeting more and more people around other conferences… MacWorld and… C4 [an independent Mac developer conference]… just consistently awesome people, one after another.
>
> (Chris Parrish Interview, March 2, 2012)

Xcoder attendees felt that this social closeness in the community was rather unique, especially compared to Windows developers, who only attended Microsoft sponsored events for professional learning and networking, and seldom formed lasting bonds with other attendees. Gus Mueller mentioned how newcomers to iOS from Windows were surprised at the level of friendship and sociality among the Apple oldtimers:

> Windows programmers... I'm sure those are button-down guys, and these guys are now sort of refugees from… wanting to go indie with their iOS apps… and I think [our friendly beer culture was] just not what they were expecting, and as far as I could tell, they were happy with it.
>
> (Gus Mueller Interview, February 21, 2012)

Nevertheless, Xcoders' beer-drinking, pub-hopping culture could exclude as much as it could include. Despite the claims by the Xcoders that newcomers are welcome, the afterparty culture could reinforce the feeling that the social core of the group was a clique. Gus Muller admitted that some newcomers expressed feeling excluded because of the focus on alcohol.

> …Some people would get pissed and would be like, some of us don't drink, you know. And it's like, well you don't have to drink there, but it just seemed like so much of the community revolved around just, you know, hanging out at a bar, well, restaurant. Some people got kind of upset about that, but I never really saw those people. I would see them on the mailing lists, but they wouldn't show up to the after-meetings… It wasn't a whole lot of complaints, but I do remember some complaints.
>
> (Gus Mueller Interview, February 21, 2012)

This beer pub culture and the inner clique of male friends it fosters is probably a key reason why the culture of Xcoders as a whole is masculine. It is not that the club explicitly excludes women; far from it, most of the men express hope that more women become Mac developers. However, the core of the Xcoders culture is a homosocial world, a fraternity of like-minded men who are bonded through drinking together, and through the shared experience of being indie developers. The identity of the Xcoder is imagined as male, like Dunbar-Hester's radio activist geeks

(Dunbar-Hester 2008). What is striking to me is that many members of this inner clique within Xcoders are also the Cocoa developers who also have the largest online presence, who consistently blog, have large Twitter followings, and are frequent guests on podcasts. This means that the masculine identity of the Xcoder gets presented to the much larger community of Cocoa developers who follow them online.

The social bonds formed at Xcoders is the primary reason for the collegiality and camaraderie among the indies of the Seattle community, who frequently help each other out in a sentiment of "I scratch your back, you scratch mine." For example, Chris Parrish, a co-founder of the now-defunct indie Rogue Sheep and a former engineer on Adobe's InDesign team, had written an image filtering plug-in that he could not find a way to commercialize and make money from. Gus Mueller had an image editing application called Acorn, and Parrish thought Mueller might have a use for some of the code, so he gave it to him for free. Acorn now ships with Parrish's image filtering code. Informal social interactions at Xcoders also provide opportunities for indies to form formal or informal business collaborations. Brent Simmons, the original developer of NetNewsWire (a Usenet newsgroup and RSS feed reader) decided he wanted to devote his time to a new project, and needed to find a likeminded developer to take over his old app, give it a home, and continue to further develop it. The personal relationships Simmons had formed at Xcoders with Black Pixel engineers, including co-founder Daniel Pasco, led to the sale of NetNewsWire to the latter company. Indeed, in interviews with me, Brent felt that the Black Pixel company itself could not have existed without the Xcoders group, as it formed as a collection of local friends who wanted to go into business together developing Mac OS X applications.

> I think for one thing, Black Pixel might not even exist as a company without the social aspect…
>
> Without Daniel—you know he had a very small company quite some time ago, and it was not doing very well in the early days, that's common for companies. But he went to Xcoders religiously, he met people, he learned a lot of stuff, and since then has been able to build

> up a really fantastic company. …It's hard to say how things would have turned out differently without that social aspect. He might not have found those employees, might not have learned the key things he needed to learn… A lot of other developers have been helped by that, me certainly. People like Gus and Chris have been helped a ton.
>
> (Brent Simmons Interview, February 17, 2012)

Undoubtedly, countless other technical exchanges and formal or informal business collaborations among indies may have occurred during the monthly Xcoders afterparty. What Xcoders shows us is that, despite the indie ideology holding up the lone indie developer as the ideal model for what a programmer should be, implying a either an atomized individual working through the market or a mere node in a network, the most successful indies in actuality depend on a local community with deep social ties, formed over consistent face-to-face interaction. Although these indies do not have any formal institutional or corporate ties to each other, their bonds go much further and deeper, and these social relations form crucial resources. Members of the Seattle community share knowledge and code, and transact business deals with each other that would be highly asymmetric if the two parties were not friends. The Seattle Cocoa community likely functions more as a gift economy than a market-based one when it comes to intellectual property.

While Xcoders might be one of the most close-knit local Cocoa clubs, its afterparty model of sociality appears elsewhere. Both CocoaHeads Atlanta and the downtown Atlanta iOS Developer's Meetup have afterparties at local bars and restaurants. CocoaHeads, given its smaller, more exclusive membership, generally has almost the entire group go to the afterparty at a local brew pub, while in the iOS Meetup, only a small group of core members (some of whom also attend CocoaHeads) consistently attends the afterparty at a Mexican restaurant, although at any one afterparty, maybe a third to half are members who are new or only sporadic attendees. Alcohol is imbibed at both afterparties, although it does not seem to be quite as central to them as it is to the Xcoders crowd in Seattle. Organizer of the iOS Meetup, Rusty Zarse, has made encouraging socialization an important part of the group:

> I've tried very hard to maintain that dynamic, that sort of social aspect of our groups. For a little while it was getting much more presentation-centric…
>
> So I've really tried to encourage, and I think the beginner topics have helped, I try to encourage dialogue and we make sure to have our social afterwards and go have drinks and make sure it's not just a presentation. You're not just going there to hear someone talk about a technology topic that's going to make your job better.
>
> (Rusty Zarse Interview, September 25, 2012)

The Atlanta iOS Developer's Meetup is a significantly larger group than Atlanta CocoaHeads, and because many attendees are relative newcomers who are only interested in iOS because they want to make a mobile app, and are only interested in information, professional development, and possibly business opportunities. (Recruiters often attend the meeting.) Rusty tries to maintain a welcoming atmosphere, but also wants to foster the sense of community, friendship, passion and commitment that is the feature of other Cocoa clubs:

> You're going there because you love what you're doing and you're passionate. There are people out there that want to learn, there are people there that want to share, so I think the difference, R. D. [A former Apple engineer] said it once, when he started joining our group… "I was really looking for my tribe. And people that I really have an affinity with, and share a passion with as well as my interests and what I like to do and what I like to talk about."
>
> Apple, it's a social community, you're going to learn, you want to learn something, but I think the goal—the goal is to learn a little bit at the meeting, if there's a good presentation that's interesting, but really you're also trying to meet the people who know about that stuff, so you can then network with them, so that when you have a problem you can work with them, and likewise if they have a problem they might rely on you, but also just to extend your network of friends, but I think just beyond just professional.
>
> (Rusty Zarse Interview, September 25, 2012)

Rusty, as the organizer of the group, is trying to appeal to two different audiences. Every meeting has two presentations, and one of them is always on a beginner topic. When I attended, I observed that the vast majority of attendees are there for purely professional purposes, and don't interact with anybody.

Unfortunately, this includes most of the women and minorities (which includes mostly South Asians and East Asians, but also African Americans and Latinos in smaller numbers) who attend. These people are the likely target of the beginner presentation. The more core members, who often also attend Atlanta CocoaHeads, are interested in much more than learning about iOS. Like at Seattle Xcoders, they are there to hang out and geek out with like-minded people and form social bonds. These members are the relative experts who participate, volunteer to give presentations, actively post on the group's mailing list, and more often than not, are white men. Because of its relative inclusiveness compared to CocoaHeads, the group uneasily straddles the line between a professional organization (the Microsoft model) and a hobbyist club/community of interest (the Xcoders model). The afterparty winnows out those who only attend to passively receive information.

> It's like an open club. There are groups that go biking together… Everybody shares an interest and a passion for biking and they get together and go on bike rides, and then afterwards, typically… if it's an established group, they're going to follow up with an after-party of some sort, at a restaurant or coffee house or whatever. So that same sort of mentality is definitely a part of our Cocoa group, is we socialize as well, and when there's something interesting we try and—when we go to WWDC [Worldwide Developer Conference]… people coordinate, say, let's go together, let's get together there, so we tend to like each other and we tend to be friends as well.
>
> (Rusty Zarse Interview, September 25, 2012)

Like at Xcoders, the social makeup of the afterparty reveals the level of social centrality of the attendees. Only those who want to form social bonds and friendships with each other attend the afterparty, typically between six to a dozen, compared to the thirty or forty in the regular meeting. Also like Xcoders, additional socializing occurs even after the afterparty is over. A small group of the more committed members often hangs out and continues the conversation in the parking lot of the restaurant, sometimes until quite late.

> I can't get people to leave. I've cleaned up the whole place and I've got the chairs reset and I've still got twenty people standing around talking. And we had one night at La Fonda [Restaurant] where there were six of us who stood around outside after the La Fonda closed and

> we were still there at three in the morning. And I left; I was like, guys I gotta go home. And we were standing in the parking lot, in Howell Mill [Road]. Probably not the smartest thing in the world. And it turns out like three of them stayed until four in the morning talking. Standing outside their car. They just couldn't let it go.
>
> (Rusty Zarse Interview, September 25, 2012)

For those members who stuck around to talk late into the night, the iOS Developer Club was where they could hang out and discuss their favorite topic, Apple technology, with their buddies, just like at Seattle Xcoders. I myself participated in a number of these late night conversations.

As we have seen, Cocoa developer clubs are sites where people with a common interest in developing software for Apple platforms congregate and share knowledge, but it is also a place where sociality transcends mere professional association and becomes genuine friendship. Of course, if Cocoa developers only met in local groups, it would be more accurate to say that there is not one Cocoa community, but many local Cocoa communities. And in a sense this is true. However, these local groups are linked together through the online discourse of blogs and Twitter feeds, and more powerfully through conferences.

Apple Worldwide Developer Conference (WWDC)

In Coleman's examination of the sociality of free software hackers, the "Con" (short for conference) emerges as the "single most important site of in person sociality." (Coleman 2013, 45) Unlike the free software world, however, the lifeworld of the Cocoa developer exists in relation to the proprietor of the platform they have devoted their lives and allegiance to: Apple. In the Apple developer world, the most important conference is Apple's official Worldwide Developer Conference (WWDC, verbally abbreviated affectionately as "Dub-dub," short for "WW") and there are numerous community-run conferences. WWDC itself is a tech industry developer conference hosted by a platform proprietor whose purpose is to announce and "evangelize" to developers new software features (often in the form of APIs) in its platform that it wants developers to use in order to show off the competitive

advantage of apps on the platform over others. Most software platforms dominated by a single company have run such conferences now or in the past to promote their platform, such as Sun (Java), Palm, Microsoft, and Google (Android).

Because these platforms typically are proprietary, the developer conference is usually the first place third party developers can learn about the latest APIs, and for many professionals it is a must. For longtime attendees of Apple's WWDC, it serves three purposes (Dempsey 2014). First, the talks and sessions given in the main venue of the conference itself provide vital information for developers, and for Apple it is also a way to shape developer norms by discouraging certain practices and encouraging others—in part because Apple reserves the right to unilaterally change implementations or deprecate APIs and recommending to developers to be good citizens and follow best practices (especially not using private APIs) will best prepare developers for such changes in the future. Secondly, Apple provides labs where third party developers can ask Apple engineers, often the very engineers who wrote the framework code and APIs that developers are using, for help in solving problems. Coleman notes that at hacker cons, developers working face-to-face often "overcome some particularly stubborn technical hurdle, thus accomplishing more in two days than they had during the previous two months." (Coleman 2013, 53) This is even more true for WWDC Lab sessions, where because of the closed nature of Apple's code, getting face-to-face interaction with the engineer who wrote the code is often vital to fixing a difficult bug—this alone can often justify the $1699 cost of a ticket to WWDC, not to mention plane tickets and a week of hotel accommodations in downtown San Francisco. Thirdly, WWDC, as the most important conference in the Apple developer world, provides networking, socializing, and community building opportunities that are unmatched by any other conference.

Coleman notes the ritualized nature of conferences for reinforcing group solidarity (Coleman 2013, 47), and WWDC is no exception. The most talked about event at WWDC, of course, is the Keynote speech by the Apple CEO. For one, since Apple stopped participating in the consumer tradeshow MacWorld Expo, WWDC has become the only public venue for the Keynote speech, and especially when Steve

Jobs was alive and actively running the company, his Keynote presentation, with its legendary product revelations, was a hot ticket in the information technology industry—people regularly camped out overnight to get into the Keynote. Given Apple's continued influence in IT and mobile computing, Tim Cook's Keynotes continue to be major draws for both developers and the technology media alike. Keynote speeches are part technology demonstration, part marketing, and part church revival meeting. As we discussed earlier, Apple's presentations at WWDC are supposed to communicate new technologies and developments to its third party developers so that they can update their apps to take advantage of the latest capabilities Apple is providing them. The Keynote, however, has a slightly different purpose than the presentations during the rest of the conference. It is open to the press, and is live streamed, so its audience is not just the developers sitting in the room, but includes the general public. This means that it sometimes contains eagerly anticipated product announcements, in addition to PR material about Apple's business, material that is not strictly technical. The Keynote also often serves to present Apple's official philosophy (its ideology) to its developers, users, and the general public, and to enroll them into this ideology. In Steve Jobs' heyday, the Keynote speech was the site where even the most jaded observer sitting in the audience might become taken in by his famous "reality distortion field," generated in large part through the infectious atmosphere of the audience itself, whipped up to a frenzy through Jobs' masterful manipulation. Like a Catholic Mass or the President's State of the Union Address, the Keynote is very carefully choreographed to elicit responses from the audience (applause and even cheers) at particularly dramatic moments when a new product, its features, or its price, are revealed. The Keynote speech was where Jobs' charismatic authority waxed highest. In a Durkheimian sense, the emotional euphoria experienced by the audience, which becomes a kind of "laity" in their unified response to Jobs' ritual unveiling, regenerates and reinvigorates the collective "mana" or spiritual energy, reinforces the feeling of sacrality of Apple products over profane alternatives (Windows, Android, etc.), and reaffirms, for indie Cocoa developers, the passion and commitment to continue to devote their lives to writing apps exclusively for Apple's platforms (Durkheim 1965).

As we saw earlier, STS literature on demonstration can illuminate the role of the Apple Keynote. Demonstrations of are often key to selling a new technology, as often users may not know what to do with it; the demos help create the use and thus the demand for the product, as Pinch and Trucco's study of Van Koevering's demonstrations of the Moog synthesizer show (Pinch and Trocco 2002). Apple has managed to successfully create or vastly increase demand for new computing products that a consumers never thought they needed before, including PCs, MP3 players, smartphones, and tablets. Successful product launches like the iPhone's depend critically on a successful Keynote presentation, for which Steve Jobs was legendary. It is in large part for this reason that Apple follows a strict policy of secrecy about products in development, in order to make the biggest splash possible during its Keynotes. The product demonstrations themselves are carefully rehearsed, as their purpose is to dazzle the audience with the capabilities of Apple's new technology to make them believe that they can do things no other company's technology can. Because these technologies are often still works in progress, however, there is the possibility that things may go wrong. Multiple backup machines are usually made available in case this happens so that the presenter can smoothly go on; the demos are never supposed to turn into experiments. For this reason, there is an element of stage magic to the demonstration; elements might be canned, and engineers usually brief the presenter to make sure that they don't deviate from the script, risking the demo not working. And like magic, these demos are intended to produce in the audience a similar affective response, of being utterly amazed by what they are seeing. The difference with stage magic, and like scientific demonstration, the audience is being persuaded to believe that what they are seeing is not illusion, but reality.

Other events at WWDC are also important. The rest of the conference is composed of talks that go into successively more technical detail, and the audience is restricted to conference attendees who are usually programmers, although managers sometimes attend. The first afternoon after the Keynote is filled with "State of the Union" talks (sometimes called "Kickoff" talks in recent years), where an vice

president or other upper-level executive usually presents an overview of a subset of particular developments in the platform of interest to developers: Developer Tools, Frameworks, Graphics and Media, the Operating System, and in the past, Enterprise or Server technologies. The sessions that begin Tuesday and run through Friday are usually specific to particular new APIs, frameworks, or developer tools, and they are organized into "tracks" that follow the theme laid down by one of the "Kickoff" talks.

There are a couple of events that fall outside of the main sessions. Usually, during lunch, there is a lunchtime session that is often presented by an outside (non-Apple) speaker. In the past couple of years, one of these sessions has included a presentation from a major media personality of interest to geeks, such as the Star Trek actor and Reading Rainbow host Levar Burton, or J.J. Abrams, director of the recent Star Trek films. In both cases, the celebrity was there to talk about how Apple's platforms have aided them in their current creative endeavor—Burton, for example, discussed his Reading Rainbow app that he was promoting to encourage child literacy. In WWDCs of the 2000s, one of the most popular sessions was given by an employee of Pixar, Dr. Michael B. Johnson, known in the Cocoa community by his nickname "Dr. Wave," who demonstrated how he used Cocoa to build in-house applications to support its production pipeline for animated films. Another popular event has been the Beer Bash, a giant party/concert held generally Thursday, in the late afternoon to early evening. The Bash used to be held in the quad on Apple's corporate campus in Cupertino, and after the Conference was moved to San Francisco from San Jose in 2001, attendees had to be bused down and back up. This continued to be extremely popular because it allowed attendees to pay a pilgrimage to the corporate "Mothership" and was an opportunity to meet Apple engineers who were not attending the conference, as well as pick up souvenirs from the Apple corporate gift shop. However, in recent years, the Beer Bash has been thrown at Yerba Buena Gardens in San Francisco instead, probably to avoid the cost and time of busing people to Cupertino.

Two additional events stand out. One is the Apple Design Awards, an event where Apple awards certain third party apps for being the best example of good

design (both aesthetically and in terms of usability) and for use of new Apple APIs. Winning an Apple Design Award is considered to be extremely prestigious, especially among the indie community; on the other hand, the Design Award is also treated with some cynicism, as it is understood to be part of Apple's technical marketing machine, a way of promoting both the adoption of new technologies to differentiate Apple's platform and a way of promoting particular aesthetic values among third party apps that fit in with Apple's own. The second event, usually taking place immediately after the Design Awards, is a game show called "Stump the Experts" which celebrates hacker culture among Apple developers. At "Stump," members of the audience (third party developers) compete to ask a panel of current and former Apple engineers arcane technical questions and hope to stump them. Apple engineers can ask tricky questions of the audience. The audience members who are able to stump the Apple experts receive a prize, ranging from a T-shirt to a free copy of a software package. The game show host is snarky and humorous, takes special joy in making fun of audience contestants, and makes no pretense of being fair in judging answers. The audience understands that the game, while not being explicitly rigged, is biased against them, and that it is not usually possible to actually "win" the game by outscoring the Experts, but that is all part of the fun.

In addition to these official events, there are a significant number of afterparties, often sponsored by a smaller technology company (like Testflight, a company which offered solutions for deploying mobile apps for user testing, since acquired by Apple) or sometimes a technology media company such as ArsTechnica, and held inside a major San Francisco hotel. These are usually loud, crowded, glitzy affairs, with lots of alcohol and finger foods. There seem to be a lot more women at these parties than at the conference itself, most of them fashionably dressed. Since one party I attended was put on by the media site ArsTechnica, some of them could have been press, such as ArsTechnica writer Jacqui Cheng, while others may have helped organize the party. Nevertheless, one Asian woman I interviewed at a party was a mobile developer for a large company. A lot of the after-hours activity involves hopping from one party to another. Some oldtimers, however, favor lower-

key venues, where networking and socialization are easier, such as hanging out at the Chieftain, an Irish pub in the SOMA district of San Francisco. Some indie companies, such as OmniGroup, sponsor their own smaller parties. In recent years, a couple of more notable parties, especially from the point of view of the oldtimers, have emerged. Dr. Michael B. Johnson of Pixar has for the past few years organized a charity event at the Cartoon History Museum in San Francisco as an opportunity for ex-NeXT engineers and developers attending WWDC to congregate and reminisce about the history of their platform. James Dempsey, who became famous at WWDC as an Apple employee for performing songs he wrote about Cocoa and other programming topics, has with his band, The Breakpoints, been performing sets of his songs at a local bar. Apple engineer Bill Bumgarner, meanwhile, has for years hosted a gathering at Tommy's Mexican restaurant and tequila bar in the Richmond district of San Francisco. This gathering is rather exclusive and is dominated mainly by Apple engineers, but one can get invited if one happens to personally know an Apple engineer headed there, as it is a considerable taxi-ride away from the SOMA and Yerba Buena districts where the conference takes place and where most of the other action is occurring. Indies who are socially well-connected to Apple personnel are also in heavy attendance, especially a significant portion of Black Pixel.

At WWDC, members of local Cocoa communities tend to travel as a group, as their frequent face-to-face connections have formed stronger personal bonds. "When we go to WWDC, the Seattle guys usually stick together too. It's kinda funny." (Gus Mueller Interview, February 21, 2012) However, online social connections formed on Twitter and blogs help to arrange physical meetings at WWDC that strengthen these connections and create new ties between members of the various local communities. Often younger community members will meet in the flesh for the first time a famous developer who they follow on Twitter or whose blog they read. Coleman notes the importance of such physical interaction for reinforcing relationships that are otherwise wholly digital: "The prospect of finally meeting (actually in person) people you often interact with, although typically only through the two-dimensional medium of text, is thrilling." (Coleman 2013, 49) A number of

developers commented on the importance of face-to-face interaction for maintaining the sense of community:

> Like this week [at WWDC], I met a bunch of people who I've been following a long time, or who's work I've been interested in for a long time, but me being me, I don't feel comfortable talking to someone that I haven't been introduced to, so… So it wasn't until this week that I actually, pumping flesh and whatever. And now I feel completely comfortable, I consider them my peers rather than some guys over there whose work I like.
>
> (Chris Clark Interview, June 12, 2009
>
> On-line is great and wonderful, but there is no replacement for kind of this one-on-one meatspace kind of thing.
>
> …Blogs and that kind of stuff… are great, because I learn a lot of stuff, but the meatspace stuff, you know, you hang out and have a beer or two with somebody. Quite a different tangent. So most of the people I consider more than just acquaintances, kind of up to friends and beyond are all people I've met in meatspace.
>
> Started off as friendship or something on line, but eventually we've hooked up in meatspace and "you look nothing like I imagined you would look like."
>
> …And once you kind of know that there is a human being behind that IRC nick[name], or that email address, then [it's harder to say or do mean things to them], those are harder to do when you're dealing with somebody that you have met; they know you, you have shared experiences together, so generally the level of discourse becomes friendlier. Because you still agree to disagree on stuff and have different technological or political beliefs. But there is that commonality of mutual respect, mutual understanding, mutual affection, in cases that kind of transcends that which you don't get from a purely online experience.
>
> (Mark Dalrymple Interview, April 11, 2012)

Thus, social bonding and conversation in person, often over alcohol, plays a significant role in the construction and maintenance of personal friendships and professional relationships among indie developers. At WWDC, it also fosters such relationships between third party developers and Apple employees, creating a sense of shared community between them. Like the afterparties of local club meetings, for quite a few of the oldtimers, the social scene of WWDC is almost more important to

them than actually attending the conference itself, as it is the one time a year they reconnect in person with friends from around the country. Oldtimers are much less interested in the technical content of the talks given by Apple at WWDC, especially as Apple now puts videos of the talks up online soon after the conference. There seems to be an inverse relationship between relative inclusion and centrality in the community and how much emphasis attendees place on the formal sessions at WWDC.

WWDC most certainly is a venue where Apple actively courts and enrolls developers in its ideology, that they are improving society by making their apps more useful, pleasurable, and usable to everyday people, and where it advocates normative practices, such as encouraging certain design patterns such as delegation, or avoiding the use of private APIs Apple has not published, because these practices are more in line with the Cocoa idiom, because they will help make apps more maintainable or usable, and because they will help developers more easily adapt to changes Apple is making to its system over time. Because Apple's platforms are proprietary, it reserves the right to change things, and the Apple developer community has largely accepted that it is better to conform to the way Apple does things than to try to fight it. The attitude is, if you want to program for Apple, you have to play by their rules, because it is their platform. Apple, like other companies, also hands out gift items that come with the purchase of a WWDC ticket. In the past, this has included laptop bags or backpacks and jackets, all emblazoned with the WWDC logo and the year of the conference. Developers who attend WWDC wear their jackets with pride, as a marker of special access and a sign that one has joined the community. Former Apple engineers who have participated in the annual Stump the Experts game show sometimes wear commemorative T-shirts that display their membership in an elite group.

It is clear, however, that the more experienced Apple developers, which include most of the Cocoa indies, see the conference in a slightly different light than most of the attendees who are relatively new to WWDC. Oldtimers have attended many past WWDCs and can easily parse Apple corporate marketing and public

relations messages, and often take a snarky, cynical, ironic stance to such messages. Being relative experts, they are less interested in the technical content of WWDC sessions and are more interested in parsing the meta-message of what directions Apple and its technology and platforms are headed, and more specifically, how it impacts their interests as third party developers. This plays into the relative disinterest of many oldtimers in continuing to attend WWDC at a time when it has become more difficult for them to get a ticket. As the social and networking aspect of the conference is, on a whole, more important to them, they are able to satisfy this need by being in San Francisco during the conference to meet people and hang out, but not necessary attend WWDC proper. What is important, however, is that this ironic stance does not mean that these oldtimers are opposed to Apple's agenda. Some of these oldtimers are former Apple engineers themselves, or have social connections to Apple engineers, and can thus separate Apple's official corporate message from the intentions of the well-meaning individuals who actually work there. Oldtimers, being highly included in the Apple techno-cultural frame, including its ideology, largely agree that what is good for Apple and its platforms is good for them; the difference is that they reserve the right to criticize Apple for pursuing policies or releasing technologies that hurt developers, which not only hurts their interests but, they argue, harms Apple's platforms. This may be very different from many of the newcomers who are attending WWDC for the first time. These newcomers may be more suspicious of Apple as a whole and are less able to differentiate between Apple policies or technologies that are "good" or "bad" in the opinions of more experienced Apple developers, instead seeing Apple as more acting in more monolithic fashion that it actually does. Alternately, newcomers may simply take in all of Apple's messages from its sessions uncritically. For less highly included newcomers, Apple's messaging is more all-or-nothing, take-it-or-leave-it.

Community-run conferences

WWDC is not the only game in town, however. Since the 1980s, Apple developers have organized their own independent conferences, and these are much closer in flavor to free software hacker cons—one of the early ones was, in fact,

called “MacHack.” There have been a number of various community-run conferences over the years related to Apple, the Macintosh, NeXTSTEP, Cocoa, and/or iOS, and as some have gone away, others have taken their place: C4, 360|iDev, Voices that Matter, NSConference, CocoaConf. In the mid-2000s, C4, meeting in Chicago, was the central locus of the indie Mac developer community, but fell by the wayside after organizer Jonathan “Wolf” Rentzsch moved on. NSConference, one of the few non-North American conferences, meets in London. 360|iDev, which usually meets in Denver, is one of the newcomers, being exclusively devoted to mobile iOS development, whereas many of the other conferences include both Mac desktop and iOS and are often dominated by speakers who are indie developers of desktop Cocoa Mac apps. While most of these conferences meet once a year, CocoaConf is a smaller event that holds multiple meetings a year (as often as once a month), and travels to various cities: San Jose, Columbus, Las Vegas, Seattle, Boston, Atlanta. Community-run conferences have become increasingly important to indies in recent years as the interest in iOS has resulted in explosive demand for tickets to WWDC, making it difficult for small shops and individuals, even those with name recognition and/or close ties to Apple employees, to acquire tickets. As mentioned earlier, demand for WWDC tickets has increased so dramatically that it now sells out within minutes of availability. Longtime members of the community, who in past years could count on getting a ticket and relied on WWDC to make face-to-face contact with friends they see once a year, have been increasingly shut out in the cold, as Apple tries not to play favorites. These developers have needed a physical site of their own to congregate. As a result, since at least 2012, the community has organized its own AltConf (originally known as AltWWDC with its own Twitter hashtag), also taking place in San Francisco during the same week as WWDC, to facilitate developers who come to town for the socializing and networking aspect of “Dub Dub.” Because of AltConf, and because Apple now puts WWDC session videos online shortly after the conference, two out of the three functions of WWDC for many oldtimer Cocoa developers can be sufficiently met without attending WWDC at all. Unfortunately, the Lab sessions, which grant developers exclusive access to help from Apple engineers, cannot be replicated by the community.

At CocoaConf and similar community-run conferences, the keynote speeches are given by well-known and respected members of the community, often by a local luminary. At CocoaConf Atlanta in 2013, Aaron Hillegass gave the keynote speech. Many of the other talks are technical talks, similar to the talks given at local club meetings, while other talks are normative talks about best practices in coding and designing apps. However, occasionally at these conferences one finds talks where a prominent luminary (often an indie developer) either gives a testimonial about his experience as an indie, or more vision-type speeches on Apple's direction or mobile technology in general.

Such speeches at community-run conferences also help create solidarity among developers, but because they are given by one of their own, they have a different effect. If the Apple Keynote is a like an evangelical pastor giving a sermon, the keynote speech by an indie developer is more like a testimonial from a member of the laity. These accounts derive much of their impact through their personal nature, and speak to the moral import of the indie developer identity. Aaron Hillegass spoke about how, despite how grateful he was that he had managed to carve a career out of doing what he loved, teaching Cocoa programming, he could not help feeling as if he should do more to help the "next 5 billion people" on the planet—those in the developing world without access to information technology. His talk spoke to the limits of the moral project of the indie developer world—that somehow, making an app will improve the lives of everyone on earth. His dissatisfaction with not being able to do something about the world's larger problems motivated a talk in which he tried to get developers to think on a grander, more socially conscious scale. Nevertheless, because of his audience of indie technology entrepreneurs, Hillegass, though acknowledging the important role of government in innovation (something he thought might be controversial with his audience), his proposed solutions for such problems as clean water and sanitation necessarily had to be not only technological, but market-based products.

The closing speech of Atlanta CocoaConf 2013, was in some ways the exact opposite. The central argument of this talk, given by "Stan," was inspired in part by

Apple's own marketing, and premised upon the idea that not trying to solve the world's big problems, but by focusing only on the needs of a single imagined user (often one's self) was the proper normative stance for the indie developer. By fixing a single user's problem, a developer will create an app that has an intimate feel, as if the app spoke directly to the user, similarly to how a radio DJ (which Stan had been) speaks directly to a mass audience as if it were an individual, creating a sense of intimacy. This attitude spoke very well to the realities and pragmatics of indie app development, in which individuals and small teams simply do not have the resources to try to fix every problem or address mass markets and thus tend to focus on narrow market niches of users.

This "build it and they will come" mentality was very powerfully reiterated by replaying two Apple advertisements. The first was the famous "Think Different" campaign, written by Steve Jobs upon his return to Apple in 1997, which exhorted listeners to be iconoclasts in order to change the world, but was also a mission statement, a manifesto for what Jobs saw Apple's role was to be in the coming decade. The second was an Apple ad first shown to developers at WWDC in 2013, but later replayed to the public at the beginning of one of Tim Cook's product announcement events. This ad reiterated the central importance of focus in Apple's approach to products—doing one thing as best as it can for a particular purpose, honing a design to perfection, and saying no to everything else.

Stan, who is not an Apple employee, then reiterated how important this message was for developers like himself, as this virtue of focus is one that developers must emulate in order to make products as elegant and transformative as Apple. Succumbing to market pressure to include too many features in one's app does not benefit the user—it makes a product incoherent and difficult to use, like Microsoft Windows. Tailoring one's app to a narrow subset of users but making that experience the best it can be will not only make for a better app, but will create a loyal, passionate following of users, just like Apple has done with its own products.

To strike home his point, Stan told a story about how Apple's focus on creating the best user experience had powerfully affected his life. Throughout the talk, he carefully sprinkled photographs of his daughters in the slides. At the end, he revealed that one of his daughters had passed away before her eighteenth birthday, and that he and his wife had lamented that they had no pictures of her with her baby teeth missing. But because Apple had designed its iMac computers with built-in cameras and software easy enough to use that his children could pick it up themselves, his other daughter revealed that they had taken photos of themselves with it, and these irreplaceable memories could be found on their iMac. Thus, Apple's focus on the user, its perfectionism for design, had not only empowered his children, but intervened in his life in a powerfully meaningful way. For Stan, this was how Apple's products changed people's lives, not just in a utilitarian sense, but also in an affective sense, and this was what every app developer should aspire to. The moral and normative aim of every app developer should be to improve the lives of their users in both utilitarian and affective ways, thereby improving the world an individual at a time. This is the reason why aesthetics and usability is all important to Apple developers—their moral imperative, indeed their existential purpose as developers, is to affect their users on an emotional level. This ideologically frames their work and their identities—they need to believe that their labor constitutes a higher contribution to society.

Conferences and local developer clubs, by facilitating physical interactions between Cocoa developers in "meatspace," make the Cocoa community more than a networked public that exists only through its interaction with online discourse. Conferences, as ritualized occurrences, in which a heightened affective experience is shared with likeminded others, powerfully bonds developers into a sense that they are not alone, but working in a social, moral, and technical world where others share their values and commitments. Relationships that began purely as disembodied, textual conversations are made flesh. Local club meetings, by virtue of taking place more frequently, and drawing together developers who live in close proximity, have a more quotidian quality. Yet, such interactions are also powerful, as they engender

friendships that go beyond purely professional associations. This can go both ways, as friendships can also lead to business partnerships and startups. In all of these ways, physical meetings and interactions are as important as online interactions for creating a sense that a community of Cocoa developers really exists. Physical and virtual interactions reinforce each other and strengthen what neither can create alone.

Tensions between Elitism and Populism in Knowledge Sharing in the Cocoa community

Coleman has noted that the free software hacker community contains a tension between meritocratic elitism and populism, manifested particularly in its attitude towards newcomers who ask what they consider stupid questions and do not take the time to learn the basis for themselves, summarized by the joking but biting epithet, “Read the Fucking Manual” (RTFM) (Coleman 2013, 107-111). Some of this tension can be found among Cocoa developers as well, although there may be a higher tolerance for asking questions among the community simply due to the fact that the Cocoa frameworks are closed source. Despite Apple’s documentation of its APIs, many developers continue to feel that it is inadequate, and this has fostered vibrant knowledge sharing within the community. Core members, while experts, are still constantly learning from their peers and from Apple, as every year at WWDC, Apple releases new software capabilities and new APIs that developers are encouraged to use. Official WWDC sessions present both this new information and also enroll their audience in the normative practices for how to best use these new technologies in a way that plays well with the rest of Apple’s platform. Material from Apple is always the first source of the knowledge necessary for taking advantage of new APIs, but this knowledge alone is not enough. Apple can not anticipate how its technologies will interact with the legacy code and the real-world needs developers actually have, and developers’ real-world experience is in many respects more valuable to other developers. If knowledge from Apple is top-down, this kind of knowledge, from other developers, can be thought of as “lateral.” There are multiple sources of this lateral knowledge. One important source is a website called Stack Overflow, which is a kind of crowd-sourced online knowledge

repository for programming for any number of platforms. The reliability of answers to questions posted on Stack Overflow is managed through user voting that in some ways has become gameified: readers of the website vote for the best answers, and contributors compete to see who can, in aggregate, achieve the highest reputations for reliability of their answers. Stack Overflow thus constructs the same kind of reputation economy of its contributors as that of Amazon reviewers (David and Pinch 2005). Oldtimers in the community have contributed significantly to Stack Overflow. This includes key Apple engineers such as Bill Bumgarner, whose answers are generally taken as reliable and definitive.

The kinds of questions one asks, however, can mark one as a newcomer who doesn't get it. Robert Walker noted asking elementary questions that could simply be answered from reading Apple's documentation or a book on Cocoa programming (say, Aaron Hillegass's) clearly signals that the poster did not do their homework. Knowing what to ask shows that one has actually done some research on their own, tried out various solutions, done some hard thinking on a problem, and has an actual, legitimate problem with how to do something using Apple's frameworks. Thus, according to Walker, there are two tiers of people on Stack Overflow—the experts, who are trying to find answers to really difficult questions, or providing those answers, and "newbies" who are asking general questions and are looking for "the right answer," for someone to tell them exactly what to do, rather than do their homework and research to find the solution on their own. This speaks to the same "RTFM" attitude among hackers that Coleman examined. It may very well be that the finding out answers on one's own is a virtue upheld by programmers in general. Walker himself is in some respects relatively new to the Cocoa community compared to most Mac indie developers, but because of his earlier WebObjects experience, was already highly included in the Cocoa techno-cultural frame before the iPhone. He is a frequent participant in the local Atlanta clubs, has social connections to employees of the Big Nerd Ranch, and has done extensive work in iOS and Cocoa in his free time. For Walker, who has been accepted into the community through his hard work,

participation, and passion for Apple's platforms, not doing one's homework is a clear marker of peripheral status.

Walker, like many of the hackers in Coleman's study, had little patience for developers who asked stupid questions and did not take the time to find answers for themselves. Similarly, Dan Wood also denigrated programmers who wanted to be told how to solve simple problems. He noted that when he asks Apple's Developer Technical Support (DTS) for help, he has a real difficult question that he has thought about and is stuck on, but when newcomers ask DTS for help, they are asking basic questions, according to his friend who worked for Apple DTS:

> …I remember him saying …that most of the things that they get are from people who don't know what they're doing at all. And their code is just junk and just horrible and they don't really know what they're doing. And they are the ones who are going through DTS incidents to get help on stuff that is just really basic, and so you know, when I have a DTS incident, it's usually something that I have just sweated over and I've had multiple looks at it, and we've looked on Twitter, or a lot of times I'll even talk to an Apple engineer who said, "Oh yeah, you need to file a DTS engineer report so we can actually deal with this." As opposed to, just you know, why does this program crash?
>
> (Dan Wood Interview, April 9, 2012)

This led Wood to do boundary work, placing these programmers outside of the community. He immediately followed this statement with his comment on the "unwashed masses" of iOS programmers only "out to make a quick buck," explicitly connecting a lack of knowledge of Cocoa to the profit motivations that marked such developers as peripheral or even non-members.

These sorts of attitudes had led in the past to accusations of elitism in the Apple developer community. In 2007, a developer ranted on his blog that "we also had our first encounter with the curmudgeons we otherwise refer to as… the 'landed gentry of Mac OS X development'. They're a very unpleasant rude lot." Ironically, this developer appears to have been a NeXT fan who was using the label of "landed gentry" to accuse old classic Mac OS/Carbon era developers of being hostile and elitist: "We were attacked left and right by some of the most incompetent—and

simultaneously 'acknowledged'—developers on the platform. Most of these were not legacy NeXT people—they were people who'd stuck to Apple through thick, thin, and the Redwood City years [the NeXT years, when it was headquartered in Redwood City], only nodding occasionally to Chesapeake Drive [location of NeXT's headquarters]."[39] By labeling Mac developers "landed gentry," this developer is trying to say that these developers are an exclusive elite group who are not interested in welcoming anyone into their club, and are interested only in perpetuating their elitist culture. Aristocracy is counter to meritocracy and open participation, which ought to be how programmers are accepted into communities. This bears not a little resemblance to the "cabal" that Coleman found many hackers accused the gatekeepers of the Debian project of having formed when they made decisions that appeared to abuse their power.

The specific context of this developer's comment has since gotten lost. The Cocoa developers who brought this to my attention thought that this accusation was hurled at the big names in Cocoa development, which included former NeXT developers, who presumably were not among those the original writer was attacking. Nevertheless, because populism and meritocracy are so ingrained as virtues among programmers, this accusation apparently created significant controversy, precisely because it went against the view that much of the Cocoa community had for itself. Mike Lee, for instance, vehemently denied that his community was exclusive, despite the fact that a who's who seemed to populate the community's inner core, asserting that in his experience, he had been welcomed into the community as a newcomer.

> He coined the phrase, the landed gentry of Mac development. Which referred to exactly this phenomenon of the exclusivity of the names in Mac development. Where it's like, if it is the case that there is a group in development who you know by name, where this is a Wil Shipley project, or this is a Daniel Jalkut product, or this is a Mike Lee product,

---

[39] Rick, "The Longest Screed," January 3, 2010, *Rixstep Developer's Workshop,* Accessed June 8, 2014, http://rixstep.com/2/2/20100103,00.shtml

> sort of name developers, it only stands to reason that there would be this kind of exclusivity. Yet, my own experience says otherwise. And is in fact why I'm so bullish on the community. Is because when I was doing corporate Java programming, it was extremely insular… And when I switched over to Mac application development, I feared a lot of the exact same thing. And I specifically avoided the community and avoided talking to people because I didn't have to run into these problems where people were constantly trying to get in my way for whatever reason of insularity or whatever. And it was Lucas who disabused me of that. "These guys are actually really cool, you know they're just like you, they're just a bunch of nerds who look to get around, drink beer and talk about programming and all of this stuff and you would really do well to meet these guys. And so I did. If I had to put a date on my formal introduction to the community, would have been at last C4 [conference]. So you know, about a year ago.
>
> (Mike Lee Interview, July 23, 2008)

Other Cocoa developers, however, acknowledged that there was some truth to the accusation of exclusivity or cliquishness.

> It's true, and I think that… some of it is really well earned…
>
> …There are some people that are very outspoken, but haven't really been producers. And so it's kind of, it's—at some point, I'm like, dude, well, have you actually done something?
>
> (Daniel Pasco Interview, June 12, 2009)
>
> Some years ago, somebody referred to us as the 'landed gentry' of Mac developers. It must have been like 2006 or something, probably even before iOS.
>
> And I forget who it was that came up with that; I knew it was meant to be insulting, but you know, yeah I see the point. It is, people who've been around for a while, who are well known, and I don't think it's ever been an exclusive club, or a non-welcoming club, you know. I think that's an important distinction, right? Anybody who cares about the same things, it's like, hey, great, join us.
>
> (Brent Simmons Interview, February 17, 2012)

Despite this elitism among some members of the community, others feel that the community has been relatively welcoming to newcomers, as long as the newcomers share similar values about making apps for the love of the platform rather than money:

> …We are open to new people, and whenever someone new shows up, you know, like occasionally we'll be sitting at the bar, and people would hear about the after meetings on Twitter, thanks to Brent, and we'll see someone in the corner with a MacBook or whatever, and we're like, do you think that guy's with us or looking for us? And I've gotten up and been like, hey, are you looking for us? And the guy's like, No. [laughs] and it just happens to be a guy at a bar with MacBook… But definitely we're open to new people. We like new people.
>
> (Gus Mueller Interview, February 21, 2012)
>
> A year ago, I didn't exist in the community! Nobody knew who I was! I could tell them the products I was working on and they would recognize it, but I was not a part of the community, definitely. You know, I didn't have fuckin' Jalkut and Hockenberry and Shipley in my speed dial… And that is because, when I decided I wanted to be part of the community, the community embraced me. They welcomed me… And so I think really what it is, is I think that the community is basically, we don't—when somebody kind of pops into the scene, there's a certain amount of worry. There's a certain amount of "who are these guys." And a lot of that is because we've built ourselves a really pleasant little city here, and we don't want people coming in and fucking it up. Now you come in and you've brought cookies and you say hi to everybody, and you say, I really love what you guys are doing and I really want to be a part of this, everybody is just more than happy to have you.
>
> (Mike Lee Interview, July 23, 2008)
>
> As long as they love what they're doing, I think they can be part of the community. We can relate to that. Doesn't necessarily matter whether they love it because they love the device, or they love it because they love the industry, or whatever. I think where I have trouble relating is when they're it, and they don't love it. I mean that was sort of what the Windows community was about, from my perspective.
>
> (Ken Case Interview, March 23, 2012)

Elitism in the community exists in tension with openness to newcomers, partly because the community is interested in "evangelizing" the Cocoa way of development to others. While Cocoa developers believe that everyone ought to experience the same pleasure they do when programming Cocoa (all programmers ought to experience "grace," as it were) they are also concerned with reproducing their community membership, and care deeply that their values are sustained and

transmitted to the next generation. This also means that, despite much of the boundary work that goes on online, at clubs and conferences Cocoa developers are concerned with helping newcomers learn, as their activities are centered around technical presentations where speakers share their knowledge and practices with others. Knowledge sharing among the community does not take place only among experienced developers but is intended to help grow the community.

Because Robert Walker is himself a relative newcomer compared to longtime Mac Cocoa developers, he feels that introducing new people to the platform is beneficial, as it is a source of new ideas. However, having been acculturated himself, he tempers this by noting that it is important to make sure that the newcomers learn the "right way" to do things on the platform, respecting established coding idioms and conventions. This elevates the importance of knowledge sharing in the community as a way to make sure newcomers learn the correct values, attitudes, and practices, and to give them a sense of being able to discern expertise and right practices from wrong ones.

> It's just I think that now that this growth is sort of plateauing somewhat, finally, I think it's time to go back and refocus on, OK, here's really the right way to do this task, or the right way to do [that] task.
>
> So I think in a way it could use some additional, sort of, start focusing more on maybe building a community repository that is vetted by the experts.
>
> More likely third parties [rather than Apple], but people that you can trust to know the right way of doing things. And it's just a matter of just knowing who those people are.
>
> That's what we hope it to be, for sure. [The programming community to be a repository of knowledge]. That's what we hope it to be.
>
> (Robert Walker Interview, May 19, 2012)

Clearly, if the community is to grow in the right way with the right values and practices, then its members need to know who the masters are that they ought to emulate. It is these masters that can be trusted to provide newcomers with the right

guidance. This is doubly important because of all of the noise and incorrect information on the Internet.

> …We have the web [to find programming information] now, but… it's very easy… since there's so much on the web, to come across something that isn't quite right. So you end up incorporating code that actually has got problems. And you run into those problems and so that's where I think that community building can really help out… And I think that it probably stems from just the immediate explosion of the iOS platform itself. It went from nobody knowing much about Objective-C Cocoa programming to a whole new community of people that have never been Apple programmers in their life before.
>
> (Robert Walker Interview, May 19, 2012)

Because of the noise on the Internet, according to Walker, it is doubly important for programmers to have a local community of people to interact with and learn from. Social interaction in physical co-presence with others is key to improving one's technical skills.

> I spent several years kind of away from programming communities—you know in college it was a tight-knit programming community, there was four or five of us that would constantly be working on projects together in college and after a few years out and away from college, I started missing that connection with other programmers outside of the work context…
>
> Where I think these community meetings really come in handy, is building that community, that spirit, and… just being able to sit down with another programmer and say, yeah, I've been through that pain myself, and here's how I got through it… Because sometimes when you're stuck on a problem, and you're not in a community like that, you feel like you're on your own.
>
> (Robert Walker Interview, May 19, 2012)

Moreover, participating in the community by sharing one's knowledge with others doubly improve's one's own knowledge. So having a community is beneficial for improving one's programming skill in this way also.

> I'll try to focus on giving a talk on something that I'm interested in myself in learning. So I think it's very important if you're in a community, to participate. Because you can really learn a lot by just

> going through the process of figuring out how to show someone else how the concept works…
>
> (Robert Walker Interview, May 19, 2012)

For Walker then, knowledge sharing is the central activity around which a programming community is organized. Through knowledge sharing, newcomers are enrolled and socialized into the norms of the community, and taught what it means to be a proper member. This also means that newcomers ought to participate, defined as volunteering to present at a club meeting or conference, writing blogs, posting to Twitter, or answering questions on Stack Overflow. Proper community members participate in knowledge sharing because the social work of the community occurs through knowledge sharing activities. Moreover, because good programmers are supposed to be constantly improving their skill, and participation helps the sharer do this, it is also considered an act of self-cultivation. It is because the Cocoa community is a "community of practice" in Lave and Wenger's sense (1991) that sharing and transmitting its knowledge and practice to new generations is so central to its activity.

Knowledge sharing in the Cocoa community is not only about transmitting expert knowledge to new generations, however. It is also concerned significantly with the continual upkeep of knowledge among experts themselves. Programmers in general understand that their skills must be constantly updated because the pace of change in the technology industry is so fast. This is as true of Apple's platform as any other, but because of the particular nature of Cocoa development, with its tie to Apple, this means that in the Cocoa community, developers are constantly responding to technological change originating from Apple itself in a top-down fashion. Every year at WWDC, when Apple announces new APIs for developers to take advantage of, a whole new round of learning must take place within the community, as the experts try out the new functionality made available to them. Because of the proprietary nature of Apple's technologies, the actual bugs and kinks of working with the bleeding edge are worked out in the initial weeks and months after WWDC, and pitfalls and workarounds are valuable information that experts

need to share with each other. When the Cocoa community was small, they only had each other to share this knowledge with, and although most of their information came from Apple, frank, unbiased discussion of the pros and cons of Apple's new changes required community input, prompting the kind of tight social network we've seen. Despite the authoritative nature of Apple's official documentation and WWDC talks, developers understood that Apple had a vested interest in pushing them to adopt new technologies in order to show off what they could do with it. In practice, not all of Apple's new features work as well as they ought to—in 2012, Apple's iCloud APIs had been considered notoriously problematic and faced significant rejection among developers (Cheng 2013). Developers are thus not uncritical consumers of Apple's latest software technologies—their own experiences of how useful versus how difficult and frustrating Apple's software is for their apps shapes their views of whether a particular Apple technology is actually any good. These actual experiences are critically important to share among other developers who may be contemplating the decision to adopt new APIs. Because Apple's Cocoa frameworks are closed source, and documentation on new APIs may not yet be completed, it becomes even more important for the community to share knowledge among its members, especially considering Apple is constantly introducing changes that developers need to react to. Sharing of knowledge among experts is crucial to the maintenance of the existing community in an environment of constant change, and is separate from the reproduction of the community through enrolling new members.

The small, tightly knit, collegial nature of the Cocoa community prior to the iPhone and the centrality of knowledge sharing to the activity of the Cocoa community fostered a culture of openness of information that one might normally associate with open source programmer communities. Many Cocoa developers, in comparing their experiences in the Mac community with the Windows community, feel that the Mac community is much more open. This is ironically in direct contrast to the highly secretive policies of Apple itself, which is known for being draconian about leaks of future product announcements and technologies. Within the developer

community, however, open sharing of information, when not restricted by Apple NDAs, rivals that of free software hackers.

> Yeah, well, the interesting thing about it was the Mac community not being as large as the Microsoft [or] Unix community, tended to share. In a very large way.
>
> (Bill Moorhead, Black Pixel Interview, June 12, 2009)
>
> It was very open. More than I found elsewhere. And the people, there was a huge weeding out of that. So the people that were doing it were doing it and solving the problems. If you were coming to find a problem you very—you would find a lot of voices, a lot of the same voices. And answers to your questions. That was my experience, was trying to oh, how do you do this, there's somebody there.
>
> (Daniel Pasco, Black Pixel Interview, March 28, 2012)

Mark Dalrymple explicitly felt that Apple's secretive nature actually encouraged openness in the community as a reaction:

> Maybe it's in reaction to Apple being so kind of tight-lipped that our to kind of get our, you know, our interaction—to get more knowledge about the platform—we're not getting it from Apple, as good as their Tech Pubs department is, they can only do so much. They keep on moving the docs around so they keep on breaking the link, so sometimes it's hard to find chapter and verse on some stuff. Where if you go to Wil Shipley's blog there's nice—the Pimp My Code thing—where he takes somebody's code and he says, "here's what they did right, here's what they did wrong, and here's why I believe it." And that kind of stuff you don't get from Apple.
>
> (Mark Dalrymple Interview, April 11, 2012)

Dalrymple analogizes the sharing culture in Cocoa to a sharing culture among skilled Photoshop users, a culture that changed from one of secrecy and hoarding to one of openness and sharing.

> [Cocoa], It's not an information hoarding culture. …Actually it's kind of like [how] the Photoshop culture was before and after Scott Kelby formed the National Association of Photoshop Professionals… Before he did all that stuff… a select group of folks used [Photoshop], they had all these techniques but they wouldn't tell you about it. Maybe they'd sell it to you for $500… And then Scott Kelby came on the scene and then all of a sudden, very free sharing of techniques and

> ideas, a lot of it is so he can sell the books, but also… I want to show off because, hey, this is cool… So he and his crew pretty much revolutionized the Photoshop world into being a very open and sharing kind of thing.
>
> And I consider that the core of the Cocoa world is the same kind of thing… Scott Stevenson, he was another big name [in the community] before he went to Apple. The Cocoa Dev Central [website]. …This one site of this guy who wrote all this amazing introductory material just because he loved the platform and he loved the people and he's an awesome person. So it's like the nucleus of everything.
>
> (Mark Dalrymple Interview, April 11, 2012)

Dalrymple explains the Cocoa culture of openness and sharing in terms of another community of practice he is familiar with, Photoshop users, and drawing a direct analogy to it. He hints that this culture is not totally un-self-interested—open sharing sells books. Moreover, the desire and ability to "show off" something "cool" builds up the reputation and credibility of the sharer. Reputation economies and knowledge economies have been shown to go hand in hand in the sociology of science. Bourdieu has noted that authority, prestige, reputation, and competence are forms of social capital that are the true stakes of scientific fields, (Bourdieu 1975) and similarly Latour and Woolgar have observed that credibility, not money, is the currency that circulates in science (Latour and Woolgar 1986). Similarly, Coleman's work shows that open source hackers also work for prestige and reputation, often showing off through the cleverness of their code (Coleman 2013). The move in the Photoshop community from more closed, proprietary knowledge to open sharing of knowledge Dalrymple describes might have been a move from a view of knowledge as a form of economic capital to a view of it as social capital (a source of reputation and authority), which, if given away, could be converted into far greater economic capital than if it had been hoarded.

The culture of openness and sharing in the Cocoa community has thus helped to foster the rise of some of the famous names in it. Men like Scott Stevenson, who created websites and online repositories of knowledge, became central nodes in the social network of the Cocoa community. Similarly, Aaron Hillegass, whose books and training courses taught almost the entire generation of new Cocoa programmers

between OS X's release in 2001 and the opening of the iPhone App Store in 2008, is a central figure in the community, whom everyone knows. Hillegass's books and courses were almost an obligatory passage point for entry into the community during these years, and his opinions as to what practices were good and bad have almost as much weight as official pronouncements from Apple. It did not hurt that Hillegass himself was a former Cocoa instructor at Apple and NeXT.

Why is openness and knowledge sharing so important to the Cocoa community? Recall that communities of practice according to Lave and Wenger are centrally concerned with their social reproduction through the transmission of knowledge and practice to new generations of practitioners. To this end, knowledge must be openly shared within the community. However, because newer practitioners also necessarily change the existing practice through either innovation or the importation of foreign ideas, this can create conflicts with the old guard, which may wish to maintain the existing practice.

Nevertheless, valuing openness serves an important moral purpose for Cocoa developers, especially for indies, whose identity sits uneasily between that of open source hackers and corporate programmers. On the one hand, indies sell their work for a profit, but on the other, make a living off of their own ingenuity and creative production, in which personal reputation is as important, if not more, to making a sale than money spent on marketing. As we have seen, indies also regularly share code with each other or make code libraries open source for other indies to use. It is possible that when the Cocoa community and the Mac market for indie applications was relatively small, the social capital of reputation was worth far more economically than the raw economic value of any knowledge that could have been hoarded, and contributed to the open and collegial culture of the Cocoa indies. This has since run up against a more traditional view of programming knowledge after the iPhone gold rush, as large corporations and a lot more money has entered the field.

Knowledge sharing for indies is not just important for social capital, however. Indies, by definition, are not embedded in institutions or bureaucracies. Lacking, and

even disavowing, such resources, indies turn to their social community instead. Knowledge sharing is crucial in the Cocoa community because the community provides intellectual resources and social support for indies who otherwise would be completely isolated. As discussed earlier, the deep social connections among Seattle Cocoa indies, which provides the context for collegial gifting of intellectual property amongst friends, is likely a crucial reason for the success of so many Seattle Mac developers. Despite the indie rhetoric of "independence," which stresses the idea that every developer makes it on his (and the indie identity is largely masculine, as we have seen) own merit, indies rely on the resources of their community and its central knowledge sharing activity.

Openness serves ideological purposes as well. Aligning themselves more closely to the openness of the hacker ethic over corporate proprietary values may also allow indies to proclaim that their identities are programmers first, entrepreneurs/businesspeople second. It allows them to maintain their stance that they are in it for the pleasure of programming, rather than pure profit. "Giving back" to the community via sharing their knowledge allows them to participate in community and identity building—while simultaneously building up reputation, which itself has economic rewards, in terms of greater opportunities for employment, or the selling of books or training courses, or at the very least, increasing the likelihood of someone else sharing their own knowledge with them, thus improving their own skills. It also serves to help differentiate themselves from the enemy, Microsoft. Joseph DeCarlo, a former Windows and now iOS programmer, explained that Microsoft Windows programmers typically hoard information or sell code libraries for a price, rather than freely sharing it, and attend conferences sponsored by Microsoft purely for professional gain, not to make friends or build community. (Joseph DeCarlo Interview, May 27, 2012) Ironically, this openness puts the Cocoa community at odds not only with the norms of Windows programmers, but with Apple itself.

Returning to Lave and Wenger's notion of communities of practice helps explain the tensions between elitism and populism in the attitudes of Cocoa oldtimers

in regard to newcomers. On the one hand, the survival and growth of the community and its knowledge of practice depends on sharing this knowledge with newcomers and welcoming them as legitimately participating peripheral members. However, to the extent that these newcomers bring with them different or new practices that may challenge the old orthodoxy, this causes conflicts with oldtimers who conservatively wish to preserve their existing practices as is. In the language of Bijker's technological frames, as newcomers move more centrally into the technological frame, from low to high inclusion, they subtly shift the frame, something which oldtimers try to guard against. We will see an example of this in the next chapter with the debate over dot notation in Objective-C. Experts in the Cocoa community try to maintain and reproduce among newcomers a techno-cultural frame involving not only knowledge about technical practice but normative values about that practice. This involves not just practice but issues of identity as well. In the following chapter, we will look in depth at the kinds of normative "best practices" advocated by Cocoa developers, and the reasons for their extremely strong opinions regarding them.

The Cocoa Community's relationship to Apple

This chapter up till this point has been focused on describing the social organization of the Cocoa developer community. However, we have not focused much on the role of Apple in relation to this community, except where WWDC was discussed. We have seen how Apple uses WWDC to enroll third party developers in its ideology and to proselytize the normative practices it wants developers to use. Core members of the Cocoa developer community tend value the social aspect of WWDC much more than learning technical material from its talks, and are more cynical of Apple's corporate messages. Nevertheless, these core members, while often critical of Apple, are also the most highly included in the techno-cultural frame of Cocoa development, which is significantly shared between Apple and the third party Cocoa developer community. We saw in chapter 1 that the most committed Apple developers, those indies who form the core of the Cocoa community, have bought into an ideology in which they see themselves as engaged in the same mission as Apple, and see the company as a partner. Unlike most of the johnny-

come-lately iPhone developers who may be more motivated economically, indie Mac developers are motivated ideologically. They do not need to be proselytized to at WWDC because they are already die-hard fanatics. They agree with Apple that usability and aesthetics of applications are important priorities, and they have the Apple Design Awards to prove it. These developers are not only dependent on Apple for their livelihoods but chose to become so dependent because they are Apple fans, and share similar values with what Apple stands for. Mike Lee put it plainly:

> The culture of the users and the aesthetic… comes from Apple. If there was no Apple, there would be no Apple fanboys. Certainly, there's kind of a culture and whatnot that have evolved separate from Apple, but ultimately that's where we take our cues.
>
> (Mike Lee, Interview, June 23, 2008)

Certainly, most of these men self-selected to become Apple developers, and understand that the ultimate source of the technology they love comes from Apple the company. However, there is more to tie these developers to Apple than fanboyism, shared values or shared economic interest in the success of the platform. Just as the community itself coheres because of the real social relationships that have formed amongst its members, the community remains close to Apple because it has social ties to individuals at the company, formed over many years.

For some Cocoa developers, these ties go back to the NeXT era. After NeXT's closure of its hardware business caused an exodus of developers from its platform, NeXT went out of its way to cultivate relationships with the loyalists who remained. The NeXT developer mailing list allowed the small population of NeXT developers direct access to the very NeXT engineers who wrote the frameworks they were using, and provided a way for them to understand how NeXT APIs worked despite them being closed source. NeXT developer conferences provided a way for developers to personally meet with NeXT engineers and form personal bonds.

> NeXT was so tiny, the engineers knew us. So we went to NeXT conferences… all the engineers were on the floor. You could walk up to them, "Hey AppKit guy tell me about this—" [and] they would come and they'd talk at the local user group meeting. William

> Parkhurst [one of the early AppKit managers] would come and talk about what's coming on the next version of NeXTSTEP. He was like, next up for us is going to be this and this. Can you imagine this today? Like the guy who is running the whole group, coming to a user group and saying here's what we're doing for 4.0, blah, blah, blah. You know, it was a really close-knit community and we knew, like there's a tiny group of engineers that had done NeXTSTEP and there's a tiny group of engineers outside of it who is [the] indie community who had contributed to it, and we were all doing these professional apps that looked better than anything else on the market, [written by] one or two… guys.
>
> (Wil Shipley Interview, April 18, 2012)

Similarly, like Wil Shipley, Andrew Stone was able to cultivate a personal relationship with the original AppKit engineers at NeXT conferences:

> So it's really cool from a sort of NeXT fanboy thing, is that you know who these people are personally [because you see their names in the source code header files], and then you want to meet 'em! And then you do meet 'em! And then you see 'em once or twice a year and it's just a great thing because when you have a problem, you can go to these guys, kind of not necessarily through official channels, because you're not big enough. You're an indie.
>
> (Andrew Stone, Interview, June 7, 2011)

As we discussed earlier, independent developers rely much more on personal social relationships than on institutional or bureaucratic resources, and this pattern also applies when it comes to Apple. As we see from Andrew Stone, because he had a personal relationship with NeXT personnel, he was able to circumvent official corporate bureaucracy. Because NeXT was a fairly small company, and because it was so important not to lose developers, it was quite easy for third party NeXT developers to form such relationships in the mid-1990s.

For men like Shipley and Stone, these bonds continued on to a certain extent after the Apple acquisition, and they were rewarded for their loyalty. For example, at Macworld Expo 1997, the first conference where the new acquisition and Apple's new operating systems strategy was announced, Steve Jobs invited Andrew Stone up on stage to demonstrate how quickly he was able to modify his NeXT-based graphics application, Create, to run on "Rhapsody," the code name for the NeXTSTEP-based

operating system that would eventually become Mac OS X. Another example of the special status of these small NeXT developers comes from my personal experience as the Quality Assurance (Test) Engineer for the Cocoa Framework Group (which was responsible for the AppKit and Foundation frameworks) at Apple from 2000-2005. At the time of the initial release of Mac OS X in March of 2001, there were very few third party Cocoa applications to test, and the two applications that I was told to include in my test suite were Andrew Stone's Create and OmniGroup's OmniWeb web browser. In addition, one day, my manager, Ali Ozer, had a special task for me. Andrew Stone was running into a bug in Create that he needed some help to reproduce and track down. I was assigned to personally help him discover the circumstances for the bug, a task Stone was so grateful for that he gave me an acknowledgement in the application's credits. I was never asked to do this for any other third party developer. That a third party developer would be given such exclusive, direct access to Apple personnel speaks to the status that Stone's loyalty to NeXT over the years had earned the influential former NeXT managers who ran Apple.

Without such special access, developers must go through the "official" bureaucratic channels for help, which means dealing with Apple's Developer Technical Support group (DTS). DTS is dedicated to helping third party developers resolve issues with their applications. This support costs money, but can be critical to helping developers fix bugs, support new features, or avoid App Store rejection. Getting help involves opening up a "DTS Incident," which is like a technical support incident for programmer issues. Developers who pay the $99 a year for membership in Apple's developer program get at least one incident for free, with additional incidents costing extra. DTS support staff are knowledgeable engineers, but are not the original authors of the code; as such, with more advanced or arcane questions, they may query the engineer who wrote the code for a more in-depth answer. Thus normally, third party developers are not given direct access to the engineers actually writing the code in OS X or iOS, because these engineers are too busy developing

the next version of the OS to help answer every developer's questions. That is what DTS engineers are for.

Some developers, such as OmniGroup, become important enough that DTS assigns a support engineer specifically to handle the relationship with that company. Daniel Pasco and Bill Moorhead of Black Pixel spoke of this process as "getting an angel." Not only does this help out significantly with technical support issues, but it can lead to Apple directing business towards the indie company.

> There's a point at which you get significant enough that somebody at Apple is kind of informally assigned to… reach out to you once in a while and tell you about developer kitchens or something like that.
>
> And also to tell you if you're screwing up, or… just to be there for help. But we work with a lot of partnership managers at Apple. Frequently, because for one thing, we keep running into them when we're seeing different customers, so we're, like, doing a project and there's the same partnership manager again. It's the Black Pixel guys again. And so then they start saying, wow, we see you guys all the time and we know what you've shipped and even if you can't talk about it—
>
> So when somebody else comes to them and says we're looking for a team, they'll say, well, check out Black Pixel.
>
> That's an angel at Apple. And… you reach a point where you're kind of getting established enough that somebody at Apple at some point, makes a point of getting to know you, like whether it's at Dub-Dub [WWDC] or something like that.
>
> (Daniel Pasco Interview, March 28, 2012)

As Pasco notes, not all of the "angel" relationship is formal. What begins as a purely bureaucratic relationship can, over time, grow into an informal social relationship. This social relationship can yield advantages for longtime developers that newcomers simply do not have.

Another way that indie developers can become known to Apple is by writing an exemplary application that Apple may decide to feature with a Design Award in other to encourage others to emulate them. Dan Wood's Watson, an application that aggregated multiple web services into a single interface, was one of these. When I

was part of the Cocoa framework team, my manager Ali and my colleagues pointed out Watson to me as an example of an innovative Cocoa app that I should make sure to test.

When the Cocoa world was still small, it was relatively easy for up and coming indies to make personal connections with Apple personnel simply by attending conferences and workshops. Dan Wood says that this was how he became personally acquainted with the AppKit team:

> I actually think there's an advantage to WWDC being small… the chance to talk to Apple engineers and I've had other opportunities to go to other small Apple events where there are engineers there, they call them workshops or they used to call them "kitchens" or stuff like that. I mean, just having a personal correspondence with members of the AppKit… or WebKit team, or whatever. Like, I know half the people on the WebKit team, just because we've kind of gotten to know each other over the years and they know we're a big user of their technology, and I go to their parties every year and so it's fun to see them.
>
> …Sometimes I'll go down to Cupertino and have lunch with the WebKit team… and just kind of see what comes up and meet people I've never met before and talk about something… And all the engineers are really good at knowing what they're not supposed to talk about and it's not like you're getting anything—there's no real advantage, except just getting a little help and time and perspective on things and stuff like that.
>
> …If they think you're going to be spamming them, they're not going to be making their contact information available to you in the first place, so there still has to be some kind of personal connection I think. Which again, if you have millions of iOS developers out there, there's too many out there to make that small personal connection. So it is nice to have—it's nice to be part of that inner, sort of inner community where there's developers and Apple employees and we're all just people. The Apple employees, there's the official corporate wall, but there's just people in there, so.
>
> (Dan Wood Interview, April 9, 2012)

As Wood notes, personal relationships with Apple engineers gain special access to third party developers who make sure they do not abuse their special privileges. This applies to not asking for or expecting to receive secret information

about upcoming products or technologies, which Apple engineers are constantly reminded not to divulge, even to fellow Apple employees who have no need to know, in order to prevent leaks. By not spamming these engineers with questions or expecting secret information, third party developers build trust with Apple engineers. Although Wood says that there's no real advantage, the social relationship itself creates opportunities for informal interaction, such as over lunch, through which information of a less classified, but still important, nature, might be passed on—say, a workaround for a tricky bug that might be difficult to discover on one's own. Moreover, informal interactions allow third party developers to find out Apple engineers' individual opinions about various issues separately from Apple's official stance.

Daniel Pasco of Black Pixel believes that all of the "superstars" in the Cocoa community have constant communication with Apple personnel that has led to building direct, personal relationships:

> [Apple], as an entity that you would interact with, maybe, directly, multiple times a month. Whether you are filing Radars [bug reports] or talking to people that actually work at Apple and that sort of thing. But, one thing I would say is that almost everybody in that Who's Who routinely has one-on-one conversations with people at Apple. I would almost bet my life on it.
>
> (Daniel Pasco Interview, March 28, 2012)

This constant communication with Apple, building up a positive relationship with specific Apple personnel, differentiates the indie developers who are central to the community from the majority of those at the periphery.

> And a lot of these other people don't. They don't have tight connections at Apple. …There's a lot of the people that we would kind of self-select for saying that person's part of our group [Xcoders], are people that have established tight contacts with people at Apple. Like they have a good working relationship with them.
>
> (Daniel Pasco Interview, March 28, 2012)

These sorts of personal connections also lead to a phenomenon that ties the third party community and Apple even tighter together. Just as there exists a

revolving door between the Federal government and the industries it regulates, there is a revolving door between Apple and the indie developer community. In the mid-2000s, a number of employees of Wil Shipley's Delicious Monster were poached by Apple. On the flip side, Mike Jurewitz, a longtime "evangelist" in Apple's Worldwide Developer Relations (WWDR) department who had devoted a significant portion of his career to building connections with the developer community, left Apple to join the indie company Black Pixel in Seattle. Other prominent indies who were once Apple engineers include Daniel Jalkut, Matt Drance, and James Dempsey. The tight social connections between indies and Apple is one reason for the close identification that indies have with Apple and their sense of a shared, collective mission with the company, as the social boundaries between the company and the indie community are somewhat porous. These connections also mean that, for indies, Apple is not just a faceless corporation but a place where likeminded individuals, even friends, can serve as advocates for them and their concerns within the company. In a sense, the Cocoa community exists both inside and outside Apple. This makes perfect sense if we remember that Cocoa community is a community of practice. According to Brown and Duguid, technical knowledge travels easily between firms precisely because the community of practice is composed of social networks that transcend organizational boundaries (Brown and Duguid 2001, 206). In the case of Apple, not only technical knowledge but also ideological, normative, and affective commitments also travel into and out of Apple itself. This explains why core Cocoa developers are so committed to the same ideology as Apple itself—some of the Cocoa community is in some sense an Apple diaspora, and continue to spread its values even without being employed by the company.

This does not mean that the relationship between Apple and its developers is free of tension or conflict, however. Although Apple and its developers depend on each other in a symbiotic relationship, this relationship is nonetheless highly asymmetric, which is inevitable when one party happens to be the largest company in the world. As the vastly more powerful actor, Apple is not only able to unilaterally make changes to its technologies, it is also able to make changes to its policies in

dealing with third party developers. Occasionally such moves are seen by developers to be harmful enough to engender significant resistance. For example, when the iPhone SDK was first made available in a prerelease version for third parties to write apps in March 2008, Apple required developers to sign a restrictive NDA that prohibited them from discussing any details of the SDK in public media such as blogs. This made it very difficult for developers to share knowledge about how to make iPhone apps at a time where very little was known and no one outside Apple had any experience. Developers wanted to be able to help each other work out the difficulties of writing apps using a version of Apple's iOS libraries that still had considerable bugs. The NDA prevented authors like Aaron Hillegass from writing books teaching iPhone development, and even placed a gag order on discussion of iPhone SDK details at Cocoa conferences not sponsored by Apple. (Foresman 2008a; Foresman 2008b) Moreover, Apple's policy violated the norms of the Cocoa community, which valued the open sharing of knowledge. A Twitter meme, FuckingNDA, was started by Craig Hockenberry, developer of the Twitter client Twitterific (Hockenberry 2008). Even when the App Store went live in July of 2008, allowing developers to sell their apps on the App Store, the NDA stayed in place, which prevented developers from comparing notes on why their apps were being rejected. After significant outcry, the NDA was finally lifted in October. Given Apple's corporate opacity, it is difficult to discern if plans to lift the NDA had been in place from the beginning, or if the developer resistance and its discussion in the tech media played a role in getting Apple to change its policy.

Another area of tension surrounded the decision by Apple in 2011 and 2012 to require Mac OS X applications sold on the Mac App Store to submit to a new security policy called "sandboxing," which restricted applications' access to files on the user's hard drive, as well as other systemwide access. Apps under this scheme can only read and write files into a "sandbox" area within the application itself. For example, on the iPhone, all documents created by an app, say, notes taken by the Notes app, are sequestered inside the Notes app itself. This makes it difficult or impossible for different apps to modify the same document, and files cannot be

easily transferred by the user. This policy had been in place on the iPhone App Store when it launched, as Apple could reasonably argue that on a mobile phone, security needed to be heightened by preventing third party apps from writing files across the file system, a typical attack vector of viruses and malware. However, on the Macintosh desktop platform, applications had always been able to freely write to locations on the user's hard drive outside of the application. Imagine if, for example, you could not save a Word document to any folder you wanted, but instead Word documents were saved to an opaque location inside the Word application itself, preventing you from moving it or copying it. In such an environment, the only way for the user to back up the file would be for the application to provide built-in synchronization over the Internet, using cloud-based services such as Apple's iCloud.

Sandboxing on the Mac platform became extremely controversial because some kinds of applications, such as utilities, which customized behavior across the operating system, or needed certain access to hardware, could no longer function under the security restrictions of the sandboxing regime. Developers who sold apps through their own channels, rather than on the App Store, were not subject to the restriction, but some features, such as Apple's iCloud internet synchronization, could only be used with apps sold on the App Store, leading many developers to fear that apps not sold on the App Store would become second-class citizens, or that at some point, Apple might even disallow them. One longtime Apple journalist, Andy Ihnatko, argued that sandboxing's restrictions would disempower users and erode the Macintosh's identity as the premier platform for creative content producers. (Ihnatko 2011) Moreover, technical difficulties with Apple's sandboxing APIs made it difficult for developers who tried to comply to actually do so by the deadline Apple imposed. Apple ended up postponing the deadline multiple times, from November 2011 to March 2012, and again to June 2012 (Caldwell 2012). Things got so bad that some developers publicly announced that they were no longer going to be distributing their apps through the App Store, as sandboxing would cripple their apps (Postbox 2012; Atlassian Blogs 2012).

Outspoken indie Cocoa developer Wil Shipley suggested that sandboxing was not only draconian, but the wrong solution to the problem of security on the Mac desktop platform. On his blog, he suggested an alternate solution: require all Mac developers to obtain from Apple a security certificate, a file that could prove that they were who they said they were, and that the software they were distributing came from them and was not modified by another party along the way. As long as Apple was the gate keeper for developer security certificates, if any developer created malware, Apple could revoke their certificate and alter the Mac OS X operating system to refuse to run software without certificates (Shipley 2011). By the next February, Apple announced that the next version of Mac OS X, which was 10.8 Mountain Lion, would introduce just such a feature, to be called Gatekeeper. Again, while it is possible that Apple came up with the feature independently, it is also possible that Shipley's blog post suggested to engineers at Apple that such a solution was the right one. Certainly, the controversy over sandboxing in the Mac App Store was alienating prominent Mac developers, and whether or not Shipley's blog was the source of the Gatekeeper idea, nevertheless it was still most likely a response by Apple to mollify developers.

Another tension in the relationship between developers and Apple was described by Michiel Van Meeteren: the phenomenon known as "Sherlocking" (van Meeteren 2008, 60). Sherlocking is a verb that refers to what happens when a third party developer's application is made obsolete when Apple releases an application of its own, often bundled with the OS X or iOS operating system, that does the same thing. The third party application is at a competitive disadvantage because Apple can tightly couple its version to its operating system, and additionally, if it bundles it, it is free, while the third party app is not. While Microsoft got into trouble with the Justice Department for similar bundling practices with the Internet Explorer web browser, Apple's low PC marketshare keeps it off the anti-trust radar, and it engages in this kind of anticompetitive, eat-its-own-children behavior frequently. The term "to Sherlock" specifically refers to an Apple application named Sherlock that was bundled with both Mac OS 9 and Mac OS X. Originally, Sherlock had been an

application that performed Internet searches on multiple search engines at once, at a time before Google became the dominant player in web search. It also performed local file system searches based on indexing of files' contents, before such functionality was subsumed by the Spotlight feature in Mac OS X 10.4 Tiger. Indie developer Dan Wood created an application that hosted, in a single place, a collection of widgets or modules that interfaced with web services, such as movie times, flights, and recipes. (Today, such functions might be provided on the iPhone by an app specialized for each purpose.) Playing off the name of Sherlock, Wood named his application "Watson." Wood provided an API that allowed other developers or sophisticated users to create their own widgets, opening Watson up to user-created modules. Watson was so innovative and well designed that Apple gave Wood an Apple Design Award for it in 2002. As we discussed earlier, it was highly regarded by my group, the Cocoa framework group, within Apple. However, in that same year, Apple released Sherlock 3, bundled with Mac OS X 10.2 Jaguar. Sherlock 3 eliminated the web and file content search functionality of the original Sherlock, instead becoming a virtual clone of Wood's Watson application. Wood, unable to compete, eventually sold the technology behind Watson to Sun Microsystems (Karelia Software 2005).

This incident caused wide sympathy for Wood in the indie developer community, and the term "Sherlock" became a verb referencing this episode. Since then, a number of other third party applications have been copied and put out of business by Apple, including an application that provided similar web-service widgets on the user's desktop, Konfabulator (Gruber 2004). Even in 2014, new features of iOS have made numerous third party apps obsolete (Tabini 2014). The ire that Apple provoked among indies was its blatant copying of their ideas. Cocoa developers took to their blogs to excoriate Apple's actions, but to no avail. Because the indie developers who were harmed were single individuals, they would not be able to afford an expensive lawsuit against Apple, which they would have to face alone. The indie community is not a labor union with collective bargaining power. Apple's power over indies is so asymmetric in this regard that it can run completely

roughshod over its biggest supporters and still mostly get away with it, because as devoted as they are, these indies are likely to still continue developing for Apple's platform. Even if they don't, Apple doesn't seem to care about one or two individuals who leave for more equitable pastures: Konfabulator's developer abandoned the Mac and ported the software to Windows. Dan Wood sold Watson to Sun and worked there for a time. However, eventually Wood left Sun and returned to Mac development with a new application, the graphical website tool Sandvox. Those developers who loudly proclaim that they would rather be sheep farmers than develop for any platform other than Apple's do not really have a choice to move on, even if they get Sherlocked. Many Cocoa developers like Gus Mueller simply acknowledge that Sherlocking is simply an acknowledged risk and a fact of life if one wants to hitch their wagon to Apple. Mueller feels that the proper way to respond to Sherlocking is not to get angry, but to simply be prepared; developers should diversify into more than one application, like he has, so that if one of them is Sherlocked by Apple, he can still make a living off the other. Another way is to be strategic and try to avoid making apps that are likely to get Sherlocked—applications like Mueller's Acorn, which is a graphic editor, are unlikely to be targets of Sherlocking, while utilities or system enhancements to the operating system, which could easily be functionality incorporated or bundled into the OS, are simply asking for it. For his part, Dan Wood, although initially upset over the Watson affair, has since made peace with what happened. His story provides the example for the right course of action: if Apple steals your app's idea, be creative enough to make another one.

Apple's actions are not always harmful to developers, however. In some controversies, third party developers actively seek to enroll Apple's support on their side. In 2011, the iOS developer community was concerned over legal action by Lodasys, which was suing indie developers, who could not afford to fight expensive legal battles, for infringing on its patent for purchases made from inside an application (In-App Purchase), a facility made available by Apple that Apple had already paid to license. (Such companies, which make their money mostly from

suing other companies for infringing on the patents they hold, have become known as "patent trolls.") In the face of this, developer Mike Lee organized a defense fund for indies (Cheng 2011; Lee 2011). Initially, it seemed that Apple would stand on the sidelines leaving its own developers to fend for themselves, which would only hurt Apple as it would prevent those developers from contributing to Apple's App Store. Developers called for Apple to intervene legally on their behalf, and eventually Apple stepped up (Foresman 2011).

To conclude, Apple's relationship with its developer community is complex. Developers and Apple need each other in this symbiotic relationship. Developers are highly completely dependent on Apple, which provides them with a platform, the primary development tools they use, a large potential user base, and now with the App Store, their electronic distribution and payment infrastructure to reach those users. Apple clearly benefits from having a lot of third party developers; both the NeXT and the Mac platforms' relative marginality compared to Windows is associated with a dearth of available software, while iOS's dominance over Windows Mobile is largely due to its command over mobile developers. The network effect applies: lack of software creates a vicious cycle where users decide to go where there is more software, leading developers to go where there are more users. Nonetheless, the more developers Apple has, the less power any individual developer has vis-à-vis Apple. Many Cocoa developers have complained that after the expansion of the community post-iPhone, Apple no longer cares about them. The fact that many of them can no longer get tickets to WWDC anymore might be indicative of this—or rather, they are no longer special, with Apple now trying to cater to the masses of new developers flocking to the iOS platform. Indies do understand that, Apple being a mega-corporation, it pursues what is in its interest first, and sometimes this means that it pursues short-sighted policies that hurts developers. Nevertheless, indies feel that Apple would be smarter to keep its developers happy for its platform to thrive. This is certainly self-interested, but it does not negate the fact that Apple benefits if its developers do well, and developers do not hesitate to point this out.

The Apple developer community, of course, is not monolithic, as we have seen. The inner core, consisting of indie Mac developers who were devoted to Apple long before the iPhone, experienced the Apple world in which they could not, and did not, hope to sell out for billions of dollars. Although they still control the discourse through their blogs, may have special access to Apple personnel, and thus can have considerably more leverage over Apple's actions than any random individual, Apple cannot ignore the masses of new developers. This can lead to policies or technological developments that existing Cocoa developers, including many who were former Apple employees, feel is contrary to the spirit of how things on the platform should be done. In the next chapter, we will examine a technical controversy over a change that Apple made to the syntax of its Objective-C language in 2007. This controversy pitted oldtimers in the Cocoa community against both newcomers and Apple itself, and due to Apple's power and the much greater masses of newcomers, it is a debate that the oldtimers have largely lost.

Chapter 6: The Dot Notation Controversy

As discussed in previous chapters, the Cocoa developer community is a community of practice committed to the self-cultivation of their craft, and to the techno-cultural frame associated with writing software using Cocoa technology. Part of this frame involves norms of practice for being a "good" Cocoa programmer. Within the community, the actors' term for this is "best practices."

The proper use of "best practices" mark a Cocoa programmer as an experienced member of the community, with high inclusion in the techno-cultural frame surrounding Cocoa. Because best practices are normative, they also figure strongly in the boundary work that demarcates Cocoa insiders from novices and outsiders. Such practices, however, are not static, but change over time, often due to changes Apple makes in the tools and technologies Cocoa programmers use. Such moments of change can cause controversies in the community, as older members may variously welcome or resist Apple's purported "innovations" to the way they do things. Such controversies take place in the various physical and online forums of the community: blog posts, conference talks, and debates at local club after-parties. These discussions can become highly emotionally charged, and reveal what is at stake in the maintenance of normative best practices. In this chapter, I will examine a particular controversy that took place from 2009 to 2011 over a feature Apple introduced into the Objective-C programming language, called "dot notation," which was a new way of writing Objective-C code that looked more like code written in other C-based languages like Java or C++.

How does the community encourage the use of best practices? Some Cocoa developers work for larger companies such as Google, which can enforce particular practices. However, many Cocoa developers are indies, who work for themselves. Moreover, Cocoa developers advocate best practices to others regardless of company affiliation. The means by which Cocoa developers enforce the use of best practices is not primarily coercive but normative. Cocoa developers follow best practices not merely because they are following managerial or institutional directives (although

some may be), but also because they face moral disapproval from their peers in the community. The proper use of best practices in the community is thus seen not as a discipline enforced upon them by management or even by Apple, but as a form of self or peer discipline for the purposes of improvement in one's craft. Moral policing of best practices by the community can be especially keenly felt by newcomers, who often commit transgressions of the norms during the learning process. Over time, such norms become internalized as the newcomer assimilates and moves from low to high inclusion.

The stakes of the Dot notation controversy

To illustrate the importance best practices have for Cocoa developers, and the normative and social stakes involved, I will turn to a controversy that occurred in the community from 2007 through about 2012. Studies of controversies and contestation have been a central methodological tool in Science and Technology Studies for examining moments in which scientific or technological consensus has not yet been achieved, when the role of social, cultural, and political factors, normally hidden or "black boxed," can be openly observed shaping the interpretations of scientists over what "counts" as a discovery. A period of open "interpretive flexibility" is followed by a process of "closure" in which consensus forms around a particular interpretation of data; such processes are shown to be social in nature. This was a fundamental tool of the Sociology of Scientific Knowledge (SSK), especially that of Harry Collins and Trevor Pinch of the Bath school (Collins and Pinch 1998b; Collins 1981a; Collins 1981b; Collins 1985; Pinch 1986), and was extended to technology by Trevor Pinch and Wiebe Bijker with the Social Construction of Technology (SCOT) heuristic (Bijker 1995; Pinch and Bijker 1984), and by Bruno Latour and Michel Callon with Actor-Network Theory (ANT) (Callon 1986; Latour 1987). Donna Haraway has separately made contestation an important method in feminist science studies (Haraway 1989). Scientific or technical controversies are often resolved through demarcating who gets to decide; thus controversies often involve "boundary work" around who is, or is not, a valid member of the expert community (Gieryn 1983).

In the dot notation controversy we will explore in a moment, the normative commitments of the Cocoa community are in flux and open to debate, and serve to bring into relief the ethical stakes of otherwise mundane practice. This controversy occurred during the iPhone gold rush, a period of enormous expansion in the community of Cocoa programmers. Existing members of the Cocoa community were decrying the decreased quality of applications available through Apple's App Store, attributing this to the influx of programmers from other platforms who did not share their values in making "quality" software. With the older core of the community feeling under siege by a group one informant called "carpetbaggers," at stake was the future character of the community: could the newcomers be taught the right values and practices and be properly acculturated into the existing community, or would they take over? Pedagogical questions arose. What constituted the "right values" and "right practices"? For Cocoa developers, these came to be articulated in terms that would facilitate the making of "quality" software for users. As a result, the debate over dot notation became conflated with concerns over the boundaries of the community and the identity of its members. The older practice marked practitioners as old timers, who attempted to mark the new practice as illegitimate, correspondingly marking its adherents as outsiders or newcomers.

As we saw in chapter 1, "quality" can mean several things for Cocoa developers. For some longtime developers, it means "usability," or "user-friendliness," having a well-designed, easy to use human interface. Sometimes conflated with this is "aesthetics" or "design"—having beautiful graphics, pleasant or friendly-looking animations, which give the user pleasurable feelings when using the app. For others, "quality" can mean attention to what the app does—its functionality, which for many means its utility, how useful it is. This can also become conflated with "usability" as the design of the user interface can directly affect its functionality. These definitions of quality all have to do with aspects of the application at the level of the idea, the design, the business plan, or marketing—which could be separate from the actual implementation of the app in code, although for indie entrepreneurs, who must be jacks of all trades, all of these tasks are part of

the job description of an independent app developer. On a technical level, however, software quality often takes on the meaning of whether the software works as advertised, whether it is stable, and does not have too many bugs. In the software industry, the job description "quality assurance" usually refers to a software tester, whose job it is to find bugs and describe under what circumstances the bug occurs so that software developers can fix them. "Quality" from a software engineering perspective, then, often comes to refer to the "reliability" of software.

As we saw in chapter 2, concerns over software reliability and risk created the impression by the late 1960s of a "software crisis." Increasing reliance on software infrastructures, coupled with widely publicized failures of large software systems such as IBM's OS/360, generated fears that existing programming techniques or skills were inadequate. Even today, big failures such as the problems with the rollout of the HealthCare.gov website in 2013 generate concern. To combat the purported crisis, "software engineering" techniques and methodologies were created as a response, to discipline programming practice to improve its reliability. Janet Abbate argues that "software engineering" never became much more than a slogan, an umbrella term for a smorgasbord of different (sometimes conflicting) techniques, and primarily a metaphor to boost the professional and social status of programmers (masculinizing it in the process) by subjecting it to quantitative and systematic rigor. Ensmenger, Mahoney, and Slayton show that software engineering took on managerial overtones, and many techniques that fell under the umbrella were appropriated for managerial (which were proxies for corporate, government, or military) interests. (Ensmenger 2010; Mahoney 1990; Mahoney 2002; Slayton 2013b) Sociologist Philip Kraft warned about the deskilling of programming by software engineering techniques. (Kraft 1977) Despite this warning, software engineering techniques were advocated by and taken up by programmers. "Software engineering advocates like [Edsgar] Dijkstra and [Fred] Brooks identified as programmers themselves." (Abbate 2012, 107) "Rather than resisting structured methods, some programmers adopted them without any management pressure to do so, because they saw them as skill-enhancing, not skill-reducing." (MacKenzie 2001,

40) For this reason, “Rather than making programmers obsolete, software engineering methods became simply another skill that programmers could claim,” increasing their value as professionals in the job market (Abbate 2012, 108). Disciplinary practices enhanced rather than eroded programmers’ capabilities, and became part of the professional identity of programmers.

Crucial to making software more reliable, Ensmenger argues, is making it easier to maintain. The issue is that software is a socio-technical system in which social and organizational relations become objectified in the materiality of code, which becomes obdurate and acquires its own technological momentum, according to Ensmenger. “Software is history, organization, and social relationships made tangible.” (Ensmenger 2010, 227) Yet the social and technological environment in which software is embedded is constantly changing, and software must change with it, but the facility of making these changes is constrained by the existing software’s design. Moreover, because even in its initial construction, software projects are often moving targets, in which specifications are changing, the line between maintenance and production of software is rather blurry.

New techniques for disciplinary software practices became embroiled in controversy at the very beginning of software engineering. As recounted by Donald MacKenzie, computer scientist Edsgar Dijkstra’s 1968 article, “Go To Statement Considered Harmful,” (Dijkstra 1968) was a polemic that generated a firestorm among computer programmers of the time. Many procedural programming languages in the 1960s included an instruction named “`go to`” (or “`goto`”) that allowed a program to arbitrarily jump from one line of code to another. In practice, use of `goto` often resulted in confusing, tangled “spaghetti code” that was difficult to read and understand, making it error-prone, hard to modify and fix, and thus less maintainable. (In fact, even as recently as February 2014, a security bug in iOS and Mac OS X was caused in part by the use of a `goto` statement in C code.) (Poulsen 2014) Dijsktra was the leading exponent of a methodology called “structured programming” that quickly became associated with software engineering. Structured programming banned the use of `goto`, in favor of controlling program flow with

"structured" jump mechanisms, using conditional branches, loops, and procedure calls. Similar to the dot notation controversy, arguments about `goto` were not only normative, but came to be expressed as issues of code style and aesthetics. "To [Peter] Naur [a leading developer of the ALGOL language], excessive use of **go to** was inconsistent with the ideals of rigor and careful structuring that inspired the language: it was not 'good ALGOL style'; it was '*ugly*... *inelegant* and uneconomical.'" [bold in original, emphasis mine] (MacKenzie 2001, 38) The response from opponents similarly complained that Dijkstra was being dogmatic in the face of pragmatic reality: "John R. Rice of Purdue University wrote… that he was 'taken aback' by Dijkstra's '*emotional'* attack on 'an obviously useful and desirable statement.…How many poor, innocent, novice programmers will *feel guilty when their sinful use* of **go to** is flailed in this letter?'" [bold in original, emphasis mine] (MacKenzie 2001, 38–39) While the controversy over `goto` is possibly the most famous controversy over a programming practice, debates over such practices and "styles" are commonplace among programmers, who are often described as "religious" about their particular favorite language or platform. (Examples include Coleman's description of a Python hacker's animosity towards Perl (Coleman 2013, 95–98), or the animosity between users of the vi and emacs text editors). Their emotional responses to what seems like technical minutiae to outsiders matter because for programmers, such practices structure how they think, and consequently, what they can make their code do. Disciplinary, normative "best practices" become associated with an "idiom," with the ways of thinking and doing of a community of practitioners, and adherence or rejection of such practices become crucial to their sense of identity and membership in the community.

For Cocoa developers, an aspect of a skilled, experienced, professional software engineer is how well he or she designs and implements software so that it is flexible and easy to change and fix in the future; in other words, "maintainable." On a technical level, and on the level of the programmer's skill, software "quality" thus comes to mean "maintainability." Technical debates over particular norms and practices purported to promote "maintainability" take on larger stakes—whether one

is or is not a skilled, professional software developer. Those who did not follow the "right" practice were implied to not care about following professional standards of software engineering, making them subject to moral disapproval. Because in the Cocoa community, "maintainability" is part of a constellation of meanings under the larger rubric of "quality," not following practices promoting maintainable software can become conflated with other values marking a developer as "other"—such as prioritizing revenue generation over usability, aesthetics, utility, or reliability.

Dot notation explained

At WWDC 2006, Apple announced that it would update its Objective-C programming language to version 2.0, with several new features. One of these features introduced a new notation for writing code, known as "dot notation," that departed from traditional Objective-C notation while making the code look more like well-known languages such as Java.

For example, in the new dot notation, to access the visibility status of a button on the screen, and store the resulting value in a variable named *myVar*, a programmer would write:

```
myVar = button.isVisible;
```

In the older notation, known as "bracket notation," the code would be written in this way:

```
myVar = [button isVisible];
```

These two statements make use of different syntax within the Objective-C language. However, to the Objective-C compiler, the program that translates such high-level language statements into the ones and zeros of machine code that the computer natively executes, both statements instruct the computer to do the same thing—access an attribute of the *button* object and store the value of that attribute into the variable *myVar.* To the compiler, the two statements mean the same thing.

So what is the problem? The reaction to this new feature among oldtimers in the Cocoa community, the primary users of the Objective-C language, was mixed.

While some embraced the change, others reacted with hostility. Over the next few years, as newcomers flocked to Apple's iOS platform and learned Objective-C for the first time using the new notation, people such as Aaron Hillegass, who taught Objective-C and Cocoa to these newcomers, began to discover common sources of errors among novices who used dot notation. Things came to a head in 2009, when Joe Conway, a fellow instructor at Big Nerd Ranch and Hillegass' co-author on *iOS Programming: The Big Nerd Ranch Guide*, wrote in a blog post, "When I teach… I make sure to tell students never to use it ever, ever, ever again." Conway was using his position as a public authority on iOS programming to call for a ban on the use of dot notation, in contradiction to official Apple policy. One commenter to Conway's post agreed, calling dot notation an "abomination," language which Conway subsequently took up (Conway 2009). Across the Cocoa blogosphere, other blog posts emerged, both pro and con. The negative opinions became increasingly polarized, such as one example proclaiming dot notation to be "100% Pure Evil" (Reid 2012).

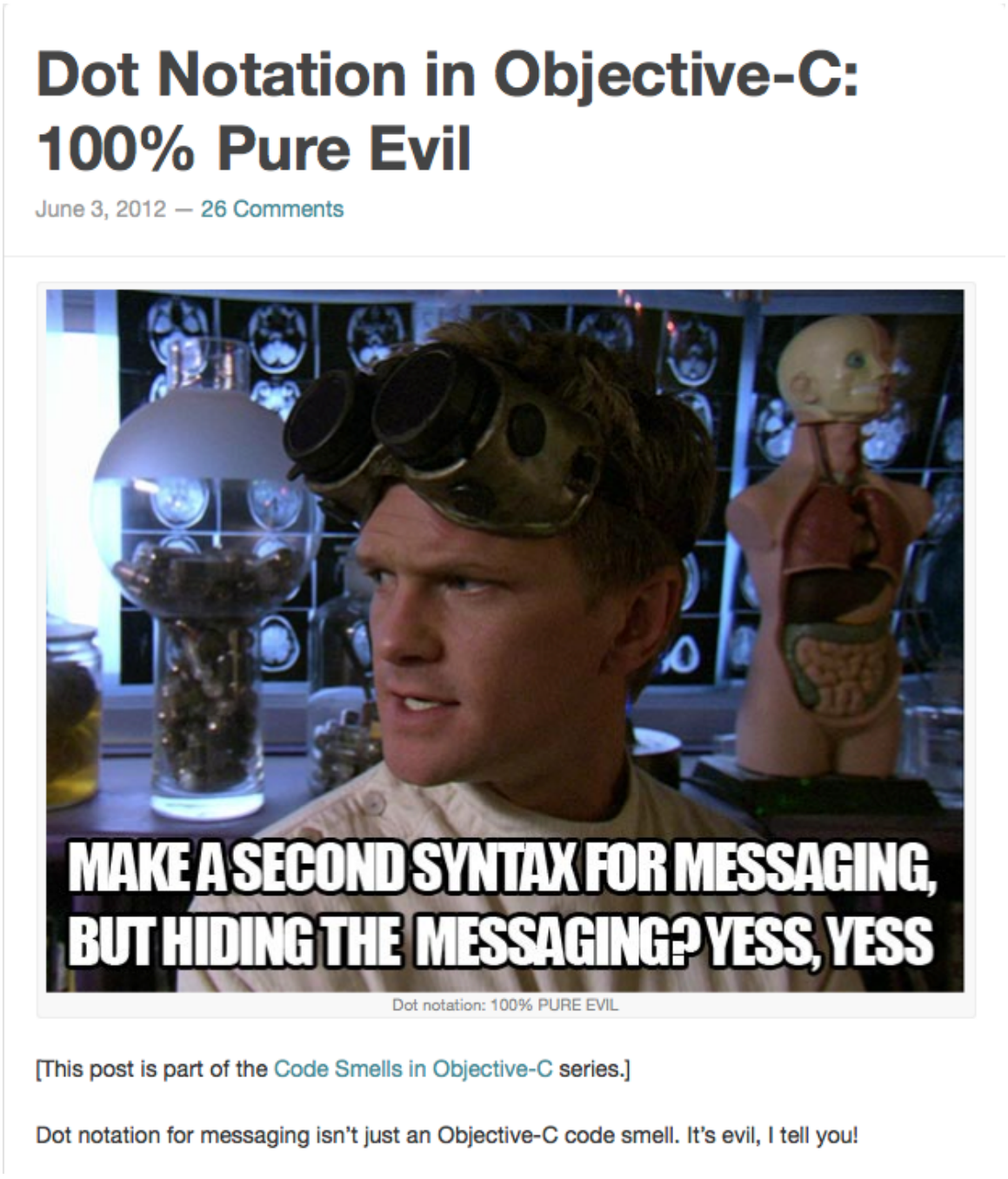


Figure 7: Blog denouncing dot notation.

"Dot Notation in Objective-C: 100% Pure Evil," John Reid, June 3, 2012, *Quality Coding*, accessed September 13, 2013, http://qualitycoding.org/dot-notation/.

To understand the reactions, we must return to the code. Code in a high level language (in other words, *source* code) is a text that always has at least two audiences—the computer (more specifically, the compiler), but also, the human

programmers who read and write the text. Human programmers are constrained by the syntax of the language the compiler accepts, which must be formal and unambiguous so that the compiler can straightforwardly translate the high-level source code into machine code.[40] Although, in the above example, the two statements are formally equivalent to the compiler, and thus generate the same machine code for the computer to run, to human readers of the code, they can mean different things. In other words, although to the computer, the two different syntaxes have the same semantic meaning, to many human readers, the two syntaxes can represent *different* semantic meanings, depending on the human's previous experience with various other programming languages. The code, which is unambiguous to the compiler, can be interpretively flexible to human readers.

Because the normative stakes of the dot notation controversy are rooted in its technical details, before going deeper, I need to first provide some technical background on Objective-C and object-oriented programming more generally.

Objective-C is an object-oriented programming language, like Java, Python, Ruby, Smalltalk, C++, and others. Programs in these languages are constructed by modeling things (called "objects" in these languages) and their relationships to each other. Most programming languages prior to this, called "procedural" languages, constructed programs as groups of processes or "procedures" that the computer runs. (Depending on the language, these might also be called either "functions" or "subroutines.") A grammatical metaphor sometimes used by programmers to describe this distinction is that object-oriented languages create programs in which "nouns" are primary, whereas in procedural languages, it is "verbs" which are primary (Yegge 2006). Procedural programs are often diagramed using flow charts, which depict the flow of execution of the program as it traverses conditional

---

[40] This is why it is called "source" code—it is the source of what the compiler uses to create "object" or "machine" code, or in other words, the low-level instructions handled directly by the hardware, encoded in ones and zeroes.

branches, loops, and subroutine calls. Object-oriented programs, instead, are diagramed by modeling objects, their relationships and communications with each other.

At the level of machine code, all programs are procedural—low-level assembly language programs (which are made up of human readable mnemonics that represent, in one to one correspondence, binary machine code instructions) that tell the computer to execute one instruction at a time, generally taking one or more data inputs and returning an output, such as adding two numbers. High level programming languages, of which all object-oriented languages are a subset, were devised beginning in the 1950s to allow programmers to express their intentions in ways more familiar to them, with syntax that either emulated mathematical notation (FORTRAN) or natural human languages, almost always English (COBOL). High-level languages also "shield the machine from the programmer" (Mahoney 2002, 96), providing a layer of abstraction between a programmer and the specifics of the particular machine (central processing unit or CPU) the program needs to run on. As long as compilers exist for various processors, a program written in a high level language is *portable,* meaning it can be quickly recompiled to run on different processor architectures. The first object-oriented language, Simula-67, arose from attempts to create a language for simulation, in which expressing a program by modeling objects which represented things in the real world was a more natural way to organize a program than as a collection of procedures for a computer to execute (Zepcevski 2012). At Xerox PARC in the 1970s, Alan Kay combined ideas from Simula with ideas from LISP, a very abstract language used in artificial intelligence, to create Smalltalk, initially intended to be a language and programming environment simple enough for children (Kay 1993). Kay coined the term "object-oriented programming," and the highly abstract and dynamic design of Smalltalk would influence a number of subsequent object-oriented languages, including Objective-C, Python, and Ruby.

Objective-C largely follows the Smalltalk model for thinking about programs. Here's an example. Let's say I want to model a dog. I have a particular dog 'object', which I can assign a name, "Fido."

Most object-oriented languages have separate concepts for the general category of things to which an object belongs (in other words, the general category of "Dogs") versus a particular instance of that category ("Fido"). Abstract categories of things are called "classes" while particular instances of that class are called "objects."

Dogs (or, rather, objects belonging to the class, "Dog") can have a number of attributes, such as its color, size, and whether he is awake or asleep. These attributes, called "instance variables" in Objective-C, are defined in the code for the Dog class.

If I want to know if Fido is awake, in Objective-C, I write this code:

```
[Fido isAwake];
```

to ask him if he is awake.

Dogs also can respond to commands such as "fetch," "roll over," or "play dead." These commands, called "methods" in Objective-C, are similarly defined in the code for the Dog class, and are analogous to "functions" or "procedures" in purely procedural languages. They represent the "verbs" that the "noun" "Dog" knows how to "do."

Thus, if I want to tell Fido to do something, I write the code:

```
[Fido fetch];
```

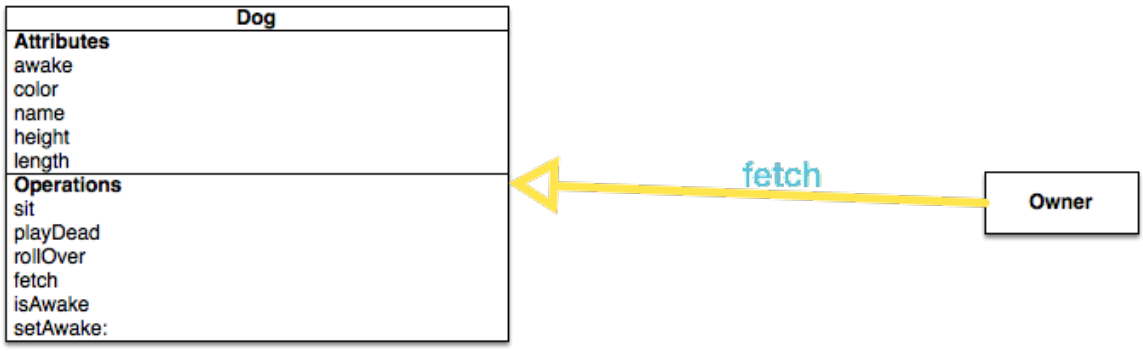


Figure 8: Message "fetch" sent to "Dog" object.

Both of these statements are written using "bracket notation" in Objective-C. Between the two square brackets, the first term is an object and the second term is a "message" being passed to it. The programmer tells the object inside the brackets to do the action corresponding to the message sent to it.

Note that asking Fido if he is awake and asking him to fetch look the same. This is because the mechanism to do so is the same.

In the second instance, I send Fido the message "`fetch`" and Fido looks inside his code, and because he is a Dog, finds that a method has been written that tells all Dogs how to fetch, and runs that code.

Similarly, if I want to know if Fido is awake, I send him the "`isAwake`" message, and he looks up the command telling him to reveal to me his state of awakeness.

This concept is called "message passing" and it is fundamental to how Objective-C programs are conceptualized. Message passing was also the central premise in Smalltalk, and Alan Kay argued that in his mind, it was the central premise of all object-oriented programming (Kay 1998; Stefan L. Ram 2003), in contradistinction to some popular object-oriented languages (C++, Java), where telling an object to invoke an action was understood as a variant of a function call. "Functions," also known as "procedures" or "subroutines," are the way procedural languages, that is, languages organized around processes rather than objects, break up code into modular sections. This allows functions to be called by other processes

(or even by the function itself) over and over again, allowing their code to be more easily reused and generalized. Because procedural programs are organized around processes, functions in such languages exist independently from the data that they manipulate. Thus, the language of "calling" a "method," like "calling" a "function," is based on a procedural, not object-oriented, model of programming, and has bled into Objective-C discourse in recent years, diluting the notion of message passing that Objective-C's bracket syntax reinforces. For example, the same code in Java would be written like this:

```
Fido.fetch();
```

In Java, all method calls are written using a period, or "dot" separating the object from the method's name. Java, Python, and many other languages use similar dot syntax for method calls. Writing method calls like this reinforces the notion that methods are simply functions attached to objects, and are called just like functions are. This way of thinking eliminates a key distinction (message passing) between procedural and object-oriented languages that Alan Kay believes is fundamental, because Kay's model of object-oriented programming was based on a metaphor of biological cells or individual computers on a network that could only communicate with each other via messages (Stefan L. Ram 2003).

Now, in the above example, I have actually glossed over a detail. I mentioned that I could access one of Fido's attributes, his state of awakeness, by sending him the message, "`isAwake`." Fido actually has an internal setting, an "instance variable," to store this value. However, like other objects in Objective-C, Fido is a black-boxed, opaque object. This means that any data stored within Fido in its instance variables can only be accessed directly by code inside Fido. Other objects, including other Dogs, cannot see or poke around and change Fido's attributes without politely asking Fido by sending him a message first. This principle of "encapsulation" or "data-hiding" is also commonly understood to be a fundamental feature of object-oriented languages. Why is this important? Such black-boxing of code keeps different code components modular, preventing them from accidentally changing each others' data, which is a common source of bugs. Moreover,

preventing objects from seeing into each other's attributes keeps them more loosely coupled, which allows for a program to be more flexible and thus more easily changed. It also helps to prevent client objects from depending on a particular class's implementation details, which can change over time. All of these practices help produce code that is not only more stable, but more easily maintained.

However, in many cases, such as in our case, we do want other objects to know Fido's `awake` state. To do this, Dogs provide an interface to allow other objects to access and modify this setting.

This is the "`isAwake`" command that I send to Fido:

```
awakeStatus = [Fido isAwake];
```

"`isAwake`" is a very simple type of command that just returns the value of some internal piece of data. In Objective-C terminology, it is a method that returns the value of an instance variable (in this case, the variable named `awake`, which is a Boolean type value, meaning that it can only store `YES` or `NO` values). This kind of method is called a "getter."

Here is what the code implementing `isAwake` looks like:

```
- (BOOL) isAwake {
    return awake;
}
```

What if I want to tell Fido to go to sleep? I need a way to modify the value of Fido's `awake` setting. Luckily, dogs have another command, which I can use to change Fido's `awake` setting to `NO`. This type of command is called a setter.

```
[Fido setAwake:NO];
```

The code implementing this setter looks like this:

```
- (void) setAwake:(BOOL)newAwakeValue {
    awake = newAwakeValue;
}
```

Getter and setter commands are very simple, but with every setting inside a Dog that I want to be accessible and modifiable by other objects, let's say, the Dog's

owner, I need a corresponding getter and setter. Writing "boilerplate" code like this over and over again gets tedious.

So in 2007, Apple added a way to automatically generate these getter and setter commands automatically. It called this feature, "properties" to signal to programmers that these values can be thought of as properties (or, in other words, attributes) of the object, and that there should be a standard way of accessing them and changing their values. It also creates a layer of abstraction between the idea of the object's "properties" from how such attributes are actually implemented—the fact that the object stores these values in its memory as instance variables. Now, instead of thinking of whether or not Fido has a variable that stores the `awake` setting, I think of whether or not Fido has the `awake` property, and whether or not that property is readonly, meaning that it cannot be changed (in other words, immutable), and thus only has a getter, or whether it is read-write, meaning that it can be assigned a new value (in other words, mutable).

With the properties feature in Objective-C, one can simply write this code to declare that `awake` is a property of dogs that can be read or written to, and the appropriate getter/setter methods, and the instance variable storing the actual value, are all automatically synthesized by the compiler. Declaring a property is now just a simple one-liner:

```
@property BOOL awake;
```

A programmer no longer needs to create an instance variable inside Dog objects called `awake`, nor manually write getter and setter methods. While these tasks are not difficult, if a class has many of such properties, it can become quite repetitive. And until 2011, when Apple added a way to automatically manage memory in Objective-C, setter methods could be tricky to write correctly without causing memory bugs.

Note that there are other ways I could have written code for Fido to wake up or go to sleep. I could have written methods that look like this, which internally just set the `awake` instance variable to `YES` or `NO`.

```
[Fido wakeUp];

[Fido goToSleep];
```

These methods are implemented like this:

```
- (void) wakeUp {
    awake = YES;
}

- (void) goToSleep {
    awake = NO;
}
```

These are just regular methods, not getters and setters, because the name of the message I send to Fido isn't a one-to-one mapping with Fido's internal `awake` setting.[41] Although these methods do the same thing internally as standard setters, to human programmers, they mean *semantically* different things. For example, let's examine the following two lines of code again:

```
(1) [Fido isAwake];

(2) [Fido wakeUp];
```

---

[41] The reader might have noticed that `isAwake` is not necessarily a one-to-one match with the `awake` instance variable either. In Objective-C, a convention for writing getters and setters for a variable named awake normally is to write a getter named awake and a setter named `setAwake`. Such conventions originate as social practices and are usually enforced normatively. These conventions are followed by the compiler when automatically synthesizing getters and setters for a property declared awake. (This convention differs from Java, where a getter would be named `getAwake`.) However, in this case, `awake` sounds like a verb, not a noun, and leaving it like this would be confusing. Another convention in Objective-C states that Boolean (or On/Off) values should have a getter starting with the word, "is," as in, `isAwake`. However, the compiler must be told explicitly to allow for a custom getter name in this case, as this is not the default. To get the compiler to automatically synthesize a getter called `isAwake`, the property can be declared like this:

```
@property (getter=isAwake) BOOL awake;
```

Line 1 is like an adjective. I ask Fido if he is `awake` or not. Line 2 is more like a verb, a command. I tell Fido to `wakeUp`.

Now, I could have written Line 1 like this:

```
(3) [Fido awake];
```

The normal convention for getter methods in Objective-C states that a getter method should have the same name as the instance variable it is retrieving. Thus, the getter accessing the `awake` instance variable should also be named `awake`. This differs from the Java convention, which would prefix the getter with the word "get," so that it would be named `getAwake`. Such rules are known as "conventions" among programmers because they are social practices enforced normatively, not mechanically by the compiler.[42] The compiler will allow a programmer to manually write a getter called `getAwake`, but the programmer will be looked at askance by the Cocoa community. All of Apple's APIs follow these conventions, and thus if one does not follow them, there will be inconsistencies of style between one's own code and when one needs to call code in the Cocoa frameworks.

Note, however, that Line 3 is semantically confusing, because it is not clear if `awake` is a verb or an adjective—am I telling Fido to `awake`, or asking him if he is `awake`? This is also considered bad style by the Cocoa community, because of the ambiguity in meaning. There is another convention that addresses this. Because `awake` is actually a Boolean value, which models On/Off state, the convention states that Boolean properties should have a getter prefixed with "is," and then the name of the instance variable the getter is accessing, so we have the resulting getter, "`isAwake`."[43]

---

[42] The compiler will, however, generate getters and setters following the standard Cocoa naming conventions when declaring properties.

[43] This is all well and good when writing getters manually, but the compiler follows the normal convention when automatically synthesizing getters. To tell the

Whether the method is named `awake`, `getAwake`, or `isAwake`, however, does not matter at all to the compiler. It understands that in all three of these cases, a message is being sent to Fido, telling it to run a command, and it looks up the method implementation corresponding to the name of the message, `awake`, `getAwake`, or `isAwake`. It will then translate this into equivalent machine code. The name and its meaning matter only to human programmers.

All Dog objects (objects belonging to the class `Dog`) would know how to run these commands, but only Dogs. A Car object wouldn't know how to wake up if the `wakeUp` message were sent to it, because no method named `wakeUp` has been implemented in the code for the Car class, and this would trigger an error at runtime.

So far, all of these examples have been written in the traditional Objective-C bracket notation. Recall line 3:

```
(3) [Fido awake];
```

With dot notation, we can rewrite line 3 like this:

```
(4) Fido.awake;
```

Again, to the compiler, this is equivalent to line 3, and translates to the same machine code.

According to Apple engineer Chris Hanson, dot notation was introduced to reduce the confusion between something that looks like a "verb" (what he calls, "behavior") and something that looks like an "adjective" (what he calls, "state") (Hanson 2009). Apple introduced dot notation alongside the property feature in Objective-C, and intended that it only be used to access the property-like attributes of an object, aspects of the object's current state stored internally in variables. Dot

---

compiler to synthesize a getter with a custom name when declaring a property, one would write this code:

```
@property (getter=isAwake) BOOL awake;
```

notation was not supposed to be used to send objects commands, even though properties are implemented by the compiler as calls to getter and setter methods. The reason that human programmers are used to seeing dot notation used for accessing state is because in languages whose syntax is derived from the programming language C, this looks like it is reading Fido's `awake` variable directly. C has been one of the most popular languages in the industry, and newer languages, such as Java, have tended to use modifications of C's syntax because it is familiar to most programmers.[44]

Note that line 2,

```
(2) [Fido wakeUp];
```

can also be written in dot notation like this:

---

[44] In Java, the same line of code (both 3 and 4) would be written:

```
Fido.awake();
```

In keeping with Java conventions, however, it would most likely be rewritten as:

```
Fido.getAwake();
```

The parentheses mean that `awake()` and `getAwake()` are method calls—parentheses hold inputs, and an empty pair of them means the method takes no inputs. It is possible, in Java, to declare instance variables in a class to be "public," meaning that those variables are now accessible to other objects without requiring getters and setters. If the `awake` variable was declared public, we could access it directly using dot notation like this:

```
Fido.awake;
```

Note the similarity with the getter of the same name, the only difference between going through the getter and direct access being the presence of parentheses. Perhaps this is the reason for Java's convention of prefixing getters with "get," so that there is more visible difference between the two.

```
(5) Fido.wakeUp;
```

There is nothing preventing a programmer from doing this. Apple's Objective-C compiler will happily accept this code, as line 5 is equivalent to line 2. However, Cocoa programmers frown upon this and consider it bad style. Why? Because `wakeUp` is supposed to be something that a Dog *does*, part of its *behavior*, not something that represents an attribute, or the state, of a Dog. The community's opinion here is in line with Apple's intentions—Apple discourages programmers from writing code in this way in its style guidelines. However, its compiler will not mechanistically enforce this convention. Apple, and the community, both rely on persuasion to spread the acceptance of this convention.

Our original getter, `isAwake`, written in bracket notation looks like this.

```
(1) [Fido isAwake];
```

In dot notation, this is now rewritten like this:

```
(6) Fido.isAwake;
```

From Apple's perspective, Line 6 now maximally expresses a programmer's intention that `awake` is a property of Fido and that we are accessing it. It does this both through the naming of the getter (`isAwake`) and through the use of dot notation. There is now a clear difference from Line 2:

```
(2) [Fido wakeUp];
```

In using a verb as the name of the message, and in using bracket notation, the programmer signals that this is telling Fido to do this behavior, meaning, we are telling Fido to `wakeUp`.

Thus, Apple's official stance, explained in detail in a blog post by Chris Hanson, is that bracket notation should be used to trigger an object to perform behavior and dot notation should used for accessing an object's state (Hanson 2009).

Herein lies the controversy, however. Opponents of dot notation, such as Big Nerd Ranch's Joe Conway, argue that even Apple-sanctioned, "proper" use of it, such as in line 6, is bad. Conway makes the argument that no Objective-C

programmer should ever use dot notation. Why? Conway and others claim that it is actually more confusing than the problem that it is intended to solve. Conway argues that rather than making code less confusing to read, dot notation makes it *more* confusing, because the meaning of the code now becomes ambiguous.

Here I must take an aside to explain that Objective-C is actually a hybrid of two different languages. Firstly, it is fully compatible with C, the most popular procedural language in use by the software industry, meaning that existing C source code can be mixed into an Objective-C program without alteration. For a high level language, C is actually very low-level, with features that allow access to hardware that most other languages prevent. For many programmers, C is the closest one can get to assembly language programming without paying the complexity and portability costs of actually programming in assembly, and because it allows such low-level manipulations, programs in C, like in assembly, can be tuned for maximum performance, making C attractive for coding infrastructural software such as operating systems. C was created at Bell Labs in the 1970s in conjunction with the Unix operating system and was used to program Unix, and its popularity spread alongside that of Unix. As mentioned earlier, its dominance in the industry by the 1980s led many subsequent programming languages to adopt C-like syntax for its familiarity.

Because C is a procedural language, it is organized not around objects but around processes (which are broken up into *functions* in C). Unlike Objective-C methods, C functions are not tied to objects, but exist autonomously in the code. However, procedural languages like C still need a way to collect heterogeneous types of data together and pass them around as a group. In C, this is called a structure, or `struct`, for short. Think of it like a box into which you can throw a bunch of different things. Unlike Objective-C objects, structures are not black-boxed, but transparent. The data stored inside can be freely accessed by functions with no restrictions.

Objective-C was created by Brad Cox in the 1980s. Cox was an admirer of Alan Kay and Xerox PARC's Smalltalk language, but noted that Smalltalk had limited commercial success because it was not compatible with the large corpus of programs being written in C. In order to bridge the gap, Cox added a small set of extensions to C to add objects and message passing, thus creating a language that was a superset of C. Cox introduced bracket notation to clearly delineate when a programmer was writing Objective-C code in the message passing style, and when one was writing standard C code. This allowed Smalltalk-styled object-oriented code to coexist simultaneously with procedural C code that could be taken from existing legacy code bases, giving Objective-C flexibility and compatibility. The object-oriented style allowed a programmer to design the program at a highly abstract level, but when the program needed to be optimized for performance, he could drop down to the C level to get at the bits.

In C, to access a piece of data inside a structure, a programmer uses the dot operator. Let's say we have a structure which represents a window on the screen, which as a height and a width. We would access its height like this:

```
x = window.height;
```

Thus, structure access in C uses a form of dot notation. Because Objective-C is fully compatible with C, this is also true in Objective-C. This code is unambiguous in C, and also in Objective-C before the introduction of dot notation for properties.

However, in Objective-C, I can also have an object that represents a window. Before the introduction of dot notation, if I wanted to access the `window` *object's* `height`, I would call its height getter method using bracket notation:

```
x = [window height];
```

But now with dot notation for properties in Objective-C, it is possible to write this to access the `window` object's height:

```
x = window.height;
```

Now, there is no way to tell if `window` is an object or a structure!

This is the heart of Conway's argument. A programmer, reading this code, can no longer tell, at a glance, which of the two possibilities she is seeing. One of the norms within the Apple programming community is that code should be readable, clearly understandable with a minimum of effort. One should not have to write extra comments in the margins to explain what a piece of code does; in other words, it should be "self-documenting." Code readability is a normative virtue held by many programming communities, though not necessarily all—as Coleman has shown, Perl hackers show considerable pride at writing purposefully obfuscating code in order to demonstrate their own cleverness (Coleman 2013, 93). Even among programmers who value readability, however, different groups have different criteria for what they consider "readable." As we will see later, Objective-C programmers consider expressive, verbose code to be readable, a preference that distinguishes them from programmers who prefer other languages.

Another opponent of dot notation, Marcus Zarra, wrote on his blog: "[Dot notation] makes the meaning of your code unclear. Objective-C is known for its self documenting nature." Because of its ambiguity, "Dot Syntax removes that." (Zarra 2008)

Similarly, Conway argues:

> The naming conventions used in Cocoa and Cocoa Touch are clear and straightforward. When we look at well written Mac or iPhone code, we can tell exactly what is going on by glancing at it. That's the power that Objective-C gives us. If I want an object to take a drink of water while doing a cartwheel, I send it the message:
>
> ```
> [obj takeDrinkOfLiquid:water
>      whileDoingCartwheel:YES];
> ```
>
> We know *exactly* what that means. There is no room for interpretation. (Conway 2009)

Note, the "we" Conway is referring to is an experienced Objective-C programmer. Obviously, an average person off the street would have no clue what this code means. Additionally, an experienced Java programmer might also have trouble parsing this statement.

Jeff Lamarche, a dot notation proponent, however, disagrees that the meaning of any line of code is immediately obvious without putting it into the context of other code.

> But, when do you ever look at a line of code in a vacuum? You don't. Code has no meaning taken out of context…
>
> You can't obscure something that you don't have a reason to know. The amount of information that bracket notation gives us over dot notation is trivial and not enough to make an informed decision about what the code is doing anyway, so you **have** to consider its context. If it's not your code, you have to look at the preceding code to understand it anyway. [emphasis in original] (Lamarche 2009)

Lamarche makes another argument as well. It is not the job of the language to discipline programmers from doing bad things, but rather to give them tools to solve problems. Tools can be abused, but it is part of being a good programmer to use them wisely. All programming languages give programmers the ability to do bad things (programmers commonly call this, "giving them the rope to hang themselves"), but some languages are designed to be more permissive, and thus more powerful, than others. Objective-C is such a language. To prevent abuse of these features, Lamarche argues, it is up to the community to inculcate proper norms of use, and individual programmers to learn good judgment.

> Any programming language, to be useful, has to allow some kinds of bad code. I doubt it's possible to create a programming language that doesn't "allow" an inexperienced programmer to do all sorts of completely horrible things. I could come up with dozens of examples of ways that Objective-C 1.0 [before dot notation] "allows" you to do bad things. (Lamarche 2009)

Lamarche contends that bad code is bad code, regardless of whether dot notation or bracket notation was used to write it. The problem lies not with the language or the syntax itself, but with the practices of the programmer. This argument is "predicated on a programmer doing something wrong and can both be demonstrated just as easily without using dot notation." (Lamarche 2009)

Furthermore, Lamarche argues that permissiveness, rather than safety, is one of the fundamental elements of the Objective-C language, as designed by Apple.

> "I actually find it hard to believe that an experienced Objective-C programmer would even attempt this argument because, frankly, it sounds like an argument you'd get from a C++ programmer. Objective-C is a permissive language. It's in Objective-C's DNA to let you do things. […]
>
> These are intentional design decisions. This language is designed to give you a lot of flexibility and puts trust in the developer that you'll use its features appropriately. Objective-C's dot notation doesn't run contrary to that in the slightest. In fact, it's a logical extension of that underlying philosophy. They're faulting dot notation for something that's inherent in Objective-C. (Lamarche 2009)

Lamarche is implying that one of the values driving Apple's design of Objective-C is that it allows developers to "do bad things" because by doing so, it also gives them the power and flexibility that they would not have if the language was designed to prevent them from making mistakes or writing malicious code. This involves Apple putting "trust" in its developers to use this power "appropriately." Because the language does not enforce certain practices but is permissive, it is up to the community of programmers to develop the norms of practice to guide them in writing code that does not "do bad things."

The specific target of Lamarche's critique is a coding style guideline (Pinkerton, Miller, and MacLachlan 2014) written by Google to establish standards of practice by Google employees writing Objective-C code for OS X or iOS programs, as well as contributors to Google open source projects for these platforms. Google's arguments for banning dot notation follow the same lines of logic as Conway and Zarra. Google as a corporation did not really have a large stake in the debate, given that Objective-C likely makes up only a tiny fraction of the all of Google's code. Dave MacLachlan, one of the authors of the Google style guideline, writes in the comments to Lamarche's post that the decision was the result of internal Google discussion and voting (with those on both sides in the discussion), and that the result was not meant to be "used by someone as a definitive statement on the correctness of using dot-notation in Objective-C." He also notes that another reason for the ban, which was never mentioned in the guideline itself, was that at the time of the writing of the guideline, Google had to support older code that would run on Mac

OS X 10.4, which did not support dot notation. He also acknowledges that the style guideline is a living document and that it is open to change, and that practices within the company, including his own, were also starting to change. As of today, the universal ban has been removed from the guideline, although it still recommends that dot notation only be used for properties (Pinkerton, Miller, and MacLachlan 2014).

What is interesting, however, is that though it was the result of internal discussion and debate at Google itself, in 2009 Google's public style guideline lent institutional legitimacy to the proponents of a dot notation ban, who were otherwise drawing on their own authority as experts in the Cocoa community of practice. This made the style guideline an important target to debunk for dot notation's supporters, like Lamarche:

> Google makes the case that dot notation is bad because it can result in confusing code when a developer pays no attention to established naming conventions or makes really poor design choices. But these problems have nothing to do with dot notation. Poorly written code is poorly written code. The simple fact of the matter is, if you're trying to read code like that, nothing is going to help. With, or without dot notation, the code will be hard to read because it's bad. The correct solution in that situation is to fire or train the developer who wrote the offending code. (Lamarche 2009)

Note that Lamarche argues here that it is not the use of dot notation per se, but the improper use of it by an unskilled developer that is the problem. Lamarche is claiming that skilled developers can agree that objectively, there is such as thing as "poorly written code," and that they can all recognize it when they see it. His larger point is that the problems people are blaming on dot notation are not technical or formal problems with the language itself, which can be solved simply by banning the use of dot notation. For Lamarche, the real problem is the low level of skill of the many programmers (likely newcomers) who are writing this code, who have an inadequate understanding of Cocoa and are insufficiently trained in its normative practices. A strict language cannot by itself force an unskilled programmer to write better code. The solution cannot be dogmatic or mechanical—as a social problem, it must be solved managerially or pedagogically. These programmers need to be better

trained to follow proper naming conventions or make better design choices—the community must do more to integrate them into their practices.

When it introduced dot notation and properties in Objective-C 2.0, Apple likewise decided against mechanistically enforcing correct practices. It could have, for example, written the compiler to emit errors if a programmer tried to use dot notation for normal method calls rather than property access. However, it did not do this. It is possible that Apple developers are used to a certain degree of freedom in their coding practices and the community might have interpreted such a move as overly draconian. Instead, Apple relied on persuasion to encourage the adoption of best practices. Apple has a number of ways to persuade developers—its online documentation and learning materials can clearly state guidelines, and sample code can give examples of proper use. Apple also has a bully pulpit at its developer conference, WWDC, where it often explicitly recommends practices in presentations. Apple is not unique here. Companies that provide toolkits and platforms for developers rely on varying combinations of technological enforcement versus normative persuasion to exhort their developers to follow certain practices, and the particular mixture can change over time. Microsoft, similarly to Apple, has technology "Evangelists" who promote the adoption of new APIs and ways of writing code. Apple itself has gradually tilted towards increasingly mechanized enforcement of practices, as changes to the Objective-C compiler have increasingly begun to count on code following established conventions. Beyond the official corporate line, individual Apple engineers like Chris Hanson can also give their opinions on their own, though it is likely that they self-censor opinions they know to be contrary to company policy, and thus the ones you see are probably in-line with mainstream thinking at the company. Because such individuals appear to represent themselves, rather than the official line, however, this can actually lend their comments greater weight as coming from the horse's mouth, rather than through a PR filter.

One crucial difference between Hanson's position and Conway's is that for Hanson, dot notation is useful because it allows programmers to see the conceptual

difference between the properties of an object (its state) and the actions it can run (its behavior) instantly, at a glance. This allows programmers to think at a higher, more abstract level. Hanson is privileging higher level understanding.

For Conway, however, the syntax masks the fact that when dot notation is used, an action has to take place (a 'getter' method is run) to return the property that is being asked for. Depending on how the action is implemented, additional things could be happening that are now hidden from the programmer, making bugs hard to track down. What looks like simply reading a piece of data is actually running some more complicated code. This obfuscation decreases transparency and thus understandability. Conway is arguing that understanding what the code is actually doing, meaning its lower-level implementation, is important for practical reasons. Conway is privileging lower level understanding.

Both sides are arguing that the use or non-use of dot notation makes code more readable. It is a struggle between two different idiomatic conventions. Hanson is arguing for a new idiomatic practice whereby the use of the new dot notation syntax clearly signals state versus behavior, thus making a programmer's intention more clear.

Conway and others, on the other hand, claim that dot notation confuses the idiomatic convention that has been in Objective-C from the beginning: message passing.

Again, from our example above:

```
x = [window height];
```

Bracket notation indicates that the first term in the bracket, `window`, is being sent the message `height`. The `window` object then runs a method that returns the value of the `height` instance variable stored within `window`. It makes clear that an action is occurring.

```
x = window.height;
```

However, when one uses dot notation, like the above, it doesn't look like an action is occurring. It looks like the `height` variable is simply being read directly, as if `window` were simply a transparent structure and not an object with access restrictions. The ambiguity arises because of Objective-C's hybrid nature, being compatible with C. With dot notation, it is impossible to tell whether `window` is a C structure or an Objective-C object; bracket notation, on the other hand, is exclusive to Objective-C objects and message passing.

Why is this important to Conway? Being involved in training new Objective-C programmers at Big Nerd Ranch, the primary reason for him and his boss and co-author, Aaron Hillegass, is pedagogical. Due to the mobile app market explosion, hordes of new programmers now want to learn iOS, and the Cocoa Touch SDK that Apple provides for writing iOS apps are all written in the Objective-C language, previously ghettoized into the NeXTSTEP and Mac OS X minority platforms. Thus, a majority of programmers new to iOS have experience in more widely used languages as C++, Python, PHP, and Java. Java is heavily used in university computer science courses, and thus one can expect most recent computer science graduates to be familiar with its syntax. C++ was dominant for applications programming in the 1990s, Java became dominant for enterprise server software, and PHP and Python for web servers.

All of these languages have syntax derived from C, and thus use dot notation. However, they use it for both accessing attributes (instance variables) and running actions (methods). Methods, however, would be indicated by a pair of parentheses, to allow for inputs. In Java, the same line of code to call `window`'s height getter might look like this:

```
x = window.height();
```

However, in Java, instance variables can be configured with different read and write permissions to allow "public" access to them directly, without having to go through getter methods. To access the height variable directly, the code would be written like this.

```
x = window.height;
```

Now, say our Java programmer is now learning Objective-C. This code could appear in either language, but they mean different things. While a human programmer might write this code to express the same intention at a highly abstract level (accessing some attribute of this window), to the Java and Objective-C compilers, they mean different things, and they will implement different code paths. Our Java programmer may assume that this code works the same way that it does in Java, which does not call the getter for the `height`, but is simply reading it directly. However, this same code in Objective-C hides the fact that a getter is being automatically generated for the programmer by the compiler. Dot notation is simply a convenience, but it hides operations that Apple's tools are doing for the programmer behind the scenes: automatically synthesizing getter and setter methods so that the programmer doesn't have to spend time on these repetitive tasks. Understanding what Apple's tools are doing is critical to learning how Objective-C and Cocoa works, and to build up knowledge and skill. This also reveals a persistent trait of Apple's tools and frameworks—while they often make things more convenient, they also hide and obfuscate what's really going on, making them more difficult to understand. It can be very simple to do simple things, but more complicated and advanced functionality requires a deep understanding of what Apple's tools or code libraries are doing behind the scenes in order to know the right places or times to "hook" into them in order to inject custom behavior. Thus, Apple's tools paradoxically make programming both easier and more difficult—they do a lot of things for the programmer, freeing her to focus on more abstract design tasks, but require substantial time in learning what they are actually doing. Cocoa is an enskilling, rather than a deskilling, technology.

This highlights the importance of pedagogy in the Cocoa community. Because, as Aaron Hillegass argues, the full power and productivity benefits of Cocoa technology can only be unleashed after deep understanding of how Cocoa works is obtained, learning is vital. Thus, it is primarily for pedagogical reasons that Hillegass and Conway insisted on avoiding dot notation.

Dot notation, because it mimicks the syntax of other languages, does not force learners to radically reconceptualize how objects are told to run commands. In Java, C++ and other object-oriented languages, methods are simply functions to be called, and dot notation reinforces this conceptualization. Bracket notation, following Smalltalk-style syntax, emphasizes the conceptual notion of messaging passing. Proponents of bracket notation argue that understanding message passing is akin to a Kuhnian paradigm shift (Kuhn 1996). The roots of this thinking lies in the appropriation of Kuhn's term "paradigm" into computer science to describe different categories of programming languages, such as procedural, object-oriented, imperative, or functional, that are so different from each other that they require completely different ways of thinking in their use. Part of the reason for this appropriation is that the originators of these programming "paradigms" found it discursively useful to argue that their new method represented a fundamental break from the past, which required programmers to upend or invert their previous ways of thinking, despite the fact that the new languages borrow or build on concepts in earlier ones. For example, Zepcevski comments on differences of opinion between computer scientists Niklaus Wirth and Tony Hoare as to whether object-oriented programming merely aggregated and emphasized concepts such as modularization that were already known, as Wirth argued, or whether it represented a fundamentally new way of thinking about how programs are organized, as Hoare argued (Zepcevski 2012, 275). This bears similarity to the oxygen/phlogiston debate in chemistry described by Kuhn. The existence of hybrid-paradigm languages, such as C++, would seem to disprove the idea that object-oriented programming is a different paradigm from procedural programming. After all, to be a paradigm in Kuhnian terms, the two sides must be incommensurable, and how can they be incommensurable if it is possible to blend the two paradigms together in a single language? Nevertheless, proponents of "pure" paradigm languages, such as Smalltalk, argue that hybrid languages, by providing training wheels for programmers new to the paradigm, do not force them to make the requisite mental shift necessary to see things in a completely different way, resulting in them never truly understanding the

language properly. It is as if a physicist continued to think in classical Newtonian terms despite working in a relativistic system.

Although Objective-C is also a procedural/object-oriented hybrid language, its partisans have argued since the NeXT era that programming in it forces programmers to shift into a more object-oriented way of thinking, because of its emphasis on message passing, a concept derived from Smalltalk, which Objective-C borrowed from liberally. Alan Kay, who developed Smalltalk and coined "object-oriented" to describe it, considered message passing fundamental to object-oriented languages, and for this reason, would probably not consider C++ a true object-oriented language (Stefan L. Ram 2003). Although Objective-C mixes procedural and object-oriented code together in the same language, it features completely different syntax for each, so that even when working in the same language, a programmer is forced to visually shift between procedural and object-oriented code, reinforcing the mental shift between procedural and object-oriented problem-solving. C++, with its unified syntax, does not force this mental shift, as it was designed by Bjarne Stroustrup to be an evolutionary step from procedural C, gradually adding in object-oriented features, rather than a revolutionary break (Stroustrup 1993).

Similarly, Objective-C uses terminology that would be completely foreign in a procedural language. For example, in Objective-C, the object receiving the message is still called the "receiver." C++'s terminology of "member functions" and "data members," by contrast, refers back to more procedural concepts. Objective-C purists try to ensure terminological purity in not referring to terms that imply procedural concepts. For example, when I was at Apple between 2000 and 2005, John C. Randolph, at the time a Cocoa Evangelist for Apple, frequently reiterated to me that it was incorrect in Objective-C to "call a method," and that the proper term was to "send a message" to an object (the "receiver"), the message containing a "selector" which would tell the object which "method" to dynamically "invoke." Because "methods" are known as "member functions" in C++, the procedural notion of "calling" a function naturally transfers over. Although Java refers to "member functions" as "methods," like Smalltalk and Objective-C, it uses C++ style dot

syntax to invoke them, with the result that the term "to call a method" is widely used in Java. Objective-C's bracket-notation, on the other hand, is derived from Smalltalk's prefix notation for message passing.

Thus, part of the stakes of the dot notation controversy itself is whether or not object-oriented programming, and in particular, Objective-C's message passing version of it, constitutes a different paradigm from procedural programming or more static object-oriented languages like C++ which contain procedural vestiges or compromises. To the old guard, the proponents of bracket notation, syntax clearly matters for signaling to programmers learning Objective-C that they need to be making a mental shift and not hold onto procedural mental analogues that will just hold them back. They argue that thinking of method invocations as simply function calls, as C++ style dot syntax encourages them to do, obscures the way things really work in Cocoa. It keeps newcomers thinking in a paradigm they are familiar with, without allowing them to move into the new way of thinking emphasized by message passing. Remaining in the old paradigm maintains practices that are foreign to the way Cocoa works, and leads to mismatches in code, and thus bugs. Thus, the dogmatic purists argue, it is crucial in Objective-C to maintain the visual distinction between message passing code and procedural code.

This rehashes older debates between Smalltalk and C++ adherents over the conceptual purity of object-oriented languages. C++ deliberately blurred procedural and object-oriented concepts together in order to remove the need for procedural C programmers to have to learn a radical new paradigm, translating such concepts as "methods" into procedural terms like "member functions," making for a gradual, evolutionary learning process. This hybridization with procedural programming was in fact, as Zepcevski argues, one of the reasons for C++'s very success in popularizing object-oriented programming in industry where Smalltalk had largely failed (Zepcevski 2012). However, for adherents of Smalltalk, this miscegenation destroyed the distinctiveness and advantage of object-orientation altogether, and for them, it was debatable whether a C++ programmer really understood, or was even doing, object-oriented programming at all. Objective-C's own hybridity is not a

problem because the clear separation of its object-oriented and procedural halves are visually delineated by Smalltalk-like bracket syntax versus ordinary C syntax. This separation keeps the object-oriented half pure; the introduction of dot notation into the object-oriented side of Objective-C desecrates this purity. Back in the 1980s and 1990s, NeXT programmers were already arguing that Objective-C, being closer to Smalltalk conceptually, was a vastly superior language to C++. "The rest of the world has gone chasing after C++ and has only slowly discovered its limitations for object-oriented design…" (Webster 1992) "as an object-oriented language, Objective-C leaves C++ in the dust." (Garfinkel 1992a) "'There's nothing about C++ that invites you to write good object-oriented programs,' says [Andrew] Stone [of Stone Design]." (Garfinkel 1992c) By mixing the paradigms and muddying the concepts, dot notation threatens to bring with it the mediocrity Objective-C programmers associate with C++ and the rest of the software industry.

Purists, however, have lost ground to pragmatists in recent years. Message passing has lost some of its conceptual dominance in the Cocoa community, so much so that the language of "method calls" has even become common within Apple itself. Newcomers, and increasingly, converts from the old guard, don't think dot notation is really that big of a deal. For the proponents of dot notation, the notion of message passing as expressed in bracket notation is not incommensurable with the notion of function calling, as expressed in dot notation. In this new view, the two notations, and hence the two ideas, are not only compatible with each other, but one can simply be thought of in terms of the other. The recession of message passing language goes hand in hand with the recession of the notion of object-oriented programming as an incommensurable paradigm from procedural programming.

For example, comments in both Conway's and Lamarche's blog posts refer to particular language features in Objective-C being "incompatible" with the principles behind object-oriented programming. A reader named "Izidor" commented:

> "State" and "behavior" are concepts from procedural programming - you have inert "data" (structs,records,etc.) and active "functions", operating on data.

> Object-oriented programming is *exactly* about removing this split and hiding everything inside object, which handles every request appropriately, without outside world knowing what needs to be done.
>
> […] So dot notation is introducing inferior concepts into OO…

Lamarche responded to this by arguing:

> It is true that OO programming hasn't traditionally recognized or utilized the split between state and behavior. That doesn't mean it can't be a valuable addition to the language. Paradigms evolve as we use them and gain experience with them.
>
> (Lamarche 2009)

Similarly, one Donald, responding to another commenter who argued that for his taste, Objective-C was insufficiently like Smalltalk, posted in Conway's blog, "I also totally agree that the procedural control flow in objective-c [sic] breaks with object orientation." (Conway 2009)

Because so much of the dot notation controversy involves judgments about the practices and skill of newcomers in the community, a critical issue at stake in the dot notation debate is the pedagogy of practice. Does it matter, pedagogically speaking, to teach message passing as a wholly different, incommensurable paradigm from function calling? Because much of the rancor against bad code written in dot notation is about novices who have imported foreign models of programming from previous experiences, which cause translational mismatches, the argument for incommensurability is for throwing out what they think they know and starting over, to learn the native model of Cocoa and Objective-C from scratch.

To emphasize the pedagogical rationale against dot notation, I wish to point out that although Hillegass will not teach its use in books or courses, and prefers to avoid it himself, he does not object to its use by his own employees. In his eyes, if an experienced developer understands that dot notation in Objective-C really means message passing and not Java-style direct access of data, he has no problem with it. A personal experience I had with him illustrates this point. During my first field trip to Big Nerd Ranch in July of 2011, I was assigned to add new features to an

application for a local orthodontic practice, and ran into a coding question. Despite being the CEO of the company, Hillegass had made himself available for answering questions, and on this day he happened to be in the office for me to ask for his help. Knowing his opinion on dot notation, I typed out all my code in bracket notation, which takes slightly longer. He told me I could use dot notation if I wished, as he trusted that I understood what the code really meant. Although pedantic for training purposes, in practice, Hillegass's attitude was pragmatic. Other instructors of Cocoa at other institutions had similar attitudes. Hal Mueller, the instructor of the introductory Cocoa programming class at the University of Washington's extension program, similarly does not object to the personal use of dot notation. He does not teach its use for the first three weeks of his course, however, noting the confusion of his students. However, after that period, he allows them to use whatever notation they prefer. It is possible that this laissez-faire attitude could be related to virtue of individual autonomy that software programmers hold in common, a value that reasserts itself once teachers are assured that their students have the proper conceptual understanding. Of course, "correct" practice and autonomy continue to exist in tension even among experts, as the dot notation controversy demonstrates. As we will see in the next section, such questions are really about group membership—what practices draw the boundaries of who is in or out? Prior to the iPhone gold rush, assimilating community members performed their membership through using bracket notation or by proclaiming a disdain for dot notation. Increasingly, this is becoming unimportant.

Boundary work, pedagogy, and aesthetics in the Dot notation controversy

While a few notable experts in the Cocoa blogosphere such as Jeff Lamarche and Chris Hanson argued in support of dot notation, including influential voices at Apple itself, much of the online backlash against dot notation took on the cast of boundary work against novices and outsiders, programmers who had low inclusion in the techno-cultural frame of Cocoa programming. As we have seen, dot notation looks much more like Java syntax, and developers such as Marcus Zarra and Joe

Conway felt that its introduction by Apple was a move to pander to newcomers. Thus, while the debate, in one sense, could be seen as one between prominent and established community experts such as Hillegass against Apple, it could also be seen as an effort by a conservative group of oldtimers to stem the practices of a new generation of immigrants, with Apple in support of the latter. The oldtimers had reason to be concerned. In the late 1990s, a few years after Apple's acquisition of NeXT, the compiler group at Apple had contemplated updating Objective-C to use Java-style dot syntax for all method calls, calling it "modern syntax." Before the NeXT acquisition at Apple, the exclusive use of Objective-C and its non-standard bracket syntax had been discussed as a possible reason for NeXT's lack of popularity and applications:

> The irony is that one of NeXTSTEP's biggest advantages—use of Objective-C and supporting tools—also serves as a major roadblock. The rest of the world has gone chasing after C++ and has only slowly discovered its limitations for object-oriented design… On the other hand, developers using Objective-C on the NeXT run into problems when they want to move to other environments. These barriers, real and perceived, have caused many developers to avoid NeXTSTEP development in the first place. (Webster 1992)

According to a former NeXT engineer, Steve Jobs dismissed this reason, citing NeXT's low sales and user base as the real problem:

> That's one of the things that I think is most fascinating about Objective-C's journey is the fact that Steve Jobs stood behind it the whole time. […] Because when I was at NeXT, I remember when we were having problems as a company not thriving as much as we all wanted—I remember saying to Steve when we were sitting down having coffee or something, you know, listen, is Objective-C part of the problem, Steve? Is the reason developers don't want to program to our platform Objective-C's fault, because if it is, I could give a crap, let's try something else, I'm fine, we could [write] translators. He said, "Steve, Objective-C isn't our problem, the problem is, I'm not selling enough boxes. When I sell enough boxes, people will program in Objective-C. It's a fine language, you guys are doing good work. (Steve Naroff Interview, December 22, 2012)

As we saw in chapter 2, NeXT's struggling business meant that it had a small installed base of users, which made it difficult for developers to sustain a business

making software for it. This was a bit of a catch-22, as having less software itself may have further contributed to a small installed base. Was NeXT's use of Objective-C was one of its advantages, or one of its drawbacks? Was the obscurity of the language dissuading people from developing for the platform? For many of NeXT's most loyal engineers and developers, Objective-C was a key to NeXT's competitive advantage over other platforms. For them, C++ was like Microsoft Windows—the more popular, but inferior solution. Jobs seemed to agree. According to Steve Naroff, the engineer Jobs hired away from Stepstone to NeXT to work on the Objective-C compiler, Jobs felt that it was NeXT's struggling hardware business and marketing strategy that was the real problem, and not the technology choices made by NeXT's engineers, which Jobs felt were sound.

This would not be the first time that being tied to an obscure programming language was linked to the problem of being a minority platform, for the company that made the technology. After Apple's acquisition of NeXT in 1997, it faced a critical transition—to move its installed base of users and developers from the classic Mac OS to its new NeXT-based OS X. It was in the years during which OS X was in development, soon after the acquisition, that Apple considered adopting a "modern syntax" for Objective that used Java-style dot notation for all method invocation. Naroff feels that Apple's "modern Objective-C" effort might have been aimed at encouraging the adoption of the Cocoa frameworks among existing Mac Carbon developers who were used to C++. This effort encountered significant resistance among the Cocoa frameworks division at Apple, and was ultimately scuttled, much to the relief of the NeXT-turned Cocoa community.

> Smalltalk is this whole environment that frankly doesn't fit well with most C people's view of life. So [because Objective-C was a hybrid of Smalltalk and C] it was always this weird balancing act of bringing these features to the C world without pissing the C people off. Complicated. And to this day, a lot of the C people hate the syntax, right?
>
> A lot of people coming from C++ really moan about brackets and the colons. And there was an effort, I forget what year it was, but I was at—I was working for Avie [Tevanian, Apple VP of Software] at the

> time at Apple and I had written this paper about doing this modern version of Objective-C…
>
> We had all the [API] interfaces translated… the compiler was modified, we really were ready to release it, and there was a lot of soul-searching in Apple, and I think Ali's [Cocoa frameworks] team, who typically I looked to as—okay guys, should we do this? Right? It's not a big deal from a compiler perspective. Your interfaces are the face of our platform. Is this really what you want? And they couldn't pull the switch. And Ali [Ozer, manager of the Cocoa frameworks team] wasn't really that happy a camper; Avie was fairly convinced we should do it… the frameworks guys were fairly against it. My only technical argument for not doing it, was I felt strongly that… keeping the Objective-C syntax looking distinct from C++ is goodness… modern Objective-C looks so much like C++ that it's going to be a lot harder to mix the languages. The compiler was able to tell, just to be clear. But humans might have a harder time with it.
>
> (Steve Naroff, Interview, December 22, 2012)

However, as most large Mac applications in the early 2000s continued to be written in Carbon, Apple tried another approach to encourage Cocoa use. Java had significant support inside Apple in the early 2000s, and a "Java-bridge" was created to allow Cocoa developers to write applications using the Cocoa frameworks in the Java language instead of Objective-C. It was hoped that, because Java was in widespread use in enterprise software and computer science pedagogy, this would remove Objective-C as a barrier to Cocoa adoption. It turned out, however, that complaints about Objective-C among newcomers and Carbon developers were merely complaints. According to Cocoa developers, the Objective-C language itself was never the real problem—they claim that the language is simple and enough rudiments to be useful can be picked up in a few days. "…the surface area of the language is very small… it is such a small, simple language, that I don't have to spend a lot of [mental CPU] cycles dealing with the sharp corners." (Mark Dalrymple Interview, April 11, 2012) "I grabbed [Aaron] Hillegass's [Cocoa] book and I was like, all right, I'm going to learn Objective-C. And two days later… I think I know Objective-C! This is great! It was like the overhead of becoming competent enough to produce stuff in Objective-C was so low—it was like, this isn't a big deal. I was picturing the nightmare that is C++." (Chris Parrish Interview, March 2, 2012)

According to Cocoa developers and Apple engineers, the real challenge of learning Cocoa is actually learning the Cocoa frameworks themselves—how they work, the design patterns they use, the sheer number of APIs available. Java developers were not only used to using the Java language, but also the user interface libraries that came with Java. Cocoa essentially replaced these libraries. Julie Zelinski, a former NeXT and Apple engineer who became adjunct faculty at Stanford, and helped develop its unique Cocoa programming course, explains:

> The [Java] bridge did not really pan out. […] I mean, what are you doing, you want to use dot syntax instead of brackets? OK, whatever. But I can remember… at the time of the bridge, I think that the motivation for that [was because] Objective-C was still seen as something that made us an oddball and people saying, "I don't want to learn Objective-C." You know what? Objective-C takes like a week to learn, right? What takes you months to learn is the AppKit, the Foundation, sort of the whole [Cocoa] toolkit. And people balking at, "I can't learn Objective-C," are like, "I can't learn your stuff, I can't learn to put it on Cocoa," like when we had the old-style Mac programmers trying to move them forward with us and they'd be like, "I can't learn Objective-C."
>
> And so that's why the bridge came out. What if we made it so you could talk to it all in Java? Would that help? It turns out that wasn't the problem. It wasn't the syntax of how you address the AppKit that made your learning curve, right, it was the AppKit, right? So I think that's how the bridge was kind of a false endeavor, right? Because it was never really the problem and in the end you can change the syntax out but still there's a huge learning curve and you have to get people to invest in the learning curve to get to the other side. And that side will want to just do it in Objective-C, [so] who cares, right? Take a week and learn that too. (Julie Zelinski, Interview, April 24, 2012)

As we saw in chapters 3 and 4, learning the Objective-C language is only the tip of the iceberg to learning Cocoa. The learning curve of Cocoa is due to the conceptual understanding of design patterns that must first be experienced in use before they are truly internalized. Peter Galison used the metaphor of pidgin languages in describing his notion of "trading zones," where different scientific subcultures meet and converse. A pidgin contains simplified elements of the two languages spoken natively by the groups involved in exchange, but is not itself the native language of either group and exists only for the purpose of facilitating contact

between them (Galison 1999, 153–4). People from one group learning words in the pidgin that originate in the other group's language do not acquire the full meaning of those words in their original language, but only a surface meaning. Galison sees this as similar to the ideas that are exchanged between groups in a trading zone. These "trading zone objects" "carry radically different significance for the donor and recipient" and may mean different things to each side (Galison 1999, 146). For example, "mass" had different meanings for Einstein, Lorentz, and Abraham, but within the trading zone between theoretical and experimental physics, a simplified notion of mass allowed the results of experiment to arbitrate between the different theories, despite what might appear to be Kuhnian incommensurability between them. This is useful in considering our earlier discussion of the term "method call," which Objective-C purists say is a misnomer because it treats methods like functions when in actuality messages are being passed. "Method call" can be seen as a kind of pidgin term that is useful for Java programmers learning Objective-C. The Objective-C language itself can be learned in a shallow fashion using a pidgin of concepts that can be borrowed from programmers' familiarity with their native languages, such as Java or C++, but to become a full "native speaker" of Cocoa requires considerably deeper understanding. Similarly, although Java could be used to program Cocoa applications, because Cocoa was written natively in Objective-C, Cocoa-Java was a kind of idiomatic mismatch. If Objective-C is shallow to learn, why bother dealing with the weird pidgin of Cocoa-Java?

For this reason, Cocoa-Java use never accounted for a significant share of commercially available Cocoa applications, and it was largely ignored by the existing NeXT/Cocoa community, which continued to advocate the use of Objective-C as the platform's true "native" language. No self-respecting Cocoa developer would use Cocoa-Java if they wanted to be taken seriously. Thus, Cocoa-Java was a kind of no-man's land—pure Java developers who did not want to spend time learning anything new stuck to using Java's Swing libraries for developing cross-platform applications, while Cocoa developers continued to use Objective-C to develop native Mac OS X applications tailored to take advantage of its capabilities.

By 2006, Apple had decided that the Java-bridge was not worth continuing to maintain.

The introduction of dot notation with properties in Objective-C 2.0, which took place between 2006 and 2007, took on a similar cast when it dovetailed with the explosion in Objective-C use with the iPhone SDK. Unlike with Mac OS X, Apple had not allowed iOS programmers to use any language other than Objective-C for the first few years of the App Store, and it could use the App Store approval process to enforce this. This forced a whole new generation of newcomers, who had economic incentive to write iPhone apps, to learn Objective-C. As we saw in the previous chapter, existing Cocoa Mac OS X developers were already wary of the influx of newcomers possibly destroying the values of the community. Their use of dot notation in Objective-C only seemed to reinforce their status as outsiders, despite official support from Apple. At stake was whether these barbarians at the gates could be taught the "right" values and practices and assimilated into the community, or whether they would end up having the run of the place.

A number of Cocoa developers noted their initial reactions to dot notation in terms of outsiders versus insiders in the community:

> …when I first saw it, I was like, come on, really? The Java programmers can't figure out brackets? Is it so hard? That was my initial reaction, no lie. I'm just like, come on, that's just absurd…
>
> (Luke Adamson, Interview, February 22, 2012)

Adamson himself has since been converted to using dot notation, but in examining his own initial hostility, he reveals that some developers considered dot notation a foreign incursion into the purity and coherency of the Cocoa idiom:

> Change is uncomfortable. Especially when you've been working in a system that's so internally consistent. And then to introduce this change that's so representative of an outside force, like this other. I don't know, I think it's uncomfortable for that reason alone.
>
> I think it's just a foreign element, and whether people are willing to admit it or not—I mean, I certainly do not like someone coming along and introducing a foreign element into my nicely cohesive thing that I

> understand. All of a sudden there's a kidney bean in my pudding. It's confusing. (Luke Adamson, Interview, February 22, 2012)

Here, Adamson gives voice to what he believes the uproar over dot notation is really about. Oldtimers felt that Objective-C, as they had been using it since the early 1990s, had been coherent and relatively pure, keeping its Smalltalk-inspired object-oriented additions cleanly separated from its procedural C elements through syntax. This syntax had stabilized by the mid-1990s and had not undergone significant change until 2006. While developers from other platforms, and even some engineers within Apple itself, saw this lack of change in the language as a sign of stagnation, for longtime developers, its unchanging features may have taken on an air of tradition and sacrality. Change, as Adamson noted, felt uncomfortable. This was doubly so if the change meant incorporating elements that the community felt were "foreign" or not "native" to the platform and its established idioms, which would ruin its conceptual consistency and purity. Dot notation was seen as coming from Java or C++, while bracket notation was fully "native." "Foreign elements" might be seen as a kind of pollution that would desecrate the language and make it profane. Anthropologist Mary Douglas has discussed how cultures define what is "dirty" or "dangerous" to be "matter out of place," that which, according to a culture's categories of classification, does not belong (Douglas 1966). A "kidney bean in my pudding" would certainly fit this description.

Nevermind that Objective-C had been, from the beginning, a hybrid language, a pragmatic compromise. Steve Naroff, who had been with Objective-C since the beginning and helped develop it in the direction it took at NeXT, never did feel that Objective-C was pure or elegant. "If you talk to [Apple's Cocoa framework engineers] they'd say 'I think [Objective-C is] a really pretty language.' Because they were the people that evolved it with me. But as a language [designer]… trying to be nonreligious about what I do, I frankly never thought it was a pretty language, but pretty—pretty wasn't the goal, pragmatic was." (Steve Naroff Interview, December 22, 2011)

By introducing "matter out of place," dot notation seemed to destroy one of Objective-C's most sacred qualities, its consistency. Previously, in Chapter 3, we discussed how coherency and consistency is one of the primary virtues Cocoa developers ascribe to how the Cocoa frameworks work, an important factor in Cocoa's ability to amplify a developer's productivity. This coherency is also important for learning Cocoa—although an initial investment must be made up front to learn Cocoa's idioms and design patterns, those patterns are applied throughout Apple's frameworks on both iOS and Mac OS X. Thus, once a developer learns how to use a few of the Cocoa APIs, she can expect that others, including new APIs Apple has yet to release, will work similarly, amortizing her learning. Brian Turner, a former instructor at Big Nerd Ranch, explained this in his words:

> …even though iOS is only five years old, Cocoa isn't. So Cocoa Touch is also not really that old. So the libraries are refined and they're clean and they're intuitive. And they follow similar standards… Once you're kind of familiar with how the accelerometer works, then interacting with—you know, they [Apple] will stick with one general category [of ways to do things]. They'll say, we're always going to [use this design pattern] for these types of things, and for these types of things we're always going to [use this other pattern]… blah, blah, blah.
>
> And it gets to the point where I can literally just kind of guess, like you know, when somebody says, hey, do you know how to use this, like no, but let's see what we can do. And with quickly glancing at Xcode's autocomplete, most of the time I don't even have to bring up documentation, you can just—so that's how Apple is good. (Brian Turner, Interview, October 4, 2011)

Here, Turner is referring to "autocomplete" in Xcode. Xcode is Apple's integrated development environment (IDE), which is a unified environment providing the majority of tools that a programmer needs to write software—including a text editor for code, the compiler, a debugger, and other tools. When a developer uses Xcode to write code, when she needs to make use of an API defined in Apple's libraries, she only needs to type in the first few characters, and Xcode will automatically suggest the possible API code that she might want. In this way, an experienced Cocoa developer may not need to even consult Apple's documentation

for the API. Because Cocoa follows a unified set of conventions for naming its APIs, once learned, a developer can often just guess at the name and it will often be correct, and if not, autocomplete will fill it in correctly. This greatly improves the speed and productivity of writing a Cocoa program.

Among existing Cocoa developers, these feelings of purity and consistency are articulated as aesthetic arguments. In regard to dot notation, a developer who Tweets about the elegance of bracket notation is performing his membership as an insider in the community. Bracket notation is unique to Objective-C, while dot notation is common in the industry, and every conversation I have had with a developer familiar with another language who has tried iOS development has included a statement about the "ugliness" or "weirdness" of brackets. Many experienced Cocoa developers I interviewed noted that they too found bracket notation difficult to adjust too when they were novices, but they all described a conversion experience upon which they began to appreciate bracket notation. Ken Case, co-founder of OmniGroup, noted, "My first reaction to Objective-C was, what's with all of these brackets? They just seemed really foreign to me… as a C programmer… But, it only took a week or two to get over that." (Ken Case Interview, February 10, 2012) For these developers, a switch occurred whereupon Objective-C transformed from an "ugly" language to a "clean" or "elegant" language:

> In Objective-C, like just everything is so clean. So I really like how method-sending looks, like, I really like brackets, you know? Which I know is completely unique to Objective-C and nobody else likes it but us, but [I] love the bracket and nested message sending…
>
> (Brian Turner Interview, October 4, 2011)

This aesthetic appreciation is a deeply affective experience that connects writing Objective-C code with using the Cocoa Touch APIs to write iOS apps. Note how Turner used the same adjective, "clean" to describe both Objective-C, and the Cocoa Touch APIs in the earlier quote.

Other developers similarly described the exclusive use of bracket notation as aesthetically pleasing. Marcus Zarra noted in a blog post that, because bracket

notation is still necessary for sending most kinds of messages to objects, using dot notation for properties results in an ugly mixture:

> Dot syntax breaks up the flow of the code. Code should be elegant. It should be graceful and beautiful to look at. This is ugly, nasty code that you want to [hide] as fast as possible so that you can stop looking at it. The message passing lines are completely at odds with the dot syntax lines. The difference is striking and distracting.
>
> When we write code, we care about its formatting. We care that the code is properly indented and that the indentation is consistent. We should care equally as much about its consistency of style and design. Switching from message passing over to dot syntax and back is not consistent. (Zarra 2008)

Code aesthetics is related to consistency of idiom, for Zarra. Two different conceptual and linguistic idioms should not mix, but be kept separate and pure. For Zarra, this is more than just an issue of personal preference, or a feeling of violation of purity by unseemly mixing, though Zarra is relying on this for persuasive effect. For many programmers, "elegance" means conceptual purity and simplicity, in an almost mathematical sense, in the way that physicists describe Maxwell's equations as elegant. Simple solutions to complex problems require cleverness and ingenuity, and are considered elegant in part because they are so difficult to construct. It is in this sense of elegance that Coleman's Python hacker Espe described his "high tower of control and purity." (Coleman 2013, 95) Moreover, elegant and consistent code has a pragmatic advantage—it is easier to read. And for Zarra, readable, consistent code is maintainable code.

Readability, Maintainability and Software Engineering

"Maintainability" is a trait of concern to software engineering organizations, where code readability is critical for collaboration. This concern with maintainability among software developers has a long history, going back at least to the 1960s, the perception of the "software crisis" and the emergence of software engineering, if not earlier. Nathan Ensmenger has noted that software maintenance is a crucial, if

somewhat low-status, task in the production of software. Fixing bugs, or software "repair" is endemic to what a programmer does. "In theory, software should never need maintenance because software does not break down or wear out, at least in the conventional sense… Except that software does break—all the time, at great expense and inconvenience to its users… *There is a strong argument to be made that the software crisis of the late 1960s was essentially a maintenance problem*…" [emphasis mine] (Ensmenger 2010, 224–225)

More importantly, however, software maintenance cannot be easily separated from the process of producing software from scratch. Ensmenger notes that a lot of "maintenance" work in software is actually adding new features in order to adapt an existing piece of software to changing needs and environments. Fred Brooks noted in *The Mythical Man-Month:*

> All successful software gets changed… As a software product is found to be useful, people try it in new cases at the edge of, or beyond, the original domain. The pressures for extended function come chiefly from users who like the basic function and invent new users for it.
>
> Second, successful software also survives beyond the normal life of the [hardware] machine vehicle for which it is first written… and the software must be conformed to its new vehicles of opportunity.
>
> In short, the software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product. (F. P. Brooks 1987, 12)

Ensmenger similarly argues, "The majority of software maintenance involves what are euphemistically referred to in the literature as "enhancements," "new functionality, as dictated by market, organizational, or legislative [developments] and changes in the larger technological or organizational system in which the software is inextricably bound." (Ensmenger 2010, 226) The reason for this is because software is only one part of a socio-technical system—although seemingly intangible and changeable, "legacy" software, such as the COBOL programs running on banks' mainframes that needed to be updated for Y2K, has high longevity and durability due to the fact that they are deeply embedded in working organizations,

institutions, and practices. Moreover, software itself is a materialization of such social relations and structures. "Because software is a tangible record, not only of the intentions of the original designer but of the social, technological, and organization[al] context in which it was developed, it cannot be easily modified… *Software is history, organization, and social relationships make tangible.*" [emphasis mine] (Ensmenger 2010, 226–227)

Software almost never is created *de novo*. "…the degree to which software is embedded in larger, heterogenous systems makes starting from scratch almost impossible… 'We [programmers] never have a clean slate,' argued Bjarne Stroustrup, the creator of the widely used C++ programming language…" (Ensmenger 2010, 226–227)

Thus, much of the work software developers do is modifying existing code already written, either by someone else, or by themselves at previous moments in time. This is necessary because the process of writing software is iterative, starting out with a bare-bones skeleton that gradually accumulates additional functionality. New code should work seamlessly with old code, and thus be conceptually on the same page, even if written by different authors. Conceptual mismatches, or worse, misunderstandings of the original programmer's intent, result in new features being clumsily tacked onto the old code rather than working in the same spirit. Such mismatches are a large source of bugs in software, making such code a maintenance nightmare. Even in brand new software, many programmers begin by laying down a basic architecture, and gradually adding features incrementally. The process does not stop once all the features are complete—the features must also work correctly. The focus then shifts to testing and fixing as many bugs as possible until the software seems stable enough to ship to users. Thus, making a code-base "maintainable" is a benefit not just in subsequent releases, but also in the initial production of a piece of software, particularly in complex, large-scale software systems. The production of the "new" is not easily separated from maintenance and repair.

Nonetheless, when writing new software from scratch, less experienced programmers may not be adequately thinking about future maintenance issues in the initial stages, when decisions made have ramifications down the line. Ensmenger notes "…software systems are often coded before they are completely specified. Many programmers find it easier to 'just start coding' than the develop design documents. Most programs are poorly documented (if at all), and so most maintenance works involves intensive on-the-job learning." (Ensmenger 2010, 227) Many inexperienced programmers do not follow rigorous, disciplined practices to make future maintenance tasks easier—in other words, they do not see coding as a continuing maintenance activity but as a task of creation happening only once. This means that they do not design before they begin to code, which often leads to poorly thought-out designs that are difficult to modify without breaking. Poor documentation makes it difficult for future maintainers to understand the code in order to fix or enhance it. These are pervasive problems in software, and Ensmenger argues they are part of the reason for the persistence of continued software "crises."

Almost all of the Cocoa developers I spoke with have had some experience dealing with code that was difficult to maintain in some way or another. I myself have had many such experiences from my earliest days as an intern at Hewlett Packard in 1998. When Lamarche speaks of "bad, poorly written code," or another developer mentions "spaghetti code," these experiences of frustrating late nights trying to untangle the complex interdependencies inside existing code immediately come to mind. There is a point at which the hacks and kludges to add new features to old software make the whole edifice so fragile and precarious that the whole structure threatens to crumble. "It starts becoming a pile of cards which eventually collapses under its own weight." (Mark Dalrymple, Interview, April 11, 2012) Past this point, it is more productive to simply throw the whole thing away and start over. In between, there may be intermediate points where the software needs to be partially rewritten and redesigned because the old architecture does not adequately model the new situation that the software must address in the present. In the software industry today, this process is called "refactoring." Although refactoring does not add new

functionality and thus new features, and can be a substantial undertaking, it may be necessary to do this to shore up the foundations in order to allow for future enhancements to be made. As an example, Mac OS X 10.6 Snow Leopard was a software release from Apple that added few features that benefited users, but provided new foundations for the operating system that allowed Apple's engineers to significantly advance the software in subsequent releases.

Having faced such experiences, expert Cocoa developers have learned the hard way that following disciplined practices to make code more maintainable will pay off in the future. Thus, among experienced Cocoa developers, maintainability has become a professional virtue—it is simply what professional software engineers ought to strive for in their code. Code reusability, modularity, and the separation of concerns among different components, principles and practices promoted by structured and object-oriented programming methodologies are now seen as integral to what a good programmer should do to produce reliable, quality software because it makes it easier to maintain.

Community norms may be insufficient against managerial or market pressures, however. Cocoa developers such as Marcus Zarra understand that developers under tight deadlines may have reasons to cut corners. Software projects are constantly under short-term time pressures, militating against long-term planning for maintainability. Given the constraints of managerial or client deadlines, the temptation to cut corners, be sloppy, or simply take clever shortcuts can be justified in the name of expediency. For contractors (which many iPhone developers are), once a contract is fulfilled and accepted by the client, the project ends and the developer never works on the code again. It may not be necessary to follow best practices to write software in a maintainable fashion on every project.

Thus, Zarra argues not only from normative reasons, but instrumental, practical ones: making code more readable, and thus more maintainable, may take more time in the short term, but will greatly pay off in the long run—even if the developer is an indie and has no manager to enforce such discipline. (In fact, in some

situations, programmers are actually under pressure from managers or clients not to follow disciplined practices to make code maintainable, but simply implement features as quickly as possible in order to hit overly aggressive deadlines.) Even for such indies, software should be written in collaboration in mind, because for Zarra, a programmer is always collaborating at minimum with one's past or future self.

> We often forget about this aspect [maintainability] when we are working by ourselves. However, maintainability is not just a factor when we hand off code to another developer, it is also a factor when we have to come back to our own code 6 months from now! I don't know about you, but when I look at code I wrote a year ago, all I can think is: "That moron! What the HELL was he thinking!" (Zarra 2008)

Zarra thinks that a good programmer should treat his future self as essentially a different person than his present self, and thus communicate in code his current intentions and other contextual details that may fade with time. This is because, when writing code, a programmer is deeply embedded in the mental context of that moment, a context that becomes lost in the future.

Likewise, Conway, in his own blog post, asserts that maintainability, not personal preference for style, is the main reason for his call to ban dot notation:

> Why this seemingly irrational hatred of dot-notation? Is this a style choice and [are we] being hard-headed? The answer is no, we are not being hard-headed, we are keeping our code consistent and maintaining readability… the main goal of software development [is] writing, maintainable, effective, efficient, easy to read and bug-free code. (Conway 2009)

A commenter in his post vehemently disagrees, asserting that Conway has confused the means of software development with its ends. "Well, no, I don't think so. The main goal of software development is solving someones [sic] requirement. If you code looks dirty or [whether] you use the dot syntax is not *that* important, as long as it [sic] the software helps the people who will be using it." (Conway 2009)

Both are right. What Conway and Zarra mean is that software that solves someone's requirement not just in the immediate present, but over the long-haul, must be maintainable. This is because they both recognize that software, if it is

successful, must change over time in response to its users. Inevitably, design decisions made in the context of the original target use/user and hardware have constraining effects on the ease of making future changes. Extending an inflexible design to do new things increases its complexity, creating more opportunities for bugs. Eventually, this can become so difficult that the entire design may need to be "refactored," restructured, or rewritten from scratch for further progress to proceed. This can happen even in the lifetime of a single project, as client or employer requirements change, and initial design assumptions no longer hold. Thus, planning for the long-term changeability (or maintainability) of software means following practices that encourage flexible designs.

Despite constant pressures to cut corners, Cocoa developers have learned from experience that taking the extra time to make the proper architectural decisions now will pay off later. Thus, they advocate getting into the habit of making maintainable code, because not doing so will inevitably haunt them in the future. Rather than a managerial imposition, maintainability is a professional and vocational norm among Cocoa developers, who advocate best practices for maintainability in public forums. For Cocoa developers, "maintainable" signals that code is "high quality," relatively "bug-free," "flexible" and "future-proof": virtues of the programmer's craft. For developers like Zarra and Conway, readability is one key aspect of maintainability. Among Cocoa developers, unlike programmers of other platforms, this manifests uniquely in a preference for less terse, more verbose and expressive code.

Verbosity as a Virtue in Objective-C code

This concern with maintenance in the Cocoa community can be seen in the community's preferences with regard to coding style. While maintainability itself is not exclusive to Cocoa developers, and is a trait sought for by many professional software developers, the association of verbosity (or expressiveness) in Objective-C code with readability and thus maintainability is rather unique. We can contrast this with the virtue of cleverness that Gabriella Coleman describes as primary in much of

the free software community. Coleman illustrates the virtue of cleverness by illustrating a single line of Perl code that expresses the functionality of what normally would take six lines. “Perl is a computer language in which terse but technically powerful expressions can be formed (in comparison to other programming languages). Many Perl coders take pride in condensing long segments of code into short and sometimes intentionally confusing (what coders often call ‘obfuscated’) one-liners.” (Coleman 2013, 93) Such code performs technical mastery, genius, and playfulness, virtues valued among hackers that highlight their individual autonomy despite being involved in the communal activity of programming open source software. Clever, obfuscated code is a form of boasting in a meritocratic and competitive (and masculinized) community in which intelligence is particularly celebrated. It is also a celebration of creativity, as such “hacks” are seen as akin to “poetry.” (Coleman 2013, 94) Free software programmers know that their peers in the community will read their code, and such code is as much a way of asserting their abilities among those peers, as it is about functionality. And because of the elitism in the community, people are less likely to complain about hard to read code, because they will not wish to be seen as less intelligent. This dovetails with hackers’ propensity for jokes and wordplay, which Coleman notes are alternate expressions of cleverness for hackers, being “hacks” of natural language. No one would want to admit that they didn’t “get” the joke.

Cocoa developers are programmers, and do enjoy similar expressions of cleverness and ingenuity, jokes included. However, for Cocoa developers, cleverness is not a virtue that trumps all others; it must be subordinated to disciplined professional software engineering concerns for maintainability. Clever code, particularly if it is terse, is considered by Cocoa developers to be bad if it is unreadable. In chapter 3, we saw how some Cocoa developers seemed to uphold writing less code as a virtue. Wil Shipley claimed, for example, that “The only method that matters is how little code did you use, overall. That’s the only thing.” (Wil Shipley, Interview, April 18, 2012) While this statement might seem to prefer condensing code into as few lines as possible, what Shipley meant was that, overall,

the Cocoa frameworks, by providing rich functionality and powerful design patterns, allowed a programmer to express the same functionality with less effort. Overly terse code, however, can actually increase the effort for a programmer in understanding what he (or someone else) previously did. It can also complicate the task of debugging. In Objective-C, for example, it is possible to chain a succession of method calls by nesting bracketed message passing expressions inside each other. For example, the following contrived code is a single expression that could be written on a single line:

```
return [[[NSDateFormatter alloc] init] stringFromDate:[NSDate date]];
```

The problem with this code is that it combines too many calls together, and this becomes extremely cumbersome to debug. Cocoa developers recommend that programmers break this expression up into separate components:

```
NSDate *date = [NSDate date]; // gets current date
NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
NSString *myString = [formatter stringFromDate:date];
return myString;
```

This now allows a programmer using the debugger to set breakpoints on each separate line, allowing him to verify that each of the objects, the date object, the date formatter object, and the string object, are all being returned with the desired values, before returning the string expression at the end. The one-line expression could be said to be more “elegant” in a mathematical sense, because it encapsulates all of the functionality together, and moreover, does not need to allocate local variables to store all the intermediate values, which could be seen as wasteful of memory. However, today, programmers feel that such memory usage is trivial compared to the vast amounts of memory available in modern computers, and such memory will be quickly recycled anyway. To them, it is more important to optimize for the readability of programmers, now and in the future.

Cocoa developers like Objective-C syntax because they feel it is expressive. Cocoa APIs are carefully named by Apple to have an English-like grammatical flow,

and names of things are usually written out in full rather than abbreviated as they often are in the conventions of other programming communities. For example, instead of naming a method `setBkgrdClr`, for instance, Apple recommends instead the more explicit `setBackgroundColor`. Programmers in other environments have acquired the habit of naming variables or functions with very terse names in order to save having to type them over and over again. In Cocoa, the text editor in Apple's Xcode IDE, the suite of tools Apple provides free for Cocoa developers, will autocomplete such long expressions for the programmer after typing in only one or two characters, eliminating extraneous typing. This means that Cocoa developers are free to use fully descriptive names in their code (Parker 2014). In fact, Apple actively encourages using longer, more descriptive names, because, as a popular saying in the community goes, code is written once, but read many times. When code becomes easier to read, its intention becomes clearer, lessening the need to document what it does separately in a comment to the side. (In the above code, the descriptive text following the '`//`' characters is a comment. This text is ignored by the compiler and allows programmers to write natural language descriptions of what their code does to document it for others and the future.) Code can become clear enough to an experienced practitioner that comments may become redundant, because the code itself clearly expresses what it does. Cocoa developers say that such code is "self-documenting" and strive to achieve this quality whenever possible.

> Objective-C is a little more formal. But it also informs the developers a lot more. The language itself, once you understand the syntax of the language, you're doing a lot less looking things up.
>
> So like in Java, a lot of times you'll see even in the Java doc they tell you here's the input parameters and what they mean, and so it's in the actual comments, which then get generated into documentation.
>
> Whereas in Objective-C that same information is right there in the method. So, Objective-C is not designed to be easy to write; it's designed to be easy to read. Which I think is much more important when you're reading through code.
>
> …And like I said, the tools—Xcode gives you a lot of help in typing it in, but I'm a lot less interested in making it terse and easy to type, than I am having it explicit and tell me what's actually going on here.

(Robert Walker, Interview, May 19, 2012)

Indeed, having too many comments can be a bad thing, because, when programmers change code, they may not always change the comments documenting the code, which become stale as they no longer accurately describe what the code is now doing.

> Actually, I used to be a real big code commenter… but I did start to learn quickly that the likelihood of that comment getting out of sync with the code is high, so it's far better for the code to just explain it. I try to just reserve my comments these days for something that seems unusual… "Here's why I'm doing it this way instead of the way you probably thought you wanted to do it." […] So I will write those comments first, implement them, erase the comments as I go. …more as a blueprint kind of thing… I've got to lay down the breadcrumbs… And then I try to delete all the evidence of that… so it doesn't become a burden later, documentation or cruftiness, you know, right? (Chris Parrish Interview, March 2, 2012)

Again, the reason such readability is important is to signal to one's future self the purpose of code, which one will inevitably forget:

> You come back to something after a period of time and you want to get the gist of it really, really quickly. You don't want to have to spend time doing research over something you wrote six months or six years ago, you just need to understand it fast. And with Cocoa, that's always, always so easy. I hardly ever have to comment anything. It's just like the code is the comment. (Brent Simmons Interview, February 17, 2012)

It is also is one of the reasons why both Apple and third party Cocoa developers spend considerable effort on what to name the entities they create in their code. For example:

> I've become a brutal revisioner of my own code… It's really to me more about the expression of the simplicity of the code and maintainability that I'm willing to go back and revisit… I'm adamant about changing the name of stuff. Like, if this thing used to mean this, but now it means this, I must rename it to mean what it means [now]. Because otherwise when I come back through, I'll be like, what? Why is it called that, that doesn't make sense, that's not what it's doing. (Chris Parrish Interview, March 2, 2012)

In addition to naming practices, certain features of the Objective-C language used extensively by Apple in the Cocoa APIs also help to make code more expressive and self-documenting. One of these is *named parameters*, also sometimes referred to as *keyword arguments* or *labeled arguments*. To understand what these are, let's go back to Conway's example from his blog post:

```
[obj takeDrinkOfLiquid:water whileDoingCartwheel:YES];
```

In this statement, the `obj` object is being sent the message `takeDrinkOfLiquid:whileDoingCartwheel:`. This result is `obj's takeDrinkOfLiquid:whileDoingCartwheel:` method being invoked and run. The method requires two inputs (also called *parameters* or *arguments*), which are placed right after each colon character '`:`'. In this example, `water` is the first argument, and `YES` is the second. The text right before each colon is part of the name of the method, but is also a keyword name labeling the purpose of each argument. It is also Cocoa convention to name these keywords in a way such that the entire method name has an English grammatical sentence structure to it. The programmer writing this code wants to tell the object receiving the message, `obj`, to take a drink of water while doing a cartwheel, and thus this code makes her intention clear to the reader without needing any additional comments written in plain English.

How would this code work in a language like Java without named parameters? Calling a similar Java method would look like this:

```
obj.takeDrinkOfLiquidWhileDoingCartwheel(water, YES);
```

The labels of intention are in the name of the method, `takeDrinkOfLiquidWhileDoingCartwheel`, but they are not tied together with the arguments. This means that a programmer could accidentally switch the order of the arguments, resulting in an error which would most likely be caught by the compiler:

```
obj.takeDrinkOfLiquidWhileDoingCartwheel(YES, water);
```

Because this method only has two parameters, it is not terribly confusing. But some methods can have 5 or more, and it such cases, it can be difficult to remember what all of them are, and in which order they are supposed to be. What is worse is

that some languages like Java and C++ allow two different methods with different numbers and/or types of parameters to share the same method name. It can thus be confusing as to which method a programmer actually wants to call. The following example is modified from code from a publicly available Apple 2014 WWDC presentation given by a former Apple colleague of mine (Parker 2014):

```
controller.presentPopover(aRect, aView, AnyDirection, TRUE);

controller.presentPopover(aBarButtonItem, AnyDirection, TRUE);
```

Although both of these methods have the same name, they take different numbers of inputs, and it is not clear what types of inputs I should pass to them, nor exactly what they mean. For example, what is the purpose of the Boolean true/false flag at the end? What it is signaling to the method?

Objective-C's keyword argument labels allow Apple to name these two methods more descriptively. Thus the above two calls would look something like this:

```
[controller presentPopoverFromRect:aRect
                            inView:view
                  arrowDirections:AnyDirection
                          animated:TRUE];

[controller presentPopoverFromBarButtonItem:aBarButtonItem
                            arrowDirections:AnyDirection
                                   animated:TRUE];
```

Now with labeled arguments, it is not only much more clear what the purpose of each argument is (we now understand that the flag is telling the method to animate or not), it is also much more difficult to accidentally put them in the wrong order.

Again, this is not actually required by the compiler, but is a convention followed not only by Apple but also by the majority of the Cocoa developer community. For example, a programmer could write these two methods without argument labels, which would result in code looking like this:

```
[controller presentPopoverFromRect:aRect :view :AnyDirection :TRUE];

[controller presentPopoverFromBarButtonItem:aBarButtonItem
                                          :AnyDirection :TRUE];
```

This code is perfectly acceptable to the compiler. While much shorter, it is no longer as explicit as to its intention. Though perfectly permissible, I have never seen Objective-C code written this way. Apple's Coding Guidelines document actively discourages this practice. The following example comes directly from this document (Apple Inc. 2013d):

```
- (void)sendAction:(SEL)aSelector
          toObject:(id)anObject
        forAllCells:(BOOL)flag;

- (void)sendAction:(SEL)aSelector :(id)anObject :(BOOL)flag;
```

The first is clearly labeled "Right," the second, "Wrong." Another example promotes clarity of naming. For example:

```
insertObject:atIndex:
```

is labeled, "Good."

```
insert:at:
```

has the comment, "Not clear; what is being inserted? what does 'at' signify?" Just before the examples, the document states, "It is good to be both clear and brief as possible, but clarity shouldn't suffer because of brevity." (Apple Inc. 2013c) The authors of this document at Apple understand that programmers have a tendency to want to be brief, and that this is a good quality in code, but that it must be subordinated to clarity. Again, this does not matter to the functionality of the code itself, as both are equally acceptable to the compiler and will generate the same code.

The self-documenting benefits of having named parameters was one of the reasons the Cocoa frameworks group at Apple rejected the compiler group's attempt to adopt industry-standard dot syntax of method calls in Objective-C in 1998. It was also one of the reasons why programming Cocoa using the Java bridge never became very popular.

> You get so used to the keyword argument, it's a huge part of how the API is designed, is to leverage that keyword argument, which I still miss when I don't get—I don't have a lot of reason to code in Objective-C for anything anymore than fun… whenever I'm looking at any kind of C++ interface I'm always, like, what the hell, is which

> argument where? So, I mean I think again both those efforts [modern Objective-C and the Java bridge] just show that… it wasn't a syntax problem, although that is the first thing that people notice and can kind of balk at, it turns out there are much deeper things that you need to address beyond that, and so it was spending all your time to work on this shallow thing, I think is not going to pay off and is only going to frustrate you, because in reality there is this deeper issue, which is, OK how can we deal with this, the sense that people don't want to learn another toolkit that's different than [Java's] Swing [toolkit] or looks different than Windows or whatever they have been doing.
>
> (Julie Zelinski, Interview, April 24, 2012)

Clearly, Apple follows these guidelines in its naming conventions for its own Cocoa APIs, and recommends to developers, in documentation such as this, and in presentations at WWDC, that they follow similar practices. The remarkable thing is that the community largely does follow these practices, and considers it a good thing to do so. The community itself normatively enforces these guidelines, which it has internalized as elements of good Objective-C coding style. Good Cocoa programmers want to write code with this expressive style, and encourage others to do so as well, in order to keep their code readable and maintainable:

> There is no requirement in Objective-C that your statements must be expressive. That, I mean there's a syntax, and you need to follow the syntax, but syntax supports very, very brief words that will compile exactly the same way as non-brief words… And yet, I mean if you look at Objective-C, 80-plus percent of Objective-C is expressive syntax, right? Because that's the culture…
>
> …I mean, reading other people's code has always been a pain in the ass. All of a sudden, it's like, wait a second, reading other people's code is less of a pain in the ass. Fantastic! I want to read some more of that. (Hasan Edain, Interview, March 12, 2012)

This developer is clear that he is following Apple's example in its APIs, but notes that he does so not because Apple is telling him to, but because it has clear readability benefits:

> … if you think of the API as sort of the first set of example code… this method name is sixty characters [but…] it's really precise… And you look at that and you go, fantastic, I know what you're going to do and I'm going to use you in a way that makes sense… And so that's self-

> reinforcing. That now you're like, oh, does my method look like their [Apple's] method, why doesn't my method look like their method? OK, I'll promptly try and make things that are more like their methods…
>
> Yeah, it wouldn't have worked had there been no actual benefit, then typing the characters is a pain in the ass because you would have said, oh, I'm going to go back to my very truncated style variable naming and… functions that are three characters because that's easier to type, right? But all of a sudden people started realizing that I am getting tangible benefits from someone else having done this. Oh, I will probably get tangible benefits from me having done it. (Hasan Edain, Interview, March 12, 2012)

For Cocoa developers, the readability and maintainability of the code sometimes even trumps not only brevity and terseness, but traditional engineering values such as performance and optimization. Mark Dalrymple explains, while simultaneously admiring the ingenuity required to optimize the original Macintosh Toolbox API machine code to fit into a 64 kilobyte ROM chip, that such highly clever code is unmaintainable:

> The original Mac…] They did so much with so little. Like 64K ROM. Not megabytes. K. That was the entire toolkit. Versus today, you can't even get an icon that's 64K.
>
> So they did amazing stuff of packing a lot of features into a tight compartment, but if you've ever optimized code… the more highly optimized something is, the harder it is to maintain and extend. So you start out with those highly-optimized nuggets of awesomeness, and then we put more awesomeness on top of it, and then it starts becoming a pile of cards which eventually collapses under its own weight. (Mark Dalrymple, Interview, April 11, 2012)

This is one of the reasons for a slogan among Cocoa developers—don't pre-optimize. What this means is that a developer should not be thinking about making performance optimizations while writing code, but should use measuring tools after the program is finished and running to discover where the bottlenecks are, and then speed those up. Making optimizations prematurely ends up wasting time on optimizing code that only runs occasionally and may make no actual difference to the user, and instead can slow down the developer's productivity and make their code harder to read. The developer's time, not the computer's time, is the most valuable:

> This is one of the things that I teach all my young programmers…
>
> What did you just do? And they're like, "See it's more efficient." And I'm like, "It's less readable, you have saved maybe one [machine] instruction. This microprocessor can handle about 8 billion of those per second, and this is in a keystroke processor—what's the fastest typist in the world? Let's say 20 keystrokes per second. So 20 times per second, you saved less than a nanosecond, let's generously say you've saved one nanosecond per second to make it less readable and you spent your time. No. No…
>
> Readability is the most important thing. Which is why small code is the most important thing… The smallest is the most readable. So it turns out that the smallest code is usually—it sort of defaults to fastest, [but] not always. So you do need to optimize. But you only optimize at the end, and that's another rule, it's the hard factor, optimize at the end, you don't optimize [as] you write code, you're not that fast, just write the smallest code. *Because the smallest code is going to be the most readable, and, it's often going to be the fastest anyway. And where it's not you just go in and optimize it and make it slightly less readable*. [emphasis mine] (Wil Shipley, April 18, 2012)

Shipley then connects this explicitly to Moore's Law and uses language that implicitly evokes the software crisis:

> In two years, computers are two times faster, but humans aren't. So the future of your code is, your code is suddenly running way faster. But you've got people who are looking at it who didn't write it. So they're more confused, but the code itself has gotten faster over time, not clearer, but faster… It gets less clear, because you forget it and the other people who come in never saw it. So it's much more important to [emphasize] the readability than the speed. *Speed solves itself, readability gets worse*. And especially *as the size of the product gets bigger,* too, *it gets less and less readable naturally, right*? But it's still getting faster! [emphasis mine]
>
> And because everything is object-oriented, …the frameworks just get faster. So even if machines [don't] work any faster all the stuff we're calling is just getting faster. I'm calling [some Apple API which Apple's engineers have optimized with a new technique.] All righty, *my program just got twice as fast*. La la la. *But it didn't get twice as readable. Magically. So it's the only thing that matters. And also we have a global shortage of good programmers.* There are so many things that it would be great to have automated, and we just don't have the programmers to automate them. (Wil Shipley, April 18, 2012)

According to Shipley, because of Moore's Law, hardware will continue to get faster but human programmers won't. And the larger a software project becomes, the more complex it becomes, and the more problematic the collaboration and management of its developers. Mirroring rhetoric from the software crisis, Shipley says that because there is a shortage of good programmers, making sure that programmers of all abilities can come in and understand code and thus collaborate on it is the most important priority for the software's maintainability, and thus its long-term quality.

Ken Case of OmniGroup summarizes Cocoa developers' general attitude towards the importance of readability, collaboration, and thus maintainability and code quality, supported by particular features of the Objective-C language and the practices of expressive naming that have evolved around it:

> …The results are much more readable code… I quickly fell in love with Objective-C as a language where you could have all the expressiveness of Smalltalk at higher levels… and those levels of abstraction…
>
> So a lot of it is really is not just what the language does, but the practices around the language. How do you write some code that is really readable? I mean the readability, I think, is a big piece of it because it makes it self-documenting and it makes it easier to work with a team. I can look at somebody else's code—if they know the best practices and use it—I can read their code pretty quickly and easily and understand what they're trying to do. (Ken Case, Interview, February 10, 2012)

Such concern over readability and maintainability can go too far, however. Even for indie developers, the desire for perfectionism and craft of one's app sometimes runs into the hard reality that a developer has limited time and must occasionally prioritize shorter-term goals, such as actually shipping an app in order to generate revenue. If the app doesn't ship, it cannot become successful enough to need a version 2.0, and thus putting too much effort into making it maintainable over the long run could hurt it in the short run. Chris Parrish spoke about balancing these conflicting tensions:

> I used to be a programmer who would just keep… futzing with it until it worked and you can do that, but then in the end you're like, "God, look at this thing I created, how will we ever change it the next time?" The next person who comes along and has to add the new sharing method that changes one more thing will be like, "Oh, my God! None of this makes any sense, why is this happening this way?" Yeah, so, I try really hard [to make my software maintainable], which may not be the smartest thing for [an] independent software developer, I don't know. The business reality is, maybe it's better just to get your stuff done and ship it. I don't strive for perfection—I know many people who write much more perfect and beautiful code that is so much more succinct than mine, but I try to at least be organized, like, I try really really hard to be organized. (Chris Parrish, Interview, March 2, 2012)

Another developer who was a contractor similarly mentioned to me that on one-off contracts, where he knows the project won't continue on afterwards, he often relaxes on some practices in order to get the app out the door as quickly as possible, even though it pains him to write less elegantly architected code. The short-term time pressures of hitting the schedule negotiated with the client can override longer term considerations, especially if the developer knows there won't be a subsequent version. These tensions speak somewhat to the competing "professional" and "market" logics at work among Cocoa developers (Qiu, Gopal, and Hann 2011).

Closure of the Dot notation controversy, and Swift

To a large extent, the dot notation controversy has achieved closure. The newcomers, and Apple, have won. Partly, this is because of Apple's asymmetric power vis-à-vis the community—its sample code uses dot notation, and most newcomers have readily taken it up. Partly, this is also due to the rapid expansion of the community in the wake of the iPhone App Store. The vast majority of newcomers are already used to dot notation from other languages, and have no accumulated years of experience to prejudice them against its use. Lastly, from a purely pragmatic point of view, even programmers who publicly have been against it in the past have begun to convert to its use—some purely because it can be more convenient to type long chains of method calls using successive dot operators rather than nesting bracketed expressions inside each other.

```
foo.bar.baz.qux = 10;
[[[foo bar] baz] setQux:10];
```

This code comes from the blog post by Jon Reid, hyperbolically titled, "Dot Notation in Objective-C: 100% Pure Evil," written June 3, 2012. (Reid 2012) The terms foo, bar, baz and qux are meaningless expressions used by programmers as placeholders in examples in which their meaning is not necessary. Again, these two lines of code, the first in dot notation and the second in bracket notation, are equivalent. The first is easier to read than the second, which non-Objective-C programmers find ugly. The second, however, is explicit about what the code is doing—sending messages. The object returned by each nested expression is then sent the next message. The dot notation expression also makes it difficult to differentiate if any of the items inside are method names, objects, or C structures. However, it is also much easier to type, even though the Xcode IDE helps with bracket notation by automatically inserting the ending bracket when the opening one is typed.

Reid has subsequently come around to dot notation. In a later post from September 17, 2013, "In Which I Embrace Dot Notation," he writes, "Dot notation *is* easier on the eyes. I've always admitted that. What I failed to realize before is that it isn't just a matter of "aesthetics." It's more readable. And readability is super-important." He also notes another reason, however: "Somewhat related to this: dot notation is mainstream. By opposing it, I was going against the flow… So by adopting dot notation, I hope my code will be less distracting, so that you can focus on the principles I'm trying to show and not go all SQUIRREL! [making a reference to the Pixar film *Up*, in which a talking dog constantly gets distracted by squirrels]" He concludes: "What about you: have you ever switched sides in a coding holy war?" (Reid 2013)

While there may still be holdouts among the oldtimers, Reid's comment shows that the vast majority of Cocoa and iOS programmers use dot notation pragmatically, and are not overly concerned about it. Only a few stalwarts, seen as opinionated outliers, still hold out. Aaron Hillegass himself has deemphasized his earlier stance, noting his objections were primarily pedagogical. At a talk he gave in

June, 2012 at an Apple Store during WWDC, Hillegass explained that when dot notation was first introduced, he had tried to replace all of his property accesses with dot notation, keeping method calls that indicated behavior in bracket notation. The problem was that not all of Apple's Cocoa APIs had been consistently updated in way such that attributes that ought to be implemented as Objective-C properties actually were. An example he gave was that of an NSArray. An NSArray is an Objective-C data structure (defined in Apple's Foundation library—the NS prefix is a reference to NextStep where the library originated) that holds any number of items, which can be accessed with an index. For example, to access the first object in the array, a programmer would could either use the older, more explicit message `[array objectAtIndex:0]` or the newer, terser syntax, `array[0]`. (Most programming languages count starting with 0, not 1). To get how many items are actually in the array, a programmer just needs to send it the `count` message. An NSArray's `count` is an aspect of state that somebody might want to know, not a behavior that the array can do. It ought to be exposed through Objective-C's properties feature, but it isn't. Objective-C still allows a programmer to use dot notation for regular message sending if the method to be invoked has no parameter inputs. However, this introduces an inconsistency in usage. Apple recommends that developers use dot notation only with properties (in other words, things declared explicitly using the `@property` declaration in code), and not to call methods that perform actions. But it also recommends that developers use it to access aspects of an object's state. However, here is a case in which its interface to NSArray's `count` has a state-like "property" that is not formally declared using `@property` syntax. So one has a dilemma, if one tries to maintain consistency. Does one follow the letter of Apple's rules, or the conceptual spirit? Following the conceptual spirit would require a breaking of the formal rule—accessing `count`, a state-like property, using dot notation. Some developers, notably Luke Adamson, have taken this more liberal stance towards dot notation, while others, such as Brent Simmons, have taken the more literal approach. Aaron Hillegass, finding neither option fully consistent, abandoned dot notation entirely in his own code. For Cocoa developers who value

consistency, Apple's own inconsistency in this regard, due to the historical legacies of its older interfaces, has left them with a difficult quandary.

Recent developments have rendered the debate even more irrelevant. At WWDC 2014, Apple introduced a new programming language, called Swift, that would be interoperable with Objective-C, incorporate many features of recent, more "modern" programming languages, and use a syntax more like the most popular languages in the industry, but would support the same idioms, expressive naming practices, and design patterns of the Cocoa frameworks and the Cocoa community. Swift has dropped interoperability with C, and opted for a uniform syntax for both object-oriented method calls and procedural function calls. Method calls and property access both now use dot syntax and look like Java syntax, with one exception: methods support named parameters. Thus, unlike the previous attempts to convert method calls to dot syntax, which failed because they did not support the expressiveness of named parameters, Swift, as a new language, has been designed from the ground up to support this. Because Swift no longer has to be backwards compatible with C, it has been able to update its notion of "structure" to be much more object-oriented—structures can have methods attached to them just like objects, and both can have properties, which are no longer abstractions over method call implementations. Thus, the problems that objectors had to the ambiguities of dot notation in Objective-C are no longer applicable in Swift.

The reaction to Swift from Cocoa and iOS developers has been ecstatic. Some quarters of the Apple community had been vocal that Objective-C, as a language built in the 1980s, has been getting long in the tooth and, despite several feature updates in recent years by Apple, would eventually hit a wall in terms of its ability to keep up with more "modern" programming languages (Bruchez 2014; Siracusa 2005a; Siracusa 2005b; Siracusa 2005c; Siracusa 2010). These voices have been vindicated, as Swift has seemed to be a grab bag of language features from seemingly every innovative language in current widespread use in industry. However, there has been trepidation among some that, because Swift, like some of these more popular languages, supports much more terse code, it will undermine the values of

expressiveness and explicitness in the Cocoa community. In addition, message passing as a concept is deemphasized. Not only does Swift use dot notation, but methods as well as procedural functions are labeled by a common keyword, "func," which emphasizes that the two are analogous rather than completely different concepts. In a number of ways, Swift takes several steps away from Objective-C and towards C++.

In addition, because the Swift language is new, and only a few hundred developers inside Apple have had significant experience using it, standard idioms in its usage have not yet developed, and there will be a shakeout period when conflicting practices contend with each other in the community. A developer in one podcast mentioned that there is no Aaron Hillegass for Swift, to recommend proper Swift coding style (Gruber et al. 2014; Ritchie et al. 2014)—there is no recognized expert authority on Swift that the community can turn to for advice outside of Apple itself. It remains to be seen how coding practices will change in the Cocoa community with the advent of Swift.

Conclusion

As we have seen, Cocoa programmers care a lot about coding practices, and actively debate, advocate, or denounce them, for multiple reasons. Cocoa programmers care about the maintainability of code, a concern arising out of software engineering and the response to the perceived persistent "software crisis." This is a concern with professionalism in software practice: a professional developer needs to become disciplined and internalize such practices in order to be able to collaborate with others at work or participate in the code sharing of community forums. And to the extent that it helps make software easier to fix and extend with new features, it will, in the long run, save time and effort for the programmer (which means saving time or money for one's employer, or oneself, if one is an indie developer.) Software that works with less bugs is also, to a large extent, a vital component of "quality," and incorporating practices to improve software "quality" will also be of benefit to one's users. Thus, a professional Cocoa developer is seen to

be one that strives for quality in one's work. In this way, software engineering practices are not seen as antithetical but as vital to the development of programming skill—discipline is necessary for such cultivation. Moreover, we can see that the embodiment of such practices is thus crucial for a Cocoa developer's identity—which is why boundary work inevitably becomes involved in arguments about practice. Professional and community identity are at stake in the minutiae of coding style.

CONCLUSION

Friday, November 15, 2013. I am sitting in a large conference room in a hotel near the Atlanta airport. I am surrounded by programmers for Apple iPhones and Macs, as I am attending CocoaConf, a traveling conference devoted to Apple's Cocoa development technology. In the front of the room, on a makeshift stage, sits a portly man with a ukelele at a microphone. He is backed by two younger men on electric guitars. The man at the microphone is singing a song, which he wrote himself, called "The Liki song" which is in Hawaiian style. The chorus goes like this: "Oooh, oooo-ooooh, oooh-oooh. Minawana meika la'a likiko…" The refrain is actually a pun—faux Hawaiian, which when sung or spoken, sounds like the English phrase "Me no wanna make a lotta leaky code." The song is actually about memory management practices in Objective-C programming on the Mac and iPhone. The deep technical content of the song is married to witty turns of phrase and observations about programming practices, and the bugs that can occur if one does not follow those practices, eliciting knowing laughter from the audience of programmers. Like the audience of any song, this song speaks to its experience—in this case, the experience of doing memory management while programming using Cocoa. And in speaking to that experience, deeply immersed in the technical practice of programming, members of the audience come to feel as if they are part of a special, insider group—a culture for which knowledge of, and appreciation for, Apple's Cocoa technologies and the practices of using them, binds them together in solidarity with the others in the room.

The performer is James Dempsey, and his backup band, the Breakpoints. James Dempsey is a former Apple employee, once an engineer on the Cocoa framework team (the same team I was a member of, although our tenures did not overlap). In 2003, Dempsey gave a talk at Apple's Worldwide Developer Conference (WWDC) to explain the concepts behind Apple's Enterprise Object Frameworks (EOF), a precursor to the CoreData framework on OS X and iOS. Apple's WWDC is an opportunity for Apple to teach developers about new technologies and APIs in the

latest versions of its operating systems, but some talks are about already established technologies, and are more explicitly conceptual and pedagogical, intended for newcomers to the platform. As a part of his 2003 talk, Dempsey decided to end with a little song about EOF that he wrote himself, and performed it live with his guitar. I recall being in the audience for this performance, and remember the enormously positive reception Dempsey received upon finishing the song. Needless to say, in every subsequent WWDC in which Dempsey gave a talk, he would always perform a song, and word soon traveled that his songs about Cocoa programming were a highlight of the show, a break in the usual dry technical presentations.

Dempsey left Apple in 2011 to "go indie," writing an app named WALT that allows fans of Disney movies to check off the films they've seen. Despite becoming an independent freelancer, Dempsey has maintained involvement with the Cocoa developer community. He has become a regular speaker at CocoaConf, a traveling community-organized conference run by and for Cocoa developers. He also continues to perform at WWDC, though no longer part of the official Apple-sponsored proceedings, but in 2012 at a special venue at a local San Francisco bar after hours, aided or backed by many former colleagues and friends from Apple (including his and my former manager, Ali Ozer.) Members of his backing band, the Breakpoints, are basically anybody with musical talents who wants to come join him on stage, and among their ranks include both Apple engineers as well as frequent CocoaConf speakers and participants. In Atlanta, Dempsey was joined by the iOS Developer Meetup organizer Rusty Zarse, and Black Pixel employee, book author, and Atlanta CocoaHeads member Brandon Alexander, on electric guitar, as well as Jonathan Penn, a CocoaConf speaker, and Dan Steinberg, the organizer of CocoaConf. James Dempsey and the Breakpoints performances have become a staple of the CocoaConf convention, and the band even sells a T-shirt. An album of Dempsey's songs, including "The Liki Song" and the mournful lament, "Almost Dropped My" [iPhone 5], is available on iTunes.

James Dempsey's songs can be thought of in a similar genre as "filk," folk songs written by science fiction and/or fantasy fans about their favorite fictional

settings and characters, and because they work only in reference to these original texts, speak to an already acculturated, "in the know" audience ("Filk Music" 2014). Like filk, Dempsey's music speaks to a very narrow geek subculture, but in his case, his songs are about programming for Cocoa and require at least a rudimentary level of experience programming in Cocoa to fully appreciate.

To not only attend but appreciate a performance by James Dempsey and the Breakpoints requires an engagement and participation in the knowing experience of its insider puns, jokes, and references, which depend on the technical knowledge of Cocoa. At the same time, it involves a deep understanding of the shared, collective and affective appreciation the audience has for the subject of Dempsey's songs, which is the experience and practice of Cocoa programming. Dempsey's songs are often pedagogical, sometimes quasi-historical, and often carry a normative message about best practices in Cocoa programming. Yet their packaging in an entertaining, humorous, and insider-specific artistic format softens the message, and helps bind together the community of listeners to whom his songs speak. Dempsey's songs are not just a collection of notes and lyrics about an arcane technical topic. To fully understand them requires an understanding of the technical practices of the Cocoa community that it speaks to, the norms and values that drive it, and the affective connection to Cocoa among the developer community that is powerfully enacted in Dempsey's performances and his audiences' reactions.

This dissertation is a thick description of the cultural world that produces the artifact and experience that is a song and performance by James Dempsey and the Breakpoints. It tries to answer the following questions: what motivates Cocoa developers' deep devotion and commitment to Apple and to Cocoa technology? Why do they believe Cocoa is a superior programming environment? What does it mean to be a proper member of the community, a "good" Cocoa programmer, both technically and in a moral sense? How do people become Cocoa programmers and join this technical subculture? How did this community get this way historically?

In answering these questions, I have shown that the culture of the Cocoa community can only be explained by simultaneously examining the values, norms, ideology, identity, affects, and practices, of the community and the way each of these interacts with each other and with Cocoa technology, in what I call a "techno-cultural frame," a concept which draws on but also extends Wiebe Bijker's notion of "technological frame," discussed in the introduction and chapter 5 (Bijker 1995). I extend Bijker's term to include ideology, moral order, and affective experience. The constellation of interacting elements that make up the "techno-cultural frame" of Cocoa programming impacts users through their experiences with the apps they buy and the app market itself, and the rest of the software industry through the interaction of the discourses and practices of Cocoa developers with other programming platforms.

My decision to combine the concepts of "technological frame" and "ideology" is related to the Science and Technology Studies literature this dissertation is in conversation with. This dissertation follows the classic Sociology of Scientific Knowledge (SSK) and Social Construction of Technology (SCOT) tradition, in getting "Inside Technology" and opening up technology's black box in order to show the deep intertwining of the social, cultural, and technical. I am purposely invoking this literature when I refer to the relationship of culture and technology as a "seamless web" (Bijker, Hughes, and Pinch 1987). Nevertheless, the SCOT literature does not address the role that ideology can play in shaping technological cultures. More recent STS literature does address ideology, however, from Gusterson's study of nuclear weapons scientists, to Kunda's study of a computer company, to Fred Turner's work on the counterculture in computing, Stewart Brand, and Burning Man, Malaby's ethnography of Linden Lab, and Coleman and Kelty's work on open source (Gusterson 2008; Kunda 1992; Turner 2006; Turner 2009; Malaby 2009; Coleman 2013; Kelty 2008). Other STS literature has focused on the affective experience people have in relation to technology, including Coleman, Sherry Turkle, Natasha Dow Schüll, and others (Coleman 2013; Turkle 1984; Turkle 1995; Turkle 2011; Schüll 2012). Rarely, however, do these works get inside the technology at a deep

level of technical detail in the way the SCOT literature does (with the exception, perhaps, of Schüll). My contribution is to bring these two approaches together, to show how a deep technical understanding of technology can reveal how deeply ideology and culture are embedded in it, and how it animates the way people interact with it in their practice.

The relationship of third party Apple developers to Cocoa technology is an excellent case study for this, because of the intense affective connections developers have to Cocoa. Moreover, it is important to understand this culture in its own right, as the committed core of the third party Cocoa developer community acts as an ideological extension of Apple itself, evangelizing for Apple technologies and practices among users, the public, and the programming world at large. Apple is among the most ideological of all information technology companies. Even prior to its current success, its influence far exceeded its actual market position, and its contemporary dominance has been largely predicated on its focus and rigid adherence to its sense of mission and values and its ability to enroll its users and developers in its way of seeing the world. By participating in technological production using Apple's tools to make technologies of their own, third party developers take part in the same techno-cultural world that engineers at Apple itself engage in, and magnify the influence of the Apple way many times more than Apple could do on its own.

Explaining this system of beliefs and values, the cosmological component of the techno-cultural frame of Cocoa programming, is the aim of chapters 1, 2, and 3. Chapter 1 focused on the affective and ideological aspects of this cosmology. In chapter 1, I showed how Cocoa programmers experience a deep pleasure in making software using Cocoa technologies. Chapter 3 showed how this pleasure is rooted in the way Cocoa is designed by engineers at Apple: it is created to be consistent, yet flexible, and provide more utility for less effort, allowing developers to achieve their goals more quickly and with less frustration. Chapter 1 also showed that Cocoa programmers share with Apple the goal of providing users with user friendly and aesthetically pleasing experiences with their software. Their pursuit of this goal is

considered to be their passion and vocation, not simply a job; their identity is that of artisans who are making fine pieces for their customers. It is in order to maintain control over their creative production that Cocoa programmers consider the highest, purest expression of their identity to be that of the indie developer, who works only on what software they want to create, according to their own visions. To do this, they must be in charge of their own businesses, not be the employee of a larger corporation, nor do they take funding from investors or venture capitalists, as their goal is not to create a powerful technology company but simply to sustain a small business that will allow them to continue doing what they love for a living. Chapter 3 showed how the increased productivity Cocoa provides to developers allows them to be indies. Cocoa provides significant built-in functionality and conveniences to developers, allowing them to delegate routine functions to Cocoa libraries and concentrate on their applications' unique and original contributions. Apple's libraries do half the work for them, allowing them to compete with the offerings of corporate software companies like Microsoft with hundreds of times the manpower. Being an independent developer is made possible by this division of labor. Indies' ecological dependence on Apple is not seen as a contradiction because Apple is not a direct employer, and does not directly control the kinds of apps that indies make, although its history of appropriation from the community means that developers must be careful. Nevertheless, ideologically indies see Apple as engaging in the same project as themselves—making useful, usable, and pleasurable technologies to empower users, enrich their lives, and improve society. In their eyes, Apple, with its tight control over its own artistic creations, is like a very large indie company. The mythology of the company and of Steve Jobs, its legendary founder, serves to cement this connection; in Jobs, perennially struggling against the mediocrity of the wider industry to realize his true radical and spiritual vision of computing, they see themselves or what they long to be. Chapter 2 showed the extent to which NeXT and later Cocoa developers, motivated by this vision and devotion to NeXT/Cocoa technology, struggled through the 1990s to keep making NeXT/Cocoa software in order to keep the dream alive, even if they had to take contracts from Wall Street or the FBI to do it. NeXT's acquisition by Apple, and the technology's subsequent use

as the basis for not only Mac OS X but iOS, was viewed as the arrival of the true believers to the promised land after a decade in the wilderness.

To attain the purity and consistency of artistic vision and design, however, requires control, according to the Cocoa cosmology. In chapter 3, I described how Cocoa developers understand that they live in a beautiful, curated, but walled, garden, and that they have chosen to give up certain freedoms in return for this experience. They have learned to trust in Apple's technology to give them this experience. This attitude is not acquired overnight, however. Because of Cocoa's high learning curve, novices do not experience these pleasures initially, but must struggle through learning unfamiliar new concepts and practices (such as design patterns) that do not make sense at first while they are learning them piecemeal. Only after the entire system has been learned and experienced holistically do the benefits of Cocoa emerge. Cocoa developers have described this experience as a conversion, and what is apt about the metaphor is that it is not only an intellectual transformation but also deeply affective and aesthetic. Developers are left feeling as if this is the way programming should always be done, and are eager to proselytize to others in order to share this experience. This deep quasi-religious feeling is the root of the devotion Cocoa programmers have for Cocoa technology, and why they place their trust in Apple, the creators of Cocoa.

Chapter 4 showed how novices are trained to begin seeing Cocoa in this way as they learn how to program iOS apps at the training company, Big Nerd Ranch. Through the disciplinary mechanism of manually typing in code, diagnosing and fixing those problems, all while under a relentless pace in the face of a daunting volume of Cocoa APIs to learn, novices work through frustration and helplessness, and as they get their programs working, begin to understand the reason for the normative practices they are learning: these practices are necessary in order to help them get their programs to work. Big Nerd Ranch instructors seek to teach such technical-normative lessons by calling preferred practices "stylish," which reinforces the social nature of these practices. Students are exhorted to follow "stylish"

practices as much for the importance of conformity to community standards as for their technical, instrumental benefits.

Chapter 5 explored the organization of this community and how such normative technical practices are spread and maintained. The community is organized both as an online networked public, through the blogosphere and Twitterverse, and physically as a collection of local developer clubs centered in cities like Seattle, which convene periodically at conferences, including Apple's WWDC. Most of the community's activities involve the sharing of the technical knowledge of Cocoa practice, which includes normative prescriptions. This certainly happens at WWDC, where Apple actively enrolls participants in its normative and ideological messages, but it also occurs in community-run gatherings as well, where community norms are enacted in sharing. Although at the highest level, the Cocoa community is large and expanding tremendously in the wake of the iOS App Store, the inner core of the community is rooted in the personal, intimate friendships formed over beer at local club afterparties. This personal connection is responsible for the moral economy of collegiality among indies that militates against cutthroat market competition between them, and facilitates sharing of knowledge and resources. The collegial sharing among the community, in particular the Seattle Cocoa community, is responsible as much for the success of the indies as Cocoa technology itself, despite the rhetoric of independence and individuality. Personal connections to engineers within Apple also help to sustain the sense of shared mission with the company, resolve tensions caused by misguided Apple corporate policies, and serve as a conduit for influential third party developers to make their concerns known to the company and possibly influence Apple's actions.

Chapter 6, which dealt with the controversy over Apple's introduction of dot notation syntax into Objective-C, illustrates the normative and technical tension between the practices held by the core of the indie Cocoa community and Apple, as well as the much larger population of new iOS programmers whose practices the older community felt a need to guard against. Chapter 6 and chapter 2 draw on and respond to the literature in the history of computing and software, particularly that

which focuses on the software crisis, software engineering, and the professionalization of programming (Mahoney 2002; Mahoney 1990; Mahoney 2004; Mahoney 2008; Ensmenger 2009; Ensmenger and Aspray 2002; Ensmenger 2010; MacKenzie 2001; Abbate 2012; Slayton 2013a; Zepcevski 2012; Haigh 2010). In chapter 6, I showed how arguments for and against certain practices were rooted in the concerns over software maintainability and programmer productivity that have been an ongoing concern in software engineering since the first perception of a software crisis in the late 1960s. Chapter 2 showed how such concerns motivated the creation and design of the Objective-C language and NeXT's object-oriented frameworks. Object-oriented frameworks, in conjunction with object-oriented design and coding practices, improved programmers' productivity by helping them make their software more maintainable. These productivity benefits made the NeXT/Cocoa frameworks advantageous for creating a wide range of applications, from financial services to mobile games, and was why Cocoa's adherents considered it to be technically superior to alternative programming environments. Although the purpose of chapter 2 is to provide the historical context for chapter 3, as a whole this dissertation is not primarily a work of history, and does not directly address questions central to the history of computing. Nevertheless, I am responding to the debate raised in this literature over the extent of managerial deskilling, disciplining, or routinization of programming practice through the methodologies and technologies of software engineering, structured programming, and object-oriented programming. I hope to dispel the specter of Philip Kraft's deskilling thesis (Kraft 1977). While use of these object-oriented environments disciplined programmers to think and solve problems in specific ways, by encouraging or requiring techniques such as modularization, loose coupling, and code reuse that could help manage complexity and improve maintainability, the incorporation of these methods into programming practice and object-oriented technologies did not deskill but became a new set of skills to master, part and parcel of the new professional programming worldview.

Nathan Ensmenger has shown how software engineering discourses and methodologies became part of the professional practice of programmers (Ensmenger 2010), and my exploration of the norms of practice of Cocoa developers in chapter 6 shows how deeply these discourses have become embedded. Experienced professional iOS developers seek to create quality, maintainable software because that is what good developers do, often in spite of managerial pressures to do cut corners and rush an app to market. Nevertheless, Ensmenger has pointed out that programming has never fully professionalized to the same extent as mechanical, civil, or electrical engineering (Ensmenger 2010). There is no requirement to be a formal member of a professional society such as the Association of Computing Machinery, nor are formal certifications required for the vast majority of programming jobs. The fact that many programmers are self-taught or dropped out of college and yet still find gainful employment speaks to programming's lack of status as a true profession. To an extent, this is something that programmers actually celebrate, as part of a vision of democratization of and participation in technological production in the DIY Maker/hackerspace movement. This popularist rhetoric underlies the utopian view of many indies that millions of small developers will overthrow the corporate software monopolies. Nevertheless, it is precisely because the formal boundaries of the profession are so fuzzy, that more professional developers feel the need to do verbal boundary work against those who they see as amateurs dragging down the reputation of their profession and the quality of their work. The fear that the Cocoa community will lose its values and its identity underlies the vitriolic rhetoric against dot notation in Objective-C. As Coleman pointed out, the hacker ethic has both popularist and meritocratic/elitist components existing constantly in tension (Coleman 2013), and this is precisely because the programming profession has been more inclusive than exclusive in comparison to traditional engineering.

Michael Mahoney exhorted computer historians to connect the history of computing more closely to the questions being asked by the history of technology, especially by looking at the context of practice in shaping the construction of computing technologies. He particularly wanted historians to examine software,

because it was in legacy software that older practices and social organization could be excavated (Mahoney 1988; Mahoney 2008). In this dissertation I reflect on the materiality of software and its relationship to the normative practice of its use, especially if that use is the further creation of software, and I show that, contrary to the deterministic impact model of technology, software technology and the practices of its users co-produce each other, a key insight of the technology studies literature and especially the Social Construction of Technology (SCOT) (Bijker, Hughes, and Pinch 1987; Bijker 1995; Collins and Pinch 1998a; Pinch and Trocco 2002; Kline and Pinch 1996; Bijker and Law 1992; MacKenzie 1996; Mackenzie 1990; MacKenzie 2001; Kline 2000; Oudshoorn and Pinch 2003). As explored in chapter 2, object-oriented languages and tools were created in order to encourage or enforce structured practices that would manage complexity and improve stability and maintainability. Nevertheless, object-oriented technologies by themselves are not a panacea, and programmers that simply adopt object-oriented languages without altering their design practices do not reap the promised productivity gains. This was the warning given by Bruce Webster in his book, *Pitfalls of Object-Oriented Development* (Webster 1995). Proper, disciplined use of object-oriented technologies, involving new ways of thinking about software design and implementation, must be acquired. This learning occurs socially and normatively through socialization into the norms of practice of the programming community, and because much of this knowledge is tacit, it must be learned through doing, rather than read from a book, as we saw in chapter 4.

Nonetheless, disciplined practices can be facilitated by object-oriented technologies such as languages and libraries, although such technologies cannot fully control what programmers do with them. Libraries such as the Cocoa frameworks are not designed in a vacuum but with an eye to encouraging best practices. Design patterns such as Model-View-Controller and delegation were built into the Cocoa frameworks. Thus, programmers who wish to learn Cocoa must learn to work in the way Cocoa demands, which includes learning these design patterns, as we saw in chapter 3. This is why Cocoa takes more effort to learn, because it is not merely

syntax or APIs but a different way of thinking and doing, involving concepts and practices that take time and experience to internalize. Over time, as practices continue to develop, the designers of the technology continue to react to such changes as they modify the technologies to better suit their users' practices. As discussed in chapter 4, the Cocoa engineers at Apple gradually began to assume that the majority of its developers followed conventional practices and modified the Cocoa frameworks to take advantage of these conventions, allowing them to introduce innovations such as Automatic Reference Counting. Changes in the technology respond to changes in practice, and vice versa. On the flip side, chapter 6 showed that the introduction of dot notation into Objective-C changed idiomatic practice in the developer community, though not without considerable cultural resistance from older Cocoa developers. In this way, Cocoa technology and the practices of Cocoa developers mutually shape each other.

I said in the introduction that the techno-cultural frame of Cocoa software development involves affect, ideology, identity, community, practices, and technology together in a seamless web of mutually shaping influences. Each of these aspects involves the others and cuts across the individual chapters of this dissertation, which illustrates how deeply culture and technology are intertwined, the key insight that the field of Science & Technology Studies has contributed to our understanding of technological development. This analysis of the Cocoa developer community makes it impossible to see technology as a force developing autonomously from society and impacting it in a straightforward, billiard-ball fashion. The values and practices that shaped the design of Cocoa technology by NeXT and Apple engineers are shared by the third party developers that use it to make their own software, because the extent of the Cocoa community crosses the boundaries of Apple itself. Cocoa software is what it is because of the values of the people that make it.

Future Directions

This dissertation has touched on a number of themes relevant to the culture and values of Cocoa development, but more work needs to be done. Although

chapter 5 discussed briefly the relationship between the Cocoa community and Apple, a deeper study of this relationship is necessary. To what extent does the community have influence on Apple? How are tensions between Apple and the community worked out? What happens to developers who are permanently alienated by Apple's actions? The answers to some of these questions may require not only additional research, but access to people at Apple itself, which is difficult to acquire.

This dissertation has focused mainly on the Cocoa developer community. As such, it heavily privileges the views and opinions of self-identified Cocoa developers, particularly those in the core of the Cocoa community, the indie Mac developers who dominate its online discourse. This project could benefit by getting a wider breadth of voices, however, especially from the many newer, and more peripheral iOS programmers that have come into the community since the iPhone, many of whom may work for larger corporations or venture-funded startups. This might alter the view of the ideology of the Cocoa community, by examining those who might hold more traditional programming jobs or subscribe to the Silicon Valley growth narrative. I did interview a few of these developers but did not have enough time to fully analyze their data or draw any significant conclusions about them in comparison to the more highly-included core Cocoa developers.

Insight could also be gained from comparing Cocoa developers to developers who share in Apple fandom but do not share in its love for Cocoa technology itself, in other words, Carbon or classic Macintosh developers. In the period from 2001-2008, for example, there still existed a sizeable group of developers who used the alternate Carbon toolkits to program applications for Mac OS X. By 2012, with the iPhone and its Cocoa-based API ascendant, and support for Carbon dropped for several years by Apple, it had become difficult to locate former Carbon developers for this study. Moreover, to keep the scope of this project manageable, I decided to focus recruitment efforts primarily on Cocoa and iOS developers, especially as I had easy access to them. Additional future work is required to find former Carbon developers to participate in interviews.

This project could also benefit from comparison to other developer communities, to compare and contrast their values, norms, ideologies, and practices, in order to locate what is singular and unique about the Cocoa community. This could include programmers for Windows, Android, Java, Ruby, Python, and other platforms or programming environments. My interviews did include three current iOS developers who had previously been Windows developers, including one former Microsoft employee. More work needs to be done to sift through this material for areas of congruence with themes that appear in this dissertation. I also took a course on Android programming at the Big Nerd Ranch, and spoke to a few of the Android developers there. However, the majority of the Android programmers at Big Nerd Ranch at the time were also iOS programmers, and favored iOS. This has likely changed as the company has grown, and additional research with Android programmers who acknowledge hostility to Apple and iOS would be interesting to compare their ideological commitments. As my interviews with Cocoa developers who have had extensive experience with other programming languages and communities suggest, there are many shared values between programmers in general, and many of the norms I have described in this dissertation could apply to other programmers. Nevertheless, there are some cultural specificities among Cocoa developers, in particular, their attachment to Apple, its technology, and its mythology, that sets them apart. Comparative work could help illuminate more precisely what these specificities are.

At the end of 2012, Big Nerd Ranch merged with Highgroove, a company focused on developing web services using the Ruby programming language. Some of my participants, notably Robert Walker, noted that Ruby and Objective-C shared many similarities, for example, devotees of both languages consider them elegant. However, another participant, Rusty Zarse, who was both a Ruby and iOS programmer alerted me to key cultural differences in the communities. The Ruby community is committed to open source participation, for one. Additionally, it adheres rigorously to a technique of component-based testing known as “unit testing.” Zarse informed me that use of unit testing, as a best practice, was not only expected,

but normatively enforced in the Ruby community—one would not be able to ask for help from someone without first writing a unit test. This is made easy because sophisticated tools and software infrastructures for unit testing are built into the Ruby libraries and well supported by the community. On the Apple side, however, unit testing has only recently acquired built-in support in Apple's Xcode development tools at the level of sophistication found in Ruby. Moreover, unit testing is not regarded as a universally required practice by the majority of the Cocoa community; while many advocate for it, some leading developers, like Wil Shipley, have opined that it is a waste of time (Shipley 2005). Future work will compare the different norms for best practices between the Ruby and Cocoa communities, especially with regard to the importance of unit testing for developing quality software.

Gender has not been a central analytic framework of this dissertation, though it has arisen as a minor theme in the cosmology and social organization of the Cocoa community, specifically in chapters 1, 4, and 5. Ideologically, the figure of the indie is gendered masculine, and almost all of the famous and influential Cocoa indies are male, as is the vast majority of the rest of the Cocoa community. The gender imbalance at WWDC was estimated by one of my participants as 30 to 1, and the WWDC afterparty culture caters to male developers. In my research at an iPhone startup in 2008, I witnessed a fraternity atmosphere, in which there were no women programmers. Even at Big Nerd Ranch in 2011, there was only one woman programmer, an intern, though by the end of 2012, with the company's expansion and then merger with Highgroove, there were significantly more, though still vastly outnumbered by men. Nevertheless, with the expansion in the community, the situation is improving. I did notice that in Big Nerd Ranch's classes, there sometimes were a number of women, including one indie developer. I also met women who attended Seattle's NSCoder night, a night when local developers met at a coffeehouse to help each other code; these also included at least one indie. Highgroove hosted a women's Ruby developer group. The Atlanta iOS Developer meetups included women, and in early 2013, I discovered through a mailing list that

the Atlanta iOS Developer Meetup had also started a separate women's group, although the decision to do so had created some tensions within the overall group. While women's participation in Cocoa programming is improving, there is still much to be done. In a future project, I would like to study more closely women Cocoa developers, such as those who have formed their own group in Atlanta, to examine the challenges they face in a masculine dominated culture, and illuminate both their contributions and the ways those contributions are marginalized in the rest of the community.

Although this dissertation has touched on the history of NeXT the company, NeXTSTEP the operating system and development environment, and the third party developer community of NeXT, the number of interviews I have been able to gather from former NeXT developers and employees remains small. Writing a history of NeXTSTEP is difficult at this time, due to most internal documents belonging to Apple. However, more former NeXT employees are retiring or leaving Apple. One long term project of mine is to compile an archive of oral histories of former NeXT employees and developers, in order to make such a history possible. NeXT is important for understanding more than just Apple and its later development. Certain NeXT technologies influenced the development of the Java programming language, which became an industry standard in web programming. NeXT software technologies could be precursors to various ideas taken up by later web programming technologies, including Ruby. Additionally, as we saw, NeXT was used extensively by Wall Street, and following the use of software technologies in the finance industry could be a very revealing future project.

Finally, developments in the Apple technology world, and in the Cocoa community, have continued in the three years that it took to complete this dissertation, and these developments could change the norms and practices of Cocoa developers. As this dissertation is being completed, Apple will have shipped its new wearable computer, the Apple Watch, which contains its own SDK for app development. In addition, the iOS SDK has been expanded with new frameworks, HomeKit, HealthKit, and ResearchKit, that provide new APIs for developing apps

that track personal information in conjunction with other devices. These frameworks show that Apple is heavily invested in the recent technology trend known as the "Internet of Things" (IoT). The "Internet of Things" refers to how computers and sensors are increasingly being embedded into everyday objects, in the home, in automobiles, and worn on the body. These "smart" appliances record data constantly and usually send that data to a remote server to be aggregated with data from millions of other devices, giving institutions the ability to mine this "Big Data" to discover patterns that could not have been otherwise achieved. For individual consumers, the constant monitoring of data allows them to optimize their own lives—say, their home energy usage, or their exercise regimen. This trend towards self-surveillance has acquired its own term, the "Quantified Self." A recent survey of mobile developers shows that more than half of app developers are working on IoT-related projects (VisionMobile Ltd 2015). Many such apps were released prior to Apple providing any special support, but with the HomeKit, HealthKit, and ResearchKit libraries, Apple is now providing direct system-level support for these kinds of applications, making it easier than ever to develop them, while also using them to enforce privacy protections to prevent third party developers from misusing users' data. HomeKit is being built for home automation devices, such as smart thermostats and remote lighting controls. HealthKit supports Quantified Self applications such as measuring how many steps a person has taken in a day. ResearchKit allows users to opt-in to studies that use data collected on iPhones for medical research that involves massive data sets. All of these frameworks come with Objective-C APIs and are built in the Cocoa idiom, using design patterns such as delegation. It will be interesting to track how the techno-cultural frame of the Cocoa community intersects with the Internet of Things among developers working on apps with these new frameworks.

As was discussed in chapter 6, Apple introduced Swift, a new programming language for use with Cocoa, in 2014. In a significant way, changes made to Objective-C from 2007 to 2014, including the introduction of dot notation, paved the way for its eventual replacement by Swift. Swift syntax reverses some of the

normative preferences of Objective-C code: verbosity is out, terseness is in, dynamic typing is out, static typing is in. Many of the accumulated idioms of the existing Objective-C culture no longer apply, though some may have been incorporated into Swift itself. Nevertheless, the community is presented with a new slate in which to develop a new set of norms and idiomatic practices for Swift code that take some cues from Objective-C but may also branch out into completely new directions. Moreover, Swift itself is still a work in progress at Apple, responding to feedback from the developer community on what aspects of its current design work and what do not. Nevertheless, Swift adoption has been very rapid considering how new it is and the fact that it is only available on Apple platforms. A recent survey has shown that as many as 20% of mobile developers are already using Swift. Although Google and Facebook have introduced their own proprietary languages Go and Hack, both primarily for server-side programs, neither language has seen the kind of adoption that Swift has had in less than a year of availability. Moreover, the survey also shows that the majority of Swift adopters are newer iOS developers, whereas older iOS developers with extensive Objective-C experience and existing code bases have been slower to adopt the language (VisionMobile Ltd 2015). This suggests that Apple is very much interested in Swift's appeal to the recent wave of new programmers in the iOS community, similar to its support of dot notation in Objective-C. Will this engender significant resistance to Swift among the Cocoa oldtimers? It will be very interesting to study how the community and its culture will develop in response to Swift.

BIBLIOGRAPHY

Abbate, Janet. 2012. “Software Crisis or Identity Crisis? Gender, Labor, and Programming Methods.” In *Recoding Gender: Women’s Changing Participation in Computing*, 73–111. Cambridge, Mass.: MIT Press.

Akera, Atsushi. 2007. “Voluntarism and Occupational Identity: The IBM Users’ Group, Share.” In *Calculating a Natural World: Scientists, Engineers, and Computers During the Rise of U.S. Cold War Research*, 249–74. Inside Technology. Cambridge, Mass: MIT Press.

Alicia Robb, Susan Coleman, and Dane Stangler. 2014. *SOURCES OF ECONOMIC HOPE: WOMEN’S ENTREPRENEURSHIP*. Ewing Marion Kauffman Foundation.

Althusser, Louis. 1999. *Ideology and Ideological State Apparatuses (Notes towards an Investigation)*. Alexandria, VA, USA ;;Cambridge, UK : Chadwydk-Healey,.

Amelio, Gil. 1998. *On the Firing Line: My 500 Days at Apple*. 1st ed. New York: HarperBusiness.

Ames, Morgan G., Jeffrey Bardzell, Shaowen Bardzell, Silvia Lindtner, David A. Mellis, and Daniela K. Rosner. 2014. “Making Cultures: Empowerment, Participation, and Democracy - or Not?” In *Proceedings of the Extended Abstracts of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, 1087–92. CHI EA ’14. New York, NY, USA: ACM. doi:10.1145/2559206.2579405.

Anderson, Benedict R. O’G. 1991. *Imagined Communities: Reflections on the Origin and Spread of Nationalism*. Vol. Rev. and extended. London: New York.

Apple Inc. 2013a. “Cocoa Core Competencies: Delegation.” *iOS Developer Library*. https://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/Delegation.html.

———. 2013b. “Start Developing Mac Apps Today: Design Patterns.” April 23. https://developer.apple.com/library/mac/referencelibrary/GettingStarted/RoadMapOSX/chapters/08_DesignPatterns.html.

———. 2013c. “Coding Guidelines for Cocoa: Code Naming Basics.” October 22. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CodingGuidelines/Articles/NamingBasics.html#//apple_ref/doc/uid/20001281-BBCHBFAH.

———. 2013d. “Coding Guidelines for Cocoa: Naming Methods.” October 22. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CodingGuidelines/Articles/NamingMethods.html#//apple_ref/doc/uid/20001282-BCIGIJJF.

———. 2014. “Apple - Press Info - Apple Reports Fourth Quarter Results.” October 20. http://www.apple.com/pr/library/2014/10/20Apple-Reports-Fourth-Quarter-Results.html.

———. 2015a. “Apple - About - Job Creation.” *Apple*. http://www.apple.com/about/job-creation/.

———. 2015b. “Apple - Press Info - App Store Rings in 2015 with New Records.” January 8. http://www.apple.com/pr/library/2015/01/08App-Store-Rings-in-2015-with-New-Records.html.
“Application Program.” 2015. *PC Magazine Encyclopedia*. http://www.pcmag.com/encyclopedia/term/37919/application-program.
Atlassian Blogs. 2012. “Between a Rock and a Hard Place – Our Decision to Abandon the Mac App Store.” *Atlassian Blogs*. February 16. http://blogs.atlassian.com/2012/02/between-a-rock-and-a-hard-place-our-decision-to-abandon-the-mac-app-store/.
Bardini, Thierry. 2000. *Bootstrapping: Douglas Engelbart, Coevolution, and the Origins of Personal Computing*. Stanford, CA: Stanford University Press.
Barlow, John Perry. 1992. “Will They Ever Come? Sales and Marketing at NeXT.” *NeXTWORLD EXTRA*, February.
Belk, Russell W., and Gülnur Tumbat. 2005. “The Cult of Macintosh.” *Consumption, Markets & Culture* 8 (3): 205–17.
Benkler, Yochai. 2006. *The Wealth of Networks: How Social Production Transforms Markets and Freedom*. New Haven [Conn.]: Yale University Press.
Benkler, Yochai, and Helen Nissenbaum. 2006. “Commons-Based Peer Production and Virtue.” *Journal of Political Philosophy* 14 (4): 394–419. doi:10.1111/j.1467-9760.2006.00235.x.
Big Nerd Ranch. 2014a. “Corporate Training.” *Big Nerd Ranch*. Accessed November 12. http://www.bignerdranch.com/we-teach/corporate-training.html.
———. 2014b. “Our Work.” *Big Nerd Ranch*. Accessed July 24. http://www.bignerdranch.com/we-develop/our-work.html.
Bijker, Wiebe E. 1995. *Of Bicycles, Bakelites, and Bulbs: Toward a Theory of Sociotechnical Change*. Cambridge MA: MIT Press.
Bijker, Wiebe E., Thomas P. Hughes, and Trevor J. Pinch. 1987. *The Social Construction of Technological Systems: New Directions in the Sociology and History of Technology*. Cambridge Mass.: MIT Press.
Bijker, Wiebe E., and John Law. 1992. *Shaping Technology/Building Society: Studies in Sociotechnical Change*. Inside Technology. Cambridge, MA: MIT Press.
Bilton, Nick. 2014. “The 30-Year-Old Macintosh and a Lost Conversation With Steve Jobs.” *Bits Blog*. January 24. http://bits.blogs.nytimes.com/2014/01/24/the-30-year-old-macintosh-and-a-lost-conversation-with-steve-jobs/.
Blumenthal, Karen. 2012. *Steve Jobs: The Man Who Thought Different*. 1 edition. New York: Square Fish.
Borsook, Paulina. 1992. “Striking It Rich: Oil Trader Leverages the Future on NeXT.” *NeXTWORLD*.
———. 2000. *Cyberselfish: A Critical Romp Through the Terribly Libertarian Culture of High Tech*. 1st ed. New York: PublicAffairs.
Bourdieu, Pierre. 1975. “The Specificity of the Scientific Field and the Social Conditions of the Progress of Reason.” *Social Science Information* 14 (6): 19–47. doi:10.1177/053901847501400602.

———. 1977. *Outline of a Theory of Practice*. Cambridge; New York: Cambridge University Press.
———. 1986. “The Forms of Capital.” In *Handbook of Theory and Research for the Sociology of Education*, edited by J. Richardson, 241–58. New York: Greenwood Press.
boyd, danah. 2006. “A Blogger’s Blog: Exploring the Definition of a Medium.” *Reconstruction* 6 (4). http://reconstruction.eserver.org.proxy.library.cornell.edu/Issues/064/boyd.shtml.
Brand, Stewart. 1968. “Whole Earth Catalog.” *Whole Earth Catalog*.
———. 1972. “Spacewar: Fanatic Life and Symbolic Death Among the Computer Bums.” *Rolling Stone*, December 7. http://www.wheels.org/spacewar/stone/rolling_stone.html.
Brooks, David. 2000. *Bobos in Paradise: The New Upper Class and How They Got There*. New York: Simon & Schuster.
Brooks, Frederick P. 1987. “No Silver Bullet Essence and Accidents of Software Engineering.” *Computer* 20 (4): 10–19. doi:10.1109/MC.1987.1663532.
———. 1995. *The Mythical Man-Month: Essays on Software Engineering*. Anniversary ed. Reading, MA: Addison-Wesley Pub. Co.
Brown, John Seely, and Paul Duguid. 1991. “Organizational Learning and Communities-of-Practice: Toward a Unified View of Working, Learning, and Innovation.” *Organization Science* 2 (1): 40–57. doi:10.1287/orsc.2.1.40.
———. 2000. *The Social Life of Information*. Boston: Harvard Business School Press.
———. 2001. “Knowledge and Organization: A Social-Practice Perspective.” *Organization Science* 12 (2): 198–213. doi:10.1287/orsc.12.2.198.10116.
Bruchez, Erik. 2014. “Erik’s Ponderings: Thoughts on the Swift Language.” June 3. http://blog.bruchez.name/2014/06/thoughts-on-swift-language.html.
Bruns, Axel, and Joanne Jacobs. 2006. *Uses of Blogs*. New York: Peter Lang.
Buechley, Leah, Daniela K. Rosner, Eric Paulos, and Amanda Williams. 2009. “DIY for CHI: Methods, Communities, and Values of Reuse and Customization.” In *CHI ’09 Extended Abstracts on Human Factors in Computing Systems*, 4823–26. CHI EA ’09. New York, NY, USA: ACM. doi:10.1145/1520340.1520750.
Butcher, Lee. 1988. *Accidental Millionaire: The Rise and Fall of Steve Jobs at Apple Computer*. New York: Paragon House.
Caldwell, Serenity. 2012. “Sandbox Deadline Delayed yet Again to June 1.” *Macworld*. February 21. http://www.macworld.com/article/165502/2012/02/sandbox_deadline_delayed_yet_again_to_june_1.html.
Callon, Michel. 1986. “Some Elements of a Sociology of Translation: Domestication of the Scallops and the Fishermen of St Brieux Bay.” In *Power, Action, and Belief: A New Sociology of Knowledge?*, 196–229. Sociological Review Monograph 32. London ; Boston: Routledge & Kegan Paul.
Campbell, H. A., and A. C. La Pastina. 2010. “How the iPhone Became Divine: New Media, Religion and the Intertextual Circulation of Meaning.” *New Media & Society* XX (X): 1–17.

Campbell-Kelly, Martin. 2003. *From Airline Reservations to Sonic the Hedgehog: A History of the Software Industry*. Cambridge, MA: MIT Press.
Carlton, Jim. 1997. *Apple: The inside Story of Intrigue, Egomania, and Business Blunders*. New York: Time Business/Random House.
Case, Peter, and Erik Piñeiro. 2006. "Aesthetics, Performativity and Resistance in the Narratives of a Computer Programming Community." *Human Relations* 59 (6): 753–82.
Castrogiovanni, Gary J., B. R. Baliga, and Roland E. Kidwell Jr. 1992. "Curing Sick Businesses: Changing CEOs in Turnaround Efforts." *The Executive* 6 (3): 26–41.
Chen, Brian X. 2012. "Facebook's Challenge: Making Money in Mobile World." *The New York Times*, August 23, sec. Technology. http://www.nytimes.com/2012/08/24/technology/facebook-rewrites-its-code-for-a-small-screen-world.html.
Cheng, Jacqui. 2011. "iOS Devs Put out a Call to Unite against Lodsys, Other Patent Trolls." *Ars Technica*. August 1. http://arstechnica.com/apple/news/2011/08/ios-devs-put-out-a-call-to-unite-against-lodsys-other-patent-trolls.ars.
———. 2013. "Frustrated with iCloud, Apple's Developer Community Speaks up En Masse." *Ars Technica*. March 28. http://arstechnica.com/apple/2013/03/frustrated-with-icloud-apples-developer-community-speaks-up-en-masse/.
Coleman, E. Gabriella. 2013. *Coding Freedom: The Ethics and Aesthetics of Hacking*. Princeton: Princeton University Press.
Collins, Harry M. 1981a. "Introduction: Stages in the Empirical Programme of Relativism." *Social Studies of Science* 11 (1): 3–10.
———. 1981b. "Son of Seven Sexes: The Social Destruction of a Physical Phenomenon." *Social Studies of Science* 11 (1): 33–62. doi:10.1177/030631278101100103.
———. 1985. *Changing Order: Replication and Induction in Scientific Practice*. London ; Beverly Hills: Sage Publications.
———. 2011. *Gravity's Ghost: Scientific Discovery in the Twenty-First Century*. Chicago: University of Chicago Press.
Collins, Harry M., and Trevor J. Pinch. 1998a. *The Golem at Large: What You Should Know about Technology*. Cambridge UK ; New York: Cambridge University Press.
———. 1998b. *The Golem: What Everyone Should Know about Science*. 2nd ed. Cambridge [England] ; New York: Cambridge University Press. http://encompass.library.cornell.edu/cgi-bin/checkIP.cgi?access=gateway_standard%26url=http://encompass.library.cornell.edu/cgi-bin/scripts/ebooks.cgi?bookid=53593.
Conway, Joe. 2009. "Dot-Notation Syntax." Blog. *Big Nerd Ranch Weblog*. August 6. http://weblog.bignerdranch.com/83-83/.
Conway, Joe, and Aaron Hillegass. 2011. *iOS Programming: The Big Nerd Ranch Guide*. 2nd ed. Atlanta, GA : Indianapolis: Big Nerd Ranch; Exclusive

worldwide distribution of the English edition of this book by Pearson Technology Group.
———. 2012. *iOS Programming: The Big Nerd Ranch Guide*. 3rd ed. Atlanta, GA : Indianapolis: Big Nerd Ranch; Exclusive worldwide distribution of the English edition of this book by Pearson Technology Group.
Corneliussen, Hilde. 2009. "Cultural Perceptions of Computers in Norway 1980-2007: From 'Anybody' Via 'Male Experts' to 'Everybody.'" In *Gender Codes: Women and Men in the Computing Professions*, edited by Thomas Misa, 165–85. Hoboken, NJ: Wiley.
Cowan, Ruth Schwartz. 1985. "How the Refrigerator Got Its Hum." In *The Social Shaping of Technology: How the Refrigerator Got Its Hum*, 202–18. Milton Keynes ; Philadelphia: Open University Press.
Cox, Brad J. 1983. "The Object Oriented Pre-Compiler: Programming Smalltalk 80 Methods in C Language." *SIGPLAN Not.* 18 (1): 15–22. doi:10.1145/948093.948095.
———. 1990a. "There Is a Silver Bullet: A Software Industrial Revolution Based on Reusable and Interchangeable Parts Will Alter the Software Universe." *BYTE*, October 1.
———. 1990b. "Planning the Software Industrial Revolution." *IEEE Software* 7 (6): 25.
Cringely, Robert X. 1996a. *Accidental Empires: How the Boys of Silicon Valley Make Their Millions, Battle Foreign Competition, and Still Can't Get a Date*. New York: HarperBusiness.
———. 1996b. "Triumph of the Nerds: The Transcripts, Part III." *PBS.org*. July. http://www.pbs.org/nerds/part3.html.
Crossan, Mary M., Henry W. Lane, and Roderick E. White. 1999. "An Organizational Learning Framework: From Intuition to Institution." *The Academy of Management Review* 24 (3): 522–37. doi:10.2307/259140.
Csikszentmihalyi, Mihaly. 1994. *Flow: The Psychology of Optimal Experience*. New York: HarperCollins.
Daston, Lorraine. 1995. "The Moral Economy of Science." *Osiris*, 2nd Series, 10: 3–24.
David, Shay, and Trevor J. Pinch. 2005. "Six Degrees of Reputation: The Use and Abuse of Online Review and Recommendation Systems." http://ssrn.com/abstract=857505.
Dempsey, James. 2014. "Life Without WWDC." *James Dempsey*. April 18. http://jamesdempsey.net/2014/04/18/life-without-wwdc/.
Deutschman, Alan. 2000. *The Second Coming of Steve Jobs*. 1st ed. New York: Broadway Books.
Dijkstra, Edsger W. 1968. "Letters to the Editor: Go to Statement Considered Harmful." *Commun. ACM* 11 (3): 147–48. doi:10.1145/362929.362947.
———. 1971. *A Short Introduction to the Art of Programming*. [Eindhoven, Netherlands]: [Technische Hogeschool Eindhoven].
DiNucci, Darcy. 1992. "Once More, with Feeling." *NeXTWORLD*.
"DIY Hardware: Reinventing Hardware for the Digital Do-It-Yourself Revolution." 2009. In *ACM SIGGRAPH ASIA 2009 Art Gallery & Emerging Technologies:*

Adaptation, 66–67. SIGGRAPH ASIA '09. New York, NY, USA: ACM. doi:10.1145/1665137.1665186.
Douglas, Mary. 1966. *Purity and Danger; an Analysis of Concepts of Pollution and Taboo*. New York: Praeger.
Dredge, Stuart. 2013. "If Android Is so Popular, Why Are Many Apps Still Released for iOS First?" *The Guardian*. August 15. http://www.theguardian.com/technology/appsblog/2013/aug/15/android-v-ios-apps-apple-google.
Dunbar-Hester, Christina. 2008. "Geeks, Meta-Geeks, and Gender Trouble: Activism, Identity, and Low-Power FM Radio." *Social Studies of Science* 38 (2): 201–32.
Durkheim, Émile. 1965. *The Elementary Forms of the Religious Life*. Translated by Joseph Ward Swain. New York: Free Press.
Eco, Umberto. 1994. "The Holy War: Mac vs. DOS." *Porta Ludovica*. September 30. http://www.themodernword.com/eco/eco_mac_vs_pc.html.
Ensmenger, Nathan L. 2009. "Making Programming Masculine." In *Gender Codes: Women and Men in the Computing Professions*, edited by Thomas Misa. Hoboken, NJ: Wiley.
———. 2010. *The "Computer Boys" Take Over: Computers, Programmers, and the Politics of Technical Expertise*. Cambridge, MA: MIT Press.
Ensmenger, Nathan L., and William Aspray. 2002. "Software as Labor Process." In *History of Computing: Software Issues*, edited by Ulf Hashagen, Reinhard Keil-Slawik, and Arthur L Norberg, 139–65. Berlin: Springer.
Esslinger, Hartmut. 2014. *Keep It Simple: The Early Design Years of Apple*. Stuttgart: Arnoldsche Verlagsanstalt.
Faulkner, Wendy. 2000a. "The Power and the Pleasure? A Research Agenda for 'Making Gender Stick' to Engineers." *Science, Technology, & Human Values* 25 (1): 87–119.
———. 2000b. "Dualisms, Hierarchies and Gender in Engineering." *Social Studies of Science* 30 (5): 759–92.
"Filk Music." 2014. *Wikipedia, the Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Filk_music&oldid=631867060.
Florida, Richard L. 2002. *The Rise of the Creative Class: And How It's Transforming Work, Leisure, Community and Everyday Life*. New York, NY: Basic Books.
Florin, Fabrice. 1985. *Hackers: Wizards of the Electronic Age*. VHS. Eugene, Or: New Dimension Films.
Foresman, Chris. 2008a. "iPhone NDA: Doing More Harm than Good." *Ars Technica*. July 28. http://arstechnica.com/apple/news/2008/07/iphone-nda-doing-more-harm-than-good.ars.
———. 2008b. "Pragmatic Programmers iPhone SDK Book Latest Casualty of NDA." *Ars Technica*. September 25. http://arstechnica.com/apple/news/2008/09/pragmatic-programmers-iphone-sdk-book-latest-casualty-of-nda.ars.
———. 2010a. "Apple Putting the Kibosh on Soft-Core Porn App Screenshots." *Ars Technica*. February 12. http://arstechnica.com/apple/news/2010/02/apple-putting-the-kibosh-on-soft-core-porn-app-screenshots.ars.

———. 2010b. “Apple VP Attempts to Explain Double Standard for Risqué Apps.” *Ars Technica*. February 23. http://arstechnica.com/apple/news/2010/02/apple-vp-attempts-to-explain-double-standard-for-risque-apps.ars.
———. 2011. “Apple to Lodsys: You’ll Have to Go through Us to Sue iOS Devs.” *Ars Technica*. August 9. http://arstechnica.com/apple/news/2011/08/apple-tells-judge-intervention-against-lodsys-should-be-granted.ars.
Freiberger, Paul, and Michael Swaine. 2000. *Fire in the Valley: The Making of the Personal Computer*. 2nd ed. New York: McGraw-Hill.
Galison, Peter. 1999. “Trading Zone: Coordinating Action and Belief.” In *The Science Studies Reader*, edited by Mario Biagioli, 137–60. New York: Routledge.
Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Reading, Mass: Addison-Wesley.
Garfinkel, Simson L. 1992a. “Getting Religion.” *NeXTWORLD*.
———. 1992b. “Object Lessons Sidebar: Interface Builder: Icing on the Cake.” *NeXTWORLD*.
———. 1992c. “Object Lessons Sidebar: Objective-C: Seeing Is Believing.” *NeXTWORLD*.
———. 1993a. “Objects for Sale: A New Class of Software Comes of Age.” *NeXTWORLD*, July.
———. 1993b. “Make or Buy?” *NeXTWORLD*, November.
Geertz, Clifford. 1973. “Ideology As a Cultural System.” In *The Interpretation of Cultures: Selected Essays*, 193–233. New York: Basic Books.
Giddens, Anthony. 1991. *The Consequences of Modernity*. 1st paperback ed. Cambridge, [U.K.]: Polity Press. http://encompass.library.cornell.edu/cgi-bin/checkIP.cgi?access=gateway_standard%26url=http://www.aspresolver.com/aspresolver.asp?SOTH;S10023877.
Gieryn, Thomas F. 1983. “Boundary-Work and the Demarcation of Science from Non-Science: Strains and Interests in Professional Ideologies of Scientists.” *American Sociological Review* 48 (6): 781–95.
Goffman, Erving. 1961. *Encounters; Two Studies in the Sociology of Interaction*. Indianapolis: Bobbs-Merrill.
Google Inc. 2015. “Google-Styleguide - Style Guides for Google-Originated Open-Source Projects - Google Project Hosting.” Accessed February 24. https://code.google.com/p/google-styleguide/.
Gordon, George G. 1991. “Industry Determinants of Organizational Culture.” *The Academy of Management Review* 16 (2): 396–415. doi:10.2307/258868.
Gruber, John. 2004. “Daring Fireball: Dashboard vs. Konfabulator.” June 30. http://daringfireball.net/2004/06/dashboard_vs_konfabulator.
Gruber, John, Marco Arment, Casey Liss, John Siracusa, and Scott Simpson. 2014. *Live From WWDC 2014, With Marco Arment, Casey Liss, John Siracusa, and Scott Simpson*. The Talk Show. Accessed July 1. http://daringfireball.net/thetalkshow/2014/06/06/ep-083.

Gruzd, Anatoliy, Barry Wellman, and Yuri Takhteyev. 2011. “Imagining Twitter as an Imagined Community.” *American Behavioral Scientist* 55 (10): 1294–1318. doi:10.1177/0002764211409378.

Gusterson, H. 2008. “Nuclear Futures: Anticipatory Knowledge, Expert Judgment, and the Lack That Cannot Be Filled.” *Science & Public Policy*. 35 (8): 551–60.

Haigh, Thomas. 2010. “Dijkstra’s Crisis: The End of Algol and the Beginning of Software Engineering:1968-1972.” In *Soft-EU Project Meeting*. Leiden. http://www.tomandmaria.com/tom/Writing/DijkstrasCrisis_LeidenDRAFT.pdf.

Hanson, Chris. 2009. “When to Use Properties & Dot Notation.” *Eschatology*. May 7. http://eschatologist.net/blog/?p=160.

Haraway, Donna. 1989. *Primate Visions: Gender, Race, and Nature in the World of Modern Science*. New York: Routledge.

Haring, Kristen. 2003. “The ‘Freer Men’ of Ham Radio: How a Technical Hobby Provided Social and Spatial Distance.” *Technology and Culture* 44 (4): 734–61.

Hayes, Caroline Clarke. 2009. “Computer Science: The Incredible Shrinking Woman.” In *Gender Codes: Women and Men in the Computing Professions*, edited by Thomas Misa, 25–49. Hoboken, NJ: Wiley.

Hertzfeld, Andy. 2005. *Revolution in the Valley*. 1st ed. Sebastopol, CA: O’Reilly.

———. 2013a. “PC Board Esthetics.” *Folklore.org:*. December 31. http://www.folklore.org/StoryView.py?project=Macintosh&story=PC_Board_Esthetics.txt.

———. 2013b. “Pirate Flag.” *Folklore*. December 31. http://www.folklore.org/StoryView.py?project=Macintosh&story=Pirate_Flag.txt.

———. 2013c. “Real Artists Ship.” *Folklore*. December 31. http://www.folklore.org/StoryView.py?project=Macintosh&story=Real_Artists_Ship.txt.

Hicks, Marie. 2012. “From Antisocial to Alphasocial: Exclusionary Nerd Cultures and the Rise of the Brogrammer.” May 1. http://www.sigcis.org/node/335.

Hillegass, Aaron. 2011. *Objective-C Programming: The Big Nerd Ranch Guide*. 1st ed. The Big Nerd Ranch Guide. Atlanta, GA : Indianapolis, IN: Big Nerd Ranch ; Pearson Technology Group (distributor).

Hiltzik, Michael A. 1999. *Dealers of Lightning : Xerox PARC and the Dawn of the Computer Age*. 1st ed. New York: HarperBusiness.

Hockenberry, Craig. 2008. “Twitter / ʎɹɹəquəʞɔoɥɔ Бıʁɹɔ: There Is a Huge Shortage O ...” July 23. http://twitter.com/chockenberry/statuses/866468107.

Holder, Robert J., and Richard McKinney. 1992. “Corporate Change and the Hero’s Quest.” *The Journal for Quality and Participation* 15 (4): 34.

Holwerda, Thom. 2011. “The History of ‘App’ and the Demise of the Programmer.” *OSnews*. June 24. http://www.osnews.com/story/24882/The_History_of_App_and_the_Demise_of_the_Programmer.

Huddleston, Tom. 2014. “Apple’s Market Cap Just Hit $700 Billion for the First Time - Fortune.” November 25. http://fortune.com/2014/11/25/apple-700-billion/.
Hughes, G. David. 1990. “Managing High-Tech Product Cycles.” *The Executive* 4 (2): 44–55.
Hughes, Thomas P. 1987. “The Evolution of Large Technological Systems.” In *The Social Construction of Technological Systems: New Directions in the Sociology and History of Technology*, edited by Wiebe E. Bijker, Thomas P. Hughes, and Trevor J. Pinch, 449–82. Cambridge Mass.: MIT Press.
Ihnatko, Andy. 2011. “App Sandboxing Risks Eroding the Mac’s Identity.” *Macworld*. October 2. http://www.macworld.com/article/162504/2011/10/app_sandboxing_risks_eroding_the_macs_identity.html.
“iOSDevCamp.” 2014. Accessed June 13. http://www.iosdevcamp.org/.
Isaac, Mike. 2013. “Why Facebook Is Sending Its People to Mobile Coding Camp (And Not Just Engineers).” *AllThingsD*. March 4. http://allthingsd.com/20130304/why-facebook-is-sending-its-people-to-mobile-coding-camp-and-not-just-engineers/.
Isaacson, Walter. 2011. *Steve Jobs*. New York: Simon & Schuster.
Jobs, Steve. 2005. “Text of Steve Jobs’ Commencement Address (2005).” June 14. http://news-service.stanford.edu/news/2005/june15/jobs-061505.html.
Jobs, Steve, and George W Beahm. 2011. *I, Steve: Steve Jobs, in His Own Words*. Chicago, Ill.: B2 Books. http://www.books24x7.com/marc.asp?bookid=43924.
Kahney, Leander. 2004. *The Cult of Mac*. San Francisco, CA: No Starch Press.
———. 2009. *Inside Steve’s Brain, Expanded Edition*. Expanded edition. New York: Portfolio Hardcover.
———. 2013. *Jony Ive: The Genius Behind Apple’s Greatest Products*. New York: Portfolio Hardcover.
Kane, Yukari Iwatani. 2014. *Haunted Empire: Apple after Steve Jobs*. First edition. New York: HarperBusiness, an imprint of HarperCollins Publishers.
Karelia Software. 2005. “Watson Product FAQ.” http://www.karelia.com/watson/watsonFAQ.html.
Karon, Paul. 1992a. “Digital Deputy: How NeXT Won the Shootout at the L.A. Corral.” *NeXTWORLD*.
———. 1992b. “Healthy Surprise.” *NeXTWORLD*.
Kawasaki, Guy. 1990. *The Macintosh Way*. Glenview, Ill: Scott, Foresman.
Kay, Alan C. 1993. “The Early History of Smalltalk.” In *The Second ACM SIGPLAN Conference on History of Programming Languages*, 69–95. HOPL-II. Cambridge, MA: ACM. doi:10.1145/154766.155364.
———. 1998. “Alan Kay On Messaging,” October 10. http://c2.com/cgi/wiki?AlanKayOnMessaging.
Kelty, Christopher. 2008. *Two Bits: The Cultural Significance of Free Software*. Durham, NC: Duke University Press.
Kernighan, Brian W., and P. J. Plauger. 1974. *The Elements of Programming Style*. New York: McGraw-Hill.

Keur, Christian, Aaron Hillegass, and Joe Conway. 2014. *iOS Programming: The Big Nerd Ranch Guide*. 4th ed. Atlanta, GA: Big Nerd Ranch.
King, Scott. 2012. "Changing the World." *My Favorite Apple*. January 11. http://www.myfavoriteapple.com/changing-the-world/.
Kleif, Tine, and Wendy Faulkner. 2003. "'I'm No Athlete [but] I Can Make This Thing Dance!'–Men's Pleasures in Technology." *Science, Technology & Human Values* 28 (2): 296–325.
Kline, Ronald. 2000. *Consumers in the Country : Technology and Social Change in Rural America*. Baltimore MD: Johns Hopkins University Press.
Kline, Ronald, and Trevor J. Pinch. 1996. "Users as Agents of Technological Change: The Social Construction of the Automobile in the Rural United States." *Technology and Culture* 37 (4): 763–95.
Knorr-Cetina, Karin. 1999. *Epistemic Cultures: How the Sciences Make Knowledge*. Cambridge, Mass: Harvard University Press.
Knuth, Donald E. 1974. "Computer Programming As an Art." *Commun. ACM* 17 (12): 667–73. doi:10.1145/361604.361612.
Kochan, Stephen G. 2012. *Programming in Objective-C*. 5th ed. Developer's Library. Upper Saddle River, NJ: Addison-Wesley.
Kohler, Robert. 1994. *Lords of the Fly: Drosophila Genetics and the Experimental Life*. William B. Provine Collection on Evolution and Genetics. Chicago: University of Chicago Press.
Kraft, Philip. 1977. *Programmers and Managers : The Routinization of Computer Programming in the United States*. New York: Springer-Verlag.
Krakow, Gary. 2014. "Why Developers Prefer Apple Over Google." *TheStreet*. June 27. http://www.thestreet.com/story/12759340/1/why-developers-prefer-apple-over-google.html.
Kuhn, Thomas S. 1996. *The Structure of Scientific Revolutions*. 3rd ed. Chicago, IL: University of Chicago Press.
Kunda, Gideon. 1992. *Engineering Culture: Control and Commitment in a High-Tech Corporation*. Philadelphia: Temple University Press.
Lamarche, Jeff. 2009. "Dot Notation Redux: Google's Style Guide." Blog. *iPhone & Mac Development*. August 9. http://iphonedevelopment.blogspot.com/2009/08/dot-notation-redux-google-style-guide.html.
Lampson, Butler. 1986. "Personal Distributed Computing: The Alto and Ethernet Software." In *Proceedings of the ACM Conference on The History of Personal Workstations*, 101–31. HPW '86. New York, NY, USA: ACM. doi:10.1145/12178.12186.
Latour, Bruno. 1987. *Science in Action: How to Follow Scientists and Engineers Through Society*. Cambridge, Mass.: Harvard University Press. https://catalog.library.cornell.edu.proxy.library.cornell.edu/cgi-bin/Pwebrecon.cgi?BBID=1259185&DB=local.
Latour, Bruno, and Steve Woolgar. 1986. *Laboratory Life: The Construction of Scientific Facts*. Princeton, N.J.: Princeton University Press.
Lave, Jean, and Etienne Wenger. 1991. *Situated Learning: Legitimate Peripheral Participation*. Cambridge [England]; New York: Cambridge University Press.

Lavin, Dan. 1993. “Some Assembly Required.” *NeXTWORLD*, November.
Lee, Mike. 2011. “Appsterdam Legal Fund.” August. http://appsterdamlegalfoundation.org/.
Levy, Steven. 1984. *Hackers: Heroes of the Computer Revolution*. 1st ed. New York: Penguin Books.
———. 1994. *Insanely Great : The Life and Times of Macintosh, the Computer That Changed Everything*. New York: Viking.
Light, Jennifer S. 1999. “When Computers Were Women.” *Technology and Culture* 40 (3): 455–83.
Lindtner, Silvia, Ian Bogost, and Julian Bleeker. 2014. “The End Game for Maker Culture.” presented at the Intel Science & Technology Center for Social Computing All Hands Retreat, Atlanta, May 19.
Lindtner, Silvia, Garnet D. Hertz, and Paul Dourish. 2014. “Emerging Sites of HCI Innovation: Hackerspaces, Hardware Startups & Incubators.” In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 439–48. CHI ’14. New York, NY, USA: ACM. doi:10.1145/2556288.2557132.
Linzmayer, Owen. 2004. *Apple Confidential 2.0: The Definitive History of the World’s Most Colorful Company*. Rev. 2nd ed. San Francisco, CA: No Starch Press.
Littman, Jonathan. 1991. “Bull Market: Three Securities Firms Turned to NeXT for a Competitive Advantage.” *NeXTWORLD*.
Mackenzie, Donald. 1990. *Inventing Accuracy: A Historical Sociology of Nuclear Missile Guidance*. Cambridge Mass.: MIT Press.
MacKenzie, Donald. 1996. *Knowing Machines: Essays on Technical Change*. Cambridge Mass.: MIT Press.
———. 2001. *Mechanizing Proof: Computing, Risk, and Trust*. Cambridge Mass.: MIT Press.
Mahoney, Michael S. 1988. “The History of Computing in the History of Technology.” *Annals of the History of Computing, IEEE* 10 (2): 113–25. doi:10.1109/MAHC.1988.10011.
———. 1990. “The Roots of Software Engineering.” *CWI Quarterly* 3 (4): 325–34.
———. 1993. “Issues in the History of Computing.” In *Forum on History of Computing*. Cambridge, MA: Association for Computing Machinery. http://www.princeton.edu/~hos/Mahoney//articles/issues/issuesfr.htm.
———. 2002. “Software: The Self-Programming Machine.” In *From 0 to 1: An Authoritative History of Modern Computing*, edited by Atsushi Akera and Frederik Nebeker, 91–100. New York: Oxford University Press.
———. 2004. “Finding a History for Software Engineering.” *Annals of the History of Computing, IEEE* 26 (1): 8–19.
———. 2008. “What Makes the History of Software Hard.” *Annals of the History of Computing, IEEE* 30 (3): 8–18.
Maines, Rachel. 2009. *Hedonizing Technologies: Paths to Pleasure in Hobbies and Leisure*. Baltimore: Johns Hopkins University Press.
Malaby, Thomas. 2009. *Making Virtual Worlds: Linden Lab and Second Life*. Ithaca: Cornell University Press.
Malone, Michael. 1999. *Infinite Loop*. 1 edition. New York: Doubleday Business.

Mandel, Michael, and Judith Scherer. 2012. *Geography of the App Economy*. South Mountain Economics LLC. http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&cad=rja&uact=8&ved=0CCAQFjAA&url=http%3A%2F%2Ffiles.ctia.org%2Fpdf%2FThe_Geography_of_the_App_Economy.pdf&ei=WjTAVOPsBYemgwSer4T4Bg&usg=AFQjCNGAlVmZTKs_gHbeZ4Hgi2-E-djArg&sig2=JHW0Dq-jNH0ZHBGcvgSY0A&bvm=bv.83829542,d.eXY.

Markoff, John. 2005. *What the Dormouse Said: How the Sixties Counterculture Shaped the Personal Computer Industry*. New York: Viking.

Marwick, Alice Emily. 2013. *Status Update: Celebrity, Publicity, and Branding in the Social Media Age*. New Haven: Yale University Press.

Marx, Karl. 1947. *The German Ideology, Parts I & III.* International Publishers.

Mateas, Michael, and Nick Montfort. 2005. "A Box, Darkly: Obfuscation, Weird Languages, and Code Aesthetics." In *Proceedings of the 6th Digital Arts and Culture Conference*, 144–53. IT University of Copenhagen.

McNely, Brian J. 2011. "Sociotechnical Notemaking: Short-Form to Long-Form Writing Practices." *Present Tense: A Journal of Rhetoric in Society* 2 (1). http://www.presenttensejournal.org/volume-2/sociotechnical-notemaking-short-form-to-long-form-writing-practices/.

Misa, Thomas. 2009. *Gender Codes: Women and Men in the Computing Professions*. Hoboken, NJ: Wiley.

Moritz, Michael. 2009. *Return to the Little Kingdom: Steve Jobs, the Creation of Apple, and How It Changed the World*. New York: Overlook Press.

Nelson, Theodor H. 1974. *Computer Lib: You Can and Must Understand Computers Now*. 1st ed. Chicago: Nelson : [available] from Hugo's Book Service.

NeXTWORLD. 1992a. "Special Report: NeXT Market Mosaic."

———. 1992b. "Top 40 North American NeXT Sites."

Noble, David. 1999. *The Religion of Technology: The Divinity of Man and the Spirit of Invention*. New York: Penguin Books.

Nonala, Ikujiro, and Martin Kenney. 1991. "Towards a New Theory of Innovation Management: A Case Study Comparing Canon, Inc. and Apple Computer, Inc." *Journal of Engineering and Technology Management* 8 (1): 67–83. doi:10.1016/0923-4748(91)90005-C.

Nørmark, Kurt. 2014. "Programming Paradigms." *Functional Programming in Scheme With Web Programming Examples*. January 3. http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigms.html.

Notaro, Anna. 2006. "The Lo(n)g Revolution: The Blogosphere as an Alternative Public Sphere?" *Reconstruction* 6 (4). http://reconstruction.eserver.org.proxy.library.cornell.edu/Issues/064/notaro.shtml.

Nye, David. 1994. *American Technological Sublime*. Cambridge, MA: MIT Press.

———. 2003. *America as Second Creation: Technology and Narratives of New Beginnings*. Cambridge, MA: MIT Press.

Orr, Julian E. 1990. *Talking about Machines: An Ethnography of a Modern Job*.

Oudshoorn, Nelly, and Trevor J. Pinch. 2003. *How Users Matter: The Co-Construction of Users and Technologies*. Cambridge, MA: MIT Press.
Page, Ruth. 2012. "The Linguistics of Self-Branding and Micro-Celebrity in Twitter: The Role of Hashtags." *Discourse & Communication* 6 (2): 181–201. doi:10.1177/1750481312437441.
Parish, Nick. 2014. *Cool Code, Bro: Brogrammers, Geek Anxiety and the New Tech Elite*. Robot, Robot & Hwang.
Parker, Chris. 2014. "Building Modern Frameworks." presented at the Apple Worldwide Developer Conference, Presidio, Moscone West, San Francisco, June 5. https://developer.apple.com/videos/wwdc/2014/.
Patel, Manish. 2010. "iPhone Development Costs." *OSXDaily*. September 7. http://osxdaily.com/2010/09/07/iphone-development-costs/.
Pinch, Trevor J. 1986. *Confronting Nature: The Sociology of Solar-Neutrino Detection*. Dordrecht, Holland ; Boston ; Higham, MA, U.S.A.: D. Reidel Pub. Co; Kluwer Academic Publishers. https://catalog.library.cornell.edu/cgi-bin/Pwebrecon.cgi?BBID=1549692&DB=local.
Pinch, Trevor J., and Wiebe E. Bijker. 1984. "The Social Construction of Facts and Artefacts: Or How the Sociology of Science and the Sociology of Technology Might Benefit Each Other." *Social Studies of Science* 14 (3): 399–441.
Pinch, Trevor J., and Frank Trocco. 2002. *Analog Days: The Invention and Impact of the Moog Synthesizer*. Cambridge MA: Harvard University Press.
Pinkerton, Mike, Greg Miller, and Dave MacLachlan. 2014. "Google Objective-C Style Guide." http://google-styleguide.googlecode.com/svn/trunk/objcguide.xml.
Postbox. 2012. "Postbox and the Mac App Store." July 25. http://www3.postbox-inc.com/?/blog/entry/postbox_and_the_mac_app_store/.
Poulsen, Kevin. 2014. "Behind iPhone's Critical Security Bug, a Single Bad 'Goto.'" *WIRED*. February 22. http://www.wired.com/2014/02/gotofail/.
Prentice, Rachel. 2013. *Bodies in Formation: An Ethnography of Anatomy and Surgery Education*. Experimental Futures. Durham, NC: Duke University Press.
Qiu, Yixin, Anandasivam Gopal, and Il-Horn Hann. 2011. "Synthesizing Professional And Market Logics: A Study Of Independent iOS App Entrepreneurs." In *Proceedings of the Thirty Second International Conference on Information Systems*. Shanghai.
Raja, Tasneem. 2012. "'Gangbang Interviews' and 'Bikini Shots': Silicon Valley's Brogrammer Problem." April 26. http://www.motherjones.com/print/172791.
Randolph, John. 2009, May 5.
Reid, Jon. 2012. "Dot Notation in Objective-C: 100% Pure Evil." *Quality Coding*. June 3. http://qualitycoding.org/dot-notation/.
———. 2013. "In Which I Embrace Dot Notation …." *Quality Coding*. September 17. http://qualitycoding.org/dot-notation-wins/.
Richardson, Julia, and Michael B. Arthur. 2013. "'Just Three Stories': The Career Lessons Behind Steve Jobs' Stanford University Commencement Address." *Journal of Business and Management* 19 (1): 45–57.

Richtel, Matt, and Brian X. Chen. 2014. “Tim Cook, Making Apple His Own.” *The New York Times*, June 15. http://www.nytimes.com/2014/06/15/technology/tim-cook-making-apple-his-own.html.
Rick. 2007. “CTGradient and the Landed Gentry of Mac Development™.” *Rixstep’s Red Hat Diaries*. November 21. http://rixstep.com/2/1/20071121,00.shtml.
———. 2010. “The Longest Screed.” *Rixstep Developers Workshop*. January 3. http://rixstep.com/2/2/20100103,00.shtml.
Ritchie, Rene. 2014. “Debug 34: Sexism in Tech.” *iMore*. April 24. http://www.imore.com/debug-34-sexism-tech.
Ritchie, Rene, Matt Drance, Ryan Nielsen, Daniel Jalkut, and Jason Snell. 2014. *Debug 38: WWDC 2014 Developer Roundtable*. Debug. Accessed July 1. http://www.imore.com/debug-38-wwdc-2014-developer-roundtable.
Robinson, Brett T. 2013. *Appletopia: Media Technology and the Religious Imagination of Steve Jobs*.
Rotemberg, Julio J., and Garth Saloner. 2000. “Visionaries, Managers, and Strategic Direction.” *The RAND Journal of Economics* 31 (4): 693–716. doi:10.2307/2696355.
Ruby, Dan. 1992. “Dry Bones.” *NeXTWORLD*.
———. 1993a. “Who Needs Shrink Wrap?” *NeXTWORLD*, October.
———. 1993b. “The Shirts Off Their Backs.” *NeXTWORLD*, December.
Ruby, Dan, and Steve Jobs. 1992. “Reinventing NeXT: Steve Jobs Goes on the Record about Technology, Marketing, and Ross Perot: Interview.” *NeXTWORLD*.
Sadun, Erica. 2012. “TUAW - The Unofficial Apple Weblog.” Accessed February 8. http://www.tuaw.com/editor/erica-sadun/.
Sander, Peter J. 2012. *What Would Steve Jobs Do? How the Steve Jobs Way Can Inspire Anyone to Think Differently and Win*. New York: McGraw-Hill. http://www.myilibrary.com?id=332629.
Savage, Neil. 2013. “Backing Creativity.” *Commun. ACM* 56 (7): 20–21. doi:10.1145/2483852.2483860.
Schmidt, Jan. 2007. “Blogging Practices: An Analytical Framework.” *Journal of Computer-Mediated Communication* 12 (4): 1409–27. doi:10.1111/j.1083-6101.2007.00379.x.
Schoemaker, Paul J. H. 1997. “Disciplined Imagination: From Scenarios to Strategic Options.” *International Studies of Management & Organization* 27 (2): 43–70.
Schoenberger, Erica. 2001. “Corporate Autobiographies: The Narrative Strategies of Corporate Strategists.” *Journal of Economic Geography* 1 (3): 277–98. doi:10.1093/jeg/1.3.277.
Schramm, Mike. 2013. “Why Is Facebook’s App so Much Better Lately? Ask Big Nerd Ranch.” *TUAW: Apple News, Reviews and How-Tos since 2004*. March 5. http://www.tuaw.com/2013/03/05/whys-facebooks-app-so-much-better-lately-ask-big-nerd-ranch/.
Schüll, Natasha Dow. 2012. *Addiction by Design: Machine Gambling in Las Vegas*. Princeton, NJ: Princeton University Press.

Sculley, John, and John A. Byrne. 1987. *Odyssey: Pepsi to Apple ... a Journey of Adventure, Ideas, and the Future*. 1st ed. New York: Harper & Row.
Segall, Ken. 2013. *Insanely Simple: The Obsession That Drives Apple's Success*. Reprint edition. Portfolio Trade.
Seibel, Peter. 2009. *Coders at Work: Reflections on the Craft of Programming*. New York: Apress.
Senft, Theresa M. 2008. *Camgirls: Celebrity and Community in the Age of Social Networks*. Digital Formations, v. 4. New York: Lang.
Sennett, Richard. 2008. *The Craftsman*. New Haven: Yale University Press.
Seth, Suman. 2010. "Pedagogical Economies: The 'Sommerfeld School' and the Problems of Teaching." In *Crafting the Quantum: Arnold Sommerfeld and the Practice of Theory, 1890-1926*, 47–70. Cambridge, Mass.: MIT Press.
Shapin, Steven. 2008. *The Scientific Life: A Moral History of A Late Modern Vocation*. Chicago: University of Chicago Press.
Shipley, Wil. 2005. "Unit Testing Is Teh Suck, Urr." *Call Me Fishmeal*. September 22. http://blog.wilshipley.com/2005/09/unit-testing-is-teh-suck-urr.html.
———. 2011. "Call Me Fishmeal.: Real Security in Mac OS X Requires Apple-Signed Certificates." November 3. http://blog.wilshipley.com/2011/11/real-security-in-mac-os-x-requires.html.
Silverstone, Stuart. 1992. "Black Market." *NeXTWORLD*.
Simmons, Brent. 2014. "Inessential: Who at the Table Is an Indie iOS Developer?" July 25. http://inessential.com/2014/07/25/who_at_the_table_is_an_indie_ios_develop.
Simon, Stephanie, and Erin Mershon. 2014. "Bill Gates Masters D.C. — and the World." *POLITICO*. February 4. http://www.politico.com/story/2014/02/bill-gates-microsoft-policy-washington-103136.html.
Siracusa, John. 2005a. "Avoiding Copland 2010." *Ars Technica*. September 28. http://arstechnica.com/staff/fatbits/2005/09/1372.ars.
———. 2005b. "Avoiding Copland 2010: Part 2." *Ars Technica*. September 30. http://arstechnica.com/staff/2005/09/1393/.
———. 2005c. "Avoiding Copland 2010: Part 3." *Ars Technica*. October 4. http://arstechnica.com/staff/fatbits/2005/10/1412.ars.
———. 2010. "Copland 2010 Revisited: Apple's Language and API Future." *Ars Technica*. June 16. http://arstechnica.com/apple/news/2010/06/copland-2010-revisited.ars.
Sivek, Susan Currie. 2011. "'We Need a Showing of All Hands' Technological Utopianism in MAKE Magazine." *Journal of Communication Inquiry* 35 (3): 187–209. doi:10.1177/0196859911410317.
Slayton, Rebecca. 2013a. *Arguments That Count: Physics, Computing, and Missile Defense, 1949-2012*. Cambridge, Massachusetts: The MIT Press.
———. 2013b. "The Political Economy of Software Engineering." In *Arguments That Count: Physics, Computing, and Missile Defense, 1949-2012*, 151–71. Cambridge, Massachusetts: The MIT Press.
Smykil, Jeff. 2009. "Nine-Year-Old Makes Waves with iPhone Programming Skillz." *Ars Technica*. February 6. http://arstechnica.com/apple/news/2009/02/nine-year-old-makes-waves-with-iphone-programming-skillz.ars.

Star, Susan Leigh, and James R. Griesemer. 1989. “Institutional Ecology, ‘Translations’ and Boundary Objects: Amateurs and Professionals in Berkeley’s Museum of Vertebrate Zoology, 1907-39.” *Social Studies of Science* 19 (3): 387–420.
Statista. 2015a. “Apple Revenue - iTunes, Software & Services 2013-2014, by Quarter | Statistic.” *Statista*. January. http://www.statista.com/statistics/250918/apples-revenue-from-itunes-software-and-services/.
———. 2015b. “Apple App Store: Number of Available Apps 2008-2014 | Statistic.” *Statista*. January 21. http://www.statista.com/statistics/263795/number-of-available-apps-in-the-apple-app-store/.
———. 2015c. “Apple App Store: Number of Downloads 2008-2014 | Statistic.” *Statista*. January 21. http://www.statista.com/statistics/263794/number-of-downloads-from-the-apple-app-store/.
Steel Media Ventures. 2014. “App Store Metrics.” *148Apps.biz*. June 16. http://148apps.biz/app-store-metrics/.
Stefan L. Ram, Berlin. 2003. “Dr. Alan Kay on the Meaning of ‘Object-Oriented Programming,’” July 23. http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en.
Strauss, Anselm L. 1978. “A Social World Perspective.” *Studies in Symbolic Interaction* 1 (1): 119–28.
———. 1982. “Social Worlds and Legitimation Processes.” *Studies in Symbolic Interaction* 4: 171–90.
———. 1984. “Social Worlds and Their Segmentation Processes.” *Studies in Symbolic Interaction* 5: 123–39.
Streeter, Thomas. 2003. “The Romantic Self and the Politics of Internet Commercialization.” *Cultural Studies* 17 (5): 648–68. doi:10.1080/0950238032000126865.
———. 2011. *The Net Effect: Romanticism, Capitalism, and the Internet*. Critical Cultural Communication. New York: New York University Press.
Stross, Randall. 1993. *Steve Jobs and the NeXT Big Thing*. New York ; Toronto ; New York: Atheneum ; Maxwell Macmillan Canada ; Maxwell Macmillan International.
Stroustrup, Bjarne. 1993. “A History of C++.” In *The Second ACM SIGPLAN Conference on History of Programming Languages*, 271–97. Cambridge, MA: ACM. http://portal.acm.org/citation.cfm?id=155375.
Sullivan, Kevin. 1991. “Inventing the Future: New Approaches to Management, Compensation, and Learning at Apple Computer.” *Employment Relations Today* 18 (4): 417.
Sutter, John D. 2011. “5 Memorable Quotes from Steve Jobs - CNN.com.” *CNN*. October 6. http://www.cnn.com/2011/10/05/tech/innovation/steve-jobs-quotes/index.html?hpt=hp_t1.
“System Software.” 2015. *PC Magazine Encyclopedia*. http://www.pcmag.com/encyclopedia/term/52419/system-software.

Tabini, Marco. 2014. “Sherlocked! Nine Technologies Apple Disrupted at WWDC.” *Macworld*. June 4. http://www.macworld.com/article/2359422/sherlocked-nine-technologies-apple-disrupted-at-wwdc.html.
Takhteyev, Yuri. 2012. *Coding Places: Software Practice in a South American City*. Acting with Technology. Cambridge, Mass: MIT Press.
Tanenbaum, Joshua G., Amanda M. Williams, Audrey Desjardins, and Karen Tanenbaum. 2013. “Democratizing Technology: Pleasure, Utility and Expressiveness in DIY and Maker Practice.” In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2603–12. CHI ’13. New York, NY, USA: ACM. doi:10.1145/2470654.2481360.
Thacker, Chuck. 1986. “Personal Distributed Computing: The Alto and Ethernet Hardware.” In *Proceedings of the ACM Conference on The History of Personal Workstations*, 87–100. HPW ’86. New York, NY, USA: ACM. doi:10.1145/12178.12185.
The Economist. 2007. “The Third Act,” June 7. http://www.economist.com/node/9298983.
Thompson, E. P. 1971. “The Moral Economy of the English Crowd in the Eighteenth Century.” *Past & Present*, no. 50 (February): 76–136.
Tierney, Therese. 2013. *The Public Space of Social Media: Connected Cultures of the Network Society*. Routledge Studies in New Media and Cyberculture 13. New York: Routledge.
Tomayko, James E. 2002. “Software as Engineering.” In *History of Computing: Software Issues*, edited by Ulf Hashagen, Reinhard Keil-Slawik, and Arthur L Norberg, 139–65. Berlin: Springer.
Traweek, Sharon. 1988. *Beamtimes and Lifetimes: The World of High Energy Physicists*. Cambridge Mass.: Harvard University Press.
Tresch, John. 2001. “On Going Native Thomas Kuhn and Anthropological Method.” *Philosophy of the Social Sciences* 31 (3): 302–22. doi:10.1177/004839310103100302.
Turkle, Sherry. 1984. *The Second Self: Computers and the Human Spirit*. New York: Simon and Schuster.
———. 1995. *Life on the Screen: Identity in the Age of the Internet*. New York: Simon & Schuster.
———. 2011. *Alone Together: Why We Expect More from Technology and Less from Each Other*. New York: Basic Books.
Turner, Fred. 2006. *From Counterculture to Cyberculture: Stewart Brand, the Whole Earth Network, and the Rise of Digital Utopianism*. Chicago, IL: University of Chicago Press.
———. 2009. “Burning Man at Google: A Cultural Infrastructure for New Media Production.” *New Media & Society* 11 (1-2): 73–94. doi:10.1177/1461444808099575.
Van Horn, Royal. 1996. “The Journey Ahead.” *Phi Delta Kappan* 77 (6): 454.
Van Meeteren, Michiel. 2008. “Indie Fever: The Genesis, Culture and Economy of a Community of Independent Software Developers on the Macintosh OS X Platform.” Bachelor thesis, Human Geography, University of Amsterdam. http://www.madebysofa.com/indiefever.

VisionMobile Ltd. 2014. *European App Economy 2014*. London. http://www.visionmobile.com/product/european-app-economy-2014/.
———. 2015. "Developer Economics Q1 2015: State of the Developer Nation." *Developer Economics*. February 17. https://www.developereconomics.com/reports/developer-economics-q1-2015/.
Wajcman, Judy. 1991. *Feminism Confronts Technology*. University Park Pa.: Pennsylvania State University Press.
Warner, Michael. 2002. *Publics and Counterpublics*. New York; Cambridge, Mass.: Zone Books ; Distributed by MIT Press.
Warwick, Andrew, and David Kaiser. 2005. "Kuhn, Foucault, and the Power of Pedagogy." In *Pedagogy and the Practice of Science: Historical and Contemporary Perspectives*, 393–409. Inside Technology. Cambridge, Mass: MIT Press.
Weber, Max. 1946. *From Max Weber: Essays in Sociology*. New York: Oxford University Press.
———. 1958. *The Protestant Ethic and the Spirit of Capitalism*. Translated by Talcott Parsons. New York: Scribner.
Webster, Bruce F. 1989. *The NeXT Book*. Reading Mass.: Addison-Wesley.
———. 1992. "Open Horizons." *NeXTWORLD*.
———. 1995. *Pitfalls of Object-Oriented Development*. New York: M&T Books.
———. 1996. "The Real Software Crisis: The Shortage of Top-Notch Programmers Threatens to Become the Limiting Factor in Software Development." *BYTE*, January 1.
Williams, Raymond. 1977. "Ideology." In *Marxism and Literature*, edited by Steven Lukes, 55–71. United Kingdom: Oxford University Press.
Wortham, Jenna. 2009. "The iPhone Gold Rush." *The New York Times*, April 5, sec. Fashion & Style. http://www.nytimes.com/2009/04/05/fashion/05iphone.html.
Wozniak, Steve. 2006. *IWoz: Computer Geek to Cult Icon: How I Invented the Personal Computer, Co-Founded Apple, and Had Fun Doing It*. 1st ed. New York: W.W. Norton & Co.
Xu, Weiai Wayne, Yoonmo Sang, Stacy Blasiola, and Han Woo Park. 2014. "Predicting Opinion Leaders in Twitter Activism Networks The Case of the Wisconsin Recall Election." *American Behavioral Scientist* 58 (10): 1278–93. doi:10.1177/0002764214527091.
Yegge, Steve. 2006. "Stevey's Blog Rants: Execution in the Kingdom of Nouns." March 30. http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html.
Young, Jeffrey S. 1988. *Steve Jobs: The Journey Is the Reward*. Glenview, Ill.: Scott, Foresman.
Young, Jeffrey S, and William L. Simon. 2005. *ICon : Steve Jobs, the Greatest Second Act in the History of Business*. Hoboken NJ: Wiley.
Zarra, Marcus. 2008. "A Case Against Dot Syntax." *Cocoa Is My Girlfriend*. July 8. http://www.cimgf.com/2008/07/08/a-case-against-dot-syntax/.
Zepcevski, Joline. 2012. "Complexity & Verification: The History of Programming as Problem Solving." Ph.D., United States -- Minnesota: University of Minnesota.

http://search.proquest.com/pqdtft/docview/926961600/abstract/140E50ED87C2CD7DFC3/1?accountid=10267.
Ziman, J. M. 1968. *Public Knowledge: An Essay Concerning the Social Dimension of Science*. London: Cambridge U.P.