FROM CODING TO COMMUNITY:
ITERATION, ABSTRACTION AND OPEN SOURCE SOFTWARE DEVELOPMENT

A Thesis
submitted to the Faculty of the
Graduate School of Arts and Sciences
of Georgetown University
in partial fulfillment of the requirements for the
degree of
Master of Arts
in Communication, Culture and Technology

By

Brad Weikel, B.S.

Washington, DC
April 15, 2009
FROM CODING TO COMMUNITY: ITERATION, ABSTRACTION AND OPEN SOURCE SOFTWARE DEVELOPMENT

Brad Weikel, B.S.

Thesis Advisor: Garrison LeMasters

ABSTRACT

Studies of Free and Open Source Software (FOSS) have generally focused on the programmers who write the code (including their culture, ethics, and motivations), the economic consequences of giving away code, and the legal and moral basis for FOSS licenses, in relation to copyright. However, the act of building FOSS applications, or, rather, the very act of programming, is fundamentally mediated by computer interfaces, including programming languages. In the pursuit of an understanding of FOSS development, these interfaces – and the material practices which they both derive and are derived from – cannot be ignored.

Two procedural genres, in particular, stand out in the material practices of software development: iteration and abstraction. Iteration, in this context, refers to the writing of software through incremental changes, leaving it ever subject to further modifications: bug fixes, extensions, and re-factoring. Abstraction, on the other hand, refers to the use of interfaces (programming languages, APIs, and user interfaces) to hide complexity, thereby enabling new relations between code and people.
This thesis is an exploration of iteration and abstraction in the practices of programming, as these concepts relate to the politics and production of FOSS projects. Iteration and abstraction will be introduced through simple code examples, in order to ground them as fundamental characteristics of programming, but will then be examined through a wider discussion of development methodologies and social relations. A comparison of compiled source code and interpreted source code, and the politics they embody, will serve as a proxy for the comparison of proprietary and FOSS development, respectively, again using iteration and abstraction as an entry point.

Finally, iteration and abstraction will be discussed in the context of Drupal, an open source content management system, and the community behind it. While these concepts, at the level of the individual, seem highly compatible with the politics of FOSS, studying a FOSS community reveals several sources of tension between iteration, abstraction, and the collaborative model of FOSS development.

This project brings a necessary precision and depth, and, as such, will inform future research on both FOSS and broader themes in open and participatory culture.
ACKNOWLEDGMENTS

To the author, a Masters thesis is a disruptive life event, not just a paper. This project has been in my mind for three years, and has fully consumed it for the past six months, and I thank everyone who has touched my life and work – through even the briefest of conversations – along the way.

I extend special thanks to Garrison LeMasters and Michael Nelson, my wonderful thesis committee. As my advisor, Garrison's blind confidence in my abilities allowed me to discover my own research path and to enjoy the process far more than I anticipated. As my second reader, on the other hand, Dr. Nelson continually pushed me outside of my academic comfort zone, a healthy antagonism that has resulted in a more rounded – and therefore more satisfying – final product.

I would also like to thank Dr. Garcia, Dr. Tinkcom, and the entire CCT faculty and staff. In particular, I thank Mima Dedaic for her rigorous expectations and persistent encouragement, which raised the bar for my research and writing.

My second love, during the past three years at Georgetown, has been gnovis, and I thank the entire gnovis staff for making this final year so easy – you have required remarkably little management from your Managing Editor, and my most sincere gratitude goes to Ashley Bowen and Lauren Alfrey, for laying the groundwork for gnovis to succeed long after I am gone.

Thank you to Evan Herrnstadt and Emily Garr, for letting me monopolize our common spaces and veggies; to Phillip Thurtle and Douglass Merrill, for talking me into graduate school in the first place; to Robert Soden and Jeff Miccolis, whose insights and skepticism invigorated my work at a crucial moment; to Kevin Donovan for the steady stream of helpful Tweets and links; and to the Red Derby and the Sunday Night Club, for giving me a weekly oasis from my own mind.

Thank you, most of all, to my loving and supportive partner, Emily Cohen, for tolerating my late night panics, manic writing sprints, self-indulgent emo sessions, and general inattentiveness for months on end; for providing healthy and delicious food when I was lost in my laptop; and for knowing when I needed to talk through a chapter, and when I just needed you to get out of my way. You have made this process not only bearable, but enjoyable, and you have my eternal gratitude.

Finally, this project is dedicated to my parents, Roxane and Bill Weikel, and my sister, Robyn Nuttall, for their endless and unconditional support, and to the memory of Jim Clowes, my cousin, mentor, and inspiration.

Many thanks,
Brad Weikel
# TABLE OF CONTENTS

**Chapter 1. Introduction** ........................................................................................................... 1

**Chapter 2. The FOSS Movement: Discourses and Historical Roots** .......................................... 9
  - A Very Brief History of FOSS ........................................................................................................ 9
  - The Commons Movement ........................................................................................................... 20
  - The Economics of FOSS Production .......................................................................................... 25
  - The Ethics and Politics of Hackers .............................................................................................. 33

**Chapter 3. Writing Code: Iteration & Abstraction** ........................................................................ 41
  - Hello World: Learning to Code through Iteration ..................................................................... 42
  - A Failure Tolerant Practice ......................................................................................................... 49
  - Dummy Code: Building Complexity ............................................................................................ 52
  - Imperative Programming and Code Abstraction ......................................................................... 55
  - Does Code Want to be Free? ....................................................................................................... 63

**Chapter 4. Code Readability and Distribution** .......................................................................... 68
  - Hello World, Revisited ................................................................................................................ 69
  - Compilation, Enclosure, and Iteration Flows ............................................................................. 74
  - Abstraction of Another Sort: The Virtual Machine ..................................................................... 85
  - SourceForge Data Analysis: Limits of the Metaphor ................................................................... 87

**Chapter 5. Community Case Study: Drupal** ............................................................................ 95
  - Why Drupal? / Why Not Linux? ................................................................................................. 95
  - Drupal Basics ............................................................................................................................ 97
  - A Plane of New Relations .......................................................................................................... 101
  - The Drop is Always Moving: The Limits of Abstraction ........................................................... 105
  - “Your FOSS is Showing” – Contested Measures of “Does it Run?” .......................................... 110

**Chapter 6. Conclusion** ............................................................................................................ 117
  - Beyond the FOSS Movement .................................................................................................. 121

**References** .................................................................................................................................. 126
Chapter 1. Introduction

Free and Open Source Software (FOSS) is a hot topic among academics of all sorts, including economists, political scientists, and legal scholars. Lawrence Lessig has used FOSS principles as a foundation for his assault on the copyright regime and a model for the Creative Commons (2006, p. 214); Yochai Benkler has called FOSS the “quintessential instance of commons-based peer production” (2007, p. 63); and Steven Weber considers FOSS an “antirival” good, because “the system as a whole positively benefits from free riders” (2005, p. 154). These are just the big names; beyond them, serious books on the implications of FOSS, ranging from impenetrably dense cultural theory to pop crossover pieces, seem to be published every month. FOSS seems, in a sense, to be at the very heart of academic discourse on the political economy of the Internet.

Moving from theory to practice, FOSS is also a big deal in the marketplace. While the most famous FOSS project, the Linux operating system, has remained a marginal option for consumers, it is a very popular option for servers. Apache, an open source web server typically run on Linux, has completely dominated the web server market since 1996 (“January 2009 Web Server Survey,” 2009). Firefox, the popular open source web browser, recently climbed above a 20% share of the browser market, historically dominated by Microsoft’s Internet Explorer (“Firefox market share,” 2009), and will soon surpass its billionth download (“Spread Firefox,” 2009). Up and coming
Drupal, an open source content management system, powers more than 100,000 websites, including sites for corporate giants Yahoo, Disney, and America Online, musicians from Bob Dylan to Britney Spears (Habeb, 2008), and even the recently launched Recovery.gov (Buytaert, 2009b).

Steven Weber notes that media coverage of FOSS is “an overblown mix of hype and cynicism” (Weber, 2005, p. 224), but the hype has not yet resulted in widespread public recognition of the FOSS movement, nor has cynicism prevented many FOSS applications from enjoying tremendous success. Why is open source software being used so widely? Tim Jordan has argued that, unlike most material commodities and artifacts, computer software is judged by a very simple criteria: “Does it run?” (2008, p. 52). The public does not need to understand the intricacies of collaborative software development or copyleft licensing in order to pick winning applications, or to have them picked for them by their IT proxies; in this regard, FOSS is on the march, with an ever growing foothold in the marketplace. “The open source paradigm is moving inexorably toward worldwide domination of computer software infrastructure . . . facilitating pervasive public recognition of the movement” (Deek & McHugh, 2007, p. 325). In other words, the code runs, so FOSS applications are enjoying rapid adoption by IT administrators; public recognition will follow, according to Deek & McHugh, and end-user products like Firefox and OpenOffice may be foreshadowing this shift.
FOSS development, however, runs counter to traditional capitalist thinking about production and property and, despite the movement's extensive attempts to correct this errant view, FOSS is often understood through one particularly limiting and occasionally inaccurate characteristic: its free price tag. While hackers like Richard Stallman argue that FOSS is important for its emancipatory qualities, and FOSS communities tout the superiority of their code and the social capital that accumulates along with its production, consumers, for the most part, see FOSS merely as a free (and legal) alternative to buying or copying commercial software – a gross oversimplification of the movement.

At the same time, despite its recent popularity in academic discourse, studies of FOSS have been rather limited, and have largely been organized around two questions: what are the implications of copyleft licensing, and what are the incentives for participating in these “gift economies?” Code and programming have received far less attention. While discussions of hacking sometimes explore the rhetoric of open source software, they more often focus on the illicit hacker activities known as “cracking” or lapse into polemical accounts of a political hacker ethos, often highly romanticized.

There are several notable exceptions, including insider perspectives written by FOSS programmers themselves, but generally overlooked is “the how” of open source – the languages and tools that programmers work with, and how, at the level of material practices, these technologies shape and are shaped by the political ideology of
open source. If FOSS is as unique and important as both scholars like Lessig and programmers like Stallman claim, then there must also be something unique and important about the practices that bring FOSS into being: the practices of programming.

This paper will examine the material practices of open source software development, from basic coding practices to the complex social organization of FOSS communities, in order to understand how open source software actually comes into existence and thrives in the digital economy. This paper focuses primarily on programming and source code as a whole, not merely in the context of FOSS, but in doing so explores a number of questions about the relationship between programming and the FOSS movement. Are there particular characteristics of programming and source code that help explain FOSS development? If FOSS is, at its core, about fixing bugs and extending functionality in software that has already been released, how are these tasks shaped by choices about languages, licensing, and organization? Will a macro-level genealogy of programming languages and practices help explain FOSS's relationship to other trends in software development?

Throughout this paper, discussion will be framed around two habits which stand out among the many material practices of software development: iteration and abstraction. Iteration, in this context, refers to a programmer's incremental refinement and expansion of source code, afforded by software's dynamic and complex character.
Source code is always subject to further iterations: bug fixes, extensions, and re-factoring. Abstraction refers to the use of interfaces – programming languages, APIs, and user interfaces – to hide complexity and economize interaction, thereby enabling new relations between code, between people, and between people and code. Iteration and abstraction provide a new entry point into the politics and production of Free and Open Source Software, stripping away many of the romanticized and grandiose characterizations of programmers and replacing them with a more grounded understanding of how the practices of programming and the politics of FOSS are intertwined.

Chapter 2 contains a brief history of hacking and FOSS development, as well as a literature review encompassing perspectives on the commons and the copyleft movement, the economics of FOSS, and the ethics and politics of hackers. While much of the literature is, indeed, guilty of the criticisms I have made above, a full review is an essential starting point.

Chapter 3 begins with an examination of the practice of programming, focusing on its highly iterative characteristics, as demonstrated through “Hello World” – often the first application a programmer writes when learning a new language – and the more general practice of using “dummy code” to block out functionality that will be developed later. The second half of Chapter 3 describes the evolution of imperative programming languages (from unstructured to procedural to object-oriented) as a trend
of abstraction, then considers analogous trends towards abstraction in development methodologies and software architecture. Once iteration and abstraction are understood as fundamental to the practices of programming, it can be argued that FOSS development, which embraces iteration and abstraction, is born of and powered by these practices, particularly when contrasted to proprietary software, which restricts certain iterations and erases certain abstractions by enclosing the source code.

In Chapter 4, this argument is extended to consider the distribution and consumption of software and source code, focusing in particular on the differences between interpreted and compiled languages and how these differences structure the human relationships around applications written in those languages. Chapter 4 also includes a study of trends on SourceForge, a large online repository of FOSS projects, although the analysis, which compares FOSS projects written in compiled languages to those written in interpreted languages, draws only limited conclusions and serves primarily as a guide to future research. More generally, the comparison of interpreted and compiled languages will serve as a proxy for FOSS and proprietary software, respectively, to model more precisely the effects of enclosure of code, whether by technology or law, on both the software development cycle and the relations between programmers and users.

In Chapter 5, the argument of the preceding chapters – that the material practices of programming encourage FOSS development – will be both tested and
further developed through a case study of a FOSS community. While the formation of communities is not an explicit requirement of FOSS license, and large communities are the exception rather than the rule, these communities are the face of the FOSS movement, and the subject of most existing research, so they must be considered. Chapter 5 will illuminate important tensions in the politics of FOSS communities, including a number of places in which iteration and abstraction come into apparent conflict, exposing possible limits on the sorts of problems FOSS can solve. The subject of this case study will be Drupal, an open source content management platform, predominantly used for building public facing websites and web applications. The Drupal community has taken a particularly public and inclusive approach to collaborative development, providing an abundance of material for analysis. The method of approach will be a combination of content analysis and critical discourse analysis, drawing from conversations in community spaces such as discussion boards, mailing lists, Twitter, blogs, and bug reports.

The concluding chapter, Chapter 6, will begin by considering the direct research consequences of this work, including a number of possible quantitative studies that expand on the brief examination of SourceForge data in Chapter 5, as well as the consequences for practitioners. Throughout this work, FOSS will be characterized as part of a larger procedural genre of practices related to iteration and abstraction, which includes Agile development methods, Eric Raymond's babbling
Bazaar model, object-oriented programming languages, interpreted code, and APIs, and this characterization is perhaps the most meaningful takeaway for programmers and other practitioners. “FOSS or proprietary?” is not the be all and end all of questions that programmers and project managers face, when thinking about the impact of iteration and abstraction on the human and code relations involved in their work; rather, that question is situated in a larger set of questions about development methodologies, technological tradeoffs and, of course, users and customers. This work, both directly and indirectly, can help inform those decisions.

Open source software is, of course, not the only contemporary movement of its kind. Related movements include the Creative Commons and other copyleft approaches to creative content, the open access movement for academic journals, OpenCourseWare for teaching materials, and crowdsourcing and collaborative knowledge production ranging from Wikipedia articles to Digg rankings. Chapter 6 will conclude by briefly considering the extent to which this work can inform research of these parallel movements.
Chapter 2. The FOSS Movement: Discourses and Historical Roots

The discourse of FOSS tends to focus on questions about the two closely related aspects of FOSS development which, at a glance, appear irrational. The first question concerns the creation of FOSS applications – why would a programmer just give away her work? The second question concerns the formation of FOSS communities – why would a second programmer give improvements back to the first programmer? This chapter will begin with a history of the FOSS movement, followed by a technical overview of FOSS licenses and licensing concepts, but will thereafter focus on the various ways that others have accounted for the seemingly irrational generosity of FOSS development. This discussion will be oriented around three clusters of ideas: first, copyright, copyleft, and the commons; second, social exchange, gift economies, anti-rival goods, and other economic approaches; and, finally, cultural studies perspectives focused on the ethics and politics of hackers. While some of these discourses have considered significant characteristics of source code, they have not sufficiently explored programming and software development as processes; filling this gap will be the subject of subsequent chapters.

A Very Brief History of FOSS

Prior to the 1970s, computing was largely confined to government and academia, and sharing of source code was the norm. This norm, and the future of FOSS, was aided by historical circumstance: from 1956 to 1982, anti-trust rulings kept
AT&T from entering the software industry, so the UNIX operating system (developed at AT&T subsidiary Bell Laboratory) could only be distributed for free (Söderberg, 2007, pp. 14-19).

Perhaps inevitably, though, the industry commercialized. In 1976, foreshadowing the rapid software enclosure of which his own company would become the dominant symbol, teen programmer Bill Gates wrote “A Letter to Hobbyists,” arguing not only that copying his software was theft, but that such theft would discourage the development of complex software (Söderberg, 2007, pp. 42-44). In the early 1980s, software copyrights became commonplace, and the breakup of AT&T in 1982 allowed the company to enter the computer industry and attempt to enclose the UNIX operating system (Söderberg, 2007, p. 19).

Almost immediately, in 1983, Richard Stallman, the grandfather of FOSS, began developing GNU (“Gnu is Not Unix”), a FOSS alternative to Unix. In 1985 Stallman founded the Free Software Foundation (FSF), which would become one of two major bodies that approves FOSS licenses, and in 1989 released the first version of the GPL (GNU General Public License) the most widely used FOSS license (Deek & McHugh, 2007, pp. 297-302). The Free Software Definition, which the FSF uses to evaluate licenses, is based on four freedoms (“The Free Software Definition,” 2008):

- The freedom to run the program, for any purpose.
- The freedom to study how the program works, and adapt it to your needs.
The freedom to redistribute copies so you can help your neighbor.

The freedom to improve the program, and release your improvements (and modified versions in general) to the public, so that the whole community benefits.

The second and fourth freedoms, to adapt and improve the program and to release those changes, require access to the software's source code, the human readable code files that programmers write and edit. Source code should not be confused with machine code, the binary instructions that are actually sent to the computer's CPU.

In 1991, Linus Torvalds started building a new, lightweight Unix-like kernel, licensed under the GPL. His project would come to be known as Linux. While certainly not the first or only attempt to create a viable FOSS variant of Unix, Linux was, by 1994, “selling like hotcakes” (Raymond, 2001, pp. 16-17). Torvald's style – “release early and often, delegate everything you can, be open to the point of promiscuity” (2001, p. 21) – became the model of the “babbling bazaar” in Eric Raymond's famous 1997 essay “The Cathedral and the Bazaar,” and a later book by the same name. Raymond contrasted the dynamic, if seemingly unpredictable, model of the Bazaar with that of the Cathedral, the more cautious approach of most prior large-scale programming projects at that time, both proprietary and open source (2001, p. 21). In the more traditional Cathedral model, software is built very deliberately and not released until it is “done” – a false destination that denies the iterative nature of programming.
The rise of Linux was, of course, paralleled by other important developments. Tim Berners-Lee first publicized his “World Wide Web” in 1991 (Deek & McHugh, 2007, p. 24), giving the Internet the public face – and interface – it needed to shift from a geek novelty to the mainstream, household utility it is today. PHP, a server-side scripting language, was released under the GPL in 1995 (Deek & McHugh, 2007, p. 72), and Javascript was first bundled into the Netscape browser later that year (Champeon, 2001). These two languages, along with C++, will be discussed at length in Chapter 4.

In 1998, Netscape released its Navigator source code under an open source license (Deek & McHugh, 2007, p. 244) – a move that failed to save Netscape from the browser war with Microsoft but would later spawn the wildly successful Firefox browser. Seeing Netscape's decision as a critical moment for the free software movement, Eric Raymond and Bruce Perens quickly founded the Open Source Initiative, officially splitting the FOSS movement in two (Raymond, 2001, pp. 174-176). Like the Free Software Foundation, one of the Open Source Initiative's primary functions is to approve licenses – typically written by third parties – but it does so using a slightly less restrictive definition than the FSF, meaning that some licenses qualifying as “open source” might not qualify as “free software” (“Problems with older versions of the Apple License (APSL),” 2008).
To clarify, briefly, the term “FOSS” (Free and Open Source Software) is not widely used by programmers, but is a helpful umbrella term for academics wishing to discuss the collective movement without explicitly choosing sides between the FSF and the OSI. However, following the lead of Deek and McHugh (2007), I will acknowledge the problematic nature of all the available terms, and use “FOSS” and “open source” interchangeably, unless the context indicates otherwise.

The OSI was largely an attempt to soften the FOSS movement's ideology, making it more appealing to businesses (Deek & McHugh, 2007, p. 244) and circumventing the ambiguous meaning of “free” in the English language. “Free,” of course, can mean both free of charge (“gratis”) and free from restraints (“libre”), which has caused much confusion for consumers. Despite Stallman's best attempts to clarify – famously declaring that free software is “free as in speech, not as in free beer” (“The Free Software Definition,” 2008) – free software continues to be confused with freeware, proprietary software that is only free of charge (“gratis”). The ambiguity of the word “open,” of course, leaves the phrase “open source software” just as vulnerable to misuse and misunderstanding as “free software,” though perhaps with less damaging consequences for the movement.

The more important distinction between the FSF and the OSI, however, is in the subtleties of both their licensing definitions and their expressed values, a distinction which is clouded by the tremendous overlap between the licenses they have
approved. Chopra and Dexter, in their study of the ethical implications of each
definition, observe that “the FSD appears to hold the interests of society paramount,
while the OSD . . . seems to prioritize the interests of the individual licensee” (Chopra & Dexter, 2005). Richard Stallman has broadly dismissed the OSI, arguing that “open
source is a development methodology; free software is a social movement” (Stallman, 2007) - the FSF emphasizes freedoms, the OSI emphasizes reliability and power. In the
same article, Stallman provides a very helpful example of where the two movements
diverge, at least in theory: DRM (Digital Rights Management or, in Stallman's words,
Digital Restrictions Management). DRM is software that prevents consumers from
making copies of music and movies and, according to Stallman, “this software might
be 'open source' . . . but it won't be free software, since it won't respect the freedom of
the users that actually run it.” This example illustrates why open source is generally
considered to be more amenable to businesses (in this case, the entertainment
industry).

Despite Stallman's compelling example, however, it is this author's opinion that
the FSF's ideology – and the resulting objection to the OSI – originated in a time when
both software and hardware were hard to come by, and that the supposed differences
between the two movements is largely outdated and insignificant, due to their mutual
acceptance of all the most popular licenses. While it is possible to exploit the gap
between the two movements, and despite the vocal opinions of key figures in both
movements, the split has not resulted in a dramatic functional divide among programmers. The most popular licenses are approved by both groups, so programmers have no need to pick sides (although companies that choose to write their own licenses will have important decisions to make). This convergence of licenses will be discussed further, below, after clarifying how FOSS licensing works.

**FOSS Licensing and Copyleft**

The FOSS movement achieves its goals by distributing software under licensing agreements that use the mechanics of copyright to grant privileges to users, rather than to restrict them. The logic can be translated roughly as “I hold the copyright to this source code and, as the copyright holder, I give all users the right to use this source code in these specific ways.” The license is typically included as a text file in the root directory of the distributed software, and referenced from the source code files and, in some cases, the application itself. With some exceptions, the majority of FOSS licenses have met the requirements of either the FSF's Free Software Definition (“The Free Software Definition,” 2008) or the OSI's Open Source Definition (“The Open Source Definition,” 2006), or both. These two definitions have important differences, but the following substantive similarities are sufficient to consider them collectively:

1. **Redistribution Rights:** FOSS-licensed software may be copied and redistributed, for free or by sale, without paying royalties to the original creator.
2. Source Code Included: The original source code of FOSS-licensed software must be included in the distribution, so that it may be edited and improved upon.

3. Copyleft Tolerant: FOSS licenses are allowed (and encouraged by the FSF) to use “copyleft” licensing terms. Copyleft, a play on copyright, requires that FOSS-licensed software must retain the same license it was first distributed under, even if it is modified and redistributed. In essence, copyleft prevents a FOSS application from being enclosed, converted into proprietary software.

Notably absent from these criteria is a requirement that anyone can contribute, as if the source code repository was as editable as a Wikipedia page. Anyone can edit the source code and redistribute it themselves – this is known as “forking” the code, when it leads to divergent versions of the same project – but the licenses do not require collaboration. Some licenses, such as the Artistic License 2.0 license (“Artistic License 2.0,” 2006), do require that certain code modifications be sent back to the original developer, but the developer is not obligated to accept those revisions. The curious fact that, despite these terms, forking is relatively rare and vibrant communities sometimes form around FOSS applications, is the focus of much of the literature discussed throughout this chapter.
The third characteristic of FOSS licenses, as noted above, is copyleft tolerance, which demands further clarification and will shed insight into the complexities of FOSS licenses. Both the OSI and the FSF allow copyleft, but do not require it, and both groups have approved licenses with and without copyleft. The most popular copyleft license, the GPL, is approved by both groups, as is the most popular non-copyleft license, the BSD (Berkeley Software Distribution) license. Regardless, the optional copyleft property – sometimes described as “sticky” or “viral” – is thought to be critical to the spread of FOSS applications, as copyleft licenses prevent future iterations of the source code, no matter how radically altered, from being removed from “the commons” of FOSS applications.

Above, I noted that the apparent conflict between the FSF and the OSI has not resulted in a functional split for programmers, because of the overlap of licenses approved by both organizations. In an October, 2008 snapshot of SourceForge, a repository of FOSS projects, 80% of projects – excluding the roughly 5% of projects released to the public domain, which are neither FOSS nor proprietary software but still accessible via SourceForge – designated that they used one of the three most popular licenses: the GPL (62%), its lesser counterpart the LPGL (10%), and BSD compatible licenses (7%). All three of these licenses are approved by both the FSF and the OSI, as are the next fourteen most popular licenses, which collectively account for 96% of the license designations (Madey, 2009).
Focusing on the number of projects, of course, only tells part of the story. Licenses are relevant to both the programmer and the user, so the popularity of the projects, the usage, must be considered as well. As of October, 2008, of the 100 most downloaded projects on SourceForge (accounting for 65% of total downloads), 96 were licensed under the GPL, LGPL, or BSD-style licenses, and three of the remaining four have licenses approved by both the FSF and the OSI. Once again, the theoretical distinction between the FSF and the OSI is, in practice, largely dissolved.

It should be noted, however, that some licenses, including the original BSD license, were initially rejected by the FSF before being approved in revised formats (“Licenses - Free Software Foundation,” 2009). SourceForge's data does not clearly distinguish between “BSD-old,” the original version, and the modified “BSD-new.” Figure 2.1, above, shows how thirteen licenses, including those just mentioned, represent the various combinations of OSI and FSF approval. With the exception of older BSD-style licenses, those that have not been approved by both groups are simply niche licenses – in fact, the ones on the right side

![FOSS Licenses Diagram](image-url)
(only FSF approved) would almost certainly be approved by the OSI, if anybody bothered. Figure 2.2, below, shows the more important categorization: copyleft. The real choice that programmers have to make is between BSD-style licenses, which does not have a copyleft requirement, and the GPL or LGPL, both of which do require copyleft (“Licenses - Free Software Foundation,” 2009) (Tiemann, 2006).

Overwhelmingly, according to the SourceForge data, they have chosen copyleft.

The Free Software Foundation also tends to note whether a license has “weak copyleft” or “strong copyleft” (weak copyleft allows FOSS libraries to be bundled with proprietary software, under certain circumstances) and, similarly, whether the license is compatible with the GPL. While the FSF tolerates many variations of licenses, it expresses a preference for strong copyleft, GPL-compatible licenses, believing that standard will allow the movement as a whole, and particular software, to spread most effectively (“Licenses - Free Software Foundation,” 2009).
The Commons Movement

Despite nearly three decades of copyleft in the FOSS movement, copyleft has come to be widely associated with the Creative Commons, an organization founded in 2001 that provides standardized licenses not for source code, but for creative content. “Creative Commons licenses are not designed for software, but rather for other kinds of creative works: websites, scholarship, music, film, photography, literature, courseware, etc.” (“History - CC Wiki,” 2007). Like FOSS licenses, “Creative Commons” licenses use copyright to aid distribution and use, rather than to limit them, including the optional use of “Share-Alike,” the Creative Commons equivalent of copyleft.

The Creative Commons has greatly enriched FOSS the movement by introducing a new framing device: the commons. The interplay of copyright and copyleft can, under this framing, be seen as a debate over the future of the commons, linking copyleft to societal benefits in addition to individual rights. The term “commons” is both singular and plural, with the commons referring to all commons, and a commons indicating a particular commons or a type of commons, such as a natural commons, an information commons, or even the Creative Commons.

However, while the information commons is perhaps the most studied topic around FOSS, and thus the most refined, it suffers from a rather severe limitation: it is focused almost entirely on the legal status of end products, and largely overlooks the
The fact that FOSS is distinctive not only for its legal approach to licensing, but for the way the code is produced and distributed. Those distinctive processes, and the practices embedded within them, are central to this paper, while the commons are only peripheral; nonetheless, the commons, specifically the Creative Commons, is a prominent part of the discourse and demands brief exploration here.

The Creative Commons has, in a sense, taken a softer approach than the FOSS movement, particularly the FSF. Copyright is, by convention, associated with the slogan “All rights reserved,” which has led some FOSS proponents to refer to copyleft as “all rights reversed” (emphasis added) (Stallman, 2008). Creative Commons has taken the middle way, taking on the slogan “Some Rights Reserved” (“History - CC Wiki,” 2007).

These slogans are not just rhetorical. Tim Jordan notes that while a piece of software can be continually improved, creative work more often has a finished state, determined by its creator, beyond which changes can only be considered subjective improvements (Jordan, 2008, p. 105). Put differently, while a painter might want to give away digital copies of her paintings for free, under a Creative Commons license, she might not want to allow others to edit and redistribute those copies as a new work. The Creative Commons, therefore, offers a range of licenses, allowing the creator to specify whether the work can be altered, whether it can be used commercially, and whether the copyleft property (“Share Alike,” in the language of the Creative Commons)
Commons) applies to the work. All Creative Commons licenses require attribution of the original creator (“Licenses - Creative Commons,” 2009).

The discourse of the commons is rooted in the field of economics, beginning with Garrett Hardin's famous essay “The Tragedy of the Commons” (1968), in which Hardin argued that capitalists, acting as rational individuals, will destroy common goods through overuse. “Freedom in a commons brings ruin to all” (Hardin, 1968). Standard reading for a generation of economists, Hardin's argument is hotly contested, and Hardin later softened his own stance by admitting he should have added the adjective “unmanaged” to his article's title (Hardin, 1998).

Managed or not, advocacy for the protection of existing commons and the creation of more commons has been recently remobilized on two fronts. The first front concerns the environmental movement and the preservation of natural commons. This needs only brief mention here, to aid understanding of the second front, the copyleft movement. Natural commons include such resources as air (and the airwaves), water, and silence. The organization On The Commons has called for the creation of commons trusts to prevent further private enclosure of the commons (Bollier, Barnes, & Rowe, 2006) (Bollier & Rowe, 2006).

Their advocacy is not limited to natural commons, but also includes social and intellectual commons, effectively linking the environmental front of the commons
movement with its second front: copyleft via the Creative Commons. Some might consider this a misuse of the word “commons,” which in economics has traditionally referred to resources that are rivalrous (one person's use of the resource limits another's), but non-excludable (a person can be prevented from using the resource). Copylefted information, on the other hand, is both non-excludable (assuming reproduction is cheap, as is the case with digital information) and non-rival, so it ought to be considered a public good rather than a commons. However, perhaps to distinguish it from information in the public domain, or perhaps influenced by the naming of the Creative Commons, copylefted information is frequently described as a commons rather than a public good.

The intellectual shepherd of the Creative Commons, in terms of popular awareness, has been lawyer and copyright expert Lawrence Lessig, one of the founders of the Creative Commons. Generally speaking, Lessig's work is concerned with how creativity and freedom are regulated on the Internet through laws, markets, norms and architecture (2000, p. 87). His early work presents a dystopian view of cyberspace "as it moves from a world of relative freedom to a world of relatively perfect control" (2000, p. 23). Lessig suggests that open source software can act as "a check on state power" (2000, p. 100) but warns that, contrary to popular belief, we are entering a period of increasingly restrictive copyright regulation (2000, p. 127).
In his more recent work, Lessig's focus has narrowed to the intellectual property / copyright regime. Building on Yochai Benkler's framework of layers in communication systems (the "physical layer", the "logical/code layer", and the "content layer"), Lessig argues that the Internet is unique because it has a mixture of freedom and control at the top two layers (2001, pp. 23-25). However, resuming his dystopian fears, Lessig warns of a tightening of control at the code layer (through restrictions of traffic by broadband providers, an issue that has since become known as Net Neutrality), at the content layer (through restrictive licensing schemes, DRM, business method and software patents, and copyright extensions), and at the physical layer (through the privatization of spectrum that might otherwise be used as a wireless commons).

In “Free Culture: The Nature and Future of Creativity” (2005), Lessig turns his attention more strictly to the content layer, arguing that the current trajectory of the copyright regime is suppressing cultural innovation, and calls for adjustments to copyright laws and norms to restore the balance between property owners and the public. Lessig frames his argument around two questions - "whether 'piracy' will be permitted, and whether 'property' will be protected" (2005, p. 10). Most recently, Lessig has drawn attention to generational shifts in society, regarding copyright, ownership, and creativity, and argued that the copyright regime is waging an unwinnable war that has criminalized an entire generation, in a manner similar to but
more extensive than the war on drugs (Lessig, 2008). Put differently, illegal downloading of music and other copyrighted materials has become so pervasive that it is now accepted as the norm by young people, and the solution is not to wage war on this pervasive criminality, but rather to reorient policy so that property owners still have rights and rewards, but socially accepted norms are no longer criminalized. The Creative Commons Developing Nations License, which has been retired, was an early experiment in this philosophy, an attempt to use licensing to decriminalize rampant, unstoppable content piracy in developing countries while reserving full copyright in the developed world (Haughey, 2004).

The Economics of FOSS Production

Once the basic identity questions are out of the way (“what is FOSS?,” “what is copyleft?” and “what is/are the commons?”) discussions of FOSS often turn to motivations. “Why would anyone do that?” is a common concern in the discourse of open source, often followed by “for free?” In other words, why do people participate in FOSS development, arguably giving away their property and labor? What are the incentives? Such questions are gradually losing their importance – as the networked society becomes accustomed to all the free services (many of them by for-profit companies) that have proliferated on the Internet in the “Web 2.0” era, the idea that “free” and “profit” are not mutually exclusive is less provocative than it once was, and
it is increasingly accepted that small contributions from many sources can result in a stable, high quality product, as most famously demonstrated by Wikipedia.

Yochai Benkler (discussing social production in general, not just FOSS) notes that “it is not necessary to pin down precisely the correct or most correct theory of motivation . . . . All that is required . . . is recognition that there is some form of social and psychological motivation that is neither fungible with money nor simply cumulative with it” (2007, p. 96). Lessig makes a similar suggestion, noting that the continual flourishing of FOSS is evidence enough that there are meaningful incentives (2001, p. 69). While dismissals on such grounds are tempting, and while I share in the opinion, it must be acknowledged that this is influenced, in part, by the limitations of specific disciplinary perspectives; the question of motives should not be dismissed outright, and warrants brief attention here.

FOSS development is a counterintuitive and confusing model to newcomers, both because we are so accustomed to thinking in terms of market-based production and, more fundamentally, because non-market production is necessarily fuzzier than market production. Yochai Benkler argues that market transactions “must be clearly demarcated [so they] can be priced efficiently” whereas social exchange “does not require the same degree of crispness at the margin” (2007, p. 108). Others, however, including Eric Raymond, have tried to eliminate the fuzziness of non-market exchange through an exploration of “gift economies,” drawing from roughly a century of
anthropology. “Gift cultures are adaptations not to scarcity, but to abundance” (Raymond, 2001, p. 81) In the case of FOSS, there is an abundance of hardware: “disk space, network bandwidth, computing power” (2001, p. 81). In the consumer society, success is often measured by the accumulation of wealth, and the luxuries such wealth provides, but this measure only works because wealth is scarce. Online there is no such scarcity, so “the only available measure of competitive success is reputation among one's peers” (2001, p. 81).

A gift economy should not be confused with a barter economy; a barter economy functions more or less like the exchange economy we are accustomed to, except that, lacking a common currency, all transactions require the direct exchange of goods for other goods. In a gift economy, on the other hand, individual “transactions” are unidirectional: goods are given and taken for free. If there is an exchange, it is a social one.

The discourse of gift economies is rooted in anthropologist Marcel Mauss's “The Gift: The Form and Reason for Exchange in Archaic Societies.” The gift economy, for Mauss, is built upon a set of obligations: “the obligation, on the one hand, to give presents, and on the other, to receive them,” as well as the “obligation to reciprocate presents received” (Mauss, 2000, p. 13). The obligation to give is grounded in reputation and prestige. The chief, for instance, distributes his wealth in order to “preserve his authority” by “humiliating others” and “placing them 'in the shadow of
his name’’ (2000, p. 39). From this, then, quite naturally, comes the obligation to reciprocate. If one must give to maintain his own prestige, the recipient then must reciprocate in order to maintain his prestige. Similarly, for Mauss, the recipient is obligated to receive, because to refuse a gift would be to “show that one is afraid to be 'flattened’” (2000, p. 41), or to lose face (“lose his name,” in Mauss's terms) by failing to reciprocate.

Put differently, a gift economy is driven by competitive, agonistic giving, where giving determines social status. As one gift/debt is annulled, another is created, in an endless circle. For Mauss, as for Raymond, this depends upon but also leads to wealth and abundance – “A good warrior can thus receive more than his hand can hold” (2000, p. 70). Mauss' notion of an obligation to receive, when applied to FOSS, may help explain why FOSS communities rarely fork, or split a FOSS project and community into two divergent variants, despite the fact that forking is explicitly allowed by FOSS licenses. In the reputation game of FOSS gift economies, there is a continual tension between the obligation to receive and the right to fork – Weber notes that a near-fork of Linux was triggered by Torvalds' failure to integrate a backlog of outside contributions (Weber, 2005, pp. 116-119) – and the threat of a fork might be seen as the mechanism for maintaining the cycle of gift-giving.

Raymond adds two important twists to gift giving in FOSS. First, he notes that when a gift economy intersects with an exchange economy, as is often the case in
FOSS development, “your reputation may spill over and earn you higher status there.” An obvious example of this spilling over is the prestige (and, most likely, hefty speaking fee) associated with the appearance of Linux Torvalds at a conference – Torvalds has become a highly sought after expert and celebrity, because his gift to FOSS was so enormous. However, reputation building is complicated by the immateriality of the gifts – transient bits of code that vanish into the project. “Status is delicately dependent on the critical judgement of peers” (Raymond, 2001, p. 84).

Second, Raymond notes, while reputation may have become an important reward in FOSS development, it was not necessarily the incentive for improving the code in the first place. The joy of writing the code (or the relief of solving a problem) might be the underlying motivator and, in a bit of circular causation, the abundance of gifts enables further realization of that joy. “The reputation game may be critical in providing a social context within which the joy of hacking can in fact become the individual's primary motive” (Raymond, 2001, p. 84). Söderberg calls this emphasis on personal needs and satisfaction “production for use” and contrasts it with “production for exchange,” claiming that “to a hired programmer, the code he is writing is a means to get a paycheck. . . for a hacker, on the other hand, writing code is an end in itself” (Söderberg, 2007, p. 84). Hackers, including Raymond, are fond of referring to the pleasure of hacking as “scratching a developer's personal itch” (Raymond, 2001, p. 23). I will discuss this subject in depth in the next section.
The gift economy description of FOSS, however, is not without its critics. Steven Weber suggests that the model is flawed, because no matter how abundant computing resources have become, programming resources are still quite scarce (Weber, 2005, pp. 150-151), and the programming labor is what FOSS is giving away for free. Weber uses this criticism to shift the discussion away from the economy of the gift and toward the economics of “antirival” goods, or goods which become more valuable the more they are used (2005, p. 154). A simple example of the antirival property is bug detection: “the more users . . . actively engage in using a piece of software, the more likely that any particular bug will surface,” which increases the likelihood that someone will fix it, improving the software for everyone (2005, p. 154).

An alternative approach to thinking about the economics of FOSS, however, is to focus on the firm, rather than gifts and goods. How does decentralized, peer production of open source software affect the status of the firm? In 1937, Ronald Coase attributed the existence of firms to the prohibitive burden of transaction costs in a firm-free market. "Within a firm, these market transactions are eliminated and in place of the complicated market structure with exchange transactions is substituted the entrepreneur-co-ordinator, who directs production" (1937, p. 388). This widely accepted fact is certainly the basis for firms' pursuit of horizontal and vertical integration, both in the past and present: enclosing more of the production process in a single firm increases profits by lowering transaction costs.
Clay Shirky, however, has argued that some activities, typically ignored by firms because they were below the "floor" of what a firm would willingly invest in, are now becoming possible because they can be organized outside the firm, thanks to plummeting transaction costs. "The collapse of transaction costs makes it easier for people to get together" (2008, p. 48). Weber makes the same observation, and describes it as one of the keys of the antirivalness of FOSS (2005, p. 155).

Shirky also describes the open source world as failure tolerant (2008, p. 245), meaning that most open source projects fail to take off, but the costs of failure are so low that it's just not worth worrying about. "In the open source world, trying something is often cheaper than making a formal decision about whether to try it" (2008, p. 249). This is a stunning observation, in that it completely relocates the fundamental question of whether a program – or a particular feature within a program – ought to exist. Instead of depending on a human being to argue that it ought to exist, a program depends only on that human being to write the code, and then the code (or the program, to the degree that the two are indistinguishable) argues for its own existence.

While Shirky's utopian view is compelling, it also has its limits, and it must be recognized that open source software is not completely removed from the firm. The four most recognized open source projects are probably Linux, Apache, Firefox, and OpenOffice, and all four have rich historical ties to commercial companies. Both
Weber and Raymond dedicate entire chapters to hybrid business models that use FOSS to turn a profit.

It is also worth noting, though, that while Shirky's observation referred explicitly to open source, it could easily be extended to proprietary technologies and firms in the digital economy. For instance, in the emerging subindustry of microblogging, an industry that has not yet even monetized, the number of startup companies can only be explained by a certain degree of failure tolerance – it is cheaper and easier to start such companies, even if they fail, than it is to evaluate in advance whether they will succeed. Rather than undermine Shirky's view of FOSS's exceptionality, however, this observation merely points to a parallel trend – a blurring and de-sanctifying of the traditional boundaries around work and the firm. This blurring has been explored at length with respect to the work/play performed by hackers, but is also broadly recognized at the popular level – technology startup companies are forever linked with the image of a ping-pong table in the office. While the cultural impact of hackers on the workplace is beyond the scope of this project, Chapter 4 will discuss, to some extent, how proprietary software development has absorbed some of the technological lessons of FOSS.

**The Ethics and Politics of Hackers**

While many FOSS developers, this author included, would not self-identify as “hackers,” this reluctance is undoubtably influenced by the negative media
representation of all hackers as criminals, and disregards the undeniable historical and philosophical links between hacking and FOSS development. The code-breaking activities, often illicit, that are popularly associated with hacking are better described as “cracking,” a subculture of the larger hacker culture which also includes FOSS development and has been the subject of much scholarly work. Hacking, on the other hand, emerged not from the dark alleys of cybercrime but from the halls of government and academe, including the computer labs at MIT (Himanen, 2002, p. 4), an important but often inconspicuous shadow of the rise of computing and the Internet. Still, having emerged on American campuses during the radicalism of the 1960s, hacking has always been rooted in politics (Söderberg, 2007, p. 15).

Pekka Himanen has described the work of hackers as governed by “the hacker ethic,” contrasted with the Protestant work ethic that accompanied the rise of capitalism. The Protestant work ethic, of course, is the idea that work (and working hard) is an end in itself, “which must be done because it must be done” (Himanen, 2002, p. 9). The hacker ethic, on the other hand, is the performance of work (even hard work) out of passion for the work; the fulfillment of that passion, not the work itself, is the end of work. According to Himanen, social acceptance under the Protestant work ethic comes from the performance of hard, dutiful work, even if it comes at the expense of the pursuit of individual passions. “When social motives do not find an ally in passion, they become allied with survival, and life becomes concentrated on 'making
a living” (2002, p. 52). Under the hacker ethic, on the other hand, peer recognition results from passionate action, because communities are organized around shared passions (2002, pp. 50-51).

In addition to its significance for social recognition, the hacker ethic has altered the relationship of the worker to the workplace and working hours. The Protestant work ethic, with its emphasis on the optimization of work time, has caused time optimization to creep from work life into home life, such that weekends are tightly packed with scheduled activities. Himanen calls this the “Fridayization of Sunday” – the imposition of a work week ethos onto non-work time - and contrasts it with the “Sundayization of Friday” – the integration of a playful (and time flexible) ethos into the work week – that results from the hacker ethic (2002, pp. 26-33). Both ethics result in the optimization of time and the blurring of boundaries between work and play, but whereas the Protestant work ethic optimizes time strictly to get more work done, “hackers optimize time to be able to have more space for playfulness” (2002, p. 32).

Johan Söderberg, in Hacking Capitalism, attributes to Himanen the idea that “volunteer involvement in FOSS development projects can be explained by the extent to which primary needs have been satisfied in consumer society” (Söderberg, 2007, p. 100), but this is an overstatement and oversimplification. While Himanen does discuss a simplified version of Abraham Maslow's hierarchy of needs – “survival, social life, and entertainment” (Himanen, 2002, pp. 48-49) – his argument about the rise of
hacking does not depend on excess abundance, as Söderberg suggests, except to the degree that consumer society itself depends on abundance. Rather, Himanen's argument rests on the realigning of these needs in society; the hacker ethic adjusts Maslow's hierarchy to put a greater emphasis on passions.

One way of considering this realignment of needs is through Marxist critique. While Söderberg's work is arguably the most complete Marxist analysis of open source, and will be discussed in detail in a moment, he was preceded by another important work, McKenzie Wark's “A Hacker Manifesto,” which uses a very inclusive notion of a “hacker class” to reorganize the classic Marxist class struggle around knowledge production instead of material production.

Wark's writing is playful and poetic, filled with double meanings. He writes, for instance, that “hackers are a class, but an abstract class, a class as yet to hack itself into manifest existence as itself” (Wark, 2004a, sec. 3). Obviously, the primary referent in this use of the term “class” is the Marxist notion of economic class, but the words speak on a different level to hackers themselves. In object-oriented programming, particularly in C++, an abstract class is a sort of meta-object from which other classes can be derived, but which cannot be instantiated as an actual object itself. For instance, to clarify with a more familiar example, “mammal” could be considered an abstract class. It has many derived classes (caribou, mice, leopards, and so on) which are
instantiated many times in the natural world, but “mammal” itself cannot be instantiated, because it is an abstraction.

One of the advantages of using abstract classes in code is that they allow programmers to treat a wide variety of derived classes as if they are of the same class — the abstract class — and to access all the methods and data shared by that abstract class. If those methods involve communication between objects, this polymorphism, as programmers call it, becomes a common interface allowing different objects to relate to one another without explicit knowledge of any of the features of the other derived classed in the system. So Wark, by describing hackers as an abstract class, is both broadening the hacker class to include many different subclasses (programmers, scientists, musicians and all other types of knowledge workers), and arguing that this inclusiveness can be the basis for a complex social movement: “to abstract is to construct a plane upon which otherwise different and unrelated matters may be brought into many possible relations” (Wark, 2004b, sec. 8).

The heart of Wark's argument is his update to Marx: with the increasing importance of informational goods in the digital economy, a new class conflict has emerged, between information producers (the hacker class) and the owners of the means of information production, including patent and copyright holders, which he calls the vectoral class (Wark, 2004b, sec. 21).
Söderberg makes a somewhat analogous argument, that the labor struggle of workers is being replaced by the play struggle of hackers – "resistance has here become a game" (2007, p. 183) – and that the hacker movement is part of a larger revolt "against the boredom of commodified labour and needs satisfaction" (2007, p. 44).

In other words, for Söderberg, living standards have improved enough (in developed nations) that workers, particularly middle class knowledge workers, no longer have an incentive to work harder to improve their living standards, or to participate in the antiquated labor struggle, and they are instead simply becoming bored with work (and here Maslow is clearly at least as important to Söderberg as he was to Himanen.) The expression of this boredom, for Söderberg, is play struggle. Noting that scholars frequently fixate on the question of hacker's incentives - "What drives hackers to write code when there are no direct economic incentives for them to do so?" - Söderberg dismisses the very idea that hackers are motivated by the market: "All of that is invalidated once we start taking play seriously" (Söderberg, 2007, p. 165).

Söderberg argues that the underlying productive processes of hacking are more important than the products that result, or even the methods of distribution. "It is not pirate sharing that makes peer-to-peer networks subversive . . . but the peer-to-peer labour relations of which this technology is an example" (2007, p. 123). This
perspective is highly debatable when taken to its extreme, but is also a very powerful way of expanding theories about hacking to social trends beyond hacking.

Open source software and the copyleft movement are occasionally characterized as communist, socialist, anti-capitalist or Marxist – for instance, when asked to address calls for copyright reform, Bill Gates dismissed the movement's members as “some new modern-day sort of communists” (Kanellos, 2005) – but these are shallow criticisms based on a simplistic interpretation of FOSS as anti-property or anti-business. While Gates' the pejorative tone is largely absurd, such comments – particularly when coupled with the works of Wark and Söderberg – correctly suggest that open source software development does make important ideological arguments that run contrary to some of the normative values associated with capitalism. Echoing Söderberg, is open source software provocative to capitalism because it is free, or because it is authored in a way that subverts the labor-wage-consumption relations that are so central to Post-Fordist capitalism?

However, the Marxist perspective is not the only approach to understanding the subversive qualities of FOSS development and, given the diversity of political views among hackers, it may be overly limiting. Tim Jordan, while still informed by Marxist thought, approaches the hacker ethic from a less ideological constrained perspective.
Jordan offers perhaps the most poetic definition of hackers: "warriors of technological determinism" (Jordan, 2008, p. 13) which he contrasts with the “programming proletariat” (2008, p. 98) of the major software corporations, who surrender their hacking weapons to the market, in exchange for a salary. Jordan rejects pure technological determinism, as do most scholars, because technologies are never completely asocial, but argues that we nonetheless experience something like technological determinism in our daily lives. Our daily experiences are often determined by technology, particularly when technologies fail – a broken toaster, for instance, determines that its owner cannot have toast for breakfast. This might be described more accurately as technological mediation, but Jordan prefers the connotations of technological determinism, a foreboding enemy for his hacker warriors to fight.

Hacking, then, is the process of pushing back against the experience of being technologically determined. Hackers "identify where a technology is determining them in ways they dislike . . . and they engage in altering that technology, which thereby automatically produces new ways in which technology can determine action" (Jordan, 2008, p. 15) – hackers are scratching an itch, as Raymond puts it, but for Jordan itch-scratching is a political act.

This chapter has provided an extensive (but not exhaustive) review of the existing literature about FOSS and FOSS communities. Readers interested in the finer
details of the history and current state of open source, in terms of technologies, policies, and intersections with business, are advised to consult Deek and McHugh (2007) and Weber (2005). The gaping hole in this literature, however, and the subject of the next two chapters, is a general discussion of what programmers actually do. Of the authors discussed so far, Eric Raymond (2001) has given the most attention to the physical work of programmers, but his emphasis is on the narrative of development rather than the practices. What follows will instead focus on programming languages, and how, in practice, they shape learning, development, and distribution.
Chapter 3. Writing Code: Iteration & Abstraction

In Chapter 2, I gave an overview of the literature concerning FOSS software and communities. In the next two chapters I will reveal and expand a gap in that discourse – the study of how programming actually happens, and how the practices of programming relate to the ideology of the FOSS movement. “Hacker knowledge,” writes Wark, “implies, in its practice, a politics of free information, free learning, the gift of the result” (Wark, 2004a, sec. 70). Deek and McHugh claim that “programmers are often autodidacts: open-source helps that happen” (Deek & McHugh, 2007, p. 71), but this chapter considers that relationship from the opposite direction: programming is a process of situated learning, which encourages both autodidactic self-teaching, through iterative experimentation, and the politics of free information and open source. It does this, primarily, through two genres of activity: iteration and abstraction.

There is some precedent for my practice-oriented approach. Deek & McHugh dedicate an entire chapter to the “technologies underlying open source development,” including a lengthy discussion of the CVS (Concurrent Versions System), a source code versioning system which helps programmers collaborate on code by helping them track and merge multiple versions of their files. In their close examination of various CVS commands and their implications for developers, they note in particular the importance of the “copy-modify-merge” model used by CVS compared to the “lock-modify-unlock” system used by earlier systems (2007, pp. 121-122). However, they
understate the politics of this design, noting that it “smoothes the interactions in . . . distributed development” without arguing that it actively encourages distributed collaboration (2007, p. 122).

In a similar vein, in the popular but highly intelligent YouTube video “The Machine Is Us/ing Us,” anthropologist Michael Wesch explores the political implications of XML and related technologies, focusing on the ways the language itself restructures the relationship between users and content: “with form separated from content, users [do] not need to know complicated code to upload content” (Wesch, 2007b).

Video game scholars James Paul Gee and Ian Bogost have also set precedents by studying, respectively, how computer mediation alters learning and enables a new form of rhetoric. Their work will be explored further, below, following the introduction of this chapter's primary object of study: Hello World.

**Hello World: Learning to Code through Iteration**

The first sample program in an Intro to Programming textbook is typically the “Hello World” example, a very simple program that merely outputs the words “hello, world” (or some variant) to the default display, which may be a browser, a console, or any number of other displays. The code varies from language to language, but
generally looks something like this classic from *The C Programming Language* (Kernighan & Ritchie, 1978):

```c
#include <stdio.h>
main( ) {
    printf("hello, world\n");
}
```

Hello World has at least three pedagogical purposes. The first is to introduce the appearance of code to the textbook's reader, distinguishing it from the book's non-code text. As it would be in a textbook, this purpose is achieved here with the simple example above – the snippet of code informs the reader that, in this text, code will be set apart through the use of single-spaced line breaks, a sans-serif typeface, and tabbed indentation.

The second purpose of Hello World is to provide a skeleton for more complicated programs, which includes the essential but not always intuitive starter code required to build *any* program in the particular language. The meat of Hello World, the part that performs the critical printing action, is usually just a single line of code, which the programmer can then delete and replace with her own code as she writes more sophisticated applications. In other words, by providing skeleton code, Hello World is teaching the programmer two simple lessons about the language: “*This is how you structure a working program in this language*” and “*your code goes here.*”
The third purpose of Hello World is less obvious, but equally significant. The words “code” and “program” are often used interchangeably, and in the next section I will argue that the code and the application are a single entity, but it is still important to remember that source code is just text; source code does not do anything by itself. The code provided above, when printed on paper, is not yet a program – it only becomes a program when it is compiled or interpreted by a computer and can then be executed. Every language is different, and requires different steps to move from code-as-text to code-as-application – sometimes including the compilation of the source code into machine code – and the Hello World code is often used as a fail-safe starting point to walk through those steps. At the end of a successful execution, the simple rendering of the words “hello, world” is a reassuring sign that the programmer has followed the steps correctly and is ready to start doing real work.

This reassurance has even made the leap out of the code layer and into the content layer and is now sometimes used to indicate a successful installation or configuration of third party software. For instance, when a user creates a new blog using the popular Wordpress software, a post titled “Hello World” is generated automatically, giving them the green light to start blogging: “Welcome to WordPress. This is your first post. Edit or delete it, then start blogging!” (“WordPress Trac,” 2008).

In the case of early C programmers, there were three steps to get from code to output: (1) save the code as “hello.c”, (2) compile the code into executable machine
code with the “cc hello.c” command, and (3) execute the code by typing “a.out” (Kernighan & Ritchie, 1978, p. 6). For a contemporary C++ programmer, working in a visual development environment such as Microsoft Visual Studio, these steps have been reduced to a simple press of the F5 key. A new PHP developer may have to use FTP to transfer her program to a PHP-enabled webserver, then open a browser and direct it to the server version of that file. A Javascript programmer, on the other hand, when writing inline script in an HTML document, need only open the document in a browser.

The steps vary dramatically, but the outcome of the exercise is the same: the programmer learns the basic steps to execute their code. To the two material lessons of Hello World mentioned above (“this is how you structure a working C program” and “your code goes here”), a third lesson can be added: “this is how you make it run.” Echoing Tim Jordan's “Does it run?” criteria for code, this final lesson may be the most important part of Hello World programs, not only because it highlights the importance of execution, but also because it reveals that, from the very beginning, writing code and learning to write code are reciprocal, iterative processes. An amateur programmer is not expected to learn the intricacies of a programming language before compiling her first program – in fact, she does not even need to learn the basic syntax or keywords for the language. She just has to copy the Hello World program and learn to execute it. In all likelihood, she'll then think “It works! What should I try next?” Over
the course of many, many iterations of writing and executing small changes to the code, she will learn the language. Learning by doing is hardly a new idea, dating at least as far back as Aristotle—“the things we have to learn before we can do them, we learn by doing them” (Aristotle, 2005, bk. II-2)—but for the programmer it is a bit of an understatement. For the programmer, doing is learning, and learning is doing.

“Every hacker is at one and the same time producer and product of the hack, and emerges as a singularity that is the memory of the hack as process” (Wark, 2004a, sec. 158).

James Paul Gee's notion of “situated understanding,” which he contrasts with “verbal understanding,” is helpful in clarifying the significance of the way programming languages are learned. Verbal understanding, of course, corresponds to traditional modes of learning, whereas situated understanding “starts with a relatively concrete case and gradually rises to higher levels of abstraction through the consideration of additional cases” (Gee, 2007, p. 113). For a programming language, Hello World is quite obviously the concrete first case, and the higher levels are the ensuing lessons which unveil the infinite possibilities of the language. Gee, a scholar of video games, argues that written texts about video games, specifically gameplay manuals, “are not very meaningful, certainly not very lucid, unless and until one has played the game” (2007, p. 114), because while the technical language is quite easy to understand literally— it
simply describes which buttons perform which action – the literal understanding is very hard to put into practice without also gaining a situated understanding, i.e. without trying it. Similarly, reading about a programming language is fairly meaningless – the programming language is learned through situated experiences, not verbal explanations.

Jordan has echoed the importance of situated learning in computer mediated activities, describing the cracking community as a paradoxical game of secrecy and peer recognition (Jordan, 2008, pp. 30-32), a cat-and-mouse version of mentorship in which more experienced hackers are willing to share and teach, but only to those who have proved themselves worthy through self-teaching and experimentation. The relationship between doing and learning, for coders, is somewhat analogous to that of crackers, but for coders the game of cat-and-mouse is often with the language and development environment itself, rather than an external mentor. In both cases, though, learning is tightly bound to trial and error, to small tests and experiments; as described above, the work of of learning to program is highly iterative and highly situated. At the start of this chapter I noted Deek and McHugh's claim that “programmers are often autodidacts” (2007, p. 71), but, in light of the iterative nature of programming, perhaps learning a programming language just seems to be autodidactic, even when it is being formally taught, because situated learning looks a lot like self-teaching.
Gee's work on situated learning is complemented by Ian Bogost's work on “procedural rhetoric” in video games. Bogost is particularly interested in small games that make political arguments. “When we acknowledge videogames as a medium, the notion of a monolithic games industry, which creates a few kinds of games for a few kinds of players, stops making any sense” (Bogost, 2008). By procedural rhetoric, Bogost means “the practice of using processes persuasively” (2007, p. 28) and, while he does build on theories of general rhetoric as well as visual rhetoric, Bogost is less concerned with theoretical origins and more interested in what sorts of rhetorical arguments games can make and how designers can implement them. He focuses specifically on games that explicitly teach players about a political issue by simulating the systems behind the issue. Bogost walks in the footsteps of Marshall McLuhan's “the medium is the message” (McLuhan, 1964, chap. 1), but with an emphasis on crafting those messages in the more malleable and interactive medium of programming and video game design.

Can procedural rhetoric help us understand the politics of FOSS, or the nature of programming? Individual video games, of course, are created by teams of game designers and programmers, who delicately craft the player's experience, so it is easy to understand how and why they might contain procedural rhetoric. In the case of general programming practices, however, procedures do not have such a clear origination – programming languages, environments, and practices are incredibly
varied, and no individual or collective has the influence to shape processes at such a low level, across the board. Determining and discussing the arguments being made “by the code,” then, is a bit trickier for this medium, perhaps limiting the value of procedural rhetoric in a discussion of FOSS. Bogost is mostly concerned with authoring procedural rhetoric, not reading it, so his work is not particularly helpful for identifying and interpreting the procedural rhetoric at work in programming practices, but he does make it quite clear that such rhetoric does exist, and he does attempt to move beyond video games – “outside of videogames, procedural tropes often take the form of common models of user interaction” and, later, “procedural genres emerge from assemblages of procedural forms” (Bogost, 2007, pp. 13-14). These common models of interaction, in programming, are the focus of this chapter, and the concept of a procedural genre will be invaluable in linking programming processes, development methodologies, and interface design to the relatively distant FOSS movement.

A Failure Tolerant Practice

When Clay Shirky said that FOSS development was failure tolerant, he was thinking at the project level – it is often easier and faster to write and release a FOSS application, and then let the crowd decide if it is worth further development, than to try to decide in advance whether it is worth starting the project. However, Shirky's logic actually translates quite well to the code level – trial and error, or compile and test, is a
fundamental programming task, then programming must be understood to be rather
failure tolerant. Consider these two PHP statements:

```php
$greeting = 'hello, "' . $name . '"';
$greeting = 'hello, "' . $name . '"';
```

While the two statements appear identical at a glance, the second statement
actually has a typo which will cause a parsing error: the last single-quote and the last
double-quote have been reversed. These sorts of errors are extraordinarily common in
newly written code, particularly when a programmer has been cutting and pasting or
writing large amounts of code at once. Some programming tools will catch this error
on the spot, calling attention to it in roughly the the same way that modern word
processors underline misspelt words, but the programmer will often find it another
way: by trying to execute the code, leading to the following error message:

```
PHP Parse error:  syntax error, unexpected T_STRING
in /hello.php on line 2
```

The error message identifies the line number that the error occurred on, and the
programmer will quickly learn that “unexpected T_STRING” usually indicates mis-
matched quotes – the parser both corrects and teaches. So, while a programmer will
always try to write syntactically correct code, she will not proof-read it in the same
way that a journalist proof-reads an essay – she'll let the iterative process expose and
correct them in due time. The point here is that writing code, to use Shirky's words, is a
remarkably failure tolerant process. It is often easier to write code quickly and sort out
the problems through iterations than to try to write perfect code the first time, or to
discover problems before executing.

This failure tolerance plays out not only at the compiler stage, with respect to
code syntax, but also during the execution stages. If the code runs, but does not do
what the programmer intended, it really does not matter – she can just keep tweaking
the code until it works. Even in the worst case, that a bug causes a crash, the effects are
usually relatively inconsequential, because the application can just be started up again.
When a programmer's application crashes, the experience is quite comparable to the
death of a player in a video game – the crash itself is irrelevant. The only two things
that matter are whether the crash or death causes work to be lost – programmers and
gamers alike save their work frequently – and whether something was learned from the
crash or death. Gamers, of course, learn new skills and strategies from their deaths; for
the programmer, the crash often provides clues towards its own causation through error
and exception codes, and magic numbers (like “BAADF00D”) that help the
programmer diagnose memory violations.

That coding is crash tolerant should not suggest that software with crash bugs is
tolerable. Jordan's “does it run?” criteria implies, correctly, that code that crashes will
be judged particularly harshly by users. Every owner of a personal computer during the
1990s recalls the miserable experience of the Windows “Blue Screen of Death,” and
most have been devastated by some sort of crash-related data loss, even if only a single
document. Crashing is universally abhorred by users. Rather, referring to programming as crash tolerant instead highlights the degree to which crashing bugs are a completely normal part of the iterative development process. A programmer expects to write code that crashes.

**Dummy Code: Building Complexity**

In a sense, because of the “your code goes here” message embodied within it, the Hello World example discussed above can be thought of as a specific case of a more general concept: dummy code. Dummy code is temporary code that performs only a minimal portion of its eventual functionality, but allows the programmer to pretend that the functionality exists so she can develop other parts of the application. Just as Hello World allows a new programmer to compile and execute her first application, without worrying about actually engineering that application or learning the entire language, dummy code allows an experienced programmer to build complex architecture into an application without worrying about writing every line of code in the first pass. Consider the following function:

```cpp
int document::autosave( ) {
    // TODO: autosave to temp file
    return 0;
}
```

This function, obviously, does not actually do anything. However, knowing precisely what the code will eventually do – automatically save a document to a
temporary file, in case the application crashes – allows her to continue programming as if the functionality already exists. So, for instance, if she wants the document to be saved automatically whenever the user attempts to print, she might write something like this:

```cpp
int document::print( ) {
    this->autosave();
    ...
}
```

The significance of this approach to development is not immediately obvious, and might be best understood through comparison with a form of production that follows a dramatically different developmental approach: baking. When a chef is making a loaf of bread, he only bakes it once. In fact, he only *can* bake it once. If he is making banana bread, for instance, the chef does not leave the bananas out while he bakes the rest of the ingredients – just to be sure they work – and then add the bananas in later. The bread is baked, and there is no turning back to add bananas. While a recipe can change over time, through gradual changes introduced in each iteration, and a particular batch of dough might change through iterations of tasting and adjusting, any single material loaf can only go through one full, end to end iteration.

The difference, with software, is rooted in the fluid materiality of the application, versus the fixed materiality of the loaf of bread. The material loaf, once baked, cannot be refined and baked again. The software application, on the other hand, is never permanently baked – it can be infinitely refined and still be, to the
programmer, the same application. If an application has already been distributed and
the developer wants to release it again with new features or bug fixes, she does not
release “a new application” – she releases “a new version” of the same application. Put
slightly differently, a loaf of bread with a dangerous bug would be recalled from stores,
whereas an application with a dangerous bug would merely be patched, or upgraded.

Another way to frame the distinction I am drawing between software
development and other forms of development is to consider the relationship between
the design and the end product. In software development, the design becomes the
product (and it does so very early in the process), and when changes are made to the
design they manifest immediately in the product. Once code has been written – even if
it is merely skeleton code or dummy code – the design is no longer separable from the
application. For the baker, on the other hand, the design (the recipe), quite obviously,
is entirely separable from the product (the bread).

The obvious objection here is that the application can be separated from the
design – code (the design) can be compiled into binary form (the application) and
distributed entirely separately, as is typically done with proprietary, commercial
software. However, as will be discussed in Chapter 4, this separation is only possible
with certain programming languages and, further, the separation is not bidirectional –
while copies of the application can be distributed without the code, the code cannot be
distributed without the application, because it is the application. The design and
application coexist, coevolve, and can be understood, for the most part, as a single entity.

Why does this matter? The core questions of this paper concern the relationship between the material practices of programming and the principles of open source software development, as well as the formation of FOSS communities. If, as argued through the Hello World example, the act of writing code is fundamentally iterative and if, as just argued, the code and the application are understood to be a single entity that goes through many revisions, then the application must also be understood as fundamentally iterative. Again, Tim Jordan notes that the reason that some Creative Commons licenses restrict “derivative works” is because the maker of a creative work, such as a painting, might want to declare her work “done” so that it cannot be further edited (2008, pp. 104-105). FOSS licenses, on the other hand, do not allow restrictions of this sort, because software applications are never “done.” The iterative process of programming has, in this sense, been codified in the FOSS license.

**Imperative Programming and Code Abstraction**

It is common knowledge that there are many different programming languages, but not that there are many different *kinds* of programming languages, which take different approaches to defining the relationship between the programmer and the computer. The most common paradigm, however, and the focus of this chapter, is imperative programming, in which code statements represent a chain of instructions
for the computer to process in a specific order. Imperative languages, which dominate the landscape in both FOSS and proprietary software, include all of the most famous and popular languages (C/C++, Java, PHP, perl, C#, Python, BASIC, etc), although some of these languages also support other paradigms.

The history and taxonomy of programming languages, however, is surprisingly complex. While there is a sentiment among programmers that, once you have learned one language, learning another is simply a matter of learning a new syntax, that sentiment only works within a single paradigm. Switching from an imperative language like C to a declarative language like Lisp, in which statements express general and non-sequential logic instead of explicit algorithms, involves a paradigm shift that is far from trivial. In a sense, it is analogous to the difference between moving from English to Korean, rather than from English to German; the shift to Korean involves fundamental challenges to the speaker's understanding of how ideas are expressed through language. The analysis that follows is situated in the history and concepts of imperative languages, and other paradigms have not been considered.

Despite their emphasis on sequential operations, programs written in imperative languages are not necessarily linear; in fact, only the simplest examples, such as Hello World, would remain linear. Conditional statements (including “if-then-else” statements and various looping statements) allow execution to “branch” in multiple directions, and the number of paths through the code can grow exponentially as
conditional statements are added to an application. However, local segments of those paths can always be understood linearly, and debugging tools for imperative programming languages actually allow programmers to “step” through each statement in order – inspecting the state of the application and data along the way.

Much of the work of programming consists of accounting for the many paths through the code, and the variety of conditions that can result, and many bugs result from overlooking the simplest of these conditions. For instance, the following line of BASIC code will cause a “divide by zero exception” if the QUANTITY variable is equal to zero.

90 LET AVERAGE = SUM / QUANTITY

In early imperative programming languages, like BASIC, debugging errors was extraordinarily difficult, because the programmer could only inspect the application's current state, at the point of the error. Little information was readily available about how the application had reached that point, although understanding could be reached – with difficulty – by digging backwards through the code and the memory stack. In the example above, for instance, if the quantity was zero and an exception was thrown, the programmer might have to search the entire source file to find every location where the quantity might have been set to zero. For a complex bug in a complex program, the programmer might have to read and understand every line of code.
Some imperative programming languages greatly simplified this problem by introducing what is called “procedural” or “structured” programming: named procedures or functions that can be called from anywhere in the code. While procedural programming has many advantages over straight imperative programming, one of the most important advantages is modularity – the separation of code into logical chunks. Earlier, it was shown how modular code is a useful way of blocking out skeleton code, enabling iterative development even in complex applications, but modularity is also profoundly useful for debugging. Consider the following example written in C, a function that calculates the average value of an array of floating point numbers – a task similar to but more complex than the line of BASIC discussed above.

```c
float average(float values[], int quantity)
{
    int i;
    float sum = 0.0;

    // first, total up the values
    for( i = 0; i < quantity; i++ )
    {
        sum = sum + values[i];
    }

    // then return their average
    return sum / quantity;
}
```

If a divide by zero exception were thrown by the last statement in this example, because the quantity was zero, it would be quite easy for the programmer to determine one very simple fact: the quantity was set to zero outside of the scope of this function,
because the function never modifies the value of the quantity variable. This small fact may seem trivial, but that is only because the example itself is trivial – debugging a procedural language, under a good debugging tool, is immeasurably easier than debugging a non-procedural imperative language. By extension, making sense of and extending a program in such a language is immeasurably easier, as is collaborative development.

The leap from imperative programming to procedural programming was nothing more than a leap of code abstraction. In BASIC a programmer could jump through code with the “GOTO” statement, but in C a similar step would now be accomplished by calling the name of a procedure or function. Procedural programming, however, also created a new interface (in code) for these jumps, which provided a helpful context for editing and debugging code, and which also made collaborative programming profoundly easier. No longer did a programmer have to wade through all of the surrounding code to make sense of the piece she was looking at – the purpose, scope, and context of a section of code was clearly defined by the procedure in which it resided.

To clarify, in both BASIC and C (and all other languages), the source code is ultimately converted into machine code by either a compiler or interpreter. For simple programs, that machine code could, conceivably, be exactly the same for both procedural and non-procedural languages. The key is to understand that the language is
an interface, or a toolkit, to create that machine code, and that the abstraction from imperative to procedural programming is an abstraction not in an arbitrary direction, but an abstraction away from the machine code – moving from an incomprehensible language to increasingly readable languages. In a sense, the barriers to entry, both for programming as a whole and for any particular project, were lowered by this abstraction.

After procedural programming, the next major abstraction in imperative programming was object-oriented programming (OOP), originally synonymous with the Smalltalk language (Wilson, 2001, p. 41) but popularized by C++ and Java. “Object-oriented programming took off in a big way in the 1980s, and by the late 1990s had become a dominant language paradigm” (Wilson, 2001, p. 39). In object-oriented programming, procedures – which can be called from anywhere at any time – were replaced by object methods. Methods are like procedures, except that they are tied to a class and can only be called if an instance of that class is available, on hand. The procedure call

```c
print(&document);
```

is replaced with

```c
document->print();
```

The action of printing becomes a function of the object itself, not an externally defined process. Recall, from Chapter 2, the second meaning of McKenzie Wark's
“abstract class” – in object-oriented programming, an abstract class is a class of objects that cannot itself be instantiated, but from which other classes of objects can be derived. This property, inheritance, allows object-oriented programmers to make use of polymorphism – the ability to treat an object as a member of its own class or as a member of the class(es) it is derived from.

Objected-oriented code looks a lot like procedural code, but writing it involves a much different thought process. Computer scientist Ralph Westfall has argued that the typical “Hello World” example for Java is biased towards procedural programming, at the expense of the object-oriented paradigm that is Java's strength, because “it doesn't instantiate any objects [or] use any object behaviors,” opting for the fewest number of lines rather than an object-oriented structure. Westfall proposes a slightly more complicated Hello World example for Java, in order to embed object-orientation in practice from the start: “this will help imprint the concept immediately, so they do not have to ‘unlearn' a procedural approach” (Westfall, 2001). Westfall's observations are important not only because they highlight the contrast between procedural code and object-oriented code, but also because they echo Gee's notion of situated learning: programming concepts are learned by typing them in and running them.

Wark, as noted earlier, argued that abstraction creates a plane of new possible relations (Wark, 2004b, sec. 8). It is quite easy to see how the abstraction from non-procedural to procedural programming, and then to object-oriented programming,
created many new possible code relations. Procedural programming meant that procedures could be called from any piece of code using a known syntax, that libraries of procedures could be used in many different applications, and that applications now had a callstack, a nested list of procedure calls within procedure calls, which contained contextual information about the path of execution and the state of the application in each nested procedure.

However, far more profound than these code relations are the new human relations enabled in this new plane of abstraction: the increasing abstraction of code dramatically reduced the overhead of collaboration, both by breaking the code up into manageable chunks and by replacing impenetrable machine code with increasingly structured and readable code. Without code abstraction, in all its varying degrees and forms, FOSS communities simply could not exist, because collaboration would be so much more difficult. There is a pejorative term used among programmers, for code that is difficult for anyone but the original author to understand: spaghetti code. In the domain of large, complex, 21st century applications, it is not a stretch to say that, without abstraction to hide complexity, all code would be spaghetti code.

The evolution of programming languages, of course, occurred alongside the rise of microprocessors, then the PC, then the Internet. Along the way, proprietary software was born, and a commercial software industry emerged. Not surprisingly, as applications became more complex, the commercial industry experimented with
different development methodologies – searching for the most efficient model, in terms of both speed and cost – and, like most business processes, they continue to be highly disputed.

In recent years, debate has been raging between proponents of Agile development and Waterfall development. The Waterfall development model, the conventional way of doing things, consists of a series of discrete stages – requirements, specification, design, implementation, verification, and maintenance (Chopra & Dexter, 2005, p. 3). A project does not enter the next Waterfall stage until it has fully completed the current stage. Agile development, on the other hand, de-emphasizes rigid planning in favor of short development cycles and frequent changes to the specification and design (Huo, Verner, Zhu, & Babar, 2004). Agile acknowledges the iterative nature of programming and incorporates this characteristic into its methodology.

**Does Code Want to be Free?**

Does this sound familiar? Are Waterfall and Agile merely alternate labels for Eric Raymond's Cathedral and Bazaar? “Release early, release often” Raymond wrote, describing Linux Torvalds' approach to Linux development (2001, p. 21), but the phrase has also come to be a mantra for Agile developers (Daly, 2008). Agile and the Bazaar, of course, are not equivalents; the former is a development methodology for a structured team of developers, the latter an approach to unstructured, participatory
development. The overlap between these two methods, however, validates the notion that iteration and abstraction are meaningful concepts for understanding trends in software development, including not only the trend towards Agile and the Bazaar, but also the trend towards FOSS applications.

There are other trends, of course. Lessig, for instance, has argued that modular code delivers some of the benefits of open source code, even if it is closed source (Lessig, 2000, p. 225). In its simplest form, the suggestion does not make much sense – whether code is modular or not is irrelevant if you cannot actually see the code or hook into its functionality – but Lessig was discussing not modular code within a given application, but modular relationships between applications. To some degree, these relationships can be achieved through the use of APIs (Application Programming Interfaces). APIs, in effect, allow programmers to build extensions and complementary applications that can execute code related to an application – often accessing its data – without actually requiring access to the source code. For instance, microblogging website Twitter.com has a promiscuously open API that allows programmers to “make applications, websites, widgets, and other projects that interact with Twitter” (“FAQ,” 2009). In a sense, Lessig is suggesting that modularity allows a sort of transparency and interoperability analogous to that of open source, even when the source code is not made available.
By flipping Lessig's observation and considering it from the perspective of material practices, however, it becomes considerably more potent: if the material practices of programming embody a rhetoric that encouraged the invention of object-oriented programming, of modular code, of APIs, and of Agile development, might it not also encourage the use of open source licenses?

When linking the techno-centric practices of programming to the ideological and political position of the FOSS movement, there is a danger of lapsing too deeply into technological determinism by arguing that code, as a technology, wants to be free, and influences society accordingly through the work and politics of programmers. This is an overstatement, of course, because all of the technologies involved in this discussion are both historically and socially determined to various degrees. However, as quoted earlier from Wark, FOSS must be understood as an emergent process driven by both the programmer and the code: “Every hacker is at one and the same time producer and product of the hack, and emerges as a singularity that is the memory of the hack as process” (Wark, 2004a, sec. 158).

Tim Jordan claims that the “malleability” of software has dissolved the question of determinism “not in the sense that the terms . . . – society, technology, determination – are meaningless, but in the sense that contradictory positions can be simultaneously maintained and maintaining such contradictions proves highly dynamic and effective” (Jordan, 2008, p. 140). In other words, we can talk about the ways that
programming languages and other technologies shape programmers, as through situated learning, and about the ways that trends of abstraction in programming languages are paralleled by similar trends in development methodologies, while still distancing those points from deterministic claims about code wanting to be free.

It is useful, perhaps, to return here to Bogost’s concept of procedural genres, broadened to include not only types of applications and interfaces as end products, but also the various factors of their production. In this sense, FOSS, Agile, the Bazaar, object-orientation, and APIs can be understood as overlapping procedural forms in an emergent procedural genre, a genre that embraces iteration and abstraction rather than resisting them. In order to be meaningful, this procedural genre need not culminate in the combination of each respective procedural form: Agile teams writing FOSS applications with promiscuously open APIs in object-oriented languages while taking the Bazaar approach to participation; rather, this procedural genre is meaningful simply because it offers a new lens through which to think about how and why FOSS applications are developed.

This chapter has explored, primarily, two aspects of programming: the role of iteration and experimentation in both learning a language and writing applications, and the impact of abstraction and modularity in code upon both code relations and social relations, including development methodologies. While programmers try to avoid writing bugs, they also expect to write bugs, because programming is understood as an
iterative process of trial and error, of debugging, of situated learning. Abstraction, and the relations it enables, can be seen as enhancements to that iterative process, and the removal of abstractions as a disruption of that iteration. Proprietary software, the Cathedral and the Waterfall, by definition, place rigid restrictions on when and where iteration can occur and, in that sense, they are disrupting the iterative nature of programming. Their counterparts, on the other hand - FOSS, the Bazaar, and Agile development – enhance the iterative quality. Chapter 4 will continue to explore these ideas, with the focus shifted from production to distribution and revision.
Chapter 4. Code Readability and Distribution

In Chapter 3, Wark's notion that abstraction enables new possible relations was explored in terms of the increasing abstraction of computer languages, and the effect of this abstraction on both development methodologies and the ideology embodied by developers. Implicit in that discussion was the fact that all source code is an abstraction of machine code. Assembly languages were the first abstraction from machine code (the earliest programmers actually wrote in machine code itself), but assembly was merely a remapping of machine instructions into a symbolic notation that was easier to read and write. Later language types, including those discussed above, extended this abstraction, allowing increasingly complex and distributed development models. In Chapter 3, this increasing abstraction was seen from a largely historical perspective, following the emergence of various new languages and development methodologies. Below, it will be explored again, but pursuing the directional flow from development to distribution and use – as actual software applications emerge from the iteration and are delivered to users.

Using interpreted code as a proxy for a FOSS distribution and compiled code as a proxy for proprietary software, this chapter will demonstrate how the availability or enclosure of source code determines the plane of relations between users and developers. The code examples used will result in identical output to the user, but will take dramatically different routes to reach that end, depending on the language, and
these routes have important consequences programmers and users. Source code is, of course, an abstraction of machine code, and the persistence of that abstraction through various stages of distribution and execution has profound implications for both code and human relations. This chapter argues that compiled code is enclosed the moment it is compiled while interpreted code is non-excludable by nature, so the former embodies the politics of public goods and the latter the politics of the commons, which suggests that interpreted code may be better suited for FOSS development. A reasonable question, motivated by this argument, is whether FOSS applications written in interpreted languages are quantifiably more successful then those written in compiled languages, and the later part of this chapter will explore one approach to answering this question.

**Hello World, Revisited**

The figures in this section show how the source code of a Hello World application, in each of several languages, progresses from the moment of its original authorship to its consumption by the end user. The code in these examples is slightly more complex than the Hello World program used in Chapter 3, as the “Hello World” string has been abstracted into a procedure, which is in turn abstracted into a third party library. This architecture would not be used in practice, because of the simplicity of the application, but is helpful here because it allows for a very simple construction of a more complex end-to-end process: a third party library (containing the
get_hello procedure), which is used by a programmer to write a Hello World application (including a call to the library's get_hello procedure), which is then distributed and executed by an end user.

A library, to clarify, is a package of code that does not constitute an application in itself, but which provides procedures (or, in object-oriented programming, classes and methods) that can be used in many different applications. For instance, many applications use third party libraries for common math operations, to build user-interface widgets, or access various types of databases. Applications typically interface with libraries through APIs – Application Programming Interfaces, which were discussed briefly in Chapter 3.

In a sense, libraries and APIs constitute another form of modularity, in addition to the structural modularity of procedural and object-oriented programming, which separates general purpose, reusable code from application-specific code. A well-abstracted API should, in theory, hide all of the details of its underlying implementation, to the degree that one implementation could be replaced with another without consequence, but in practice APIs often do reveal aspects of their implementation. These so-called “leaky abstractions” (Spolsky, 2002) – and their effect on human relations – will be discussed further in the Chapter 5, in the context of the FOSS community at Drupal.org.
The following examples focus on three languages: C++, a compiled language; PHP, an interpreted language typically executed on the server side; and Javascript, an interpreted language typically executed on the client side, in a web browser.

| C++     | string get_hello() { 
|         |   return "hello, world\n";
|         | }
| PHP     | function get_hello() {
|         |   return "hello, world\n";
|         | }
| Javascript | function get_hello() {
|         |   return "hello, world\n";
|         | }

Figure 4.1: Source Code for a Hello World Library

At this point, as the programmer works on her code, the only differences are the relatively inconsequential syntactical variations between languages – she writes some code, and that is about all that matters, regardless of the language she is using. When the library is distributed, however, important differences between the languages emerge.

---

1 The figures in this chapter assume that the C++ application is being executed by a web server in response to a page request from a web browser. This is a plausible but atypical scenario, as C++ applications are usually executed locally, often as desktop applications, but the scenario is helpful here for the sake of comparison.

2 Whether a language is compiled or interpreted is determined by individual implementations of the language, not intrinsic properties of the language itself, so any language can be either compiled or interpreted. However, languages tend to be very consistent in their implementations, so calling them “compiled” or “interpreted” languages is an oversimplification, but not a misleading one: C++ could be interpreted, but is in practice always compiled, and the converse is true of PHP and Javascript.
C++  Compiled machine code, as a .DLL file (in Windows), with the function declaration remaining exposed as source code:

```cpp
string get_hello();
```

PHP  Original source code.

Javascript  Original source code.

| Figure 4.2: Form of a Distributed Hello World Library |

PHP and Javascript are both interpreted languages, rather than compiled languages, so they stay in source code form until very late in the process. The C++ library, on the other hand, is compiled to machine code prior to distribution. The declaration of the `get_hello` procedure is exposed in a “header” file, the API which allows users of the library to call the procedure, but the implementation – the procedure itself – is compiled and unreadable. If a developer using the library wished to modify the `get_hello` procedure, she would be unable to do so at this stage without asking the original library developer for a copy of the source code, making the change, and recompiling the code herself.

---

3 Even though C++ is a compiled language, C++ libraries are not always distributed in compiled form. FOSS-licensed C++ libraries, for instance, must be distributed with the source code. However, while these examples consider the worst-case for C++, it is also a common case, particularly for closed source libraries. C++ libraries are often compiled before distribution, whereas PHP and Javascript are never compiled for distribution. This is the distinction being explored, and is one which has profound implications for the relations between programmers using these languages.

This chapter further assumes that applications are written entirely in one language, but this is not always the case. It is very common for web applications to combine both PHP and Javascript, as well as HTML. C++ is more often used as a stand-alone language, but not always. This oversimplification, however, helps illuminate the distinctions between each language.
### Application Source Code

<table>
<thead>
<tr>
<th>Language</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td><code>cout get_hello();</code></td>
</tr>
<tr>
<td>PHP</td>
<td><code>print get_hello();</code></td>
</tr>
<tr>
<td>Javascript</td>
<td><code>document.write(get_hello());</code></td>
</tr>
</tbody>
</table>

**Figure 4.3: Application Source Code**

### Application Distribution

<table>
<thead>
<tr>
<th>Language</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>Compiled machine code only – a .EXE file and the .DLL.</td>
</tr>
<tr>
<td>PHP</td>
<td>Original source code and library source code.</td>
</tr>
<tr>
<td>Javascript</td>
<td>Original source code and library source code.</td>
</tr>
</tbody>
</table>

**Figure 4.4: Application Distribution**

Figures 4.3 and 4.4 echo the first two figures, except that the application developers are now writing their own code and interfacing with the library. In Figure 4.3, the three languages converge again, as they began in Figure 4.1, as the library procedure is called from the application. They depart again in Figure 4.4, much as they did in Figure 4.2. While this redundancy may seem unnecessary here, it will be helpful later in this chapter.

### Output Received by User Agent (e.g. Web Browser)

<table>
<thead>
<tr>
<th>Language</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td><code>hello, world</code></td>
</tr>
<tr>
<td>PHP</td>
<td><code>hello, world</code></td>
</tr>
<tr>
<td>Javascript</td>
<td>Original source code and library source code.</td>
</tr>
</tbody>
</table>

**Figure 4.5: Output Received by User Agent (e.g. Web Browser)**

PHP is a server-side scripting language, whereas Javascript is a client-side scripting language, and Figure 4.5 shows how this attribute causes Javascript source
code to persist longer than PHP, remaining in source code form all the way to the browser. Most readers will be familiar with the “View Source” option in most modern web browser, but may not be aware that this option allows the user to see not only the HTML source code, but also the source code for inline Javascript blocks and the URLs of external Javascript libraries.

Finally, in Figure 4.6, we see that all three applications have converged by displaying the same output to the user.

<table>
<thead>
<tr>
<th></th>
<th>hello, world</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++</td>
<td>hello, world</td>
</tr>
<tr>
<td>PHP</td>
<td>hello, world</td>
</tr>
<tr>
<td>Javascript</td>
<td>hello, world</td>
</tr>
</tbody>
</table>

Figure 4.6: Output Displayed to User by User Agent

**Compilation, Enclosure, and Iteration Flows**

If they all result in the same output, what is the value of examining these languages so closely? One way to address this question is to return to the definition of the commons, specifically the non-excludable property. Copyright, of course, can be seen as a legal mechanism for enclosing a commons, converting commons goods into private goods by making them excludable under the law. While all software applications are equally dependent on copyright law to keep them out of the commons (ignoring DRM technologies, and other mechanisms for enclosure), all source code is *not*.

Source code written in compiled languages is enclosed the moment it is compiled,
so it does not need copyright laws to the same degree that interpreted languages need copyright laws. Interpreted code is non-excludable by nature, and requires copyright to provide the excludable property, whereas compiled code is excludable by nature, and requires a FOSS license in order to become non-excludable. In other words, compiled code embodies the politics of private goods, and interpreted code the politics of the commons. Both copyright and copyleft seek to dissolve this distinction, but do so in opposing directions. Copyright imposes excludability onto interpreted code, while copyleft imposes non-excludability onto compiled code.

At this point, it is important to step back and acknowledge the limitations of equating interpreted code with FOSS and compiled code with proprietary software. The most obvious problem is simply that such equation is wrong – there are proprietary PHP applications, on the one hand, in which the source code is distributed but all rights are reserved by copyright. On the other hand, of course, the most famous FOSS applications are all compiled. However, interpreted code vs. compiled code remains a useful proxy for FOSS vs. proprietary software that de-emphasizes both the politics and economics of licensing in favor of an emphasis on practice: iteration, abstraction, and the bug cycle.

The use of language types as proxies for license types also reminds us that the non-excludability of source code should be understood as a spectrum, not a simple binary, influenced by both the format of the code's distribution and the license attached
to it. However, more important than the subtlety of the spectrum are the practical implications for software development.

A software bug can be encountered at any of the stages of distribution discussed above, and there are always at least two parties interested in eliminating the bug: the end user and the original developer of the application or library. Likewise, there are always at least two pieces of information needed to diagnose and fix the bug – the code containing the bug, and the conditions that triggered it. The ease with which these pieces of information flow from programmer to user, and vice versa, is implicitly linked to the degree to which the source code is not excludable. Put differently, when the abstraction of source code is removed, a barrier appears in the plane of relations between the user and the programmer.

Figure 4.7 presents a model of code and human relations for FOSS libraries and applications written in interpreted language, i.e. the type of software which has the least excludable source code. The arrows in this diagram are necessarily ambiguous, an inclusive representation of the iterations of software development, the flow of distribution, and the many bits of information that can be passed around freely between users, programmers, and the code itself: bug reports, patches, tests, tips and questions.
Though Figure 4.7 distinguishes end users from programmers, for clarity, there is an equivalence and reciprocity in their relations that, in a sense, collapses this distinction. The two pieces of data needed to fix a bug – the buggy code, and the conditions that trigger it – are available to anyone who cares to seek it out.

Figure 4.8: A restricted plane of relations, as in a proprietary project written in a compiled language and distributed without source code. Dashed arrows indicate incomplete information.

Figure 4.8, on the other hand, models the scenario which creates the most excludable source code: a proprietary application and a proprietary third party library, written in a compiled language and distributed without the source code. In this model, code and other information flow only to the right and users and programmers are separated into silos, to such a degree that the programmers essentially vanish, reduced to mere technicians of their local development loop. The potential relations between programmers and users, as well as between their respective source code files, are cut off at the moment of enclosure, the moment of compilation.

In comparison to Figure 4.7, Figure 4.8 has an appealing precision and certainty, but this actually represents a severe limitation of this model: when a bug from an earlier phase of distribution is encountered, the two critical pieces of
information needed to fix the bug are separated by the enclosure of code, with neither a
code or human relation in place to bring them together. The programmer, of course, has
the code, but no knowledge of the conditions required to trigger it, while the user
knows something of the trigger conditions but can not access the code. Further,
particularly in the case of crashing bugs, the absence of the source code makes
identifying the trigger conditions more difficult, because simply knowing which line of
code was executing when the application crashed can yield fruitful clues about the
bug's origin.

These observations are, in a sense, quite straightforward, but they are important
to articulate because they enable an understanding of how enclosure of source code
impacts the bug cycle, for both users and developers. If source code is understood as
an abstraction of machine code, which creates a plane of possible relations, than
compilation of source code can be seen as a disruption of that abstraction, a shattering
of the plane of relations. While iterative development can still happen, within each
stage of distribution, the routes for iteration are tremendously constrained by this
disruption.

The bug cycle, and its implications for support costs, have been explored by
scholars and proponents of both proprietary and FOSS development. Perhaps the most
famous claim, from the FOSS side, is that “given enough eyeballs, all bugs are
shallow” (Raymond, 2001, p. 30) – this mantra is often used to support the claim that
FOSS software has fewer bugs than proprietary software. The opposing view, of course, is that all software has bugs, and only the proprietary model can guarantee support when a user encounters problems.

Eric Raymond, among others, has been highly critical of the relationship between proprietary software sales and commercial software support, because the sale price is meant to cover not only the cost of development, but ongoing support costs. Raymond, in a sense, suggests that proprietary software is an elaborate Ponzi scheme in which a company must continually seek out new investors (through sales) in order to pay off its old investors (through support). Like any Ponzi scheme, this model “is only viable in a market that is expanding quickly enough to cover the support . . . costs entailed in yesterday's sales with tomorrow's revenues” (Raymond, 2001, p. 121).

However, again, there are two sides to support costs: proponents of proprietary software frequently cite the concept of TCO (Total Cost of Ownership) in order to downplay the free price tag of FOSS (Deek & McHugh, 2007, p. 270) – the implication is that the end user will pay more for FOSS in the end, in support costs and lost productivity. Deek & McHugh, however, note that while “understanding these costs is a nontrivial matter and . . . acquisition costs are only a single, and possible small, part of the overall cost picture,” a number of comparative TCO studies have come out in favor of FOSS (Deek & McHugh, 2007, pp. 270-271).
The TCO question is still open to debate, as is the question of whether FOSS or proprietary software yields fewer bugs, but the commercial software industry has implicitly acknowledged the limitations of identifying and fixing bugs in a closed source, compiled application (as diagramed in Figure 4.8), through the many mechanisms it has developed to circumvent the moment of enclosure.

The most public of these mechanisms is the notoriously frustrating technical support systems – some free of charge, others paid per incident or by subscription – that software companies employ. Neal Stephenson claims that FOSS “does not have to maintain any pretensions as to its reliability. Consequently, it is much more reliable” (Stephenson, 1999, p. 108), correctly suggesting that proprietary software's technical support systems are more concerned with treating the symptoms of a bug – the customers' frustrations – than the bug itself. Not all support calls involve bugs – many calls are from customers who simply need assistance, and it is debatable whether FOSS communities or software companies are better at providing that assistance – but it is clear that the tech support model is not conducive to diagnosing and fixing bugs. Just as tech support tends to treat symptoms, Raymond points out that “non-source-aware users tend to report only surface symptoms . . . so they seldom include a reliable recipe for reproducing the bug” (Raymond, 2001, p. 33).

Part of the argument of this chapter is that source code is an abstraction of machine code which, as long as the source code is available, simplifies the complexity
of the code and enables new relations between users and developers. One possible misinterpretation of this argument would be that enclosing the source code, through compilation, simplifies complexity, hiding it behind the abstraction of the point-and-click interface of the end product. This is an understandable misinterpretation, given how challenging reading source code can be for many people, but the key is to remember that machine code does not hide complexity, it just further obfuscates it. The result of enclosing the source code and only allowing the user to access the machine code, in Raymond's terms, “is a mismatch between the [user's] and the developer's mental models of the program” (2001, p. 33).

This mismatch, when combined with the information flows illustrated in Figure 4.8, explains why proprietary software companies' support systems tend to have two frustrating bureaucratic traits: a tiered intake hierarchy and separate silos for support and engineering. Customer support calls are triaged not by an engineer, or even a highly informed support technician, but by an entry level customer service rep, possibly working from a script. If, as shown in Figure 4.8, the enclosure of source code disables the potential relation between user and engineer, and if customer support is interested primarily in treating the symptoms rather than the disease, then why would a user ever be allowed to talk to an engineer? FOSS communities, on the other hand, allow largely unrestricted relations between users and engineers. There is no formal obligation for a particular person to address a particular support request (i.e., to treat
the symptoms), so FOSS support inquiries are viewed by the community not as
distractions from valuable development time, but as important data points in
identifying the repairing the bug (i.e., treating the disease).

Customer technical support is, of course, the most blunt of the many ways that
proprietary software tries to deal with this problem. Other mechanisms include
deployment of multiple build types, hiring of test engineers, whitebox testing,
gathering of crash reports, and even the well-known practice of Beta testing. Each of
these mechanisms attempts, in its own way, to vault over the barriers to iteration
created by both compilers and copyright, or to reduce the size and significance of those
barriers; in other words, they attempt to make the arrows in Figure 4.8 point in both
directions. Discussing each of these mechanisms here would be a bit laborious, but one
example – build types – will adequately illustrate the theme.

Deployment of “multiple build types” refers to the common practice of
compiling code at a variety of different levels of optimization. When using Microsoft's
Visual Studio development tools, it is typical to start development using a “Debug”
build, and then graduate to a “Release” build for testing and distribution. In the Debug
build, the code is compiled to machine code, but is accompanied by a mountain of
debugging information which, among other things, allows the programmer to step
through the code, one line at a time, and inspect the contents of every variable.
However, this extra information carries performance costs (just as using an interpreted
language carries performance costs), so a Release build excludes much of that information in order to output fully optimized code (“Common Problems Switching from Debug to Release Build,” 2009). It's worth noting that these two build types are used by convention, but programmers can – and often do – configure a wide range of different build types, exposing and hiding different combinations of debugging information. In a sense, a Debug build allows the programmer to interface with the source code as if she were using an interpreted language.

However, while multiple build configurations create a workaround for some of the challenges of compiled code, they also draw attention to the mismatch Raymond described. A particularly illustrative example is the Heisenbug. “Heisenbugs” are bugs that disappear when you attempt to debug them (Bourne, 2004). The term is clearly inspired by the Heisenberg uncertainty principle, although the definition is actually more analogous to another idea from quantum physics: the observer effect. The implication of Heisenbugs is that the act of debugging influences execution; again, there is a mismatch between the programmer's model and the user's model – even when the programmer is the user, testing her own code.

Heisenbugs are common in use cases involving asynchronous processes and intense resource demands, so they often occur in video games. For instance, imagine a hypothetical bug where a console video game crashes frequently (but not always) when a fast moving bus enters the screen and explodes, but when a programmer uses a
debuggable version of the game and steps through the code, the game never crashes. In this example, looking for the crash prevents the crash from occurring at all.

The environmental conditions required for this particular bug to occur – i.e., loading from disc, at full speed, with fully optimized code – make it very difficult to debug. Diagnosing the bug would likely become a game of cat and mouse, in which the programmer attempts to corner the bug by writing new code in areas that might contain the bug, to help communicate – through the game interface – the details that the compiler has hidden from her. For instance, if she suspects that the bug is related to the loading of files from the DVD, she might write a few lines of code that display the name of each file before and after it is loaded, which would allow her to identify the problematic file. These lines of debugging code can be seen as a mechanism to try to circumvent the moment of enclosure that accompanies the compilation of code for the end user (as in Figure 4.8), to give the user (or, rather, the programmer-as-user) a window into the enclosed code.

Eventually, in this hypothetical example, the programmer might learn that the code was trying to render the burning bus graphics to the display before all of the needed graphics files were loaded from the game DVD – the crash was related to the hardware limits of the DVD drive – specifically, its read and transfer speeds – so the delays caused by debugging allowed the DVD drive to catch up, preventing the bug from occurring. While this is an extreme example, and perhaps an inevitable one, since
the high performance requirements of video games precludes the programmer from using an interpreted language, it nonetheless demonstrates the failure of one particular mechanism (the use of multiple build configurations), to overcome the problem of enclosure.

As mentioned above, there are a variety of other mechanisms used by proprietary software companies to overcome the restrictions on information flow resulting from the enclosure of source code. These include hiring of test engineers, whitebox testing and a variety of other testing strategies, and the gathering of automated crash reports (users of both Firefox and Windows, among many other applications, will be familiar with these reporting tools, which pop up when an application crashes). The very existence of SDETs (Software Design Engineers in Test), and the many forms of test engineering that the employ, testifies that proprietary software companies know that there is value in allowing some “users,” outside of the core programming team, to inspect the code. Simple end user testing is insufficient, both because of the difficulty in targeting code paths and, more importantly, the difficulty in communicating problems back to the development team without referring to the code – the same mismatch of “mental models” noted above, from Raymond.

**Abstraction of Another Sort: The Virtual Machine**

Compilers and interpreters are not the only options for converting source code into machine code – or, more accurately, they are not mutually exclusive options. Both
Java and most of the .NET Framework compile their code for “virtual machines.” A virtual machine is an abstraction of a computer, which allows code to be executed on any platform with an implementation of that virtual machine. So, for instance, Java source code (e.g., the file “HelloWorld.java”) is compiled into a binary file containing machine code (“HelloWorld.class”). However, unlike a traditional compiler, which outputs native machine code for a particular operating system or processor set, a Java compiler outputs Java “byte codes” - machine code instructions for a virtual machine. Any device that has an implementation of the Java Runtime Environment (JRE) – including Macs, PCs, and many cell phones – can translate these virtual machine instructions into machine code native to that particular device. The translation of bytecodes to native machine code can be performed by either an interpreter or a compiler, but “just-in-time” (JIT) compilation is the usual implementation. In JIT compiling, the bytecodes are compiled to machine code the first time they are executed (somewhat like interpretation), but the compiled machine code may thereafter be cached for rapid reuse (Suganuma et al., 2000).

Languages that compile for virtual machines are sometimes called “pseudo-compiled” languages (Crelier, 1999), a useful term here because it highlights the degree to which these languages occupy a somewhat ambiguous place between interpreted and compiled languages. Pseudo-compiled code is a different form of abstraction, which privileges code relations over human relations. Put differently, writing pseudo-compiled code for virtual machines allows a much higher degree of interoperability between multiple languages, as well as platform independence. It seems, in a sense, to be a compromise between the sustained abstraction of interpreted code and the optimized performance of compiled code, but in doing so it reduces the many social possibilities of abstraction to a simple technological outcome: machine virtualization. The human relations, and the corresponding political implications, of interpreted code are circumvented by the pseudo-compilation of the source code into bytecode. The source code is, as with compiled code, enclosed prior to distribution.

SourceForge Data Analysis: Limits of the Metaphor

Earlier, I claimed that “compiled code embodies the politics of private goods, and interpreted code the politics of the commons.” This is a bold claim, which hints at many possible corollary hypotheses, including the possibility that FOSS projects written in interpreted languages might be more successful or more numerous than those written in compiled languages. The comparison of interpreted and compiled code, however, has throughout this chapter been framed as a proxy for copyleft and
copyright, more of a metaphor than a rigid *equation*, and I have more generally argued that “the politics of the commons” are, through iteration and abstraction, embodied in *all* programming and source code. So, while the hypothesis that interpreted code will yield greater FOSS success is worth testing and, if confirmed, would certainly lend additional weight to my argument, that hypothesis should not be seen as the be all and end all of this chapter.

The hypothesis can be tested, however, by surveying SourceForge.net, an online repository for FOSS projects, currently serving more than 171,000 projects (“Software Map,” 2009). The SourceForge Research Data Archive (SRDA), hosted by the University of Notre Dame, allows academic researchers to run queries on anonymized dumps of the SourceForge database, dating back to 2003 (Madey, 2009). At present, this is likely the largest, most comprehensive source of historical data on FOSS projects, but there are a number of challenges presented by this data, two of which are particularly important here.

First, the SRDA data does not clearly distinguish between projects that use SourceForge as their primary repository and projects that merely mirror a copy of their code on SourceForge, to help it reach a wider audience. For instance, the Drupal project – the subject of Chapter 5 – is developed at Drupal.org, but can also be downloaded from SourceForge. Consequently, while Drupal has a very vibrant community of thousands of contributors, queries of the SRDA will classify Drupal as a
project with very low levels of participation. Without extensive, case-by-case analysis of every project, any study of participation levels on SourceForge is tainted by this uncertainty.

Second, SourceForge relies on developers themselves to correctly classify their projects – a problematic process on a site where many projects are competing for attention. For instance, looking at the Operating System classification, a very large number of projects written in Java are filed under “OS Independent (Written in an interpreted language).” Java, of course, is a pseudo-compiled language that runs on a virtual machine. A more accurate categorization of Java projects, using the existing options on SourceForge, would probably be “OS Portable (Source code to work with many OS platforms),” although one could certainly argue that SourceForge needs a new category for pseudo-compiled languages. Regardless, some of the most promising categorizations of data in the repository are subject to spam and human error, and are therefore unreliable.

![Table showing growth rates for different languages](image)

Figure 4.9 Total Projects and Growth Rates per language. The 10 fastest growth rates are indicated in bold.
Figure 4.9 shows the number of projects hosted on SourceForge which contain source code written in each of 21 programming languages, as of December 2008. This study was limited to the top 25 languages, and four languages were removed from the chart (XSLT and assembly because they are outside of the three categories; PL/SQL and Actionscript because they are bound to proprietary environments – Oracle and Flash, respectively). While these 25 languages represent only a third of the languages on SourceForge, those languages account for 97% of the total language designations in the entire repository. In this analysis, projects which include code from multiple languages will be counted once for each language. Also, it should be noted that runtime compiled or server compiled languages, such as JSP (JavaServer Pages) are classified as “interpreted” because, while they use compiler technology, they are distributed in source code form as if they were interpreted, and consequently occupy the same plane of relations as interpreted languages, as described earlier in this chapter.

If we inspect the two year growth rates for each language, measured from November 2006 to November 2009 and shown above in
Figure 4.9, we see that the fastest growing languages are either interpreted pseudo-compiled, and by comparison the compiled languages are growing quite slowly. All three types have grown dramatically, but Figure 4.10, to the right, shows how the distribution of compiled, interpreted, and pseudo-compiled languages (aggregated from the top-tier of languages discussed above) on SourceForge has evolved over the past six years. As a percentage of the total projects, compiled languages have dropped significantly, interpreted languages have increased modestly, and pseudo-compiled languages have increased significantly.

As a simple test of the hypothesis that interpreted code will yield more numerous FOSS projects this data is compelling but inconclusive, because it is unclear which group, compiled or interpreted, is being adversely affected by the rise of pseudo-compiled languages. It is very likely, for instance, that Java, the most common language and one of the fastest growing, is drawing more potential projects away from C and C++ than from PHP. This data would be much easier to interpret if it could be compared to corresponding trends in proprietary software development. Put differently, it is difficult to determine whether growth rates for a particular language reflect a trend in FOSS development in particular, or in software development as a whole – for instance, is ASP.NET exploding on SourceForge because it is well-suited to FOSS development, or because it is a young language with rapid growth in popularity across both FOSS and proprietary development?
Figure 4.11 compares the respective age of each language to its percentage growth (measured in number of projects on SourceForge) to the age of the language (Pigott, 2006) (Ashton, 2001) (Cable, Pelegri-Llopart, & Ahmed, 1999) (Greene, 2007) (“PHP: History of PHP - Manual,” 2009) (Madey, 2009). While the correlation is not overwhelmingly strong, there is nonetheless a correlation: young languages are growing faster on SourceForge than older languages. Again, though, the implications of the correlation are unclear. Are younger languages thriving because they are better suited to FOSS or just because they are the latest, hottest technologies? Is the number of projects a biased metric, because programmers using older languages are more likely to contribute to existing projects than to create new projects? Are the pseudo-code languages, all relatively young, cutting into the growth of either of the other groups?

There are also important technological factors here, which may overwhelm other trends. As a general rule, compiled languages have been used to build operating systems and native applications for both desktops and computers, whereas interpreted languages are more common for web applications. There are fairly obvious reasons for
this breakdown. Compiled code is fast and efficient – it is already converted to native machine code, after all – so it is well suited to native execution, where processor speeds and memory sizes are the primary performance bottlenecks. Interpreted code, on the other hand, is less efficient but easier to work with and collaborate on in diverse environments, so it works well on the web, where performance is often dictated by connection speeds rather than execution speeds. However, as both computing power and broadband connections improve, these distinctions may be dissolving, and pseudo-compiled languages – which are popular for both web applications and native applications – might seem to some to be, FOSS politics aside, the best of both worlds.

These technological factors, however, are a reminder that – as stated at the start of this section – this quantitative analysis is not the be all and end all of this chapter. Interpreted and compiled code should be seen not as causal agents driving FOSS and proprietary software (though Figure 4.10 gave modest support to that idea), but rather as a proxy for copyleft and copyright, respectively. Put differently, and perhaps more accurately, interpreted languages and compiled languages are entries in the same procedural genres as copyleft and copyright, respectively. While the bulk of this chapter compared the planes of relations associated with each type of language, this should be understood as a comparison of broader procedural genres. The relations that emerge from interpreted language can, through the analysis in this chapter, inform an
understanding of the relations that would emerge from other entries in its genre, including interpreted code, FOSS, Agile, the Bazaar, object-orientation, and APIs.
Chapter 5. Community Case Study: Drupal

In the two preceding chapters, I examined the material practices of programming – particularly the ideas of iteration and abstraction – and argued that the principles of FOSS development are extensions of these practices. The logical conclusion to this trajectory is to consider how iteration and abstraction play out in an actual FOSS community. There are any number of communities worth considering, but this chapter will focus on the community working on Drupal, an open source Content Management System (CMS) and web application framework.

So far, iteration and abstraction have appeared to be largely complementary, even mutually reinforcing. In the context of the Drupal community, however, iteration and abstraction begin to come into tension, as innovation collides with the needs and expectations of a large base of users and contributors, as complexity leaks through the abstraction, and as the once simple “does it run?” test becomes increasingly subjective and contested in a large community. The challenges raised by these tensions are neither insurmountable nor unique to Drupal, but are vulnerabilities in the Bazaar model that have not yet been conclusively resolved.

Why Drupal? / Why Not Linux?

Linux and the rest of the LAMP stack (Linux, Apache, MySQL, Perl/PHP) have been the focus of most literature on FOSS communities, including the works of
Weber, Raymond, Stephenson, Himanen and, to a lesser extent, Deek and McHugh. Linux has also long been the focal point of popular discussion of FOSS. In a sense, Linux is the default choice, yet Drupal will be the focus of this chapter. The reasons for this decision are threefold.

First, the “clash of the titans” narrative of Windows vs. Linux is a tired old story, and will remain so. While Linux has come to dominate the server market, it has made only modest gains on the client side, which is still dominated by Windows. However, it is also unlikely that Windows will ever “win” completely – Linux is here to stay – and the three way contest between Windows, Apple, and Linux figures to plod along for years to come. Limiting discussion of FOSS communities to the OS wars overlooks the fact that, to the end user, the choice of operating systems is of decreasing importance. Desktop applications are increasingly available for multiple platforms – including many written in pseudo-compiled languages for virtual machines - file formats are increasingly interoperable, and more and more of the typical user's time is spent in a web browser. In a sense, the operating system has become an abstraction, a common interface with interchangeable implementations. While Tux (the Linux penguin) remains an unofficial FOSS mascot, continuing to frame discussions of FOSS entirely around Linux unnecessarily limits the discourse.

Meanwhile, computing and the Internet have grown up, and this change of scenery is another important reason to study something other than Linux. Drupal and
the Drupal community are, in a sense, of a different era than Linux, influenced from birth by advances in programming methodologies – such as the rise of code modularity and agile development discussed in Chapter 3 – as well as a general shift towards Web-based technologies and promiscuously open APIs. Drupal and its contemporaries might be called second generation FOSS projects, compared to first generation projects like Linux.

Finally, the third reason for studying Drupal is much more pragmatic – while I have not been heavily involved in the Drupal community, I have built several websites using Drupal 5 and Drupal 6, and have peripherally followed the community's activities for more than two years, picking up a great deal of tacit knowledge along the way. Since a full ethnographic study is far beyond the scope of this chapter, this tacit knowledge of Drupal and the Drupal community will allow a greater depth of analysis than I could achieve by studying any other FOSS community, including Drupal competitors such as Joomla, Plone, and Wordpress. A hacker might say that, by focusing on Drupal, I'm being “lazy like a fox” (Raymond, 2001, p. 27), but I'll frame it as rigorous pragmatism.

**Drupal Basics**

Drupal, which is written in PHP, an interpreted language, began as a side project, much like Linux and many other FOSS projects. Dries Buytaert founded the
project in 1998 while an undergraduate at Ghent University in Belgium and, in 2001, released Drupal 1.0 to open source development (Buytaert et al., 2009). The latest version, Drupal 6, was released in 2008 and is known to be in use on at least 120,000 websites (“Usage statistics for Drupal,” 2009). In 2007, Buytaert started Acquia, “a company that is to Drupal what Ubuntu or RedHat are to Linux” (Buytaert, 2007).

Like other large FOSS communities, Drupal contributors are decentralized geographically, but coordinate many of their activities at www.Drupal.org, groups.Drupal.org, and other Drupal.org sub-domains, including often lengthy discussions in “the issue queues” - the list of bugs and feature requests on Drupal.org, which accept comments and proposed patches from anyone with a Drupal.org account. Other community interactions, however, take place on a number of mailing lists, Internet Relay Chat (IRC) rooms, personal and collaborative blogs, real life “meet-ups” and conferences, social networking websites, and the informal personal communications that characterize any community: email, phone calls, and social events of all sorts.

Drupal's online documentation includes the following statement of “Mission and values” (although – like all Drupal documentation – it is subject to continual wiki-style revision):
**Mission and values**

*To develop a leading edge open-source content management system that implements the latest thinking and best practices in community publishing, knowledge management, and software design.*

*We value:*
- Flexibility, simplicity, and utility in our product;
- Teamwork, innovation, and openness in our community;
- Modularity, extensibility and maintainability in our code.

("Mission and values,” 2008)

This statement of values emits traces of all of the key concepts explored in this paper: “flexibility, simplicity, and utility” goes hand in hand with the “Does it run?” objective; “modularity, extensibility and maintainability” sounds a lot like the iteration and abstraction – and the resulting code relations – that were the central focus of Chapters 3 and 4; and, of course, “teamwork, innovation, and openness” are the human relations that are enabled by those code relations. These are not just stated values, either; they occupy a prominent place in the Drupal community.

Drupal has two distinct areas of development: Core and Contrib. Core is the default Drupal installation, which contains the Drupal framework and a small set of modules allowing the most commonly used features. Core is carefully controlled, in the sense that only a few key programmers can commit code changes to the central CVS repository, major versions are released rarely and only after a lengthy “code freeze,” and the direction of development is guided by an unofficial inner circle of contributors. While anyone can file “issues” (bug reports, feature requests, etc), submit
patches for those issues for consideration by the committers, and participate in
discussions about development in a number of spaces, digital and otherwise, the code
itself is tightly gated (“Core developers,” 2009). However, contrary to what many
might expect, this has not prevented extensive participation – between Drupal 5 and
Drupal 6, Core patches were submitted by 741 different contributors (Internet Archive,
2009).

Once a user has downloaded Drupal Core and installed it on a webserver, she
will likely return to Drupal.org to download additional modules (extensions which
provide a huge array of features not included in Core) or themes (interchangeable skins
that change the appearance of a Drupal site). These modules and themes are
collectively known as Contrib, the second area of Drupal development which is, by
comparison to Drupal Core, the Wild Wild West. Contrib currently contains nearly
3800 modules (“Modules,” 2009) and 450 themes (“Themes,” 2009) and, though many
Drupal sites use custom themes and modules, Contrib contains only those that have
been contributed back to the community – i.e. added to a shared CVS repository and
distributed at Drupal.org. While contributors must apply for a CVS account in order to
add their first new project (a module or a theme) to the CVS server, and can only
commit changes to their own projects, Contrib is largely unregulated (“CVS FAQ,”
2009) and projects can remain very autonomous. In particular, the individual project
maintainers determine when and how bugs in their code get fixed and when new
versions are released, if at all. Each project has an “Issue Queue,” a list of bugs and feature requests particular to that project, and anybody can file issues and submit patches, but only the project maintainer decides whether to incorporate the patch by committing it to the CVS repository, or to address the issue at all. In a sense, each contributed module and theme is its own FOSS project, merely hosted on Drupal.org for convenience, and they are as varied in quality and culture as standalone FOSS projects.

A Plane of New Relations

On the whole, the Drupal community is notably conscious of its communal and collaborative nature, and puts considerable effort into developing and growing its community, particularly by fostering higher levels of participation from the long tail of users and new potential contributors.

Recently, for instance, Angela Byron (webchick) encouraged the use of a “novice” tag to identify bugs and features that might be good tasks for new contributors to try to solve. “By specifically identifying 'novice' issues, we give them a nice short-list of 'low-hanging fruit' that they can use to get their feet wet” (Byron, 2009). Drupal has also been very active and successful with two Google initiatives: Summer of Code, a FOSS mentorship program for college students, and the Google Highly Open Participation Contest, an analogous program for secondary and high school students (“Google Summer of Code 2009 Frequently Asked Questions,” 2009)
Chapter 2 briefly discussed the role of reputation in FOSS communities, primarily in the context of the gift economy. As a point of comparison, programmers in proprietary software development are typically only recognized within their company. If they are recognized externally, it is only by their job title, not the work they actually contributed. The quality of their work, or their many hours of overtime, vanish into a non-specific “credits” list tucked away in some obscure menu item or help page.

In FOSS communities, on the other hand, reputation has tremendous value. Drupalers refer to their community as a “do-ocracy” – as opposed to the more common FOSS “meritocracy” – meaning that those with a reputation for writing great code (or lots of it) are rewarded with greater social standing.\textit{webchick}, based on the successes described above, epitomizes the do-ocracy: she did a lot in a very short time, and was quickly rewarded. Another of Drupal's most devoted contributors is Károly Négyesi (\textit{chx}), and his efforts are recognized in many ways, including a tongue-in-cheek fan site called “chx can not be distracted,” which includes photos of \textit{chx} – some of them photoshopped – working on his laptop in unlikely places (“chx can
not be distracted,” 2009). As a final example, one of the more astounding Drupal contributors is dmitrig01, a programming wunderkind who at the age of 12 was a mentor for a 16 year old GHOP participant (Gordon, 2008), among many other contributions. His recognition is less explicit than the others I have mentioned, but his expertise is highly sought after and, more subjectively, he is treated in the community with a vague sense of awe, on account of his age. Each of these reputations is highly individuated, particularly in comparison to the complete anonymity of a proprietary developer – while their reputations originate in their contributions, they incorporate their personal quirks and narratives, to the degree that they begin to seem like celebrities.

Reputation in Drupal, however, is about more than simple recognition, in the celebrity sense, because it also impacts how a programmer is treated within the community: “the folks who go beyond talk and start contributing to the project tend to get better support, have their opinions taken more seriously, rocket up the Drupal learning curve faster, and attract more Drupal clients” (Byron, 2008b).

Wark's notion that abstraction creates a new plane of relations has been central to the ideas presented in this paper, but is the model of a meritocracy or do-ocracy compatible with those relations? The obvious answer, of course, is yes: all sorts of new relations occur in FOSS, and they are all very exciting. The ability for anyone to ask questions or submit a patch is unprecedented, in comparison to proprietary software.
However, these new relations also construct a much more complex social dynamic, which may be further from an egalitarian utopia than FOSS contributors would like to admit. With proprietary software, users are more or less equal to other users (with a bias in favor of high volume customers), and programmers are more or less equal to other programmers (with an exception for incremental raises and promotions). While this flattening is obviously problematic, it also greatly simplifies society's power relations. FOSS communities exhibit no such simplification. Söderberg describes, in terms of general society and higher education, “the liberal myth of a meritocracy where class inequalities are coded as differences in intelligence between individuals” (2007, p. 173). Is the FOSS meritocracy, or Drupal's do-ocracy, also a myth?

While individuals like webchick and dmitrig01 demonstrate the community's tremendous openness to upward mobility – a stark contrast to the class inequalities that concern Söderberg – Drupal is also a technocracy; upward mobility is largely dependent on technical programming skills that only a privileged few can attain, and there is limited space at the top. Is the meritocracy, in this sense, in tension with abstraction's plane of new relations? The next few sections will tease out some answers to this question, cloaked in a broader discussion of tensions in FOSS development.
The Drop is Always Moving: The Limits of Abstraction

In the midst of a heated email exchange, debating whether APIs in Drupal 7 ought to be backwards compatible with modules written for Drupal 6, contributor Earl Miles (merlinofchaos) wrote that “people who say how easy it would be to have a backward compatibility API are people who clearly don't actually understand the codebase” (Mainardi & Miles, 2009). According to other messages in the same thread, and in other threads, this is a recurring debate on the Drupal development mailing list.

merlinofchaos maintains Views, the most widely used contributed Drupal module, and as such has been at the center of a significant controversy in the Drupal community. Drupal 6.0 was released in early 2008, but the community was slow to adopt it, largely because “it took a long time for contributed modules to catch up to Drupal 6... some essential contributed modules only became available as late as November 2008” (Molnar, 2009). Views was among these delayed modules, as was CCK, the second most used contributed module, in both cases because their maintainers “decided to do a major architectural redesign” (Buytaert, 2008a). While the uproar is a bit misguided – the delay of Views and CCK was caused by unfortunate timing more than changes in the Drupal 6 API – discussions of a backwards compatible API occur in the shadow of this controversy.

The previous two chapters have suggested that both iteration and abstraction, perhaps working together, make an argument in favor of FOSS licensing and
communities, but here, particularly in the debate over backwards compatibility, we see a tension arising between these two forces. A pure abstraction ought to hide all of the complexity that underlies it; in practice, however, as programmer and blogger Joel Spolsky has argued, “all non-trivial abstractions, to some degree, are leaky. . . . One reason the law of leaky abstractions is problematic is that it means that abstractions do not really simplify our lives as much as they were meant to” (Spolsky, 2002). Put differently, some of the complexity leaks out, so changes below a layer of abstraction – which are inevitable, due to the iterative nature of programming – can force changes above the abstraction. A non-leaky abstraction would, by definition, respect backwards compatibility, but is in practical terms extraordinarily difficult to achieve and accompanied by heavy performance tradeoffs.

For Drupal, this sometimes means that module contributors and users need to know more about the underlying codebase than they would like. However, the tension between iteration and abstraction in the Drupal community is not merely due to the intrinsic iteration of code, or the intrinsic leakiness of abstraction, but also to an explicit philosophy within the community. “The drop is always moving” exclaims the Drupal Handbook (Drupal's online documentation), “Drupal development is always on the cutting edge and with each major release there will be radical improvements. . . . While the upgrade path will reliably preserve your data, there is no backward compatibility with the previous Drupal code” (“On backward compatibility: the drop is
always moving,” 2009). The Handbook links to Buytaert's blog, where he writes the following:

“Preserving backward compatibility often requires that you drag historical baggage along, and in interpreted languages like PHP, this comes at a significant performance cost . . . Over the years, we've seen a lot of innovations happen that would not likely have happened while preserving backward compatibility . . . there will always be a tension between the need for hassle-free upgrades and the desire to have fast, cruft-free code with clean and flexible APIs. At the end of the day, we can't make everybody happy . . . As Drupal grows, [users] become increasingly verbose about the issue of backward compatibility, and as a result, it becomes harder and harder to maintain our original core values” (Buytaert, 2006).

Put differently, while the non-excludable nature of interpreted PHP may have helped enable the complex relations of the Drupal community, Buytaert suggests that it now forces a choice between the directions development can proceed. While “performance” here refers to technical speed and size, it is echoed by a broader sense of performance – backwards compatibility threatens innovation itself; the leaky abstraction threatens iteration.

Earlier I defined Drupal as a CMS, but Drupalers often describe it as a Web Application Framework (Bahey, 2007) (Farina, 2009). Which categorization is correct is a highly technical question and beyond the scope of this chapter, not to mention a sensitive debate to choose side in, but what is clear is that the contestation over categorization echoes the underlying debate over approaches to development. In general terms, those who view Drupal as a Framework focus on Drupal's power to build innovative web applications, particularly in the hands of highly skilled
developers, and would rather stay on the bleeding edge of technology than slow down to focus on backwards compatibility and usability. Those who view Drupal as a CMS would prefer the opposite.

Is this tension between iteration and abstraction just another way of thinking about Agile vs. Waterfall, or the Cathedral and the Bazaar? Is the idea of backwards-compatibility, of abstractions that remain non-leaky from one version of Drupal to the next, merely a request that the Bazaar mature into a Cathedral? Alternatively, is the tension merely the legacy of expectations established during the reign of proprietary software? These are phenomenally difficult questions to answer, in part because the debate tends to be quite dogmatic and ends with resignation instead of resolution. The debate above, for instance, which included merlinofchaos and paolomainardi, ended quite abruptly with these words from killes: “Go away and come back when you've got code to show” (Killesreiter, 2009). Whether it is dubbed a do-ocracy or a technocracy, code rules in the Drupal community, and many debates go unresolved.

In addition to the tension between iteration and abstraction, software development projects (both FOSS and proprietary) face a tension between two different sorts of iteration – iteration for maintenance and iteration for innovation – and Drupal has been notably more successful at dealing with this tension. When Drupal 6 was released in early 2008, it was numbered as version 6.0. The first digit, the 6, indicated that this was a “major” release, a dramatic upgrade over Drupal 5, and the
second digit, the 0, indicates the minor version (or patch level): since Drupal 6.0 was released there have been 10 “point” releases, so the latest version is version 6.10. While point releases are generally driven by security fixes and bug fixes, they rarely include API changes or large new features.

Meanwhile, work proceeds on Drupal 7. While the iterations that create point releases focus on maintenance, and are performed with great care, the innovative iterations that take place between major releases are greatly accelerated. This strategy of splitting iteration into two separate spaces – one stable and one innovative – is analogous to that used by the Linux community (Deek & McHugh, 2007, p. 96), although the numbering scheme is different. The latest Drupal code (known as “HEAD”) can be downloaded from CVS by anybody at any time, and is bundled nightly for downloading as “Drupal 7.x-dev” from Drupal.org. The principle of “release early, release often” is taken to the extreme, in the sense that HEAD is a permanent, immediate release, available at all times for anyone who wants to test or work on it, although only the most involved contributors will actually do so. The majority of users will stick with the stable 6.10 release – for them, Core is stable (except at the moment of a major release like 6.0), and Contrib is where rapid changes occur.

Earlier I wondered whether asking for backwards compatibility in Drupal was analogous to asking for the Bazaar to mature into the Cathedral, but the release process
described above suggests that, while it sticks to the Bazaar approach in most respects, in terms of the release cycle Drupal describes itself as both a Bazaar and a Cathedral.

“Your FOSS is Showing” – Contested Measures of “Does it Run?”

Throughout this paper, I have repeatedly invoked Tim Jordan's notion that hackers evaluate software using the “does it run?” criteria, but I have not interrogated what it means to ask whether software runs, or who can ask the question. As have others, Jordan focuses on the asking that occurs among hackers, paying little attention to non-technical end users. The limitation of this view, however, is the implication that “running” is an objective measure. Jordan himself suggests that the differences between FOSS licenses and Creative Commons licenses are rooted in the fact that creative works are judged more subjectively than code (Jordan, 2008, p. 105).

What we see repeatedly in the Drupal community, however, is ongoing contestation over the subjective meaning of “running.” At the DrupalCon Boston conference, in 2008, Dries Buytaert presented results from a University of Minnesota usability study of Drupal 6, including film of eye-tracking tests. The study captured rather embarrassing results, revealing users eyes darting around the screen for several minutes, clearly confused by routine actions on a standard Drupal installation. While the video rolled, the crowd of developers laughed – some nervously, others genuinely. “It gets less funny after a while,” Buytaert confessed (Internet Archive, 2008) (“Report
From Formal Drupal Usability Testing at the University of Minnesota Libraries,” 2008).

To a Drupal developer, Drupal passes the “does it run?” test with flying colors, because of its tremendous power, rich feature set, and flexibility. To an end user with fewer technology skills, however, Drupal does not pass the test as easily – it is often confusing and hard to use, and the line between “does it run?” and “can I use it?” (or “does it run, for me, right now?”) is quite blurry.

Usability has been a recurring problem not only for Drupal, but for FOSS in general. Weber suggests one cause: user interfaces depend on “a great deal of tacit information” which is “difficult to modularize and difficult to develop in parallel distributed settings” (2005, p. 238) – again, subjectivity is problematic. Despite the wild success of Linux on the server side, it has made only modest inroads on the client side, again because of conflicting answers to “does it run?” To a tech savvy system administrator, Linux runs – and it runs better than the proprietary alternatives. To many users, though, Linux does not run, simply because they have no idea how to make it run. Neal Stephenson devotes an entire chapter to the story of his first Linux installation (Stephenson, 1999, pp. 91-103), and seems to view the laborious process as a rite of passage. Much like programming itself, configuring Linux is an iterative process – tinkering until you get it right – and while this process is par for the course for hackers, it is a daunting undertaking the layperson. While it is a bit unfair to
compare operating systems based on the installation process, since personal computers typically arrive with an operating system pre-installed, many users may be reluctant to try Linux, and other FOSS projects, because of the “by hackers, for hackers” mystique. Linux has seen dramatic improvement in usability in recent years, and Ubuntu Linux can now be installed on a PC through a simple interface similar to a typical application installation (Nestor, 2007), but the problem persists: both Linux and Drupal have the reputation of sophisticated code burdened by a primitive interface. Firefox, on the other hand, has tremendous usability and is wildly successful. Söderberg argues that usability must be foregrounded in FOSS development: “since the least skilled users are also by far the most numerous, it is their decisions that add up to setting the standard . . . hackers too will be hurt if the majority of users side with proprietary technology” (Söderberg, 2007, p. 58). In other words, making FOSS accessible to the mainstream may be the difference between an uphill battle and a downhill battle against proprietary competitors.

It is tempting to dismiss these sorts of problems as a culture clash between geeks and laymen, or as a failure of the former to accommodate the latter, but it might be better understood as a problem of abstraction. A user-interface, much like an API, is an abstraction of the underlying complexity, both of the source code itself and of the development methodology that created it. For the layman user, the bar for a successful interface is set very high; the user does not particularly care how the source code
implements an idea (or how the programmers who wrote the code organized themselves) so long as it behaves in the expected way, based on his use of the interface.

Can a poor interface, then, be considered a leaky abstraction? If the emphasis is on the abstraction of code, it is a tricky question, at least for Drupal. Does Drupal face usability challenges because of underlying implementation details? Put differently, does the implementation of Drupal leak out into its interface, forcing users to understand the implementation in order to use Drupal? In some cases, the answer is certainly yes, particularly when vocabulary used in the code, such as “taxonomy,” confuses users by leaking into the interface: “Taxonomy. That's a funny word” (“Participant 8,” 2009) (“Label taxonomy is abstract,” 2009). On the whole, however, many of Drupal's usability issues are simply due to neglect: a menu is placed in a confusing location not because of the underlying architecture, but because it has to go somewhere, and nobody has bothered to move it to a better spot.

If, on the other hand, the interface is viewed as an abstraction meant to hide the development methodology, then it becomes very clear that the FOSS development process is leaking through the abstraction – its FOSS-ness is showing. “All it takes is a knack for usability, some development skills and a lot of time and effort,” writes Buytaert, but in a competitive do-ocracy, driven primarily by engineering prowess, some development skills do not buy a lot of leverage. There are exceptions, of course,
but, as discussed earlier, the Drupal community operates largely as a technocracy –
status is determined first and foremost by programming skill and productivity. While
other perspectives and skill sets are said to be welcomed, they tend to remain marginal
with respect to decision making. As a rhetorical point of comparison, would a well-
funded commercial software studio ever hand over its interface to a developer with
merely a knack for usability? Would a proprietary developer expect a usability expert
to choose between editing code himself or posting his ideas in an Issue Queue, a mere
suggestion box, and hope that someone else volunteered to change the code?

Skepticism aside, the Drupal community has made a serious commitment to
usability and user experience for Drupal 7, which is slated for release in late 2009 or
early 2010. The usability study mentioned above was part of that effort, as were
additional studies, the formation of a usability group on groups.Drupal.org, and several
Usability Sprints (intense face-to-face work sessions). Perhaps most importantly, Dries
Buytaert's startup company, Acquia, has hired a top design team, Mark Boulton
Design, to help with usability and user experience (Buytaert, 2009a). It is debatable
whether resorting to a commercial design team should be considered a failure of the
FOSS model to solve the problem internally, or an innovative hybridization that
demonstrates the dynamism of the FOSS approach. However, the hiring of Mark
Boulton Design does clearly indicate that the FOSS model continues to struggle with
usability and user experience.
Not surprisingly, Drupal's usability campaign, and the particulars of its goals, are contested within the community – as are most decisions. When Boulton's team launched a new website, D7UX.org (Drupal 7 User Experience Project), in March, 2009, with the slogan "When we’re done we’ll always choose Drupal over Wordpress," reactions were mixed, as these three Twitter posts show:

- “I just *love* the subtitle on d7ux.org, it is just SO smart. #drupal” (McKenna, 2009)
- “Wrong UI/UX goal for Drupal: 'When we’re done we’ll always choose Drupal over Wordpress' . . . #smallercore” (Gundersen, 2009).
- “You have to eat your own dogfood, folks!! . . . #Drupal UX redesign project site running #Wordpress” (Robertson, 2009).

In the second quote, from developer Eric Gundersen, “#smallercore” is a reference to the CMS vs. Framework debate discussed above; Gundersen clearly prefers that Drupal Core remain small, focused on providing a versatile and powerful Framework rather than a feature-rich Content Management System. Gundersen is not against usability per se, but rather the implication that Drupal ought to engage in a usability contest with Wordpres, a blog-oriented CMS. However, in aggregate, many Drupal developers express a sense of suspicion towards the usability campaign, seeming to view it as a direct challenge to Drupal's flexibility and power.

It remains to be seen how successfully Drupal will cope with these tensions. Will Drupal 7 renew the tension between iteration and abstraction, due to API changes and delays in Contrib? Will D7UX and the Drupal community's usability efforts
successfully tame Drupal for the layperson, with an interface that successfully hides the underlying complexity? If so, will that abstraction come at the expense of iteration, bloating Core and slowing innovation, or will this tension be resolved? Can it be resolved? This unanswered question is among the most important questions about FOSS development, and Drupal is poised to begin to answer it over the next few years.
Chapter 6. Conclusion

Condensed to the bare minimum, I have made, in this thesis, four closely related arguments. The first argument, implicit in the formulation of this project, is that existing studies of FOSS are inadequate, or at least incomplete, because they have not thoroughly considered programming languages or programming itself. Second, in Chapter 3, I argued that the concepts of iteration and abstraction are foundational to the study of programming, and can help explain a number of trends in software development, of which FOSS is but one. In Chapter 4, using interpreted code and compiled code as proxies for FOSS and proprietary software, respectively, I demonstrated how the enclosure of source code disrupts iteration and abstraction, implying a sense of incompatibility between the proprietary model and the practices of programming, mirrored, of course, by a sense of compatibility with FOSS. Finally, in Chapter 5, I showed how iteration and abstraction explain aspects of both the successes and limitations of FOSS communities, revealing possible tensions between iteration and abstraction, particularly over questions of usability and design.

It is quite possible that an experienced programmer, reading this thesis, could perceive the bulk of these arguments as a statement of the obvious, but this is because that programmer, through the very same situated learning processes described in Chapter 3, has become intimately familiar with the arguments of this thesis without ever articulating them, or hearing them articulated, in these formal verbal terms. This
project, then, is in many ways a project of articulation, a translation of tacit knowledge into critical discourse – an abstraction, as it were, that will allow non-programming academics to understand these processes without enduring the complexity of programming itself.

Chapter 4, of course, particularly the quantitative study of projects on SourceForge, hinted at many possible directions for related research. Most valuable would be the development of meaningful criteria for measuring the success of a FOSS project, in terms of any of the many metrics available from the SourceForge data (i.e., downloads, contributors, support requests, patches). Such criteria, however, are inherently difficult to arrive at, because each of those metrics is a potentially problematic measure – for instance, the number of patches alone is not a reliable indicator, because it can reflect either a strong development community or a sloppy development process. My language-centric queries, which merely tallied the number of projects using each language, revealed nothing about the success of individual projects and only the most simplistic measure of a language's success.

Unfortunately, the only agreed upon measure of FOSS success is Jordan's “does it run?” test. Perhaps it makes more sense, then, to proceed by refining not the measures of success but the definition of success. Is success a large number of contributors? Reliable code and a wide user base? Rapid innovation, or solving a new engineering problem? The demise of a proprietary competitor? The creation of related
jobs? Or, taking a different tack, perhaps there should be a classification for different types of success, dependent upon the nature of the project. For instance, Linux and Drupal are often considered successful on the basis of their large, dynamic development communities, but does this imply that FOSS projects without communities (i.e., the majority of FOSS projects) are not successful? Linux and Drupal – an operating system and a content management system – are well-suited to a large collaborative effort, but other types of projects, particularly small, independent extensions of large projects (such as small Drupal modules in Contrib), may not demand the support of a full community, so they should probably not be measured by the same standard.

In proprietary software, sales and profits are a blunt but effective measure of success, but such measurement is considerably more complicated for FOSS. Further research in this area would be immensely helpful, both to researchers and practitioners. For instance, Chapter 4 suggested that there would be value in a side by side comparison of language popularity in FOSS and proprietary software. While quantitative data from proprietary software development would, undoubtably, be difficult to come by, particularly data as voluminous as that provided by SourceForge, the ability to control for trends in proprietary software would allow significantly deeper insight into the meaning of the trends observed within FOSS development.
Finally, research investigating the relationship between the use of FOSS licenses and the practice of Agile methods would be profoundly valuable and, regardless of findings, would enrich and complement this work.

This thesis, however, has not only paved the way for additional academic research; it also has practical implications for users and programmers, and their employers, particularly in the way that the concepts of iteration and abstraction have allowed for the formation of links between FOSS and other trends in software development, the procedural genre that includes FOSS, Agile, the Bazaar, OOP, APIs, and interpreted code. A conservative or institutionally constrained FOSS project, for instance, practicing Cathedral-style development, might find in these pages the needed justification to adopt Agile methods for internal development, even if the inclusive Bazaar model is out of contention. Similarly, once exposed to the relationship between iteration, abstraction, and the success of FOSS, a team of Agile developers which uses a number of FOSS applications might realign their iterative development methodology to strengthen their relationship with the FOSS communities from which they draw.

Thinking about Chapter 4, while it would be foolish to suggest that project managers ought to choose an interpreted language over a compiled language based solely on my arguments, this work might nonetheless inform them about the challenges, or opportunities, that will accompany their choices. Finally, of course,
Chapter 5 demonstrates some of the limitations of FOSS development, serving as both a warning and a guide for developers and users alike.

**Beyond the FOSS Movement**

The discourse of FOSS, both popular and academic (including this text), has often been framed as an underdog story, but that story may have run its course. If the buzz on the tech blogosphere is any indication, FOSS is getting hotter and hotter in the software industry, and finding inroads into spaces previously reserved for proprietary software – even Microsoft is beginning to take FOSS seriously, not only as a nuisance and competitor, but as an option for some of their own tools (Goodin, 2009). In the words of Sun CEO Jonathan Schwartz, "Open source is now a given . . . what's interesting is what comes next" (Asay, 2009).

The introduction to this thesis began with the observation that “FOSS is a hot topic among scholars,” but at this concluding juncture it seems worthwhile to broaden that statement, at least briefly, to include participatory culture as a whole, in order to consider how this study of FOSS might inform studies of other contemporary phenomena. During the nine months in which this work was written, at least four important, peripherally related books landed in stores: James Boyle's “The Public Domain: Enclosing the Commons of the Mind” (September, 2008), Lawrence Lessig's “Remix: Making Art and Commerce Thrive in the Hybrid Economy” (October, 2008),
David Bollier's “Viral Spiral: How the Commoners Built a Digital Republic of Their Own” (January, 2009), and a book to which this thesis owes a tremendous debt: Tim Jordan's “Hacking: Digital Media and Technological Determinism” (November, 2008). While each of these books links FOSS development to broader contemporary themes, Bollier summarizes concisely and eloquently in the two passages that follow.

“Free software may have started as mere software, but it has become an existence proof that individual and collective goals, and the marketplace and the commons, are not such distinct arenas. They are tightly intertwined, but in ways we do not fully understand” (Bollier, 2009, p. 37).

“Open source' has become a universal signifier for any activity that is participatory, collaborative, democratic, and accountable. Innovators within film-making, politics, education, biological research, and drug development, among other fields, have embraced the term to describe their own attempts to transform hidebound, hierarchical systems into open, accessible, and distributed meritocracies” (Bollier, 2009, p. 39).

While these broader themes are tremendously important, and will undoubtedly continue to be the subject of much more research and publication in the coming years, this paper set out to cover FOSS not in broad strokes, but in narrow detail, focusing on aspects of programming that have not previously been brought to light in the context of this discourse: the iterative nature of code and programming, and the trend towards abstraction in software development. At the completion of this project, however, a new set of questions arise: what can this research tell us about broader themes in participatory production, the commons, and the networked society? Can iteration and
abstraction teach us anything about Wikipedia, or open access publishing, or transparency in government?

At a surface level, the answer to this last question is an obvious “no.” In this thesis I focused on the material practices of programming; as such, most of the discussion concerns those aspects of FOSS which are unique to software, rather than the characteristics it shares with many of these related ideas. There is some space to maneuver, of course – Wikipedia and iteration are nearly synonymous, for instance, and Wark's plane of relations fits nicely with any trend that includes “open” in its name – but, on the whole, any sweeping claims about the applications of this research to other fields would be self-indulgent and disingenuous, because it has remained so narrowly engaged with programming and programming languages.

On the other hand, the key contribution of Jordan's work, to this project, was his “does it run?” test, which – much as this paper has done – drove a wedge between FOSS and the Creative Commons, arguing, more or less, that code is fundamentally different from creative works. This breaking of oft-cited links, however, allowed Jordan to describe hackers with much greater precision than other authors, and he later doubled back and wrote an entire chapter on “Hacking the Non-Hack” - a reforging of those broken links, strengthened by a clearer sense of difference (2008, chap. 5). In similar fashion, this thesis – which hones in narrowly on software practices and avoids grander temptations – can be the basis for an understanding of difference, grounding
not only our understanding of FOSS, but also our understanding of things that are like FOSS, but different in important ways.

Unlike Jordan, I have not doubled back at length to reforge these links, but will do so briefly here to suggest a few areas for future research. In very general terms, this thesis's emphasis on material practices might be used as a model for closer study of related areas beyond FOSS. File-sharing and music piracy, for instance, have been discussed extensively in terms of legality and morality, as well as economic consequences, but not as a set of practices. The same might be said of any number of online activities, including crowd-sourcing aggregation sites like Digg and social media sites like YouTube, although Michael Wesch has initiated preliminary work on the latter (Wesch, 2007a).

Moving beyond the sometimes distracting realm of creative and social media, the Open Architecture Network is attempting to apply the FOSS model and Creative Commons licensing to housing architecture, in order to collaboratively design sustainable and affordable urban housing for slum regions around the world. “How do you improve the living standards of five billion people?” the OAN asks rhetorically, before answering “with 100 million solutions” (“Open Architecture Network | Improving living standards through collaborative design.,” 2009). These sorts of applications of FOSS principles are among the most exciting, but also seem at a glance to be the furthest removed from FOSS practices. Might this not be, however, an
opportunity to study the material practices of architects, perhaps in comparison to programming practices, in order to understand how the OAN can better serve its mission? It is an off the cuff suggestion, but this is one sense in which this thesis might act as a guide or foundation for more grounded studies in other areas.

The tendency of authors to describe FOSS in flowery, narrative terms is not particularly surprising, given the difficulty of evaluating its success in empirical terms. It is tempting, even here, to conclude with bold predictions about the future of the FOSS revolution, and the many related revolutions it might spawn in the commons and other spaces. However, if this work has accomplished only one thing, I hope it has demonstrated that the academic discourse of FOSS does not have to be about revolutionaries and warriors, or about David and Goliath, or even about hackers and gift economies. Instead, the discourse of FOSS – or at least some small part of it – can be about programmers, simply programmers, sitting at their computers, iteratively writing and testing their code and moving through many different layers of abstraction – both in their own code and in relation to their peers – all the while quietly embodying, through this work, an unexpected politics.
References


Raymond, E. S. (2001). *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary* (Revised & Expanded ed.). O'Reilly Media, Inc.


Stephenson, N. (1999). *In the Beginning...was the Command Line*. Harper Perennial.


