Starting from Scratch:
Open source collaboration platforms as learning environments

Jane Reynolds

A research paper submitted to the University of Dublin,
in partial fulfillment of the requirements for the degree of
Master of Science Interactive Digital Media

2014

Declaration

I declare that the work described in this research paper is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Signed: _____

Jane Reynolds

28 February 2014

Permission to lend and/or copy

I agree that Trinity College Library may lend or copy this research paper upon request.

Signed: _____

Jane Reynolds

28 February 2014

Acknowledgments

Many thanks to Glenn Strong, my Scratch *and* GitHub guru and my encouraging adviser. Thanks to Mom, Dad, and Aoibhe for being my beta-testers. And Lydia Burke, thank you too, for telling me I could do it.

Summary

This paper approaches open source software development through the lens of learning theory. The project management tools used to coordinate the distributed, democratic development process that characterizes the open source software movement—such as version control software and changelogs—are found to aid inexperienced programmers learning the practice of software development by providing worked examples and by facilitating novice observation of expert practice. An examination of the Scratch Online Community demonstrates how open sourcing and democratic development processes might be employed in a learning-oriented environment. The Scratch Online Community is a uniquely pedagogical open source collaboration platform, designed to engage school-aged children in programming, in which it has succeeded. As such, Scratch's design provides a guide for assessing more typical production-oriented software collaboration platforms. Referencing Scratch and learning theory, a comparison of four such production-oriented software collaboration platforms—The Linux Kernel Archives, Mozilla Developer Network, SourceForge and GitHub—reveals that the new "social coding" platform model, exemplified in GitHub, is expanding learning opportunities for novices in open source development. In social coding platforms, social media-influenced functionality makes developer-level activity transparent across multiple projects. Not only does social coding's developer-oriented approach to open source production support learning, but it provides a context within which developers can exploit the generative potential of open source software.

Table of Contents

Introduction

At the technological level, open source software is no different from any other software, yet the open source movement is often characterized as a transformative force in the software industry (Fitzgerald, 2006). The only thing that distinguishes open source software from proprietary software is licensing—software is 'open source' if its license stipulates that its source code be freely available and modifiable—but this type of permissive licensing has engendered (although it does not dictate) a highly efficient software development model, and it is this development model that has drawn so much attention to the open source movement.

Permissive licensing allows anyone, anywhere to contribute to an open source software project. By leveraging the connective power of the Internet and hosting project files online, open source software can harness the skills, time, and innovations of self-selected contributors all over the world. Open source advocates argue that with more people invested in a code base, fixing bugs, adding features, or just providing feedback, this democratization of the software development process (von Hippel, 2006) produces higher quality software than what any small corporate team could create—and produces it faster (Weber, 2004; Raymond, 1999).

But in order to capitalize on openness, a project community must be able to draw and coordinate new developers, who in turn must be able to learn enough about the software to decide where and how to contribute. Not just the source code but the entire development process must be open and accessible. Most open source projects, therefore, post thorough documentation along with their source code files, provide a web interface to a log of all code changes in the project history, and facilitate communication via mailing lists or forums.

This transparency, necessitated by the practical challenges of coordinating widely distributed volunteer contributors, represents a unique learning opportunity for aspiring software developers.

This paper approaches open source communities as learning environments. Learning support is integral to the structure of most open source projects and collaboration platforms, yet literature on open source development often overlooks the educational potential of the movement.

The first part of this paper, Chapters 1 and 2, explores these concepts in more detail, drawing on learning theory from the fields of cognitive science and anthropology to support the claim that democratic development processes precipitate learning opportunities for novices. Chapter 1, however, also considers how the productive forces behind democratic development may pose barriers to novice participation in some open source communities by putting pressure on contribution and discouraging amateurism.

The second half of the paper applies the theory established in Chapters 1 and 2 to real open source communities and platforms. I compare platforms with an eye to how functionality and interface design differentially encourage or discourage the participation of novices, and more specifically, how platform design affects the ability of novices to learn software development practice. In Chapter 3, I use the Scratch Online Community, a website where very young programmers—most between the ages of eight and sixteen—share and remix small open source projects, as an extreme case study in how open sourced code and distributed development infrastructures can establish pathways for learning.

Scratch, unlike other open source collaboration platforms, is designed primarily as a learning environment rather than a productive environment. With over two million users and well over four million projects shared with its online community, Scratch has been extremely successful in engaging young people of all backgrounds in programming, a historically difficult task, in large part through its unique use of open sourcing. Its design,

shaped by scientific research on learning, highlights two ways in which social coding platforms may support novice learning:

1) By de-stigmatizing the re-appropriation of others' code

2) By facilitating apprentice-like observation of expert thought processes and practices.

Using Scratch as a reference point, Chapter 4 turns to production-oriented collaboration platforms to assess whether these realize the educational potential of open source development or whether productive goals end up suppressing amateur creativity in these environments. In comparing platforms I find that while older project hosting sites tend to exclude novices, the relatively new phenomenon of "social coding" has indeed moved the open source movement closer to fulfilling the educational promise of democratic software development.

Social coding platforms, like GitHub and SourceForge, are websites that provide open source repository hosting and web interfaces for common collaborative tools—namely version control systems and changelogs—within a social network (Begel, Bosch and Storey, 2013). GitHub and SourceForge each host hundreds of thousands of open source projects, connecting millions of developers to each other[1]. They use social media artifacts, like profile pages and activity feeds, to add developer-level transparency to open source development, a new resource for novice developers. These platforms allow novices to follow more experienced developers' activity across multiple projects, facilitating a kind of apprenticeship which learning theorists have posited is the key to building skills like software programming (Hemetsberger and Reinhardt, 2006; Lave and Wegner, 1991; Brown and Adler, 2008).

[1] GitHub user statistics come from http://github.com/about and repository statistics come from http://octoboard.com;

1. Collaboration in democratic software development

College level computer science professors face a challenge in transitioning students from writing small "toy programs" to collaborating on the kind of large, industry grade software that they will actually encounter in their future careers, the kind of software generally protected by copyright and so not available for course material. To meet this challenge, a few experimental university professors—in places as far apart as Texas and Switzerland—have run courses in which computer science students study and contribute to open source projects (Allen, Cartwright and Reis, 2003; Carrington and Kim, 2003; Pedroni et al., 2007; German, 2005). Open source projects make a convenient entry point to production software engineering because, as Pedroni et al. put it, "open source software is accepting contributions even from inexperienced programmers."

More precisely, open source projects don't *necessarily* require credentials (or company affiliation) of participants. Contributions must, however, meet a community's requirements or standards; and project maintainers, in effect, hire a development team by deciding who gets 'write access' to a code base and who is denied. For example, Apache OpenOffice, a well-established open source project, warns prospective code contributors that they will be asked about the "pedigree of the code" they submit for review, implying some initial screening before code is even considered (Apache Software Foundation, 2012). Vetting is a practical necessity for projects like OpenOffice with real users expecting a usable product, or for any project hoping to have real users and use value.

Open source software presents unique learning opportunities for novice programmers, but projects are not usually learning specific environments, meaning that new entrants may have hurdles to jump in order to reap the benefits of participation in an open source community.

>> Democratized development <<

Open source licensing, in combination with network technology, has enabled the democratized software development process that characterizes most open source communities. Open source licensing *permits* universal access to a project's code, and the Internet *provides* that access. Version control software, like Subversion or Git, plays a key role too, allowing developers to manage diffuse, asynchronous or overlapping code submissions to a single repository. Together, technology and licensing facilitate a decentralized development mode in which collaborators self select both their level of participation and the content of their contributions.

This mode of development is democratic not in the political sense, but in the sense that there are no hard restrictions on participation. For this reason Weber (2004) calls open source communities "end-to-end" networks, like the Internet, whose design "takes away the central decision maker in the sense that no one is telling anyone what to do or what not to do."

There are actually "central decision makers", gatekeepers of a kind, in any open source project—namely project owners whose vision and goals may conflict with those from the outer "ends" of the network; however, they can be bypassed by copying a code base and continuing development in any desired direction, because open source licenses protect the right to do just that. Theoretically then, open sourcing promotes large-scale code reuse and a proliferation of niche software, but in practice the opposite occurs: widely distributed, independent work is concentrated into huge industry software like Linux and Apache that competes with proprietary software for market share (Weber, 2004).

The historical taboo against 'forking' is evidence of many open source communities' push towards concentration rather than diffusion of development (Glass, 2003; Fung, Aurum and Tang, 2012; Nyman, 2013). Forking is simply the copying of a code base for independent development, as described above. Attitudes towards forking, and even the connotation of the term, have shifted with the increasing popularity of GitHub (McMillan, 2012), where someone might fork a repository simply as a means of copying it for private study and

experimentation. The taboo forking to which Glass and Nyman refer is a divisive act intended to create a new *functional* software product. This type of fork is seen as competitive with its original, drawing developers away from the parent project and confusing users as to the function that each fork serves. Forking is disruptive to open source projects developing software for real use (not just for tinkering or learning purposes) and even more so for projects with existing users, because the production, let alone the quality, of these products depends on the contributors they attract.

The sheer mass and diversity of developers that an open source project can leverage are seen, and often used, as a competitive edge, especially against proprietary products, for public facing, production software (von Hippel, 2006; Weber, 2004; Goldman and Gabriel, 2005).

Open source advocates argue that software developed in this distributed, democratic mode, is, as Weber (2004) states, "more functional, reliable, and faster to evolve than most proprietary software." For example, the famous "Linus' Law", "given enough eyeballs, all bugs are shallow," (Raymond, 1999) posits that open source software is more reliable than proprietary software because it may be exposed to so many more potential bug-catchers. Eric von Hippel (2006) finds that the general phenomenon of "democratized innovation", which includes but is not confined to democratized software development, offers "great advantages over the manufacturer-centric innovation development systems that have been the mainstay of commerce for hundreds of years," because in these democratic systems users needs and interests are more accurately represented in products.

Von Hippel also argues that democratic innovation, based on user contributions, has major advantages from the manufacturer or business perspective as well. His research demonstrates that "the traditional pattern of concentrating innovation-support resources in just a few pre-selected potential innovators"—in other words, within a private organization—"is hugely inefficient. High-cost resources for innovation support cannot be allocated to "the right people," because one does not know who they are until they develop an important innovation."

Goldman and Gabriel (2005) capture this idea in the phrase "innovation happens elsewhere." They encourage businesses to consider open source licensing on the premise that it is not only naive but inefficient to assume that the best or most useful innovations will come from inside company walls. In doing so they make a very strong link between open sourcing and useful production:

> High productivity requires doing less to produce as much or more—a company that requires its own employees to labor hard and long to make its products or perform its services will be less profitable, in general, than one that can take advantage of the efforts of others.

Open source projects, in this view, become large stars whose gravity pulls in contributions just by virtue of the physics of democratic development.

This emphasis on the commercial advantages and efficiency of democratic development, epitomized in Weber's definition of open source development as "a productive movement intimately linked to the mainstream economy," obscures the other possibilities that open sourcing presents: the educational uses and exploratory code reuse touched upon above.

>> Collaboration: contribution vs. re-appropriation <<

Open sourcing clearly precipitates collaboration, but the literature discussed so far tends to identify only one form of collaboration, one in which multiple contributors become joint authors of a single product, i.e. collaboration as contribution. In the context of production software development, this may be a restrictive type of collaboration as projects will only accept contributions that improve the product in specific ways and directions, or by specific means. Novices in particular may not be able to collaborate in this way.

A study by von Krogh, Spaeth, and Lakhani (2003) suggests that standards for contributions, and even contributors, are indeed a barrier to participation in collaborative programming. The authors observed participation in the Freenet open source project over the course of a year, focusing on attempts by new mailing list members to start contributing to the project. Over a

third of all emails announcing interest in contributing code were ignored, i.e. got no responses from the development community.

Another school of thought, though, takes distributed, democratized innovation as a cultural and political phenomenon which encompasses much more than the software market, and this body of literature includes re-appropriation—that is using parts or all of others' work as a basis for a new work—as another legitimate form of collaboration that openness facilitates (Brown and Adler, 2008; Lessig, 2008; Monroy-Hernandez, 2012). Re-appropriators or remixers are more typically lone creators of diverse works.

Remixing, or re-appropriation of others' work, is also an accepted practice in open source software development, and software development in general, even though this sense of "collaboration" may seem to contradict the former, which also clearly applies to open source software development. Programmers often reuse bits of code for the sake of efficiency (Monroy-Hernandez, 2012; Raymond, 1999), so software develops through *both* contribution and re-appropriation.

Lawrence Lessig writes extensively on remixing or re-appropriation as a form of collaboration (2004; 2008), including in the context of software development. He notes situations in which re-appropriation leads to viable products or innovations in the same way that contributive activity does—open API's, like those of Netflix, Amazon, and Google, for example allow what he calls "LEGO-ized innovation", that is, a type of product development based on tinkering or incremental additions to existing work (2008)—but he focuses on how re-appropriation, or remixing, fosters amateurism. Lessig defends amateur remixing as a public good, part of the foundation of a healthy, democratic society, not as productive habit. His defense rests in large part on the educational value of amateur practice, not on the quality of amateur creation.

In fact, Lessig (2008) admits that much amateur content is very low quality. Studies of the Scratch Online Community, a website where school age programmers can share and re-appropriate open source software projects written in the Scratch language, confirm that

amateur remixes are generally lower quality than their originals, based on peer-ratings (Hill and Monroy-Hernandez, 2013). If amateurism is antithetical to quality, it may be difficult for amateur content to thrive in an environment like an open source software community in which quality is a primary goal and quality contributions are the primary mode of participation.

Evidence from Scratch, aside from reaffirming this conclusion, suggests that the opposite is also true: amateur content thrives when contribution is de-emphasized and re-appropriation is facilitated.

With over two million users and well over four million projects shared with its online community, the Scratch programming language and environment has been very successful in lowering the barriers to programming for the most novice of novices, children between the ages of eight and sixteen (Resnick et al., 2009). So the design choices behind the Scratch language and online platform are a good indicator of what supports learning in an online community of software collaborators (albeit for learners of a very different skill level than anyone attempting to master production software development; still, one could argue that an nine-year-old's skill level relative to the task of making a short animation with Scratch's drag-and-drop code blocks is roughly comparable to a nineteen-year-old's skill relative to the task of contributing a bug fix to an open source project).

The Scratch online platform emphasizes re-appropriation much more so than contribution. The Scratch creators impose an open source license (Creative Commons Attribution Share Alike) on all projects shared on the site and the interface provides a one-click path for "remixing"—essentially forking—any project. Remixing is a heavily used tool among Scratch's very amateur programmers. Remixes made up more than a third of all projects shared to the site in 2013.[2] On the other hand, the platform provides no explicit support for contribution-based collaboration. Collaborative projects do emerge on the website, but in order to coordinate efforts, users adapt interface elements not originally intended as project management tools, using project comment areas as a chat room, for example (Aragon et al.,

[2] Calculation based on graphs at http://scratch.mit.edu/statistics

2009). Chapter 3 explores other implications of Scratch's design and success in relation to open source collaboration platforms more generally.

Open source licensing, again in combination with network technologies, expands possibilities for collaborative software development, in both contribution and re-appropriation modes. Open source communities and platforms narrow these collaborative possibilities in order to meet other needs or goals—some more so than others—limiting novice participation in open source software development. Communities and platforms that discourage re-appropriation and promote contributive activity, for example, may exclude novices from full participation (von Krogh, Spaeth and Lakhani, 2003), although not necessarily from all educational benefits. On the other hand, communities and platforms, like Scratch, that emphasize channels for re-appropriating code may encourage novice participation and supplement the learning opportunities already inherent in open source software.

2. Learning support as a project management tool

The previous chapter introduced the concept of democratized software development and discussed how this production process can both open avenues for empowering, collaborative work and raise barriers to novice participation in open source communities. Re-appropriation emerged as a mode of collaboration in which novices can fully engage, in opposition to contributive collaboration, which can exclude novices. This chapter describes the learning opportunities that open source project infrastructures present novices—but only in a general sense, as each community or platform provides different infrastructure (Chapter 4 addresses these differences). In other words, while the last chapter examined collaborative opportunities that democratic software development *makes possible*, this chapter examines the types of collaborative infrastructures that democratic software *necessitates*, and how these in turn affect the novice learning experience.

>> The role of learning in open source development <<

Individual and collective learning is crucial to the productive goals of any open source project. After the core development phase of an open source project, adoption and growth depend on the project's ability to draw a critical mass of contributors and users (Schweik and English, 2012), and in order to successfully join a project community, potential contributors must be able to learn the software architecture well enough to know how and where to add their code or documentation. And because these potential contributors may be located anywhere in the world, all of the materials to support this learning and all project communications must be hosted online.

The source code itself, available by definition, is an essential guide for newcomers, but most open source projects also provide other documentation, tutorials, and communication tools to make the code more accessible (Hemetsberger and Reinhardt, 2006). Code collaboration platforms like SourceForge, GitHub, or CodePlex, which host hundreds of thousands of browsable open source repositories, provide templates for many of these kinds

of tools. In this sense, learning support is inherent in open source project infrastructure. Specific communities and platforms use or emphasize different project management tools, differentially affecting novice participation, but these differences will be explored in Chapter 4.

>> Novice-specific learning <<

There is some writing on the educational potential of open source software development, but again, because the literature tends to address only successful open source projects and how those are created, the focus is usually on how already skilled programmers start *contributing* to a project, not on how true novices may quietly use open source code resources. Von Hippel (2006) finds, for example, that personal learning goals motivate many user innovations to products, but he also finds that for the most part, those contributions come from “lead users”, which he defines as those deeply invested in the product market, ahead of trends and likely professionals or experts in a relevant field.

These experts seem to learn through contribution, but novices will inevitably learn through different channels. Learning theory literature suggests that novices learn best through observation of expert practices (van Gog and Rummel, 2010; Lave and Wenger, 1991) and through actively engaging in socially meaningful tasks (Jonassen, 1992; Brown and Adler, 2008; Papert, 1991; Fosnot and Perry, 1996; Prince, 2004; Resnick et al., 2009). Distributed software development, aside from being a concrete, motivating context in which to learn software development skills, provides many opportunities for novices to observe and learn from more skilled programmers, because their thought and work processes must be captured and recorded not only in code but in other forms of what Hemetsberger and Reinhardt (2006) call "transactive group memory" in order to coordinate the virtual community.

A public facing open source project, intended for production, is an “authentic” learning environment (Jonassen, 1992), unlike a classroom, which abstracts learning to a degree.

Learning theorists in the constructivist tradition, grounded in cognitive psychology, argue that this authenticity—provided by social context, which in this case is not only the development community but the entire software market—is the key to deep learning and understanding, as knowledge is constructed through actively grappling with physical and social surroundings (Fosnot and Perry, 1996; Jonassen et al., 1995). Pedroni et al.'s (2007) controlled study of two college-level software development classes found that students who studied and wrote code for open source projects had a greater sense of achievement at the end of the course than those who worked on software contrived for teaching, suggesting that authentic tasks are indeed more motivating for learners (and that participation in an open source project is a good introduction to production software development).

Brown and Adler (2008) hail open source communities as proof of "the power of ... social learning." Social learning here means not just group work or group study, but really learning to be part of a community of practice, a process exemplified by but not restricted to apprenticeship. Project management tools, as mentioned above, strengthen this apprentice-style learning.

According to Lave and Wenger (1991), novices enter a community of practice through "legitimate peripheral participation," watching, listening, and carrying out small but meaningful tasks, until they are ready to become full participants. An open source community from this analytical viewpoint can be a rich learning environment for novices because learning is not isolated but embedded in a "social practice"—collaborative software development—"that entails learning as an integral constituent," and because *all* community activity is digitally archived and hence totally observable. The modularity of many open source projects, another characteristic of open source software necessitated by the democratic development process, also enables incremental graduation from peripheral to full participation by identifying small, skill-appropriate tasks that novices can tackle to contribute to the community practice. Bug trackers and issue trackers make identifying these novice-appropriate tasks even easier.

Lave and Wenger approach learning from an anthropological perspective, but their ideas on observation-based learning find support in cognitive science as well. Learners observe expert practice to spare the cognitive effort and risk of blindly seeking out new information, according to cognitive load theory, and this assertion is born out in studies of example-based learning (van Gog and Rummel, 2010; Kirschner, Sweller and Clark, 2006). Alfieri et al.'s (2011) meta-analysis of over a hundred controlled learning studies discovers that novices (of all ages) learn best through semi-exploratory exercises scaffolded by worked examples and human resources.

Open source projects provide just this scaffolding for novices. Source code, supplemented by commenting, functions as both a worked example—a written and explained problem solving process for a specific example problem—and even in some ways as a modeled example—a process enacted live by a real practitioner (van Gog and Rummel, 2010). As Hemetsberger and Reinhardt (2006) explain, “Similar to natural language, code may be understood as an expression of an individual’s thoughts,” but a clear and concise expression, which may hide many unfruitful thought processes or discarded trials. In this way, code, especially when well commented, constitutes a worked example. Revision control systems, programs like Subversion and Git which allow multiple developers to modify and re-submit project files in a code repository, make the collective development process transparent by attaching all repository changes to the developers who made them. Project management tools like Changelogs, commit comments or Diff outputs, also serve to help novices track software development practices and strategies as they observe an open source community from the periphery (Hemetsberger and Reinhardt, 2006).

Commit comments are short descriptions attached to files that a developer has edited and re-submitted, or 'committed', to the code base. They describe what issues were addressed with the change and sometimes background on why the developer chose a certain method for solving a problem. 'Changelogs' display a history of all of the committed bug fixes or new features on a project, and 'Diffs' show the actual code changes—both subtractions and editions—at the places in the file where the changes occurred.

Hemetsberger and Reinhardt argue that these common open source development infrastructures allow novices to "review the whole history of code development" and to thereby "re-experience" expert practice, which they see as the key to learning online. This theory draws on Lave and Wenger and resonates with van Gog and Rummel's findings on example-based learning, but is not novice specific. They point out, for example, that the "release early, release often" ethos of many open source programmers leaves bugs in code that also serve as examples, but more as modeled examples as they reflect the actual human processes behind the programming work. Van Gog and Rummel's meta-analyses of learning studies finds that flawed modeling is actually detrimental to some novice learning, but helpful for more experienced learners. This suggests that some newcomers to an open source community may have a strong enough cognitive schema of software development practices to benefit from "re-experiencing" flawed thought processes, but others may not.

Examples are not the only type of scaffolding that open source project management tools provide novice learners, however. Novices can observe conversations and decision making that take place over mailing lists or in project forums. Experts may address questions directly through these outlets, providing feedback which is key for learning (Alfieri et a., 2011). Hemetsberger and Reinhardt call these archived (or archivable in the case of email) discussions the "transactive group memory" of an open source community. All project documentation, wikis, or tutorials also contribute to transactive group memory, which becomes a resource not just for novices but for all members of the community, as individuals and as a collective.

These typical open source project management tools support novice learning in theory, but as Allen, Cartwright, and Reis (2003) discovered with the undergraduate software engineering course at Rice University, "Not every open source project is a good candidate for use in the classroom. Many such projects have high "barriers to entry"—such as inadequate documentation, poor coding style, and the lack of comprehensive unit tests—that new developers must overcome before they can reliably extend the code base." The next two chapters analyze how specific communities and platforms support novice learning through infrastructure and interface design.

3. The Scratch Online Community: A model learning environment

This chapter introduces the Scratch Online Community, a model of open source licensing and production models implemented specifically to support learning. The analysis here informs the conclusions drawn in Chapter 4, which assesses learning supports for novices in production-oriented software collaboration platforms.

Scratch is an introductory programming language with a custom development environment designed to engage young people of all backgrounds in programming by empowering them to create digital media—animations, games, interactive stories, and more. Scratch users can post their projects to the Scratch Online Community (scratch.mit.edu), where an audience of millions of other users can play with, give feedback on, and remix the work. The Scratch Online Community does not immediately present itself as a point of comparison for software collaboration platforms like GitHub and SourceForge, but if we define these platforms as online networks of people sharing, downloading, and modifying source code, then the Scratch Online Community is no doubt an open source software collaboration platform, even if it doesn't identify as such.

All projects published to the Scratch Online Community are covered by a Creative Commons Attribution Share Alike license, and come with a “Look Inside” button to view the source code, or in this case, code blocks (Scratch uses drag-and-drop blocks for scripting), and a “Remix” button to create a new project with the source, equivalent to forking. But unlike with typical open source software development platforms, code sharing is only a feature of Scratch, not its core functionality.

The ability to share projects with other Scratch users and the requirement that the source code be open are just parts, although crucial, of Scratch's *pedagogical* strategy. That is, Scratch uses open source licensing for the purpose of supporting learning, whereas the platforms to be discussed in Chapter 4 support learning as a function of open sourcing. Scratch was created in the tradition of programming languages and environments designed to

lower the barriers to programming, starting as far back as BASIC in the 1970s (Kelleher and Pausch, 2005). But even in that tradition Scratch is unique, in large part because of how it leverages open sourcing.

Because Scratch has been so successful in empowering those with no prior knowledge to learn to program—putting into practice the theoretical learning supports discussed in Chapters 1 and 2—its design principles inform a subsequent analysis of how well other open source software collaboration platforms support novices learning to be software engineers.

The effectiveness of Scratch as a learning tool may be seen in the increasing sophistication of users' code over time. Monroy-Hernandez (2012), lead developer of the Scratch Online Community, reports that "the metric for general complexity of a project, i.e. sprites, increases as the age of the account (Figure 3-31). We see the same for code complexity but different for media complexity." Code complexity is measured by the number of scripts, average number of code blocks, and number of unique block types. "Blocks" here are single commands, functions, or logic structures (Scratch provides blocks for if and if-else statements, and several types of while loops), while a "script" is a group of blocks controlled by an event listener. Of course these statistics don't indicate exactly what causes a user's coding to improve in sophistication over time—perhaps he or she learns coding concepts through Scratch, becomes more familiar and confident with the language through practice, or matures cognitively over time—but in the absence of a scientific study on learning outcomes of Scratch users, this trend towards code complexity at least indicates that Scratch users do become better programmers during the time in which they are involved in the Scratch Online Community.

By measures of popularity and activeness of community, Scratch is at least the *most* successful current attempt at introducing programming to novices. Six years after launching, the Scratch Online Community boasts over 2.7 million registered users,[3] and more than 7,500 educators participate in the ScratchEd community.[4] Alice and Greenfoot, two contemporary

[3] http://scratch.mit.edu/statistics
[4] http://scratch.mit.edu/educators

introductory programming environments to which Scratch is often compared (Fincher et al., 2010), on the other hand, have approximately 8,800[5] and 19,200[6] members, respectively, in their online communities. To put it in perspective, SourceForge, founded in 1999, has 3.7 million registered developers (sourceforge.net/about). Moreover, community members are active and producing content. There have been over 4.8 million projects shared with the Scratch Online Community—about 1.7 per registered member—and the rate of project submissions has risen steadily since the website launched (scratch.mit.edu/statistics).

Albeit, Alice and Greenfoot have slightly different goals and target audience—both aim to increase retention in college-level computer science courses, and to transition users to more general programming environments (Cooper, 2010; Kolling, 2010), which Scratch makes no pretentions towards—and so online presence is not as central to either's design. Yet membership, site traffic (compete.org) and forum participation is an order of magnitude bigger on Scratch.mit.edu than on either Alice.org or Greenfoot.org, a big enough difference to infer that Scratch is indeed more widely used and that it is more effective at engaging its community.

>> Scratch's background in learning theory <<

Scratch was developed by the MIT Media Lab's Lifelong Kindergarten research group to introduce young people to programming with the larger goal of promoting what Lifelong Kindergarten calls “digital fluency”, a critical awareness of the processes, technologies, and computation that drive digital media. To achieve this goal, Scratch embraces three core design principles: “Make it more tinkerable, more meaningful, more social than other programming environments.” (Resnick et al., 2009)

[5] This is the approximate number of Alice forum members, posted at http://www.alice.org/community/; the forum is Alice’s only online user community

[6] I calculated this by manually counting users starting at http:// http://www.greenfoot.org/users/list/score/0

While the Scratch creators cite only Papert's constructivism and Dewey's progressivism as influences on their pedagogical strategies (Resnick and Rosenbaum, 2013; Resnick et al., 2009; Monroy-Hernandez, 2012), these three goals clearly align with other learning theories discussed in Chapter 3. *Tinkering*, which Resnick and Rosenbaum define as a problem-solving approach characterized by experimentation with existing materials, may be seen as a variation on example-based learning, which cognitive load theory upholds as the most effective learning path (van Gog and Rummel, 2010). *Meaningful* work within a *social* context basically describes legitimate peripheral participation (Lave and Wenger, 1991).

To foster digital fluency Scratch empowers users to *make* exactly the type of digital content that they, as young people, are likely to *consume* on the Internet, elevating users from the role of audience member to the role of actor. Scratch projects, just like any other open source software programs, are functional works created with users in mind. Scratch programming happens within the meaningful social context of the online community—as much a community of practice as GitHub or SourceForge—which makes simple programming exercises authentic by bringing the final product to a real, world-wide audience on the Internet. And in such a community, learning follows from participation (Lave and Wenger, 1991; Brown and Adler, 2008).

>> Novice learning support embodied in the Scratch platform <<

Based on the analyses from Chapters 1 and 2, the Scratch Online Community interface[7] biases users towards exactly the kinds of collaborative activity, and with the kinds of infrastructure, that one would expect in a novice-centric software collaboration platform. In other words, the Scratch interface encourages re-appropriation much more so than contribution, and allows users to observe the thought processes and work-related activities of other users.

[7] The Scratch desktop application is not considered here as it is more properly classified as an integrated development environment, and is independent from the Scratch Online Community. For concision, the Scratch Online Community is referred to as "Scratch" from here on.

Re-appropriation

Scratch users can access the source code of any project shared to the site with a single button click. The "See Inside" button on the project page opens up the development view and toolbar, displaying all the project's sprites (graphical objects) and scripts, and in effect eliminating the barrier between project viewing and project development. Once this panel is open a “Remix” button appears in the top right of the window. Again with a single click, this button creates an exact copy of the project and moves it to the remixer's file of unshared projects. It also opens up the project in the remixer's development page so he or she can begin tinkering immediately. Scratch automates attribution, so when a remix is published a link to the originating project appears below the “Notes and Credits” section of the project page. Users can and do attribute manually as well.

The Scratch platform design not only facilitates re-appropriation, but encourages it. While other open source communities may tacitly discourage re-appropriation, namely forking, Scratch highlights re-appropriation, namely remixing, and rewards heavily remixed projects with space on the homepage. A panel next to the comments section on each project page lists and links to all remixes of that project, which can also be viewed in a “Remix tree” accessible from the project page. Remixes are thus treated as a metric of project quality or popularity. Frequently remixed projects are also featured on the Scratch homepage, again not only normalizing but celebrating re-appropriation.

The platform design does seem to influence use as over a third of all projects published on the site are remixes. Users may remix projects for private experiments which they may not necessarily publish, so the number of actual remixes created may be much larger than the number reported.

Anecdotal evidence from users also indicates that remixing is an effective learning tool. This is a statement from a 14-year-old Scratch user:

> "*I have to say if it weren't for remixing, I would have never understood velocity or scrolling. It should be used for things other than "add yourself" and coloring contests (not that I'm against those in any way) it's a tool that makes the Scratch community stand out as a friendlier and more learning based environment.*" (Monroy-Hernandez, 2012)

On the other hand, the Scratch platform provides no support for contribution-based collaboration. These types of collaborations do happen (Resnick et al., 2009; Monroy-Hernandez, 2012) and "Collaboration" is the second most active forum topic by measure of total posts, but users must coordinate these types of collaborations through interface features designed for other purposes, like project comments, for example.

Example-based learning

Novices learn through observation of more skilled members in communities of practice, "re-experiencing" experts' thought and work processes (Hemetsberger and Reinhardt, 2006).

Source code provides a map of others' thoughts and strategies in Scratch as in all open source collaboration platforms. Scratch provides but does not advertise some code commenting functionality, which would enhance the source code as a worked-example, but it does provide space for project authors to leave notes on their work and the comments section below each project allows users to ask project creators any questions they may have about how the work was created.

Profile pages provide insight on a user's influences, interests, and strategies. Each profile contains information on who the user "Follows" (i.e. from whom the user receives updates on all activity), who in turn follows the users, which projects the user has bookmarked as "Favorites", what "Studios"—aggregated links to themed groups of projects—the user receives updates from, and what studios the user curates.

The Remix tree feature on the project page may also be used to trace changes between versions of a project, granting an observer some insight into the remixer's process.

Direct support

Scratch also provides forums for users to solicit help or instruction directly. The "Comments" area of the project page can also function as a forum. The website Help page provides video and text tutorials, and users themselves have created tutorials on specialty techniques.

Scratch demonstrates how open source licensing and the open collaboration that it enables can be used to intentionally support novice learning. Scratch's success validates claims that novices learn best through worked and modeled examples, within authentic communities of practice. The popularity of Scratch's remix functionality also indicates novices thrive in an environment that facilitates easy re-appropriation of others' work. In the following chapter I search for echoes of Scratch design in various production-oriented software collaboration platforms to determine, some of which to bare striking resemblance to Scratch even though their design intent may not be pedagogical in any form.

4. Social coding:
New opportunities for novices in open source

The Scratch creators in MIT's Lifelong Kindergarten group designed their platform specifically to support novice learning. Most other open source software collaboration platforms and communities serve a very different primary function, which is to coordinate the distributed development of production software. However, there is a significant amount of design overlap between Scratch and these production-oriented platforms, particularly social coding sites. Though these design elements may derive from non-learning-related goals or functions, they nonetheless empower novices in these platforms. The novice experience of democratic software development may be very different, and more or less fruitful in terms of building software development skills, depending from what platform she or he enters the open source world.

This chapter uses Scratch and learning theory as guides for identifying design elements that support novice participation and learning in two types of open source software collaboration platforms: single project hosting site and social coding site. Specific platforms within these categories are then evaluated for novice-friendliness based solely on what functionality, interface elements, and other infrastructure they provide. Other cultural or idiosyncratic factors influence the supportiveness of a given open source project community, independent of hosting platform, but these fall outside of the scope of this paper.

I will evaluate representative examples of both single project sites and social coding sites on the learning supports provided by their respective functionality and design. This will serve both to demonstrate the full spectrum of novice-supportiveness in software collaboration platforms and to show how social coding platforms in particular represent new opportunities for novices in open source software development.

>> Categorizing open source software collaboration platforms <<

I define an open source collaboration platform as an online hosting site for one or more open source code repositories, created to allow multiple developers to download a project or upload code changes to it through some version control system.

Single project hosting site

These are independently hosted websites dedicated to a single project repository, like the OpenOffice site (openoffice.apache.org). This category also includes websites that host multiple repositories for related projects—the Mozilla Developer Network would be an example—because they offer a single suite of developer tools through uniform interfaces.

These tend to be older, well established projects that started development before sites like SourceForge, which launched in 1999, began hosting multiple projects and offering template web interfaces and management tools for repositories.

Social coding site

Social coding sites combine code repository hosting and version control with social media tools like personal profiles, project bookmarking, and newsfeeds of other users' activity to collect and display developer-level data on activity across multiple repositories. These platforms, unlike dedicated single project hosting sites, provide a standard interface and project management functionality to multiple projects. SourceForge and GitHub, two of the most popular, host hundreds of thousands of open source repositories.

The term is relatively new and seems to come from the GitHub tag line, “social coding”, but many other software collaboration platforms, and not all of them open source oriented, fall into this category, including SourceForge, CodePlex, Bitbucket, Google Code, Launchpad and Perforce. These platforms commit to the social aspect of 'social coding' to varying degrees, some only as a token gesture it seems. GitHub is the poster social coding platform.

Social coding, as exemplified by GitHub, is seen in popular media as a major shift in distributed development systems[8]. The research community has just begun to recognize the significance of the shift: "Social coding enables a different experience of software development as the activities and interests of one developer are easily advertized to other developers." (Thung et al., 2013)

GitHub, which launched officially in 2008,[9] is the most prominent among these sites and the early research that has begun to explore the implications of this emerging trend towards social coding uses GitHub as a representative of all social coding platforms (Dabbish et al., 2012; Thung et al., 2013; Begel, Bosch and Storey, 2013). GitHub fully invests in its use of the social media tools mentioned above, which creates total transparency of developer actions on the platform (Dabbish et al., 2012).

>> Platform features that support novice learning in open source <<

The analysis of Scratch in the previous chapter took the designer's intentions into consideration in order to demonstrate how a platform might fully address novice learning needs, as a special case. To be clear, none of the platforms examined in this chapter claim better support for novices or learning as an offering. As discussed in Chapters 2 and 3 software collaboration platforms provide inherent learning opportunities and supports for novices in software development, simply *as a side-effect* of their democratic software development infrastructures. One purpose of this analysis is to determine how this democratic development infrastructure manifests in practice on different platforms and to assess whether differences in approach to democratic software development affect learning opportunities for novices.

[8] Some example online articles: "Social coding – the next wave in development", Tech Republic. <http://www.techrepublic.com/blog/it-consultant/social-coding-the-next-wave-in-development/3257/>; "GitHub: Making Code More Social", O'Reilly Radar. <http://radar.oreilly.com/2009/01/github-making-code-more-social.html>

[9] https://github.com/blog/40-we-launched

Based on the preceding discussions of learning theory and how the Scratch platform reflects learning theory in its design, I have identified the following common platform design features that support learning or novice participation. Each feature is categorized as part of either the platform's participation paradigm, worked and modeled examples, or direct learning support.

Participation paradigm

As previous chapters have shown, novices are most likely to thrive in a community whose participation paradigm emphasizes re-appropriation and experimentation over contribution and productivity. Design features or platform functionality that enable this type of participation include:

Distributed Version Control support

Version control systems are programs that manage and record changes to a repository from multiple developers. Projects that employ Distributed Version Control Systems make a significant break from older collaborative software development models based on Centralized Version Control Systems.

Older Centralized Version Control Systems, like Subversion and CVS (Concurrent Versions System), require developers to download relevant files from a project repository and make changes to those files locally, and then in order for other contributors to see and build from these changes, the Centralized Version Control System requires that the developer send a patch file to a project maintainer who then decides whether or not to merge those changes back into the repository. This version control system rests on a central server hosting an official version of a project repository. Developers need permissions just to get access to a writeable version of project files (open source projects make source code available by definition but they may not necessarily be writeable files) and permission to make changes. In a centralized project, if a project maintainer rejects a contributor's patch, that developer must create a brand new repository—a fork—with independent hosting in order to share his or her changes with others.

With Distributed Version Control Systems, like Git, Mercurial, or Bazaar, in contrast, new developers *must* fork a repository in order to make changes. These systems are "distributed" because each developer owns a clone of the entire repository, not just downloaded files from a single, central server. Publishing changes is much easier with distributed version control and doesn't require any permission because each collaborator owns an equally functional version of the repository. Requesting write access to a centralized repository may be very daunting (recall von Krogh, Spaeth, and Lakhani's (2003) finding that the Freenet community and project maintainers wholesale ignored a third of all requests to become developers on the project), but in a distributed version control system, a developer may publish changes to her own fork at any time and issue a "pull request" if she would like the parent branch maintainers to consider merging those changes once they have already been made (Chacon and Hamano, 2009) .

GitHub's marketing manager, Brian Doll calls forking "a new model of participation", "made possible by Git itself." (Begel, Bosch and Storey, 2013) The dominant model of participation that Doll defines forking against is contribution-based. This excerpt from a Wired article on GitHub, "Lord of the Files: How GitHub Tamed Free Software (And More)", explains the historical importance of this new attitude towards forking:

> Back in the 1990s, forking was supposed to be a bad thing. It's what created all of those competing, incompatible versions of Unix. For a while, there was a big fear that someone would somehow create their own fork of Linux, a version of the operating system that wouldn't run the same programs or work in the same way. But in the Git world, forking is good. The trick was to make sure the improvements people worked out could be shared back with the community. It's better to let people fork a project and tinker away with their own changes, than to shut them out altogether by only letting a few trusted authorities touch the code. (McMillan, 2012)

Distributed version control systems thus de-stigmatize forking and tinkering, just like Scratch. Git and its peers encourage proliferation of software experiments, or re-appropriation in other words. Hence version control determines the role of the novice in distributed software development to a large extent.

Rewards re-appropriation

Some platforms facilitate easy re-appropriation of code via distributed version control but do not provide hosting space for users' forked repositories, diminishing some of the implication of the distributed system just mentioned.

Serendipitous discovery

The Scratch homepage exposes visitors to new projects—both curated and activity generated, as in “What the Community is Remixing” or “What the Community is Loving”—in order to generate thinking and expand a user's view of what is possible with the platform (Monroy-Hernandez, 2012). This fosters a culture of exploration among users.

Worked and modeled examples

These are platform design features that enable novices to either study written artifacts as worked examples or to observe other developers' activity as modeled examples (worked and modeled examples are defined in Chapter 2). Source code itself and comments on that code also serve as worked examples in an open source project, but neither are platform design features (source code availability is a function of the project license and code commenting is not enforceable at the platform level, only at the cultural level of a given individual project). This analysis considers only the following features and functionality that serve as worked or modeled examples in a software collaboration platform:

Commit commenting

These are generally short comments on changes, or ‘commits’, that a developer has made to a repository. They are generally attached to the file that was modified and aid in interpreting the changes to those files.

Visible changes

Changelogs display the history of all ‘commits’, which are sets of modified files that have been merged back into a repository. 'Diffs' are files that describe the actual changes to code.

As with commit comments, changelogs and diffs are standard open source project management tools.

Mailing list

Experienced developers model decision-making processes over mailing lists. Novices may learn what considerations factor into software architecture and other aspects of design by subscribing to mailing lists (Hemetsberger and Reinhardt, 2006).

Transparent developer activity

Recordable developer activity on a collaboration platform includes code commits, pull requests (a request to merge changes from one fork of a repository to another), flagging bugs or issues, and in the case of social coding platforms can also include social activity such as bookmarking or subscribing to updates from a project, or subscribing to updates from other developers. Experienced developers can thus serve as models for novices who follow their activity.

Direct support

Platforms may provide direct learning supports that benefit not just novices but developers of all levels entering the project.

Documentation

Projects provide technical descriptions of the project architecture and functionality for all developers, expert and novice. Documentation is the primary access point for any software project.

Forum, chat

These are channels for feedback on code or for finding answers to specific questions.

Other resources (i.e. tutorials)

>> Feature comparison <<

This section compares four representative sites—two single project hosting sites and two social coding sites—on platform design features that support novice learning. Mozilla Developer Network and the Linux Kernel Archives, which hosts the Linux kernel project, have been chosen to represent opposite ends of the spectrum of single project hosting sites in terms of learning supports. GitHub and SourceForge likewise have been chosen to demonstrate the range of learning supportive features that a social coding platform may implement.

The Linux Kernel Archives

The Linux Kernel Archives hosts hundreds of repositories for different Linux kernel components. The Linux kernel was first released in 1991 and since then Linux software has become very widely used and very complex (Weber, 2004). The archives do use a distributed revision control system—as mentioned above, the distributed version control system, Git, was created specifically to manage Linux kernel development—but participation in the Linux kernel project is very exclusive and in effect very centralized. The site's FAQ makes clear that programming experience is a limiting factor in project participation, stating, "Kernel.org accounts are not given away very often, usually you need to be making some reasonable amount of contributions to the Linux kernel and have a good reason for wanting/needing an account."[10]

While the platform enables easy re-appropriation through Git, it does not reward re-appropriation or experimentation. The platform does not automate hosting for forks and does not provide non-members any channels for submitting a pull request. Novices may participate in projects only by *contributing* bugs through the Kernel Bug Tracker.

Nor does the Linux Kernel Archives facilitate serendipitous discovery of content. The Archives' homepage provides recent kernel files for download along with links to the web interface for each file's Git repository, but little else. The full list of Git repositories may

[10] https://www.kernel.org/category/faq.html

facilitate some discovery or browsing, but it provides only one-line descriptions of each repository. These descriptions are often cryptic and require prior knowledge just to interpret. Some examples: “Pahole and other DWARF utils”, “zisofs tools”, “Unnamed repository”.

The platform does provide a changelog and commit comments for all repositories, but displays them in a way that assumes prior knowledge of project culture and systems. The homepage links to plain text changelogs but in other places on the site changelogs are rendered in a clearer format.

Direct support for learners is also very limited on this platform. The site does not provide mailing list sign up but does provide instructions on how to request mailing list subscription via email. A list of Wikis is available from the site and these sometimes contain information on how to start participating in a project.

Mozilla Developer Network

The Mozilla Developer Network (MDN) is a much more welcoming and novice-supportive single project platform than the Linux Kernel Archives. However, as a dedicated platform for product development its participation paradigm is still contribution-based.

All releases of Gecko and Firefox since 2009 are stored in a Mercurial repository, which is distributed. However, older products in the Mozilla network use CVS, a centralized version control system.[11]

The MDN platform, unlike the Linux Kernel Archives does reward re-appropriation to some extent. Once users gain a requisite level of repository access, they are encouraged to publish

[11] This page, buried in the MDN website states that Gecko and Firefox moved to Mercurial after versions 1.9.1 and 3.5 respectively: <https://developer.mozilla.org/en/docs/Mozilla_Development_Tools>; for information on when these versions were released: <http://website-archive.mozilla.org/www.mozilla.org/firefox_releasenotes/en-US/firefox/3.5/releasenotes/>

experiments to a personal Mercurial repository within the Mozilla code tree so that others may see and give feedback on changes without needing to go through a central gatekeeper.[12] Again, though, the platform is centrally controlled and requires a module owner to vouch for a new contributor before she or he is granted permission to publish code within this 'users directory' of the Mozilla repository.[13]

That being said, the site does provide substantial direct support for prospective participants. The "Contributing to the Mozilla code base" page[14] articulates a standard process for gaining access to repositories, which includes completing a "Mentored Bug". These are easy bugs which have been designated for novices as a means of encouraging new contributors. An experienced developer is assigned to each bug, to answer questions and review code. The platform also provides video tutorials, mailing list subscription, an introductory chat room, and thorough documentation by way of supporting new contributors.

Developer activity is mostly hidden. Names are associated with commits in the Mercurial changelogs and optional user profiles lists statistics on developer activity but not link to examples of the developer's work. Nor is this developer-level information broadcast to other users.

SourceForge

SourceForge hosts over 430,000 open source software projects[15] and its design certainly encourages exploration and discovery of those projects; the SourceForge homepage, like the Scratch homepage, highlights software products selected by both the community and site curators.

SourceForge offers a repository interface and a suite of project management tools—a ticket tracker for reporting bugs or requesting features, wiki template, discussion forum template,

[12] https://developer.mozilla.org/en-US/docs/Creating_Mercurial_User_Repositories
[13] https://www.mozilla.org/hacking/commit-access-policy/
[14] https://developer.mozilla.org/en/docs/Introduction
[15] http://sourceforge.net/about Accessed 24 February 2014

and a mailing list manager—but does not require that a project use any of these features. The platform allows projects to use their SourceForge web space just to redirect to an official site, or even a different social coding site repository, like a GitHub project.

In this way, SourceForge acts more as a catalogue of open source software than a development community, catering to software users rather than developers. The homepage highlights projects *for download*, that is, it provides easiest access to executable files, not source files. From the homepage a user may search for a specific project or browse projects by category, and again each listing is accompanied by a prominent green download button.

So while SourceForge does foster serendipitous discovery and does support distributed version control (projects have the option to use a centralized version control system as well, and many do), the overall design promotes consumption as a primary mode of participation in open source software development—not re-appropriation *or* contribution.

Some projects employ SourceForge's Git interface, and on these projects a “Fork” button provides easy re-appropriation. SourceForge then provides hosting space for the forked repository which is accessible via the parent project. These forks may be manually shared on the developer's profile page.

The platform does incorporate social media tools and a level of developer activity tracking, although again the features are optional. A developer may or may not maintain a SourceForge profile where he or she may self-report skills, a CV, a link to an external homepage or even a wiki. A developer may also choose to list SourceForge projects to which she contributes, but the process is not automated. A user activity feed shows recent commits. The feed does not indicate to what project the commit was made, but it does link to the Diff file within the project. Users may subscribe to activity feeds of other developers, or bookmark an activity feed. The profile does not show a developer's subscriptions (the feeds are available only from the subscriber's private Account page). A developer may also send a personal message to another developer via the profile.

GitHub

GitHub, more so than any of the other platforms discussed here, encourages re-appropriation, although some of its social features also reward contribution. GitHub is essentially a web interface for the Git distributed version control system. GitHub only supports Git.

When a user forks a project, GitHub automatically allocates server space for the forked repository, builds a page for the repository that include a set of standard project management tools—file manager, changelog and commit comments, issue and pull request tracker etc.—and adds a link to the repository in the user's profile page. A fork, to reiterate, is essentially a clone of a repository that can always be linked back to its parent. As soon as a user makes a commit to a forked repository, that activity will be visualized in the parent project's "Network Graph", and in the Network Graphs of other repositories forked from the same parent. The Network Graph is a timeline which shows commit activity on all forks of a project. The graph is automatically generated, which means that all of a user's work on a fork is broadcast across the project network. Forking does not require permissions or any sort of membership in a project, so the Network Graph connects developers working on the same base project who may not ever have interacted. The GitHub platform thereby increases exposure of re-appropriation-based work.

The platform also encourages serendipitous discovery and project exposure through the "Explore" pages, which highlights "trending repositories". Trends are determined by the number of "Stars"—basically bookmarks—a repository has received in a given time period (Begel, Bosch and Storey, 2013). The Explore page also features staff selected repositories, which change week by week.

Social tools are fully implemented in GitHub. A user may bookmark or receive updates from any repository or developer. That activity is made public on the user's profile page, along with statistics on how many other developers "follow" a user (i.e. have subscribed to updates on the user's activity). Unlike in SourceForge, GitHub profiles are automatically populated based on activity.

Although, tools like the Network Graph and functionality like one-click repository forking and hosting certainly de-stigmatize re-appropriation in GitHub, the platform still rewards contribution. A heat map of "Your Contributions" over the last year, is the centerpiece of every user's profile page. A feed of recent contribution activity appears below. When other users visit a profile, they will immediately notice how *productive* this other user has been. This emphasis on contribution does put novices at a disadvantage.

GitHub accommodates only very minimal direct support for learners. It provides no mailing list sign up or forum/chat functionality. Projects may provide some documentation or external links to resources like wikis or tutorials via a rendered README file.

>> Discussion <<

While single project hosting sites seem to have more leeway to provide direct support for learners and prospective contributors, the role of a novice in these platforms is very restricted. Even in the welcoming Mozilla Developer Network novices must follow a well-defined official process of participation. There are no opportunities for exploratory learning or tinkering on this platform. The mentorship that the Mozilla Developer Network offers could prove useful to an inexperienced developer, but would not replace learning gained through legitimate peripheral participation in a full community of developers.

The four platforms examined differ most in terms of how they facilitate this legitimate peripheral participation, or behavior modeling by experienced developers. GitHub and SourceForge both allow novices to observe other developers' activity via social tools, but developer-level activity is much more visible on GitHub than on SourceForge. Because SourceForge does not automate activity reporting, much developer activity may remain hidden, giving novices a partial or false understanding of another developer's work processes.

GitHub's design enforces transparency of developer habits, strategies, and interests. Dabbish et al.'s interviews of GitHub users revealed that developers do indeed read other developers' activity feeds for information on how to build programming skills, and that the practice of following "coding rockstars" is common on GitHub. Through the social coding platform these famous developers become masters to a potentially huge pool of apprenticing software developers. Dabbish et al.'s study concludes that "the transparency on GitHub supported learning from the actions of other developers. Being able to watch how someone else coded, what others paid attention to, and how they solved problems all supported learning better ways to code and access to superior knowledge."

GitHub also distinguishes itself from other platforms by bolstering re-appropriation as a mode of participation. SourceForge goes part way to doing the same by providing Git support and web interface. Many SourceForge repositories, though, use a centralized version control system, and so support for re-appropriation is patchy. All GitHub repositories, on the other hand, may be easily 'remixed'. By restricting version control options GitHub actually creates a culture of freedom—amateurs may participate anywhere on the platform.

As described in Chapter 1, democratic software development widens the scope and scale of software collaboration in two opposite directions; it enables both large-scale re-appropriation of code and proliferation of software experiments, as well as an unprecedented concentration of diffuse human resources on certain prominent projects. The two categories of software collaboration platform detailed above exemplify these two different interpretations of democratic software development and its best applications.

5. Conclusion

The implications of democratic software development for economics and innovation are well documented and thoroughly researched, but this paper has approached the phenomenon from an educational perspective, asking how novices may learn and build software programming skills within the context of an open source software project. Democratic software development necessarily takes place over open online platforms, and my research focused particularly on how social coding platforms introduce new opportunities for novices to learn from open source software development communities.

Open source software, just by virtue of accessibility, represents a unique resource for young programmers trying to transition from writing small, purely educational programs to contributing to large production software projects.

Moreover, research on learning from the fields of cognitive psychology and anthropology suggests that deep learning happens in just the type of context that an open source software development project provides: in communities of real practitioners of a skill or art, in which novices can observe experts at work and take on small but meaningful tasks to support that work (Lave and Wenger, 1991; Fosnot and Perry, 1996). These communities of practice provide novices with both worked and modeled examples of problem solving strategies or approaches to a task, crucial scaffolds for novice learners according to cognitive science and controlled studies (van Gog and Rummel, 2010; Alfieri et al., 2011).

Because open source software development takes place online, the interface that a collaboration platform provides is crucial in determining what a novice can observe and glean in an open source community. All open source software collaboration platforms, designed to facilitate democratic development, provide standard project management tools for teams of distributed developers along with some learning resources to help developers become contributors to the project, and to thereby help the project realize the potential of the democratic development model in amassing human resources at little or no cost to the project

creators. These tools and resources, such as changelogs, documentation, or forums, however, also serve as worked examples for novices in any open source project.

Single project hosting sites, and consumer-oriented project aggregator sites like SourceForge provide only these tools. Social coding sites, however, in addition to providing these standard tools and resources, also collect and display information about developer activity across multiple projects. This information on developer-level activity constitutes a new and essential resource for novices.

On a social coding platform, novices may in essence apprentice under "coding rockstars" (Dabbish et al., 2012), tracking these experts' code changes and style. Experts' project participation and bookmarking patterns also signal to novices what projects are most valuable to pay attention to or use. In other words, more experienced developers serve as virtual modeled examples over social coding platforms.

Tellingly, the Scratch Online Community, a uniquely pedagogical software collaboration platform, qualifies as a social coding platform. Just like GitHub or CodePlex, Scratch integrates user profiles, transparent project bookmarking, and project rating with programming tools. As a community of practice expressly designed to help very novice programmers build skills—in which it succeeds—Scratch may be taken as a model of learning-supportive design. Scratch's endorsement of social coding tools again indicates the potential of more typical, production-oriented social coding platforms to foster novice participation and learning.

Social coding sites have introduced new support for novices in open source in terms of interface features and site functionality. But more than just the design, the *concept* of social coding lowers barriers to novice participation in open source software development. Social coding fundamentally shifts the focus of software production from project to developer.

Social coding's emphasis on developer activity, in combination with the de-centralized repository structure introduced by Distributed Version Control Systems allow developers to

realize a different potential of democratic software development than the one championed by economists. Git, a distributed version control system, enables easy repository cloning and hosting; GitHub makes it even easier, providing one-click forking. More importantly, as a social coding site, GitHub automatically posts the new fork in a developer's profile space and broadcasts that activity. The total effect is a new participation paradigm, one that de-stigmatizes re-appropriation and allows experimental work and collaboration to flourish.

Distributed version control de-stabilizes the idea of an 'official' copy of a code repository and the idea of project owners/gatekeepers, both intrinsic in older centralized version control systems. This technology is critical in promoting a permissive paradigm of participation; at the same time distributed revision control is not very different from centralized revision control unless all distributed repository clones can be published, i.e. face the community. This is the case for some large projects with devoted independent hosting sites, like those on the Mozilla Developer Network—Gecko and Firefox use Git repositories but only a few Mozilla sanctioned copies of the distributed repositories are available for others to use and experiment with from the Mozilla Developer Network.

Social coding sites like GitHub provide the necessary social context within which distributed version control technology can facilitate a more democratic form of democratic software development.

Again, the similarity between GitHub and Scratch is notable. Scratch promotes remixing as primary mode of interaction with others' work, just as GitHub promotes forking—both forms of re-appropriation. Scratch statistics indicate the popularity of remixing among amateur creators and further research could investigate the link between remixing and learning outcomes in Scratch.

Further research could also include an empirical study of novice activity on social coding sites, using the SourceForge and GitHub API's to determine whether user code does in fact improve more over time in GitHub's environment. This could ultimately lead to a set of

recommendations for a fully learning-supportive open source collaboration program, which would be of interest to university computer science programs.

Bibliography

Books

Chacon, S. and Hamano, J. (2009) *Pro git*. Vol 288. Berkeley, CA: Apress.

Goldman, R. and Gabriel, R. (2005) *Innovation Happens Elsewhere: Open Source as Business Strategy*. Morgan Kaufmann.

Lave, J. and Wenger, E. (1991) *Situated Learning: Legitimate Peripheral Participation*. New York: Cambridge University Press.

Lessig, L. (2004) *Free Culture*. New York: The Penguin Press.

Lessig, L. (2008) *Remix: Making art and commerce thrive in the hybrid economy*. London: Bloomsbury Academic.

Papert, S. (1991) *Mindstorms: Children, Computers, and Powerful Ideas*. 2nd ed. Cambridge, MA: Perseus Publishing.

Schweik, C. and English, R. (2012) *Internet Success: A Study of Open-Source Software Commons.* Cambridge, MA: The MIT Press.

Stallman, R. (2002). *Free Software, Free Society*. Lulu.com.

von Hippel, E. (2006) *Democratizing Innovation*. 1st paperback edition. Cambridge, MA: The MIT Press.

Weber, S. (2004) *The Success of Open Source*. Cambridge, MA: Harvard University Press.

Book Chapters

Fosnot, C. and Perry, R. (1996). 'Constructivism: A Psychological theory of learning'. In Fosnot's, C. ed. Constructivism: Theory, perspectives, and practice. New York: Teachers College Press.

Jonassen, D. (1992) 'Evaluating Constructivist Learning'. In Duffy, T. and Jonassen's D. ed. Constructivism and the Technology of Instruction: A Conversation. Psychology Press.

Journal Articles

Allen, E., Cartwright, R., and Reis, C. (2003) 'Production Programming in the Classroom'. *ACM SIGSCE Bulletin*, 35(1): 89-93.

Alfieri, L., Brooks, P., Aldrich, N., and Tenenbaum, H. (2011) 'Does Discovery-Based Instruction Enhance Learning?'. *Journal of Educational Psychology*, 103(1): 1-18

Aragon, C., Poon, S. et al. (2009) 'A Tale of Two Online Communities: Fostering Collaboration and Creativity in Scientists and Children'. *Proceedings of the seventh ACM Conference on Creativity and Cognition*: 9-18.

Bezroukov, N. (1999) 'Open source software development as a special type of academic research (a criticism of vulgar raymondism)'. *First Monday* [Internet], 4(10). Available from: http://journals.uic.edu/ojs/index.php/fm/article/view/696/606. [Accessed 18 January 2014].

Brown, J.S. and Adler, R. (2008) 'Minds on Fire: Open Education, the Long Tail, and Learning 2.0'. *Educause Review*. 43(1): 16-20.

Carrington, D. and Kim, S. (2003) 'Teaching software design with open source software'. *Frontiers in Education, 2003*. *FIE 2003 33rd Annual*. 3: 9-14.

Cooper, S. (2010) 'The Design of Alice'. *ACM Transactions on Computing Education*. 10(4).

Fincher, S., Cooper, S., Kolling, M., and Maloney, J. (2010) 'Comparing Alice, Greenfoot & Scratch'. *Proceedings of the 41st ACM technical symposium on Computer science education*: 192-193.

Fitzgerald, B. (2006) 'The transformation of open source software'. *Mis Quarterly* (2006): 587-598.

Fung, K.H., Aurum, A. and Tang, D. (2012) 'Social Forking in Open Source Software: An Empirical Study'. *CEUR Workshop Proceedings* 855: Paper 6.

German, M. D. (2005) 'Experiences teaching a graduate course in Open Source Software Engineering.' *Proceedings of the first International Conference on Open Source Systems*.

Glass, R. (2003) 'A Sociopolitical Look at Open Source'. *Communications of the ACM* , 46(11): 21-23.

Hemetsberger, A. and Reinhardt, C. (2006) 'Learning and Knowledge-building in Open-source Communities'. *Management Learning*, 37(2): 187-214.

Hill, B. and Monroy-Hernandez, A. (2013) 'The Cost of Collaboration for Code and Art: Evidence from a Remixing Community'. *Proceedings of the 2013 conference on Computer supported cooperative work (CSCW '13)*, pp.1035-1046.

Hmelo-Silver, C., Duncan, R. and Chinn, C. (2007) 'Scaffolding and Achievement in Problem-Based and Inquiry Learning: A Response to Kirschner, Sweller, and Clark (2006)'. *Educational Psychologist*, 42(2): 99-107

Jonassen, David, et al. (1995) 'Constructivism and compute—mediated communication in distance education." *American journal of distance education*, 9(2): 7-26.

Kelleher, C. and Pausch, R. (2005) 'Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers'. *ACM Computing Surveys*, 37(2): 83-137

Kirschner, P., Sweller, J. and Clark, R. (2006) 'Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching'. *Educational Psychologist*, 41(2): 75-86.

Kogut, B. and Metiu, A. (2001) 'Open-Source Software Development and Distributed Innovation'. *Oxford Review of Economic Policy,* 17(2): 248-264

Kolling, M. (2010) 'The Greenfoot Programming Environment'. *ACM Transactions on Computing Education*, 10(4): 14.

Maloney, J., Burd, L., Kafai, Y. et al. (2004) 'Scratch: A Sneak Preview'. *Creating, Connecting, and Collaborating through Computing, 2004. Proceedings. Second International Conference on.* IEEE.

Maloney, J., Resnick, M., Rusk, N. et al. (2010) 'The Scratch Programming Language and Environment'. *ACM Transactions on Computing Education*, 10(4), Article 16.

Mason, R. and Cooper, G. (2013) 'Distractions in Programming Environments'. *Proceedings of the Fifteenth Australasian Computing Education Conference (ACE2013), Adelaide, Australia.*

Monroy-Hernandez, A. (2007) 'ScratchR: sharing user-generated programmable media'. *Proceedings of the 6th annual conference on Interaction design and children*. ACM.

Monroy-Hernandez, A. (2012) 'Designing for Remixing: Supporting an Online Community of Amateur Creators'. Doctoral dissertation, Massachusetts Institute of Technology.

Nyman, L. (2013) 'Freedom and forking in open source software: the MariaDB story'. *Proceedings of the 22nd Nordic Academy of Management Conference (Reykjavik, Iceland, 21-23 August, 2013).*

Pedroni, M., Bay, T., Oriol, M. and Pedroni, A. (2007) 'Open source projects in programming courses'. *Proceedings of the 38th SIGCSE technical symposium on Computer science education*, pp454-458.

Prince, M. (2004) 'Does active learning work? A review of the research'. *Journal of engineering education*, 93(3): 223-231.

Raymond, Eric. (1999) 'The cathedral and the bazaar'. *Knowledge, Technology & Policy* 12(3): 23-49.

Resnick, M., Rusk, N. and Cooke, S. (1998) 'The Computer Clubhouse: Technological Fluency in the Inner City'. *High Technology and Low-income Communities: Prospects for the Positive Use of Advanced Information Technology.*

Resnick et al. (2009) 'Scratch: Programming for All'. *Communications of the ACM*, 52(11): 60-67.

van Gog, T. and Rummel, N. (2010) 'Example-Based Learning: Integrating Cognitive and Social-Cognitive Research Perspectives'. *Educational Psychology Review* 22: 155-174.

von Krogh, G., Spaeth, S., and Lakhani, K. (2003) 'Community, joining, and specialization in open source software innovation: a case study'. *Research Policy,* 32: 1217-1241.

Magazine Articles

McMillan, R. 2012. "Lord of the Files: How GitHub Tamed Free Software (And More)". Wired http://www.wired.com/wiredenterprise/2012/02/github/all/1

Web Pages

Apache Software Foundation. (2012) /Contributing code to Apache OpenOffice/. Available at: http://openoffice.apache.org/contributing-code.html (Accessed 17 February 2014)

GitHub. (2014) GitHub. Available at: http://github.com (Accessed 26 February 2014)

Lifelong Kindergarten Group. (2014) Scratch. Available at: http://scratch.mit.edu (Accessed 22 January 2014)

Linux Kernel Organization. (2014) The Linux Kernel Archives. Available at: https://www.kernel.org/ (Accessed 26 February 2014)

Mozilla Foundation. (2014) Mozilla Developer Network. Available at: http://developer.mozilla.org (Accessed 26 February 2014)

SourceForge. (2014) SourceForge. Available at: http://sourceforge.net (Accessed 26 February 2014)