2012

Roles in a Networked Software Development Ecosystem: A Case Study in GitHub

Patrick Wagstrom
IBM TJ Watson Research Center Hawthorne, NY, pwagstro@us.ibm.com

Corey Jergensen
University of Nebraska-Lincoln, cjergens@cse.unl.edu

Anita Sarma
University of Nebraska-Lincoln, asarma@cse.unl.edu

Roles in a Networked Software Development Ecosystem: A Case Study in GitHub

Patrick Wagstrom
IBM TJ Watson Research Center
19 Skyline Dr
Hawthorne, NY, USA 10532
pwagstro@us.ibm.com

Corey Jergensen, Anita Sarma
Computer Science and Engineering Department
University of Nebraska, Lincoln
Lincoln, NE, USA 68588
{cjergens,asarma}@cse.unl.edu

ABSTRACT

Open source software development has evolved beyond single projects into complex networked ecosystems of projects that share portions of their code, social norms, and developer communities. This networked nature allows developers moving into a new project to easily leverage knowledge about process and social norms along with reputation gained in related projects. In this paper we examine a subset of the communities found in GitHub, a large software development community that focuses on "social coding". We identify a variety of roles in the ecosystem that go beyond the previous user/developer dichotomy and find that these roles often persist across sub-communities in the GitHub ecosystem. This has dramatic implications for the way that we view open source and related software development processes and suggests that a more nuanced view of the roles and relationships in these communities would be beneficial.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management: *programming teams*: D.2.8 [**Software Engineering**]: Metrics: *process metrics*

General Terms

Management, Measurement, Documentation, Human Factors,

Keywords

GitHub, open source software, networked project ecosystem

1. INTRODUCTION

Open source software (OSS) development has spread from being employed in a niche infrastructure project, such as the Apache web server, to being a standard methodology to develop almost any piece of software for which the source code can be shared [9, 20, 21]. New project hosting sites such as GitHub, which brands itself as facilitating "social coding" are changing the way open source is perceived and how it is practiced. Rather than projects being developed in isolation and reputation accrued in individual projects that culminates in the right to directly commit code to a single project, this new style of development relies on an inherently networked ecosystem where developers and users can view and track each other's contributions across a wide variety of projects[5, 16].

This networked ecosystem reflects current software development needs where a project often includes source code written in multiple languages and utilizing multiple different development frameworks and libraries, For example, development of a web application may use the JavaScript library jQuery for the user interaction, Ruby on Rails for the backend processing, and Rack as a web server. Another project may choose to use jQuery for the user interaction, Sinatra for backend process, and Rack as a web server. This need for knowledge of multiple technologies requires users to leverage their knowledge of a wide variety of projects when contributing to an individual project. Networked ecosystems, such as GitHub, make it easy to see all the contributions of a user across all their projects and thus assess their skill.

GitHub has opened up the open source development process in radical new ways. Traditionally in open source projects the source code for a project was hosted in a central repository that only a handful of developers could directly access. Changes to the code had to be mediated through central community members. Individuals had to undergo a complicated and involved socialization process, whereby they began learning about the project norms, culture, and technical content by progressively participating in social to technical roles (e.g., progressing from mailing list participation to reporting bugs and providing patches, to gaining commit access [7, 13]).

However, in GitHub the technical barriers have been reduced vastly. For example, *forking* of the code, the process by which an individual starts a new source code repository, rarely happened and was typically seen as a last resort for dramatic social or technical conflicts in a project. Distributed version control systems such as git, upon which GitHub is based, allow anyone to create lightweight forks and immediately begin developing code. When the new code is mature the user can issue a *pull request* to have the main code repository pull their code in or they can choose to easily maintain their own external branch. This has radically changed the socialization process in open source by tearing down the barriers to entry for writing source code.

GitHub also greatly simplifies the process of starting a new project by providing a common, efficient infrastructure. Developers can create any number of open source projects with only a few mouse clicks, and, in contrast to previous hosting environments such as SourceForge and Google Code that were built primarily on top of existing non-integrated tools, such as CVS, Subversion, and Mailman, GitHub provides a robust integrated environment built and architected for collaboration from the ground up. This lack of integration in previous tools resulted in information silos around projects where it was not possible to see an individual's development and social contributions beyond a single specific project.

Finally, the social media aspects in GitHub allow developers to *watch* repositories of interest or *follow* developers whose coding style or expertise they admire. This brings awareness of activities in the community and greatly improves the socialization process.

For example, the number of followers that a user has is treated as a signal of status and some members with a large follower network are treated as local celebrities [5].

In this new model the traditional roles through which developers became code contributors may not hold true. Thus far there is no research that has empirically looked into how the traditional joining script has evolved to fit this new model of open source combined with social media. The immense popularity of GitHub with more than 1.4 million registered users and 2.3 million code repositories[1] could be attributed to the social infrastructure afforded by GitHub. It is vital to understand and characterize this new networked infrastructure that promotes "social coding" as it has the potential to change the OSS landscape, as well affect traditional organization structures. Together these issues prompt the following research questions:

RQ1: How has the basic user/developer dichotomy in an open source project evolved with the addition of social data and more robust tracking of code contributions?

In traditional OSS, it was not possible to discern between users who when simply using the software found a bug and reported it or fixed a bug related to their work and submitted a patch, and users who were trying to onboard by actively tracking bugs and contributing patches. In GitHub we can easily distinguish between these groups, which can help projects in socializing newcomers or in actively recruiting patch-fixers who would otherwise drift way.

RQ2: What are the more nuanced roles of developers in a modern networked open source project?

The visible cues of others' development patterns and social networks, along with the reduced technical hurdles in participation has allowed the creation of more nuanced roles in GitHub than the traditionally accepted user/developer dichotomy in open source. Because of the availability of a common socio-technical infrastructure, developers who specialize in certain activities can work in concert to create a better ecosystem. An understanding of these roles and their effects can help projects attract these specialists.

RQ3: With the addition of social data and tracking of relationships across an entire ecosystem, how has the understanding of participation across open source projects evolved?

Our current understanding of OSS stems from research that has studied single, monolithic projects. However, the reality is that projects do not exist in isolation; rather sub-communities exist around related technology that may be founded around a common programming language, such as RubyForge[2] – a general hosting community for tools written in Ruby, or a problem space, such as the COIN-OR[3] community that provides a variety of software packages for operations research. The common social and technical infrastructure provided by GitHub further extends this by being a general hosting site for all open source, while still providing an infrastructure to develop sub-communities, which we study by investigating the overlap of roles across projects in sub-communities.

The rest of the paper is organized as follows. In section 2 we discuss the research on joining traditional, independent, open source projects. In section 3 we discuss how interconnected project hosting sites, such as GitHub, function and what data we collected for our analysis. In section 4 puts forth the various roles, both those related to development maturity and specialized roles, that a user can assume in an interconnected project hosting site and presents a description of how these roles provide additional insight and what can be done to identify these roles. Next, in section 5 we present our analysis of the prevalence of our defined development and how those roles overlap with each other, both within a single project and also across projects. We close the paper with a discussion of our findings in section 6, we address the threats to validity in section 7, and finally lay out future implications and research in section 8.

2. ONBOARDING IN TRADITIONAL OSS

Prior research identified a variety of barriers that newcomers face in the course of immigrating to a new software project [6]. Open source projects, with their decentralized nature and frequent lack of formal roles poses additional challenges [8, 10]. Open source projects typically lack formal mentoring and training for newcomers and it is the responsibility of new project participants to learn about the social and technical norms of the project and identify the appropriate technical tasks and begin contributing [7, 13]. Newcomers are rarely directed to technical tasks or exemplars. For example in an analysis of the Freenet project, von Krogh et al. found that only 1 in 6 newcomers were given specific technical tasks to work on [13]. Instead, a majority of newcomers were given general encouragement when they expressed an interest in joining the community through the mailing list, but otherwise were left on their own to find a proper way to contribute to the project. This was sometimes confounded by the fact that, irrespective of the depth of technical knowledge that a user may possess, making significant technical contributions to a community requires social standing and identity in the community. In most projects, commit access is only given after a newcomer has proved their worth and potential to the active community members; a process that limits the overall potential contributions of newcomers to the project [3, 13].

The peculiarities of the process through which newcomers become code contributors have been studied by many researchers [4, 11]. The most common model, often called the Onion Model, postulates that members in an Open Source community evolve through different roles ranging from peripheral users to core contributors and these roles can be arranged as concentric layers – similar to an onion. More specifically, the following roles have been suggested (progressing from most central and most technical layer to outer layers that are the least technical): project leader, core developer, active developer, bug fixer, bug reporter, documenters, users (active in mail messages), and peripheral user.

von Krogh et al. proposed a slight variation of this model in a qualitative study of the transition of roles in Open Source where they proposed the concept of a "joining script" for new developers joining a community [13]. Members were categorized into three broad groups: *joiners* are members who are active only in mailing lists, *newcomers* are members who have just gained commit access, and *developers* are active members with commit access who have shown strength of contributions and a technical ability. The joiners, who are potential future developers, begin by joining project mailing lists that allow them to converse about the project and learn some of the socio-technical norms and capabilities of the project. As they participate they learn how to properly participate in the community by submitting bugs, triaging bugs, and eventually working to track down the technical details of

[1] As of March 2012
[2] http://www.rubyforge.org/
[3] http://www.coin-or.org/

bugs by submitting small patches. At this point all code contributions from a new developer must be offered through another developer through a patch. After a *joiner* has shown competence with managing bugs they may be offered the ability to become a committer (*newcomer*) to a project, which allows them to directly modify the project source code without the need of an intermediary. After an intermediary trial period *newcomers* are considered to have transitioned to *developer*, if no major concerns were raised. After transitioning to the developer role they are often free to improve the project in whatever area their skills are most applicable. In this way a user is viewed a moving through different layers toward the core of the onion.

A consistent finding is that members near the center of the model exert more influence over the technical decisions of the project and other factors affecting the community than those on the periphery [1, 7, 14]. Another consistent finding is that projects that behave similar to the onion model are frequently meritocratic. As members make more frequent and important contributions to the project code they move toward the center of the project which gives them a larger voice in the direction of the project, including the process of bringing a *joiner* into a project, overseeing them as they become a *newcomer*, and eventually a *developer* [22].

However, the above studies have largely studied standalone individual projects. Our earlier work that analyzed six projects in the GNOME desktop ecosystem found the onion model to not hold true at the individual project level [12]. Instead we found evidence of developers being socialized at the ecosystem level. That is we found developers to directly start contributing in technical medium (bug patches or code) in a project, however, when we expanded our focus to track their work across other projects in the ecosystem we saw that they had participated in the social media in other projects. This implied that the common technical infrastructure provided by GNOME allowed developers to transfer their knowledge across projects. Further, we found that tenure did not correlate with the centrality of code contribution, instead, we found the longer tenured developers to take on project management roles

3. METHODOLOGY AND DATA SOURCE

In order to address our research questions, we examined the roles and contributions of users in a set of ten projects in GitHub, a large, open source software hosting site. We selected GitHub because it provides a common scaffolding of "social coding" tools and a common technical infrastructure for multitude of projects. GitHub is especially suited to allow the growth of communities since it makes user identities, project artifacts, and actions on the artifacts publicly visible across the site. This increases the social bonds of the community and success of projects [21].

3.1 GitHub Background

GitHub allows developers to create profiles that include their name, email address, organization, location, webpage, and optionally a gravatar – a globally recognized and consistent avatar image that can be associated with many different websites. These developer profiles serve both social and technical functions. From a social perspective users of GitHub can choose to follow other users and watch projects of interest. Information about recent activity of followed users and watched projects appears in a dashboard when users visit GitHub. As of March 2012 there are over 1.4 million user profiles on GitHub.

GitHub allows all users to create unlimited open source code repositories managed using the git version control system and additional private repositories for a monthly fee. As of March 2012 there are over 2.3 million code repositories hosted on GitHub. These repositories span a variety of different purposes: development frameworks critical for the next generation of web applications, clusters of code around a particular type of technology such as graph databases, mirrors of popular projects from established organizations like the Apache foundation and Linux kernel, and small scale repositories serving the needs of independent developers, among other purposes. Each repository in GitHub by default has a wiki, issue tracker, and a system for managing pull requests from other users who wish to contribute code for the project.

When creating a project repository in GitHub the site suggests one of two possible socio-technical collaboration architectures. The first architecture closely mirrors traditional open source development patterns. Individuals who are trusted to commit to a project are given direct access to add code to the central code repository. Other developers who wish to make contributions must go through a socialization process to contact these developers and get their patches accepted by the project. The second architecture, which takes advantage of distributed development patterns enabled by git and the infrastructure provided by GitHub, has developers who wish to contribute code first *fork* the project source code repository and then create *pull requests* that are managed through GitHub. This makes all of the potential changes readily apparent and makes it easy to manage pull requests.

Developers can communicate around code-related actions by commenting on a commit, an issue, or a pull request. The site also allows subscription actions that include *following* and *watching*. Developers can *follow* other developers and *watch* projects. Developers can also subscribe to be notified when a new issue (anything else?) is created, therefore, getting notified of ongoing actions in their projects and other projects of interest.

3.2 Project Selection

Our project selection was based on finding successful communities for which we could obtain deep knowledge about the underlying socio-technical practices of the community.

The first set of projects use Ruby as the underlying language and are therefore form a part of a sub-community

- **Rails:** An extremely popular full stack web development framework for creating web applications using a model-view-controller architecture in Ruby. Rails has been in development since 2004 and was one of the first high profile projects to move to GitHub.
- **Sinatra:** A web application library and lightweight domain specific language for quickly developing web applications. Sinatra is often viewed as an alternative to full stack frameworks in that it strives to be both minimal and more flexible than other Ruby based web development frameworks.
- **Rack:** A modular library for building complex web applications in Ruby. It provides a standard interface for accessing HTTP requests and responses and is used by many Ruby based web frameworks, including Rails and Sinatra.

In contrast to the large communities around the Ruby based projects, we selected a group of related projects in an exciting domain that is increasing in prominence, NoSQL databases – specifically graph databases. Tinkerpop is a loosely organized virtual organization that develops open source tools for accessing databases that store their data as a graph rather than a more traditional relational database that uses tables. All tools from Tinkerpop are written in languages that run on the Java virtual machine, such as Java, Groovy, and Scala. What is interesting about Tinkerpop is that the projects form a stack of tools that interact with graph databases, with Blueprints at the base and Gremlin, Rexster, and Frames at the top. Pipes is sandwiched in the middle as an intermediary tool that is most often accessed through Gremlin.

- **Blueprints**: A graph database agnostic property graph framework that provides a consistent API across graph database systems. This is similar to what JDBC does for relational databases on the JVM.
- **Pipes**: A dataflow framework built on top of Blueprints for performing complex graph traversals.
- **Gremlin**: A domain specific language that is an extension of Groovy, Scala, or Java that allows data scientists to easily construct graph traversal queries. Gremlin is built on top of Pipes.
- **Rexster:** A multi-faceted tool that exposes any Blueprints enabled graph as a REST enabled web service.
- **Frames:** A property mapping framework that allows a developer to easily map Java objects to graph objects through Blueprints.

Finally, we selected three projects that are semi-autonomous. These projects were selected because they were among the most watched projects on GitHub at the time of our research:

- **Jekyll**: A tool for creating static HTML websites from a set of templates and data. Jekyll is used by GitHub to create static webpages for GitHub hosted projects.
- **Resque**: A Ruby based library for creating and managing tasks that run in the background. Resque is used internally by GitHub which contributes to its popularity on the site.
- **Homebrew**: A software package manager for Mac OS X that makes it easy for developers to compile and install a large number of software packages. Homebrew, as an artifact of the process it uses and the code structure, is the most forked project on GitHub.

3.3 Data Collection

The data were collected using a custom designed set of tools that interfaced with the GitHub public APIs. The code for the project is freely available and, naturally, published on GitHub[4]. There are three major data sources that are used: information about project repositories, information about GitHub users, and project source code. By pulling information about a project repository on GitHub we obtain information about socio-technical nature of the project. For each repository of interest on GitHub we retrieve the following pieces of information:

- Basic information about the project such as when it was created, primary programming language, project URLs, etc
- Identity of each individual who "watches" the project
- Identity of each individual who has contributed to the project. In this case we use GitHub's definition of "contributor" which means an individual who has code in the source code repository for the project
- Complete history of all issues filed against the project
- Complete history of all pull requests filed against the project
- List of all of the publicly available forks of the project

From the GitHub repository information it is possible to get a list of the individuals who have been active on a project. We define this as the set of all GitHub users watching the project, all contributors to the project, all users who have had any activity on an issue or pull request, and all users who have forked the project. Our tool then retrieves the following pieces of information for each individual:

- Account information about the user such as name, email address, age of account, and URLs associated with web pages for the account
- The set of GitHub users that user is following
- The set of users following that user
- The list of repositories the user owns

Finally, the GitHub repository information provides a location of where to download the complete project source code using the git version control system. This allows us to get the complete history of all changes applied to the master branch of the project. For each git repository we collect the following information on the master branch of the project repository:

- The set of all changesets applied to the master branch
- The set of all files in the master branch and their association with each changeset
- The identity of the individual who is marked as having authored and committed the code

The data are then linked together using an automated process on various different linkage points based on shared email addresses, shared gravatar ids, and explicit references of commits in issues and pull requests. These three automated linking methods allow us to associate 94% of commits in our dataset with user information obtained from the GitHub API. A majority of the commits that we cannot associate with GitHub users were done before projects had migrated to GitHub, possibly when using other version control systems, such as Subversion or CVS, which do not preserve the provenance of the code to the same degree that git does.

4. ROLES

We divided the roles that a user can assume in GitHub into two classes: Development Maturity and Specialized roles. The former tracks the progress of an individual as they become socialized into the community to become full contributors. The latter includes roles that a contributor can take depending on their commitment and interest.

4.1 Development Maturity Roles

Development maturity roles provide a finer grained method to follow a user through their participation in a project as they move from an interested lurker to a core project member. Although it is possible to skip roles in this progression, each individual occupies only a single role at a time.

- **Lurkers**: Individuals who have only taken action to monitor a project or issues related to a project. In the context of GitHub an individual can choose to "watch" a project, which is an intentional action a user takes that results in activity related to the project appearing on their dashboard when the user logs into GitHub. While many users will "watch" projects and contribute to them by writing code or filing bug reports, for a lurker the only trace of their association and interest in a project is that they have chosen to "watch" the project.

- **Issues**: Individuals who have been active on the project issue tracker, either by filing new issues or commenting on existing issues or pull requests. This role identifies individuals who participate in the project community but do not do anything with project source code.

[4] see https://github.com/pridkett/gitminer

Table 1. Prevalence of Development Maturity Roles across Communities

	Total	Lurkers		Issues		Independent		Aspiring		External		Internal	
		#	%	#	%	#	%	#	%	#	%	#	%
Rails	14075	9779	69.48	1726	12.26	1863	13.24	275	1.95	397	2.82	35	0.25
Sinatra	3359	2768	82.41	176	5.24	328	9.76	42	1.25	41	1.22	4	0.12
Rack	1261	710	56.30	197	15.62	228	18.08	56	4.44	56	4.44	14	1.11
Blueprints	266	200	75.19	20	7.52	28	10.53	3	1.13	7	2.63	8	3.01
Pipes	118	99	83.90	3	2.54	5	4.24	1	0.85	2	1.69	8	6.78
Gremlin	549	494	89.98	19	3.46	21	3.83	2	0.36	4	0.73	9	1.64
Rexster	150	112	74.67	23	15.33	6	4.00	1	0.67	0	0.00	8	5.33
Frames	37	25	67.57	1	2.70	1	2.70	0	0.00	2	5.41	8	21.62
Jekyll	5521	4393	79.57	409	7.41	641	11.61	32	0.58	43	0.78	3	0.05
Resque	3549	2718	76.58	374	10.54	344	9.690	91	2.56	19	0.54	3	0.08
Homebrew	10724	4724	44.05	2559	23.86	1594	14.86	1797	16.76	44	0.41	6	0.06

- **Independent:** Individuals who have forked the project source code repository using the GitHub infrastructure but have not issued any pull requests. These individuals may be experimenting with the technology, developing features that they haven't finished, or electing to maintain their own branch of the project source code with a set of private customizations.

- **Aspiring:** Individuals who have created pull requests that have been closed but have never had their pull requests merged. This indicates that the individual has a desire to contribute to the project, but has yet to successfully navigate the socio-technical norms necessary to get their code accepted by the community.

- **External Contributors:** Individuals who have created pull requests that were later merged into the project source code, but are not official contributors to the project or members of the organization that own the project.

- **Internal Collaborators**: Individuals who are marked as contributors to the project or are members of the organization that owns the project and have source code in the main project repository. In the traditional user/developer dichotomy model for open source participation these individuals would comprise the set of developers.

Note, that apart from the *Issues* and *Collaborators* role, we are able to discern the other roles because of the networked and social structure implemented in GitHub. This finer grained characterization of users who are not yet members can help in better understanding the socialization process and in mentoring to facilitate the process.

4.2 Specialized Roles

Not every individual chooses to contribute to a project in the same way. For example, in a mature project some developers may work on experimental features, other developers perform maintenance, and other developers may respond to bugs reported via the issue tracker. Although all of these individuals may have fulfilled the same general development maturity role their actions provide us with a much more nuanced view of the development process. In contrast to the development maturity roles, which are mutually exclusive, a single individual can occupy multiple specialized roles.

- **Prodder**: Individuals who identify and take on long standing issues or issues that have idle for a long time. Note that the common infrastructure afforded by GitHub lowers the technical barriers, which can in turn allow an individual to take on such a role. For example, in a regular project an individual would have to first create an account in the project, learn about the project through the mailing list and then identify issues, even then the members of the project might not take kindly to an outsider prodding the team into resolving an issue. The common infrastructure by GitHub allows any individual who has an account and is interested in the project to identify these issues. Further, the reputation already garnered by the individual in the community lends weight to the individual's recommendation.

  Formally, we define a prodder as an individual who is active on issues that have sat idle for more than 14 days, either by commenting, closing, or reopening an issue. We rank all individuals by the number of issues they have prodded and then take the top 20% of this set, subject to a floor than an individual must have been involved on at least 1% of the issues in a project. This is put in place to control for long-lived projects that may see thousands of individuals that periodically prod issues they're interested in.

- **Project Stewards:** Individuals who primarily focus on managing the project. They merge Pull Requests (from External contributors) into the project, comment on the Pull Requests, and close a Pull Request once it has been merged. Formally, these individuals are among the top 20% of individuals working on a project both in terms of number of issues closed and number of pull requests closed.

- **Code Warriors:** Individuals who have frequent and consistent commits to a project. We define a code warrior to be an individual who is among the 20% of individuals working on a project in terms of both the frequency of their commits and also the standard deviation of the time between their commits. These individuals reliably produce and make available new pieces of code for the project.

- **Nomad Coders:** Individuals who have contributed only minor code changes and then have either move onto the next projects or individuals who are participating in one project, but make minor contributions to another project. Similar to *Prodders*, this role would not have been possible in the absence of the common infrastructure provided by GitHub.

- **Project Rockstars**: Individuals who have a high visibility in their project and are significant contributors to their project. Similar to a code warrior, these individuals have the same contribution distribution are in the top 20% for the number of commits to a project, but in addition they are also in the top 20% in terms of number of people in the project who follow them.

Note, that a project need not have all specialized roles. Indeed, we expect that smaller projects will lack individuals in many of these roles.

Table 2. Prevalence of Specialized Roles Across Communities

	Total	Prodder		Steward		Code Warrior		Nomad		Project Rockstar	
		#	%	#	%	#	%	#	%	#	%
Rails	14075	144	1.02	102	0.72	45	0.32	76	0.54	73	0.52
Sinatra	3359	7	0.21	18	0.54	5	0.15	28	0.83	10	0.30
Rack	1261	15	1.19	23	1.82	4	0.32	44	3.49	7	0.56
Blueprints	266	2	0.75	2	0.75	1	0.38	1	0.38	3	1.13
Pipes	118	1	0.85	3	2.54	1	0.85	1	0.85	2	1.69
Gremlin	549	2	0.36	1	0.18	0	0.00	2	0.36	1	0.18
Rexster	150	3	2.00	0	0.00	0	0.00	0	0.00	2	1.33
Frames	37	1	2.70	3	8.11	1	2.70	1	2.70	1	2.70
Jekyll	5521	51	0.92	20	0.36	2	0.04	24	0.43	5	0.09
Resque	3549	31	0.87	15	0.42	4	0.11	24	0.68	5	0.14
Homebrew	10724	208	1.94	101	0.94	38	0.35	65	0.61	82	0.76

5. ANALYSIS

The final set of data surrounding the communities was collected using the GitHub over a course of four days in February 2012. Although numerous elements of data provided through the GitHub API provide timestamps, some critical elements, notably the dates in which an individual first watches a project or follows another individual are not present. Therefore with this data we are able to address our research questions at the current moment in time. Using this data we are able to characterize GitHub, in terms of the roles that users may take in the subset of our selected projects and its user community to answer questions such as: What are the different roles that users occupy? Do they take on multiple roles? Do users participate in multiple projects? If yes, then do they take on same roles or do they perform different actions in different projects? We answer the above questions by first empirically characterizing the different roles that can occur in GitHub, followed by an analysis of the roles of individuals in our subset of projects.

5.1 Development Maturity Roles

The prevalence of each of the defined development maturity roles across each of the projects can be seen in Table 1. Most striking is the large number of users who can be termed “Interested Lurkers”, that is, individuals who have shown an interest in the project by “watching” the project. We note that lurkers range from about 90% (Gremlin) to 44% (Homebrew) of all individuals affiliated with a community. This large number of lurkers is possibly a result of the fact that GitHub makes it so easy to register an interest in a project. It is likely that individuals may not follow activities of every repository that they are watching, but these large numbers show that members like to be aware of ongoing project activities.

Although the number of lurkers may seem very high for projects, in most cases it indicates there is a large enough population to support the other roles in the project. In the onion model the second level of participation after an interested lurker or an individual active on a mailing list, was individuals active on project bug trackers. The number of individuals active only on issues in the project ranged from approximately 3% (Frames) to 24% (Homebrew). In Homebrew this large number of individuals active on issues is probably related to the relatively non-coupled nature of the project. Most issues filed with homebrew tend to be related to packaging scripts being broken for a different software package. Therefore, their interest may not reside as much in Homebrew as it does in the variety of other software packages that Homebrew interacts with.

Another highly interesting result is the number of individuals involved in roles that were not adequately captured by the traditional user/developer dichotomy of open source software development. Between 4 and 18% of the users involved in each project are classified as individual developers – each maintaining their own branch of the project source code without having ever even requested that their code be merged into the project. What is even more surprising is that except for two projects (Rails and Pipes), the number of independents is higher than the number of users who are contributing to the project (summing Externals and Internals). This shows that a significant number of users have created their own forks and have made changes. The lightweight forking process afforded by GitHub probably leads to this high number of Independents in the project. Nevertheless, our results show that there is a large untapped potential that can be easily leveraged by projects.

When we investigate *aspiring* developers, we find that although their overall percentages are small, numerous users have attempted to have their code merged into a project but have not yet been successful in their attempts. In a large project, such as Rails, even the 2% of users who are classified as aspiring developers still adds up to 275 potential new developers. On the small projects that make up the Tinkerpop community, most which are reliant on only a handful of developers, there are numerous individuals who fall into the aspiring role, providing a pool of possible future contributors. In six of our projects, the number of these individuals is higher than the Internal contributors. This suggests that there is a large body of untapped potential that can be leveraged if these individuals are socialized into the project.

Meanwhile, the community around the Homebrew project, which encourages massive amounts of forking for facilitating largely parallel work, includes nearly 1700 individuals who fall into the aspiring category. Note that the project as compared to the contributors (Internal and external), the community has 16 times as many aspiring users. However, this is an artifact of the workflow of the project that often includes developers who merge in pull requests without actually closing the pull requests. Furthermore, most contributions to Homebrew are small independent snippets of code that allow Homebrew to build and package very specific software packages. Recent changes to the Homebrew tool have given it the ability to automatically pull in code from pull requests when the code is not present in the main repository. This further limits the necessity of developers to merge code into the main repository.

Finally, we find evidence of projects following two different workflows. Homebrew relies almost exclusively on issuing pull requests to provide new code for the project. In contrast, while the Tinkerpop stack allows pull requests, almost all of the code comes from formal project contributors. Most of the other projects are some combination of pull requests and commits from core project members. Future research into the nature of the commits by core project members versus those that arrive as pull requests could provide a great amount of insight into the open source software development process.

Our investigations show that the analysis of a hosting ecosystem such as GitHub allows us to easily identify hundreds of individuals who would have previously been left out in the user/developer dichotomy of open source development, which can help us better understand the open source ecosystem. This suggests that with respect to our first research question, it is clear that there has been significant evolution of the user/developer dichotomy in modern open source projects.

5.2 Specialized Roles

Our second research question sought to expand the traditionally mutually exclusive definitions of user/developer and automatically identify specialized roles in an open source project. As would be expected, the number of individuals who occupy specialized roles in a project is strongly correlated with the total number of individuals who contribute to a project. This is particularly clear as Rails and Homebrew, the two largest projects in the ecosystem in terms of individuals participating in the project, also have the highest number of individuals in specialized roles.

Approximately 2% of the individuals participating in the Homebrew project are prodders. This is a huge number of people (208 individuals) who go back and perform actions on issues and pull requests that have sat dormant for weeks. This is partially a result of the process adopted around Homebrew – which allows individuals to make small contributions of nearly completely independent code and share the code with a pull request. It is natural with so many individuals filing issues within the project that a substantial number of these individuals would also go back and raise activity on issues that they had filed, but which have not been resolved yet. While it is generally desirable to have individuals revisit dormant issues and pull requests, the volume of individuals prodding issues within the Homebrew project may prove troublesome for long-term project management.

Somewhat surprising was the number of project rockstars working with Rack. These individuals not only contribute substantial amounts of code, but also are followed by many people in the community. Although as a percentage it is comparable to Rails and Homebrew, it is far greater than the other midsized projects of Sinatra, Resque, and Jekyll. We haven't been able to discover a definitive answer to why this is the case; one hypothesis extended by a member of the Rack community was that it had to deal with the infrastructure nature of Rack. Because Rack is a core component upon which many tools depend there are many users who follow it, but this core nature of the project also means that there is a higher barrier for the quality of code that is added to Rack as defects could have cascading effects on numerous other projects.

Another interesting observation is that for all projects in our study (except Frames, which has only one individual in majority of the projects) the percentages of individuals who are project rockstars outnumber the code warriors. This could be indicative of the highly social nature of development in GitHub, as the rockstar role is half social and half technical. It could also be a reflection of the fact that project leaders, those that one would expect to have the most followers, often need to engage in variety of behaviors that detract from their ability to write code. For example, leaders of open source projects are desirable speakers at conferences or they may be hired on as consultants to corporations using the technology. Both of these results make it more difficult for developers to deliver code on a reliable and predictable schedule, which is what the code warrior role identifies.

The existence of nomad coders shows that the common infrastructure provided by GitHub allows users to make contributions across a set of projects. These nomads could be a result of independent developers (development maturity role) who had forked and made changes contributing those changes to the project, but not really participated in the community.

Perhaps most interesting was the fact that some projects had no individuals who fulfilled some of the specialized roles on the project, not even when we consider the project leads. In particular, no stewards, code warriors, or nomads were identified in the Rexster project. While Rexster is a healthy project at the moment, a single developer performs most tasks for the project. When this developer has other engagements the activity on the project drops off significantly. This is particularly worrisome for the long-term health of the Rexster project and any individual who wishes to use the project as a critical component in a software solution.

In summary, although the specialized roles were not common in the community, they do exist. As per our definition, these specialized roles are taken up those by users who are very active with development and we expected these numbers to be low. Our results are in line with the overall "law of the vital few" principle that governs contributions in online communities. That is, it is typical for a small minority to produce the majority of work and has been observed in open source development [18].

5.3 Overloading Specialized Roles

The evidence of existence of specialized roles in our projects prompted investigation of the degree to which individuals fulfill multiple specialized roles within a given project (note that the specialized roles are not mutually exclusive). This provides a deeper insight into the distribution of work among project participants and creates a deeper understanding of the project health and future growth prospects.

To investigate the extent of the roles overlap, we focus only on the larger projects in our study – those that collective fall into the space of web frameworks for Ruby: Rails, Sinatra, and Rack. The commonalities of project space and programming language between these three projects should make comparisons easier. The overlap between specialty roles for Rails, Sinatra, and Rack are show in Table 3, Table 4, and Table 5. We specifically exclude projects in the Tinkerpop stack, Jekyll, and Resque from our analysis, because of the limited number of individuals who fulfilled specialized roles in those communities. Likewise we exclude Homebrew because of its unique process.

Table 3: Overlap of Specialized Roles in Rails

	Rockstar	Steward	Prodder	Code Warrior	Nomad
Rockstar	73				
Steward	9	102			
Prodder	18	17	144		
Code Warrior	5	2	4	45	
Nomad	0	13	1	0	76

Note that while the number of individuals who occupy multiple roles seems low (individuals who occupy the role of rockstars and stewards (9) or prodders (18), code warriors (5) in Rails, Table 3), these actually represent a significant fraction of individuals occupying multiple roles – about 43% of rockstars in Rails also execute other roles. Understanding the details of this overlap is critical for managing a large-scale software project.

Table 4: Overlap of Specialized Roles in Sinatra

	Rockstar	Steward	Prodder	Code Warrior	Nomad
Rockstar	10				
Steward	2	18			
Prodder	2	1	7		
Code Warrior	2	0	0	5	
Nomad	0	1	0	0	28

Table 5: Overlap of Specialized Roles in Rack

	Rockstar	Steward	Prodder	Code Warrior	Nomad
Rockstar	7				
Steward	3	23			
Prodder	3	6	15		
Code Warrior	1	0	1	4	
Nomad	0	5	0	0	44

Beyond sheer existence of specialized role overlap, the pair-wise combinations of roles can yield valuable insight. First, as would be expected from the definitions of the roles, it is impossible for an individual to be a Nomad as well as a rockstar or code warrior. This is due to the fact that the nomad role expressly requires a small amount of commits to a project while rockstars and code warriors require numerous commits to a project.

The role that has the most overlap with other roles is the project rockstar. The rockstar role, which is based on a combination of social factors, the number of individuals that follow a given individual, and technical factors, the number of commits made to a repository, is naturally related the code warrior, which identifies individuals who frequently commit to a project. Our results indicate that only a handful of code warriors have the high social following to be rockstars. This shows that social visibility and prominence arises not just the amount of contributions, but also the type of commits made or the files that one changes.

However, it is surprising that it is more common for an individual to be a rockstar/prodder or rockstar/steward combination than a rockstar/code warrior. This may be indicative of the fact that individuals who are active on issue and pull requests, which is a direct and obvious way of interacting with community participants, influences social status. A positive interaction on an issue or pull request may make the issue submitter to be more likely to follow the project member who handled that issue. Alternatively, a team working in an agile manner and using an issue tracker for work items would have similar role combinations [17]. Upon examination, we did not find evidence of this practice in our data, instead issue trackers were primarily used to report bugs and not track new features or changes to the architecture.

We found evidence that there are some nomads who also serve as stewards on a project. This role combination refers to those individuals who close many issues and handle many pull requests on a project, but do little in terms of actually writing new code for the project. For a large project, an individual with a moderate technical background can contribute to the project by vetting contributions from other users and handling issues and on some occasions delivering code to a project repository.

Note that we could not have identified such combination of roles in the traditional centralized versioning system (e.g. CVS and Subversion) that lacked the robust provenance of code contributions made visible in git. We also note that the majority of participants in the projects hold single specialized roles, with a minority who serve multiple roles. This might be evidence of the law of the vital few, where there is a small core group that takes on multiple roles and are critical to the project, a phenomenon seen in other open source projects [18].

5.4 Overlapping Roles across Projects

For our final analysis we sought to understand how roles are For our final analysis we sought to understand the similarities and differences in roles taken by individuals across projects in a well-connected software ecosystem. In such an ecosystem, we would expect to see some overlap in development maturity roles as individuals are able to leverage their knowledge of the socio-technical processes surrounding one project to participate in other projects in the ecosystem.

We focus on the five projects within the Tinkerpop stack because there are clear relations between the projects (i.e. all projects build on Blueprints), and they all have similar socio-technical norms. For each project, we collected the set of individuals in each development maturity role and compared these sets across projects within the Tinkerpop stack to generate an overlap matrix. For example, Table 6(a) shows that there were 65 individuals in both the Blueprints and Gremlin communities, while Table 6(b) shows there were 45 individuals who were lurkers on both Gremlin and Rexster. The diagonal shows the total number of individuals in each role for each project within the stack.

Table 6: Overlap of Users in the Tinkerpop Community. (a/left) Total Users (b/right) Lurkers

	Blueprints	Pipes	Gremlin	Rexster	Frames	Blueprints	Pipes	Gremlin	Rexster	Frames
Blueprints	266					200				
Pipes	65	118				40	99			
Gremlin	129	70	549			88	48	494		
Rexster	63	42	69	150		38	24	45	112	
Frames	33	24	29	23	37	17	11	17	9	25

There were significant overlaps in the participation of projects inside of the Tinkerpop stack, as shown in Table 6(a). Blueprints, the foundation of stack, had the most overlap with other projects. While the smallest, Frames, had at least 23 out its 38 community members associated with other projects in the Tinkerpop stack. Further analysis showed there were 18 individuals who were in some way associated with all five projects in the stack, although 8 of these individuals were lurkers on all five projects. Table 6(b) shows that the majority of the population overlaps between projects comprised individuals who were lurkers in both projects. For example, in the case of Gremlin, the majority of individuals that participated in Gremlin and another project in the Tinkerpop stack were lurkers in both projects. This shows there is a large population of individuals interested in multiple projects who have registered interest and are tracking its changes, but not interested enough to contribute.

Table 7: Overlap of Users in the Tinkerpop Community. (a/left) Independent (b/right) External Contributor

	Blueprints	Pipes	Gremlin	Rexster	Frames		Blueprints	Pipes	Gremlin	Rexster	Frames
Blueprints	28						7				
Pipes	1	5					1	2			
Gremlin	2	0	21				1	1	4		
Rexster	1	0	0	6			0	0	0	0	
Frames	1	0	0	0	1		1	1	1	0	2

Despite the fact that Blueprints and Gremlin had significant numbers of independent developers, we found little evidence of individual developers being active on multiple projects, as shown in Table 7(a). This is somewhat surprising given that higher level components in the Tinkerpop stack, such as Gremlin and Rexster, build on Blueprints and Pipes, it is likely that an individual using these higher level projects would have some of the technical knowledge and desire to modify the lower level projects.

Somewhat surprisingly within the set of projects that make up the No individuals were independent developers on more than two projects. Additional analysis showed that of the 56 *independent* developers across all projects in the Tinkerpop stack only 5 were active in the more advanced development maturity roles in another project within the stack. Similarly there is little overlap of the external contributor role across projects in the stack, as shown in Table 7(b). No user was an external contributor on more than one project in the Tinkerpop community. Our results indicate that although the projects in the Tinkerpop community are related, most projects have their own core group. This could largely be an artifact of the small size of the projects.

Somewhat surprisingly there were no users that were aspiring developers on multiple projects within the Tinkerpop stack (table not shown because all cells are zeros), which might indicate that developers did not attempt to contribute to multiple projects simultaneously. Five of the seven aspiring developers in the community had no accepted code in any project in the ecosystem, while two were external contributors to other projects (Pipes and Rexster) in the stack. We investigated these two developers further. We found that the aspiring developer on Pipes was well regarded in the community as a designer of additional tools and libraries that interfaced with the Tinkerpop stack. He was also an external contributor to Blueprints. The aspiring developer for Rexster was an external contributor on Blueprints, Pipes, and Gremlin. This indicates that there may be a relationship between the progression from aspiring developer to external contributor. An alternative explanation could be that developers have their "home" project and because of the contribution policies in GitHub, they participate in other projects as external contributors.

The overlap of the specialized roles between projects in the Tinkerpop stack is not shown because of the low number of individuals filling these roles. There was some overlap, particularly with the role of rockstar, code warrior, steward and prodder on the blueprints, pipes, and gremlin projects. This is due to the fact that a single individual that fulfilled all four roles undertakes much of the work in Tinkerpop stack. This creates a strong concern for the long-term success of the stack if this individual, who is the founder of the stack, were removed.

6. DISCUSSION

In this work we have distinguished the different stages through which a user progresses as they become more involved in the community at a fine-grained level. This classification includes six stages, starting from registering interest (lurkers) to being a part of the organization (internal contributor). Roles such as independents – individuals who have created a fork and have worked on it privately, aspiring – individuals who have submitted pull requests which have not been accepted yet, and external contributors – individuals who contribute to the project via pull requests that need to be merged by a member of the organization are new roles that are visible because of the way git tracks the provenance of changes. These roles allow us to break away from the current user/developer dichotomy view of open source development. This fine-grained view of the development maturity model can help projects adapt their socialization process to target different contributor types. For example, *independent* developers already have made changes to their code base and might need a different socialization process as compared to *aspiring* developers.

Our analysis showed that the majority of users in the projects we analyzed were lurkers. While most online communities have a large majority of lurkers [15, 19], we believe that in our case developers are interested in being aware of changes in these projects because they have registered their interest by "watching" the project. When we investigated a subset of related projects in the Tinkerpop stack to identify the extent of overlap of developers and their roles across projects, we noted a similar trend in the case of lurkers; most people involved in multiple Tinkerpop projects were lurkers.

We were surprised to find little overlap among aspiring, independent, or external developers across projects in Tinkerpop. We believe this is because Tinkerpop is a relatively small project ecosystem and that individuals largely focus on their projects. Further investigations showed the existence of one leader who transcends project boundaries and keeps the community together.

Our findings provide evidence of the existence of specialized roles in our subset of projects, albeit in small numbers. This is inline with the "law of the vital few", which dictates that the majority of contributions come from a small core group. We note that within a project users can assume multiple roles. The most common combination of specialized roles was between rockstars and prodders, stewards, or code warriors. This trend shows that the social visibility (number of followers) is accrued to individuals who are active on issues and pull requests as they interact with the community. When we investigate the overlap of the specialized roles across the projects in Tinkerpop we found no overlap of specialized roles in the community. Again, this trend is worrisome since the entire ecosystem depends on a single individual and has implications for the future growth of the community.

In summary, we note that the common social and technical infrastructure provided by GitHub in addition to the provenance of code changes maintained by git, allows us to create a much finer-grained characterization of open source projects, including individuals who are in the process of being socialized, as well as, more specialist roles. We also found the evidence of the existence of communities, although, our example turned out to be a very small ecosystem with very few overlapping roles. The ability to have an ecosystem of projects that uses a common social and technical infrastructure can help in the design of such ecosystems within commercial organizations.

7. THREATS TO VALIDITY

As in any empirical study, our sample and methods may not be wholly representative of all aspects of the community and problem space.

Internal: By definition the identification of nomad developers requires an investigation of developers' contributions across different projects. However, since we have investigated only a small subset of projects in GitHub, the prevalence of nomadic coders is almost certainly underestimated in our study. Here, we have shown that nomads do exist, which was the primary goal of our analysis, but collection of more data will serve to enhance our findings, which are a lower bound.

We also utilized automated methods to connect entities in our data and only considered the master branch of a project. This may result in some of the same caveats regarding research using artifacts from git repositories highlighted by Bird et. al [2].

Construct: Identification of rockstar, steward, prodder, and code warrior roles required us to identify top participants using a threshold. In each of these cases we applied an additional joint filter, either the requirement to be in the top 20% of two or more attributes, or a level of overall participation in the case of prodders. In the case of a joint filter, if the two distributions were independently distributed then approximately 4% of individuals would fall into a category. However, we find this not the case. Taking the example of project rockstars, which requires that an individual be in the top 20% of both the number of followers and source code contributions, we found only a handful of individuals on the smaller Tinkerpop projects and less than 1% of the individuals in the larger projects, fit these requirements. Clearly, the level of the threshold significantly alters the results, but we experimented with several different thresholds before settling on 20% - a level that follows the Pareto law (law of the vital few), a common contribution model in online communities, including open source [18].

Further, to identify prodders we chose a time span of 14 days to consider an issue old. This time span was selected through a multi-part process that involved investigation of the distribution of time between interactions on projects and qualitative examinations of some of these interactions after our periods of idleness.

External: Our subset of projects chosen might not be representative of other projects in GitHub or GitHub in total. We chose the Ruby Rails project because of the original prominence of the Ruby on Rails community in pushing git as a version control system and GitHub as a hosting platform. As an early project this provided a significant amount history, but also may show artifacts as the norms around GitHub have evolved since project creation. The tools of the Tinkerpop stack were selected because of their prominence in a field, graph databases, and also the perceived tight knit nature of the community around Tinkerpop. The participation patterns of the project may be strongly influenced by project leaders in this relative small project. The other projects were chosen because of their prominence in the GitHub ecosystem (Jekyll, Resque, and Homebrew) or because of existing relationships with another project (Rack, and by extension, Sinatra).

8. CONCLUSIONS

Open source software development has expanded from a novel, fringe development process to an established and, at least in some contexts, dominant way of developing software. In the fourteen years since the term "open source" was first defined much has changed about the makeup of the community, the process used to develop software, and the very nature of the software itself. When these dramatic changes around open source are combined with an increasingly networked and collaborative world, it becomes clear that we should revisit some of the older assumptions about how individuals work in open source projects. The large amount of data available through the APIs for collaborative software development tools allows us to build a more robust picture of the degree to which individuals participate in open source projects, and by extension, provides insight into the learning and evolution process of a new user participating in a software development project.

In this work we have shown that the traditional user/developer dichotomy of open source software development hides a broad range of different types of participation in open source projects. By understanding the level of development maturity that an individual has with a project we can better target support, training, and mentoring to better ensure that each open source project remains a viable project for many years.

Furthermore, this research provided valuable insight into the nature of the evolving open source process. Even though we chose a relatively diverse set of projects, some of which were extremely niche, such as Frames, we found that in almost all cases there was overlap between the communities of users interested and affiliated with each project. These boundaries crossed technical domains of the projects (e.g. web frameworks such as Sinatra and Rails, database access from Tinkerpop, and infrastructure from Homebrew) and also programming language. Indeed, it seems as though the nature of environments such as GitHub, which provides a relatively uniform process for individuals to collaborate on a wide variety of projects, contributes to this fact.

Above all, this research has shown that open source is still an expanding and evolving area. Tools are continually being developed that provide greater integration with not only other tools, but also the social framework that underlies open source projects. When combined with the fact that open source projects are core components in many, software development projects, these findings suggest that there is still much to learn about the roles that individuals play in open source development and how we can best ensure that these projects are successful and that individuals get the support they need to continue to grow.

This study was a small-scale study where we focused on a small set of projects for which it was possible to obtain a deep understanding of the social and technical process of the community. In future research we plan to continue to expand our research to understand the wider network of open source projects that utilize GitHub as a hosting and project management tool. We are also collecting temporal information about participation in projects so we can better understand the fine-grained nuances that surround the evolution between development maturity roles and specialized roles in open source project development.

9. ACKNOWLEDGMENTS

This research is supported by grants NSF IIS-0414698, NSF CCF-1016134, AFSOR FA9550-09-1-0129.

10. REFERENCES

[1] Bird, C., Pattison, D., D'Souza, R., Filkov, V. and Devanbu, P. 2008. Latent Social Structure in Open Source Projects. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Atlanta, Georgia, 2008), 24–35.

[2] Bird, C., Rigby, P., Barr, E., Hamilton, D., German, D. and Devanbu, P. 2009. The Promises and Perils of Mining Git. *6th IEEE International Working Conference on Mining Software Repositories* (Vancouver, BC, May. 2009).

[3] Crowston, K. and Howison, J. 2005. The Social Structure of Free and Open Source Software Development. *First Monday*. 10, 2 (Feb. 2005).

[4] Crowston, K., Wei, K., Howison, J. and Wiggins, A. 2012. Free/Libre Open Source Software Development: What We Know and What We Do Not Know. *ACM Computing Surveys*. 44, 2 (2012).

[5] Dabbish, L., Stuart, C., Tsay, J. and Herbsleb, J. 2012. Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository. *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work* (Seattle, WA, USA, 2012), 1277–1286.

[6] Dagenais, B., Ossher, H., Bellamy, R.K.E., Robillard, M.P. and Vries, J.P. de 2010. Moving Into a New Software Project Landscape. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (Cape Town, South Africa, 2010), 275–284.

[7] Ducheneaut, N. 2005. Socialization in an Open Source Software Community: A Socio-Technical Analysis. *Computer Supported Cooperative Work*. 14, 4 (2005), 323–368.

[8] Farzan, R., Dabbish, L.A., Kraut, R.E. and Postmes, T. 2011. Increasing Commitment to Online Communities by Designing for Social Presence. *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work* (Hangzhou, China, 2011), 321–330.

[9] Fogel, K. 2005. *Producing Open Source Software*. O'Reilly & Associates.

[10] Hertel, G., Niedner, S. and Hermann, S. 2003. Motivation of Software Developers in Open Source Projects: An Internet-based Survey of Contributors to the Linux Kernel. *Research Policy*. 32, 7 (Jul. 2003), 1159–1177.

[11] Jensen, C. and Scacchi, W. 2007. Role Migration and Advancement Processes in OSSD Projects: A Comparative Case Study. *Proceedings of the 29th International Conference on Software Engineering* (Minneapolis, MN, USA, May. 2007), 364–374.

[12] Jergensen, C., Sarma, A. and Wagstrom, P. 2011. The Onion Patch: Migration in Open Source Ecosystems. *Proceedings of the 8th Joint Meeting of the European Software Engineering Conference and the Foundations of Software Engineering* (Szeged, Hungary, Sep. 2011).

[13] von Krogh, G., Spaeth, S. and Lakhani, K.R. 2003. Community, Joining, and Specialization in Open Source Software Innovation: A Case Study. *Research Policy*. 32, 7 (Jul. 2003), 1217–1241.

[14] Lakhani, K. and Wolf, R. 2005. Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects. *Perspectives on Free and Open Source Software*. Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim R. Lakhani, eds. MIT Press.

[15] Lampe, C., Wash, R., Velasquez, A. and Ozkaya, E. 2010. Motivations to Participate in Online Communities. *Proceedings of the 28th International Conference on Human Factors in Computing Systems* (Atlanta, GA, USA, 2010), 1927–1936.

[16] Lerner, J. and Tirole, J. 2002. Some Simple Economics of Open Source. *Journal of Industrial Economics*. 50, 2 (Jun. 2002), 197–234.

[17] Lindvall, M., Basili, V., Boehm, B., Costa, P., Dangle, K., Shull, F., Tesoriero, R., Williams, L. and Zelkowitz, M. 2002. Empirical Findings in Agile Methods. *Extreme Programming and Agile Methods — XP/Agile Universe 2002*. Springer Berlin / Heidelberg. 81–92.

[18] Mockus, A., Fielding, R.T. and Herbsleb, J. 2000. A Case Study of Open Source Software Development: The Apache Server. *Proceedings of the 22nd International Conference on Software Engineering* (Limerick, Ireland, 2000), 263–272.

[19] Nonnecke, B. and Preece, J. 2000. Lurker Demographics: Counting the Silent. *Proceedings of the 2000 SIGCHI Conference on Human Factors in Computing Systems* (The Hauge, Netherlands, 2000), 73–80.

[20] Scacchi, W. 2007. Free/Open Source Software Development. *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (Dubrovnik, Croatia, Sep. 2007), 459–468.

[21] Tsay, J.T., Dabbish, L. and Herbsleb, J. 2012. Social Media and Success in Open Source Projects. *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work* (Seattle, WA, USA, 2012), 223–226.

[22] Ye, Y. and Kishida, K. 2003. Toward an Understanding of the Motivation of Open Source Software Developers. *Proceedings of the 25th International Conference on Software Engineering* (Portland, OR, USA, 2003), 419–429.