Entering the Circle of Trust: Developer Initiation as Committers in Open-Source Projects

Vibha Singhal Sinha
IBM Research – India
vibha.sinha@in.ibm.com

Senthil Mani
IBM Research – India
sentmani@in.ibm.com

Saurabh Sinha
IBM Research – India
saurabhsinha@in.ibm.com

ABSTRACT

The success of an open-source project depends to a large degree on the proactive and constructive participation by the developer community. An important role that developers play in a project is that of a code committer. However, code-commit privilege is typically restricted to the core group of a project. In this paper, we study the phenomenon of the induction of external developers as code committers. The trustworthiness of an external developer is one of the key factors that determines the granting of commit privileges. Therefore, we formulate different hypotheses to explain how the trust is established in practice. To investigate our hypotheses, we developed an automated approach based on mining code repositories and bug-tracking systems. We implemented the approach and performed an empirical study, using the Eclipse projects, to test the hypotheses. Our results indicate that, most frequently, developers establish trust and credibility in a project by contributing to the project in a non-committer role. Moreover, the employing organization of a developer is another factor—although a less significant one—that influences trust.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Programming teams*

General Terms

Experimentation, Human Factors, Measurement

Keywords

Open-source software, developer roles, code committer, mining code repository, mining bug repository

1. INTRODUCTION

Over the past couple of decades, open-source software has had a profound impact on how software is developed, delivered, tested, shared, and maintained. The model of open-source development relies fundamentally on leveraging the global community of developers for performing a variety of tasks such as reporting bugs, submitting feature requests, contributing patches and code, providing documentation, and performing beta testing. Needless to say, the success of an open-source project depends to a large extent on proactive and constructive participation of the external developer community.

Of the many tasks that the developer community can perform on a project, the contribution of code is an important one. Active code contribution can lower the turn-around time for bug resolution and lead to faster growth of a product with the addition of new features and novel extensions. Although code contribution by a wide community is desirable, it must be controlled with a mechanism for ensuring the quality of the code. To attain this, many projects follow some sort of governance, put in place by a "core" group or committee, which regulates how code is committed and by whom. Typically, such a governance model separates the role of *patch contributor* (which is open to all) from the role of *project committer* (which is limited to the core group). For example, the Eclipse Development Process has a formal procedure through which new project committers are elected.[1]

In the loosely regulated environment of open-source development (unlike the controlled environment of commercial software development), trustworthiness of an unknown developer is the key to letting the developer contribute code. In fact, the Eclipse policy stipulates that the committer role be restricted to ". . . [people] who have the trust of the Project's Committers . . . [through] contributing and showing discipline and good judgment."[1] However, the policy does not prescribe any specific guidelines to follow in evaluating the degree of trust of a candidate—it leaves the evaluation to the discretion of the individual projects. Thus, a pertinent question is how is trust established in practice? From the perspective of an external developer, how does one establish credibility to become a code committer? Answers to these questions can provide insights into, and improve our understanding of, what makes open-source projects tick, specifically, with respect to building a community of code committers. Such insight can also help in formulating appropriate policies for the induction of members as project committers.

In this paper, we investigate this interesting aspect of the social dynamics of open-source development—the initiation of unknown developers as code committers in a project. This phenomenon is sometimes referred to as the *joining script*, and has been studied by other researchers [2, 6, 15]. Our work builds upon this research.

The phenomenon of developer initiation as code committers can be explained in different ways. For example, the initiation may manifest as a path in which external people start as product users and report the bugs that they find. Next, they may start contributing patches for the bugs, which are committed to the product code

[1] `http://wiki.eclipse.org/Development_Resources/HOWTO/Nominating_and_Electing_a_New_Committer`

base by the core group. Over time, based on the history of successful patch submissions by an individual, the core group elects that person as a project committer. In a world of increasing social networking, knowing a person through different avenues, such as discussing technical ideas over a blog, could also establish the trust. For example, Stewart [14] observes that, very often, in open-source development, developers attain good certification in a project because of the testimony of someone who is already regarded highly in the project. Other factors that can influence trust include shared employment organization. Jensen and Scacchi [8] state that, because of the presence of corporate and non-profit organizations at the core of some open-source projects, employing full-time project members is becoming a common practice.

Specifically, in this paper, we investigate three hypotheses (stated formally in Section 2) related to the initiation of developers as code committers in an open-source project. Our goal in formulating these hypotheses is to study different factors—technical and social—that can explain the formation of trustworthiness, which is the key for the conferment of commit privileges to a developer. Our first hypothesis is intended to study the extent of non-code-commit activity that developers perform on a project before committing code directly to the product code base. Additionally, we study two other hypotheses: whether a new committer to a project has played the committer role in other (related) projects, and whether a new committer is employed by the same organization that an existing core-committee member works for. Thus, two of our hypotheses focus on technical factors, whereas one hypothesis focuses in the social factors, that can influence the creation of trust.

To evaluate the hypotheses, we developed an approach based on mining code repositories, bug-tracking systems, and other sources of project information. Our approach consists of several steps. First, it computes the initial group of "core" committers of a project, and classifies the remaining committers in the project as "non-core" committers. Then, it analyzes the activities in the code repository and the bug-tracking system performed by other people, who are not core committers. For these users, it identifies the types of activities (if any) that were performed in the bug-tracking system prior to the first code commit. In this manner, our approach computes information that let us study different hypotheses related to the induction of developers as committers. Although the hypotheses can be explored in other ways, such as via surveys, the goal of our work is to develop effective, automated techniques for mining and correlating data from different project repositories and other publicly available information sources. In the paper, we discuss the challenges in developing such techniques and present our solutions toward addressing those challenges.

We implemented our approach and performed an empirical study of the Eclipse projects. We analyzed 50 Eclipse products and 478 non-core committers across these projects. We found that a large percentage, approximately 51%, of these developers made some contributions in the bug-tracking system of the projects before they performed the first code commit. Among the remaining committers, 17% could be traced back to belonging to the same organization as one of the core committers, whereas 5% had prior committer experience in other Eclipse projects. For the remaining 27%, none of our hypotheses applied. Partly, this can be attributed to the fact that we did not have complete information for each committer in the code repositories that we studied. However, this also indicates that there may be other factors that influence developer initiation as committers and, therefore, other potential hypotheses could be formed and evaluated.

We also found that only 4% of the Eclipse Bugzilla contributors got promoted as code committers. 21% (2310) of the Bugzilla users have performed some code-maintenance activity (*e.g.,* submitting patches), but only 16% (377) of these users got promoted to being official project committers. Overall, the Bugzilla users who were granted commit privileges performed more Bugzilla activity than the users who did not make it to the committer group.

The main contributions of the paper are:

- The description, and an empirical study, of the phenomenon of developer initiation as committers in Eclipse
- The development of heuristics and techniques for mining data from code repositories, bug-tracking systems, and other information sources to investigate developer initiation as committers

In the next section, we formally state our hypotheses for investigation. Then, we present our approach (Section 3) and the results of the empirical study conducted on the Eclipse projects (Section 4). Following that, we discuss related research (Section 5) and, finally, conclude the paper with a summary of our contributions and directions for future research (Section 6).

2. THE HYPOTHESES

As we mentioned in the Introduction, there are many factors that can influence the creation of the trust that is essential for the induction of a developer as a code committer. The most obvious factor is the track record of a developer on a project: by making high-quality contributions to a project, in a non-committer role (*e.g.,* by submitting code patches), a developer can establish credibility with the core committee of the project. For example, the Eclipse Charter contains normative guidance along these lines

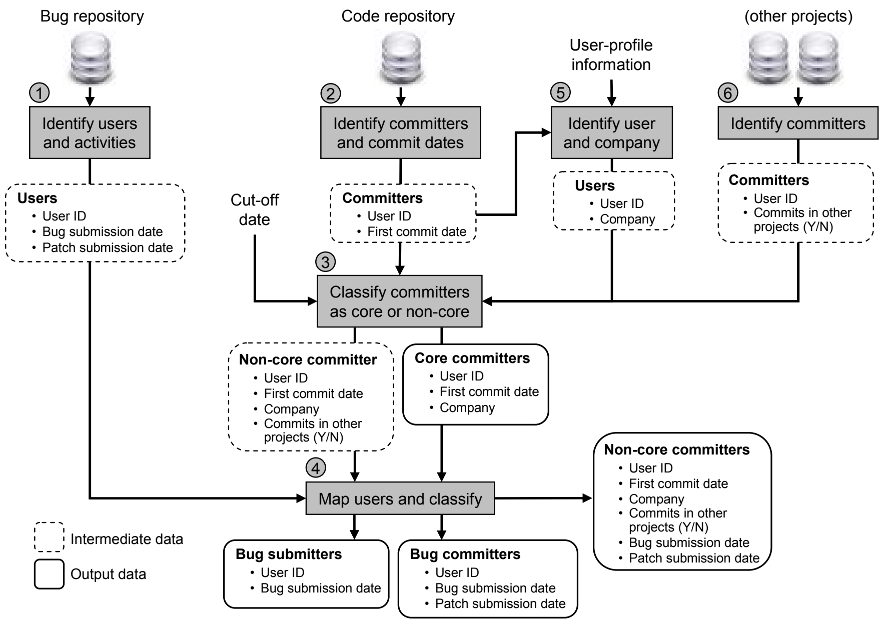
> Developers who give frequent and valuable contributions to a Project, or component of a Project (in the case of large Projects), can have their status promoted to that of a "Committer" for that Project or component respectively.[2]

Our first hypothesis is a simple one, formulated to investigate the nature of contributions made to a project by developers, before they are granted commit access to the project's code base. The most typical contributions made by developers are related to activities performed in the bug-tracking system of a project—*e.g.,* opening bug reports, contributing to bug resolution, and submitting code patches. Such activities are unrestricted, and form a good basis for establishing a good reputation with, and earning the trust of, the core committee. This hypothesis is similar to one of the hypotheses explored by Bird and colleagues [2]: they hypothesized that demonstration of skill via patch submissions increases the likelihood of attaining official developer status. More formally, we state our first hypothesis as follows.

H1: Developers make contributions to a project's bug-tracking system by opening a new bug report, or contributing a code patch, before they commit code to the project's code base.

Although developers can perform other activities in a project's bug-tracking system, we think that the two activities mentioned in H1—particularly, the submission of code patches—are more relevant for gaining commit privileges. Prior research has also indicated the importance of patch submission in attaining committer status [2, 6]. Moreover, other activities, such as assigning or closing bug reports, would typically be performed by the project core committee. Note that our hypothesis is framed to study the phenomenon that developers perform *some* activity, and not whether

[2] `http://www.eclipse.org/projects/dev_process/Eclipse_Standard_TopLevel_Charter_v1.1.php#Committers`



Bug repository
Code repository
User-profile information
(other projects)
1
Identify users and activities
2
Identify committers and commit dates
5
Identify user and company
6
Identify committers
Users • User ID • Bug submission date • Patch submission date
Cut-off date
Committers • User ID • First commit date
Users • User ID • Company
Committers • User ID • Commits in other projects (Y/N)
3
Classify committers as core or non-core
Non-core committer • User ID • First commit date • Company • Commits in other projects (Y/N)
Core committers • User ID • First commit date • Company
4
Map users and classify
Non-core committers • User ID • First commit date • Company • Commits in other projects (Y/N) • Bug submission date • Patch submission date
Intermediate data
Output data
Bug submitters • User ID • Bug submission date
Bug committers • User ID • Bug submission date • Patch submission date

Figure 1: Overview of our approach for classifying users based on an analysis of information from different sources.

they perform *significant* activity, because the amount of activity that qualifies as “significant” can be subjective in nature. However, this hypothesis does let us study, to some extent, whether the Eclipse projects adhere to the Eclipse Charter normative guidance on the granting of committer status.

In our study of Hypothesis H1, we also explored the phenomenon that developers start by performing simpler activities (submitting bug reports) and then move on to the more complex activity of submitting code patches. This phenomenon, if confirmed, would illustrate that a progression occurs in a developer's gaining confidence in a project and gradually making more important contributions that lead to the granting of committer status.

To investigate factors unrelated to the activities performed on a project that could explain the granting of commit access to developers on that project, we formulate two additional hypotheses.

H2: Developers with prior experience in contributing code to open-source projects are inducted as code committers.

H3: Developers working for the same organization that a member of the core group works for are inducted as code committers.

3. OVERVIEW

Our approach is based on automated mining of information from code and bug repositories. Before presenting the approach, we introduce terminology, which we use in the rest of the paper.

The *cut-off date* is the date on which the first release of a project occurs. To study our hypotheses, we have to identify the code committers of a project who are not part of the initial *founding group* of the project; for the founding group members, establishment of trust would not be an issue. In essence, we wish to study the initiation of those developers who start to participate in a project after its first release—and are, presumably, outside the core founding group. We use the cut-off date to classify a project committer into one of two categories. A *core committer* of a project is a user who has committed code to the project code repository before the cut-off date (*i.e.,* who has code commit privilege during the development of the first release). A *non-core committer* is a user who has committed code to the project code repository after the cut-off date only (*i.e.,* who did not commit code prior to the first release).[3] A *bug submitter* is a user who has created a bug report in bug-tracking system of a project, but has not committed code to the code repository. A *bug committer* is a person who has submitted a code patch in the bug-tracking system but who has not committed code.

Figure 1 presents an overview of our approach for classifying the users of the code and bug repositories of a project. The figure illustrates different steps of the approach, along with the data computed by each step. The shaded boxes represent the steps. The boxes with rounded corners illustrate the data flow between the steps: among these, the boxes with dashed borders represent intermediate data, whereas the boxes with solid borders represent the final data output by the approach. Thus, as illustrated, the final output of the approach consists of a classification of the users as core committers, non-core committers, bug submitters, and bug committers. For each class of user, the approach also computes attributes that are relevant for investigating our hypotheses. For example, for a core committer, we compute the first commit date and the employing company.

The *first step* of the approach identifies the users of the bug-tracking system and analyzes the activities performed by them. For each user, the approach identifies: (1) the date (if any) on which the first bug report was submitted by the user and (2) the date (if any) on which the first patch file was submitted by the user.

The *second step* of the approach analyzes the code repository to identify all the users who have committed code in the repository. For each user, it identifies the first commit date.

The *third step* of the approach classifies the committers as core or non-core. To do this, the cut-off date is required, which can be provided manually or can be approximated, for example, from the information contained in the bug repository (as we did in our study of the Eclipse projects, discussed in Section 4.1.4).

In the *fourth step*, we correlate the users of the bug-tracking system with the non-core committers. Depending on the setup of a

[3] Note that our goal is to study neither the magnitude of code commits by the founding group members during the development of the first release, nor whether the founding group remains stable over time. Our aim is simply to identify the subset of code committers for whom establishment of trust would be an issue, which we estimate by identifying developers whose first code commit occurs after the first project release.

project, matching the users between the code repository and the bug repository may not be trivial because the user may have different authentication credentials in the two repositories. This challenge has been acknowledged in the mining literature [1, 2]. To perform our study of the Eclipse projects, we developed several name-matching heuristics, which we discuss in the next section. Step 4 also computes three classes of users. A user for whom the bug-submission date exists, but the patch-submission date does not exist, is classified as a bug submitter. A user for whom the patch-submission date exists is classified as a bug committer. Finally, a user for whom a commit date exists (and the patch/bug-submission dates may exist or not), is continued as a non-core committer.

To validate Hypothesis H3, we require employment information about the code committers. The *fifth step* of the approach identifies the employing company for each code committer. Depending on the project being analyzed, this information may or may not be easily available from the user profiles. The increasing popularity of social networking sites, such as LinkedIn or Facebook, makes them potential sources for extracting employment-related and other demographic information.

Finally, for each committer, we need information about whether the user has committed code to related open-source projects prior to committing code to the project being analyzed. Many open-source projects are "umbrella" projects under which several projects exist. *Step 6* of the approach identifies committers in the other projects within the same umbrella project. Using the first commit date for a user in the project being analyzed, we determine whether any commits were performed prior to that date in other projects.

4. APPLICATION OF APPROACH TO ECLIPSE PRODUCTS

We conducted an empirical study in which we applied our approach to investigate the developers' initiation as committers in the Eclipse projects. The Eclipse projects use CVS or Subversion as the code repository and Bugzilla as the bug-tracking system. We implemented our approach for mining these repositories, and extracting information from other sources available for these projects. In all, we started by analyzing 219 projects. However, in the some of the steps in the method of the study, we had to eliminate many of the projects.

In the this section, we (1) discuss the method that we followed to conduct the study (Section 4.1), (2) present the results of the study (Section 4.2), and (3) discuss the limitations of our evaluation, including the threats to the validity of our observations (Section 4.3).

4.1 The Method of the Study

4.1.1 Identifying Committers and Commit Dates

Periodically, the Eclipse projects collate and expose the commit history via the Eclipse Project Dash.[4] Instead of mining the commit information from the code repository, we extracted the commit data from the Project Dash. For each project, we identified the users who had committed code to the project. Moreover, for each code committer, we obtained the commit dates and the employing organization. Using this data, we identified the first code-commit date for a user.

4.1.2 Analyzing Bugzilla Activities

Next, to compute the cut-off date, classify the committers, and compute the bug-submission activities, we analyzed the Eclipse Bugzilla repository. For the purposes of the study, we used the Bugzilla MySQL snapshot provided for the MSR 2011 Mining Challenge.[5] From the records contained in the `bugs` table in the database, we identified, for each bug, the login ID of the user who had created the bug report and the date on which the report was created. Finally, from the `attachments` table, we got what files were attached to a bug, when, and by whom. We only considered those attachments for which the `isPatch` attribute was set to true.

4.1.3 Mapping Eclipse Projects to Bugzilla Products

One of our tasks was to compute, for each Eclipse project, the bug reports created for the project. This turned out to be a challenging task. One of the fields in a Bugzilla report specifies the product in which the observed failure occurs. Thus, we had to map Eclipse *project names* to Bugzilla *product names*. In general, there is a one-to-many mapping between Bugzilla products and Eclipse projects. For example, the Bugzilla product `Data Tools` maps to many Eclipse projects: ***e.g.,*** `datatools`, `datatools.connectivity`, `datatools.incubator`, `datatools.sqltools`, `datatools.modelbase`, and `datatools.enablement`. To compute the mapping, we used different heuristics.

Each bug report, in addition to the product name, also contains a *classification name*, which indicates a high-level categorization of the bug. The classification is determined by the Eclipse administrators in the Bugzilla repository. We used a combination of the classification and project names as `<class name>.<product name>` to match projects and products: if `<class name>.<product name>` contains an Eclipse project name, we consider it to be a match. Another heuristic that we used performs matching based on acronyms expansions. For example, based on this heuristic, we mapped the Bugzilla product `WebTools.Java Server Faces` to the project `webtools.jsf`. We manually verified all the mappings that were computed by the heuristics. However, for some projects, *e.g.,* `tptp.performance`, we could not determine the mappings. Of the initial 219 projects that we started with, we were able to map 156 projects to Bugzilla products. These mappings identified 102 Bugzilla products.

4.1.4 Computing Cut-off Dates

The next task was to compute the cut-off date, which is required for classifying code committers as core or non-core committers. We computed the cut-off date by identifying the date on which we found the first bug report that was marked with the first released version of the product. Out of the 102 Bugzilla products, we were able to identify the cut-off dates for 50. For the remaining products, either no version information was available in Bugzilla or the core committer/non-core committer list we calculated was empty. The core committer list could be empty, if a bug was opened in the bugzilla repository with a valid version id even before the first commit was made in the code repository. The non-core committer list could be empty if there were no code commits made after the cut-off date. We limited the scope of our study to these 50 products only. These 50 products covered 80 Eclipse projects.

4.1.5 Mapping Eclipse Committers to Bugzilla Users

Because a user has different authentication credentials in the Eclipse code and bug repositories, one of the steps in the implementation of our approach requires matching Eclipse committers and Bugzilla users. Bugzilla assigns a unique integer ID to each user, and the `profiles` table in the MySQL database lists the user name for each ID. The mapping between user name and user ID is also available via the Bugzilla query interface. Because the CVS

[4] `http://dash.eclipse.org/dash/commits/web-app`

[5] `http://home.segal.uvic.ca/~schadr/msrc2011/eclipsebugs.sql.gz`

user login and the Bugzilla user names can be different, appropriate heuristics needs to be used to map the users between the two systems. For example, we found a case where the CVS login ID was `mclay`, the Bugzilla ID was 46978, and the Bugzilla user name was "Michael Clay." Fortunately, there is another information source that simplified the user-mapping task. Each Eclipse project provides a publicly available list of active project committers.[6] Using automated screen scraping of the HTML pages, we could compute the user name for a CVS login ID and then match the user names between the code and bug repositories. However, this technique was not applicable for emeritus or alumni users, who were not listed as active committers. To perform the matching for such cases, we used five heuristics, which we discuss next.

Previous research has used different heuristics to match names and email IDs. Bird and colleagues [2] created tuples of ⟨`name`, `email`⟩ of Apache developers and applied the Levenshtein edit distance [12] to identify similar names, similar email IDs, and match names to email IDs. They considered two names to be similar if either the complete names were similar or both first and last names were similar. When comparing names with email IDs, they identified a match if (1) the email ID contained both the first and last names, or (2) the email ID contained the initial of one part of the name and the entirety of the other part. Our name matching heuristics match CVS login IDs to Bugzilla user names and is, thus, similar in goal to the name-to-email matching in Reference [3].

Our first heuristic (NH1) tackles the cases where the CVS login ID follows the template `<first name str><last name>`. A match is found if `<first name str>` is a substring of the Bugzilla user first name, starting at the first position, and `<last name>` exactly matches the user last name. The second heuristic (NH2) checks whether the CVS login ID matches exactly the user's first name or last name. The third heuristic (NH3) is similar to NH1, but it reverses the order of the first and last names: it handles the case where the CVS login follows the template `<first name><last name str>`. The fourth heuristic (NH4) is a less stringent version of NH1 in that it does not require an exact match of the last name. Similarly, the final heuristic (NH5) is a less strict version of NH3—it checks for substring matches for the first name in an analgous way as NH4 does for the last name.

We applied these heuristics in the order presented. Our goal was to apply those heuristics first that had better likelihood of computing correct matches. Because NH4 and NH5 perform less strict matching than NH1–NH3 (*i.e.,* NH4 and NH5 check for substring matches of the last and first names, respectively, instead of exact matches), they can compute false positives. Therefore, we applied them after NH1–NH3.

Our heuristics can return multiple matches. In such cases, we resolved the conflicts by an analysis of the dates on which the CVS and Bugzilla activities occur. The intuition underlying this analysis is that a CVS code commit might often (although not necessarily) be accompanied by a Bugzilla activity. Therefore, if a CVS login ID and a candidate matching Bugzilla user name are associated, respectively, with a CVS commit and a Bugzilla activity on the same project on the same date, it is highly likely that the user name and the ID match. More concretely, suppose that our heuristics map a CVS login ID l to two Bugzilla users u_1 and u_2. Our analysis examines the dates of all the CVS commits performed by l; let $D(l)$ be this set of dates. Similarly, the analysis computes $D(u_1)$ and $D(u_2)$—the set of dates on which the Bugzilla activities were performed by users u_1 and u_2, respectively. Then, we compare $D1 = (D(l) \cap D(u_1))$ and $D2 = (D(l) \cap D(u_2))$, select the maximum of $D1$ and $D2$, and consider l to match the corresponding user. In case of ties in the maximum value, we attempt to resolve the conflict manually.

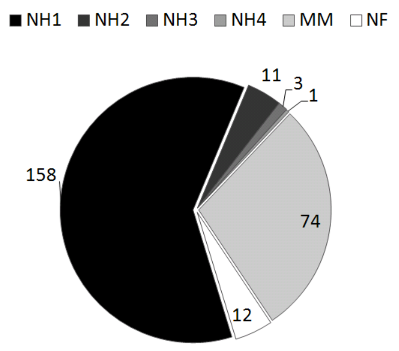


Figure 2: Effectiveness of the name-matching heuristics. No mapping was computed by heuristic NH5. MM represents the manually mapped IDs, whereas NF represents the IDs that could not be mapped.

To illustrate with an example, we found a case where the NH2 heuristic mapped CVS login ID `philippe` to 12 Bugzilla user names, some of which are "Philippe Mulet," "Philippe Coucaud," "Philippe Ombredanne," "Philippe Faes," "Philippe De Oliveira," and so on. Using the date-based ranking analysis, we were able to narrow down the match precisely to "Philippe Mulet". For cases where heuristics computed multiple mappings and where no match was found, we attempted to manually compute the mappings.

For the 80 Eclipse projects in scope for our analysis, there were 879 unique CVS login IDs, which had to be mapped to Bugzilla user names. For 620 of the login IDs, we could compute the mappings using HTML scraping. Thus, for the remaining 259 IDs, we applied the five heuristics. Figure 2 presents data about the effectiveness of the heuristics and manual mappings. As the chart illustrates, the heuristics computed the mappings for 173 (67%) of the 259 IDs. NH1 was quite effective: it successfully computed the mappings for 158 login IDs (61%). The number of IDs matched by NH2, NH3, and NH4 were 11, 3, and 1, respectively. The application of NH5 did not result in the computation of any mappings. We manually identified 74 mappings (shown as "MM" in the figure); Finally, for 12 of the IDs, neither the heuristics nor manual examination could identify the mappings. In related work, Anvik, Hiew, and Murphy [1] attempted to map 83 Eclipse CVS login IDs to Bugzilla email addresses; they were able to map successfully 83% of the login IDs.

4.1.6 *Computing Data for the Hypotheses*

Using the cut-off date and the first code-commit dates, we classified the committers as core or non-core committers. Next, for the non-core committers, using the name mappings, we identified the Bugzilla activities performed by them before their first code commit. This data set let us investigate the first hypothesis.

For the second hypothesis, we determined, for each non-core committer of a project, whether the user had committed code to other Eclipse projects before committing code to that project.

To evaluate the third hypothesis, we had to identify the employing organization for each core and non-core committer. To do this, we examined the company information provided in the Eclipse Project Dash.[3] We found instances where a user's affiliation was listed as "individual"—we could not compute the employing organization for such users.

[6]For an example of the information about active committers, see `http://www.eclipse.org/projects/project_summary.php?projectid=modeling.tmf`.

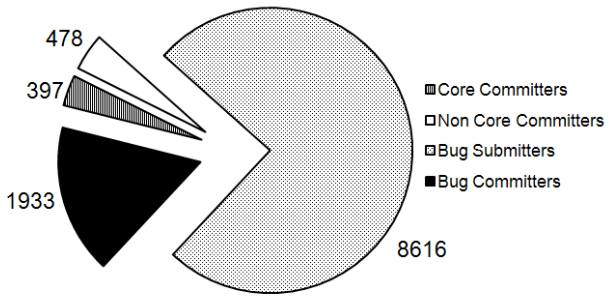


Figure 3: The classification of all the code committers and Bugzilla users for the 50 Eclipse products.

4.2 Results and Analysis

First, we present the data that we collected about the classification of the Eclipse code committers and the Bugzilla users. Following that, we present the data about the three hypotheses. Finally, we present data to investigate further the Bugzilla activities of the users: we study the duration and magnitude of activities performed by the non-core committers who matched Hypothesis H1, and we analyze the magnitude of activities performed by the bug committers who did not become code committers.[7]

4.2.1 Classification of Code Committers and Bugzilla Users

Figure 3 presents data about the distribution of core committers, non-core committers, bug submitters, and bug committers across the 50 products. The total population of users of the code and bug repositories is 11424. Of these only 8% (875) are code committers; 75% (8616) are bug submitters (*i.e.,* these users have created Bugzilla reports); and 16% (1933) have submitted code patches (in addition to possibly creating bug reports). Among the code committers, 45% (397) are core committers, whereas 55% (478) are non-core committers.

To enable an analysis of the variations in user classifications among the products, Figure 4 presents the classification of users per product as a box plot. The figure shows one box per user category; 50 data points (representing the 50 products) are plotted for each box. The vertical axis represents the percentage of users. Each box represents the middle 50% of the Eclipse products; the remaining 50%, with the exception of some outliers, is contained within the areas between the box and the whiskers. The line inside the box represents the median and the location of the median within the box illustrates skewness in the distribution.

Consider the data for bug submitters. The location of the median line indicates that, for 50% of the products, at least 64% of the user population consists of bug submitters. Furthermore, the lower end of the box indicates that, for 75% of the products, at least 46% of the users are bug submitters. There is a very wide spread in the percentage of bug submitters—from a low of 5% (ignoring three outliers products with no bug submitters) to a high of over 90%. Compared to this, the core-committer data has a much smaller spread. The percentages of core committers is generally small across the products: 50% of the products have less that 5% core committers, and another 25% have no more than 12% core committers. Thus, more products appear to have a small core team (from the user population) that works on the first release of the product. The distribution of the non-core committers is similar to

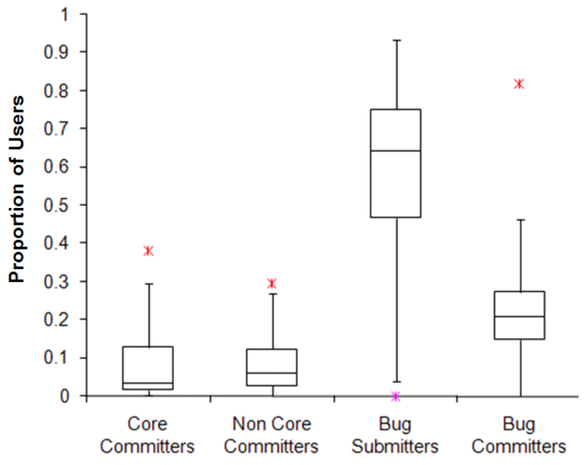


Figure 4: The distribution of Eclipse products based on the percentages of core committers, non-core committers, bug submitters, and bug committers.

that of the core committers, but exhibits less skewness. The data for bug committers has greater spread than the code-committers data. With the exception of a few outliers, for all products, less than half of the user population has submitted patches.

In terms of absolute numbers, for five out of 50 products, the user population size was less than 10; 17 products had user populations between 10 to 100; 26 had between 100 to 1000 users; and two products had more than 1000 users. To give a sense of the individual variations in, and sizes of, the user populations, Table 1 presents the user classifications for some of the products.

Overall, the data about the user classification illustrate that most products have good support in terms of a having a significant percentage of bug submitters and committers. Previous research [9] has observed that, for an open-source project to show super-linear growth, a large population of users should be performing supporting activities, such as submitting bugs and fixing bugs. Given the general success and popularity of Eclipse products, our data appears to corroborate the observation made in Reference [9].

4.2.2 The Hypotheses

We evaluated the three hypotheses for the 478 non-core committers in user population of the 50 products. Figure 5 presents the data to illustrate the application of the hypotheses. In the figure, the donut graph in the center shows that, for 51% (246) of the 478 committers, Hypothesis H1 holds. These developers have created a bug report, submitted a patch, or performed both prior to their first code commit. The chart classifies the remaining 232 committers into two categories. The first category consists of 93 committers who performed their first code commit before any relevant Bugzilla activity. The second category consists of the committers for whom our approach could find no Bugzilla activity—there are 139 such committers.

We applied the remaining two Hypotheses H2 and H3, in order, to the two categories of committers who did not satisfy H1. The two pie charts in the figure show the distribution of how these hypotheses held. Of the 93 committers who had some, but no relevant, Bugzilla activity, 12% (11) have prior commits in other Eclipse projects (they meet H2), 33% (31) have the same employer as one of the core committers (they meet H3), and the remaining 55% (51) satisfy neither H2 nor H3. For the second category of 139 committers, the distribution is quite similar: 10% (14) satisfy H2, 35% (49) satisfy H3, and 55% (76) comply with neither.

[7]The data that we collected in our study is available at `http://sites.google.com/site/committeranalysisinos`. The data includes the mapping between the Eclipse projects and the Bugzilla products, the classification of users, and the results of applying the three hypotheses.

Table 1: User classification for select Eclipse products.

Product Name	Projects	Core Committers	Non-core Committers	Bug Committers	Bug Submitters
AspectJ	tools.aspectj	8 (1.2%)	1 (0.2%)	60 (9.1%)	590 (89.5%)
Birt	birt	30 (1.7%)	49 (2.7%)	200 (11.1%)	1523 (84.5%)
BPEL	technology.bpel	1 (1.7%)	7 (11.9%)	14 (23.7%)	37 (62.7%)
E4	eclipse.e4	47 (31.1%)	11 (7.3%)	41 (27.2%)	52 (24.4%)
M2T	modeling.m2t, m2t.modeling, m2t.acceleo, m2t.jet, m2t.xpand	2 (1.2%)	20 (12.1%)	29 (17.6%)	114 (69.1%)
PDT	tool.pdt	8 (1.1%)	13 (1.7%)	65 (8.8%)	657 (88.4%)
WTP Common Tools	webtools.common	6 (1.4%)	17 (3.9%)	86 (19.5%)	331 (75.2%)

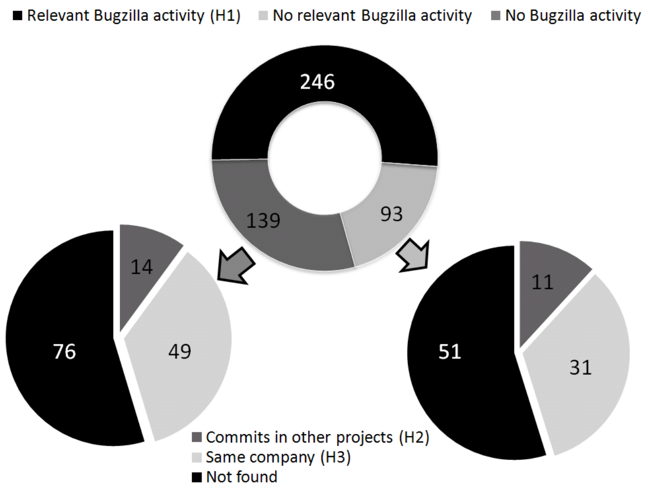


Figure 5: Data to illustrate the evaluation of the three hypotheses for the 478 non-core committers.

Overall, for 27% (127) of the non-core committers, none of the hypotheses held. This can be explained by three factors. First, our hypotheses did not cover other possible ways in which trust could be established. For example, it could be the case that developers earn reputation and trust by participating in discussions on mailing lists or via wiki activities. Second, some of our heuristics may have led to incorrect classifications. For example, incorrectly estimating the cut-off date could cause some core committers to be classified as non-core committers. Thus, it is possible that some of these 127 non-core committers could, in fact, be core committers. Finally, in some cases, the information available to us was insufficient. For example, ideally, we would have been able to compute the employment information for each non-core committer. But, as mentioned in Section 4.1.6, many of the core and non-core committers were listed as individuals with no company affiliation.

Figure 5 does not illustrate that there is overlap among the hypotheses (*i.e.,* some users satisfy multiple hypotheses). Also, because we applied the hypotheses in the order H1–H3, the absolute applicability of H2 and H3 is not evident in Figure 5. Figure 6 presents this information as a Venn diagram. As the diagram illustrates, 14 of the 351 non-committers who satisfied at least one hypothesis, in fact, satisfied all the hypotheses. Among the three hypotheses, H1, H2, and H3 were uniquely applicable for 114, 25, and 69 committers, respectively. H2 was the least applicable, with 80 committers satisfying it; H3 was satisfied by 182 committers.

Figure 7 presents finer-grained data about the hypotheses: it splits the distribution of the users per Bugzilla product. The vertical axis represents the percentage of non-core committers for a product; the segments within a bar represent the distribution of the committers who match the three hypotheses. For example, for product 6, 60% of the users match H1, and 20% each match H2

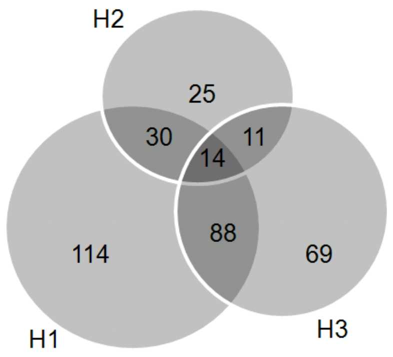


Figure 6: Overlap among the applicability of the three hypotheses. For 351 of the 478 non-core committers, at least one of the hypotheses is applicable.

and H3. In cases where our hypotheses cover all the non-core committers, the height of the bar is 100%—this occurs for 23 of the 50 products. On the other end of the spectrum, for two products (92 and 156), none of the hypotheses covered any of the committers. Both these products have very small user populations (seven and eight), and only three non-core committers. Moreover, no affiliation was available for the users, so H3 could not be applied.

For four products (103, 144, 146, and 158), all users became committers by satisfying H3 only. This case would typically occur for products that have been initiated by a small group of people belonging to the same company; participation in these products is yet to grow outside the company. Also, the products may have yet to build a large user base. We manually examined the four products (`ESL`, `Webtools.pave`, `Webtools.JSDT`, and the `Tigerstripe`) and found that they have a very small group of committers. `ESL` and `Webtools.pave` have only two committers each; `Webtools.JSDT` and `Tigerstripe` have eight committers each. Moreover, the products are at most three years old, and the Bugzilla activity for them is very low. The only exception to this is `Webtools.JSDT` for which 226 users have created bug reports or submitted patches. However, in three years, none of these users have become committers.

Overall, we find that, although there is variation in the user classification across the products, H1 is dominant in most of the products: for 13 products, all the users satisfy H1; for 33 products, at least 40% of the users satisfy H1. Thus, the Eclipse normative policy of promoting users based on contributions appears to be adhered to in most projects, with a few notable exceptions where the affiliation seems to be the dominant factor leading to trust.

4.2.3 *Further investigation of Bugzilla Activities*

For Hypothesis H1, Figure 5 presents data to illustrate whether the non-core committers performed some Bugzilla activity prior to the first code commit. We collected additional data to get further insights into the nature of the activities performed. Specifically, we studied whether developers perform only one or both of the relevant

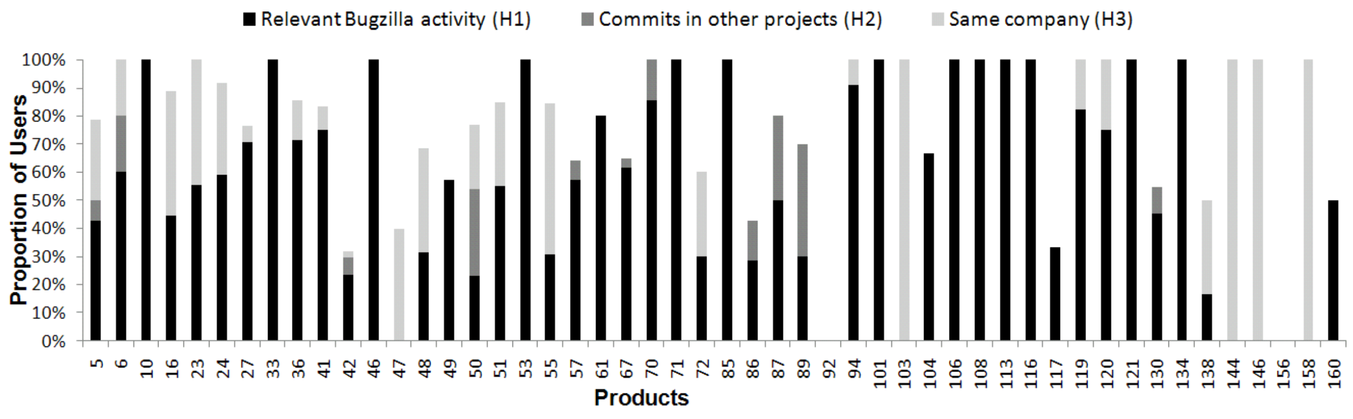


Figure 7: A product-wise illustration of the hypotheses: in each bar, the segments show the proportion of non-core committers in that project who satisfy the three hypotheses.

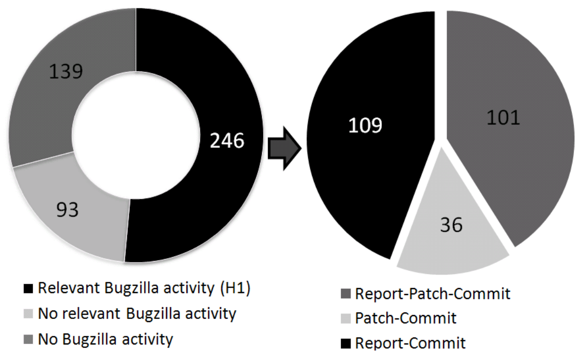


Figure 8: Patterns of activities performed by the non-core committers who satisfied H1: whether they submitted bug reports and patches, patches only, or bug reports only before becoming committers.

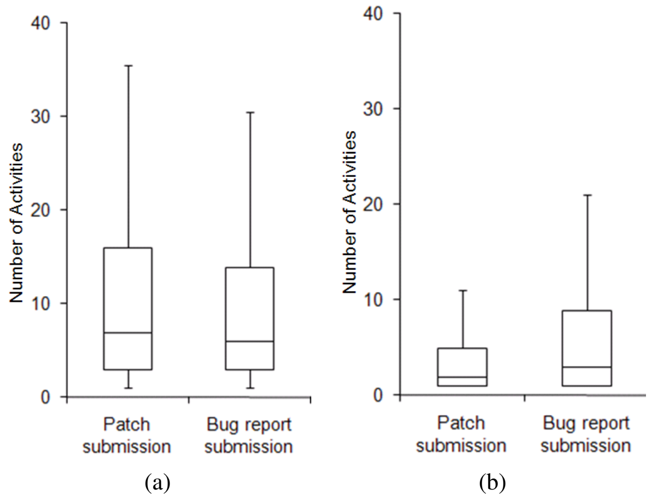


Figure 9: (a) For each non-core committer satisfying H1, the magnitude of Bugzilla activity before the first commit; (b) for each bug committer, the magnitude of Bugzilla activity.

activities (*i.e.,* bug report submission and patch submission); and the magnitude of the activities performed by non-core committers and bug committers. We also examined the duration between the first Bugzilla activity and the first code commit, and the proportion of Bugzilla users do not become committers in a project despite being committers on other projects.

The pie chart on the right in Figure 8 shows, for the 246 non-core committers who performed some Bugzilla activity, whether they submitted bug reports and patches (with bug submission being the first activity), patches only, or bug reports only. As the data illustrate, 41% (101) of the non-core committers started by reporting bugs, and subsequently submitted patches, before getting promoted as committers; 15% (36) submitted patches only and reported no bugs; 44% (109) submitted only bugs and no patches. The data suggest that users who submit patches are more likely to become committers: 56% of the users had submitted at least one patch prior to becoming committers.

To get a sense of how much Bugzilla activity precedes the granting of committer status, we examined the number of Bugzilla activities performed by the 246 non-core committers before their first code commit. The box plot in Figure 9(a) presents this data. Furthermore, we compared this data with the activities of the Bugzilla committers—users who have submitted patches but have not been granted committer status. The box plot in Figure 9(b) presents this data. A comparison of the two plots reveals that the code committers, generally, have performed more activities than the Bugzilla committers. For the non-core committers, the median bug submission and patch activity is six and seven respectively. However, for bug committers, the median is much less—three for bug submission and two for patch submission. This further seems to validate H1: it suggests that users who performed more Bugzilla contributions eventually got promoted as code committers. The data also suggest that, as users make more contributions via patch submissions, their likelihood of becoming committers increases.

For ease of comparison, the outlier values have been excluded from the two charts. For patch submission, there are two outlier values (out of 137 values, with maximum value 114) for non-core committers, and 15 outlier values (out of 1185 values, with maximum value 324). For report submission, there are five outliers (out of 242 values, with maximum value 236) for non-core committers, and 48 outliers (out of 1716 values, with maximum value 1986).

We also examined, for the 246 non-committers, the duration between their first relevant Bugzilla activity (bug-report or patch submission) and first code commit. Figure 10 presents this data: the horizontal axis shows different ranges of the duration in months, whereas the vertical axis shows the number of users in the duration ranges. As the chart illustrates, 21% (50) of the committers performed their first code commit within a month of their first Bugzilla activity; as many as 83% performed the first code commit within six months. Thus, it appears that a quick progression from the first relevant Bugzilla activity to the committer status is a common phe-

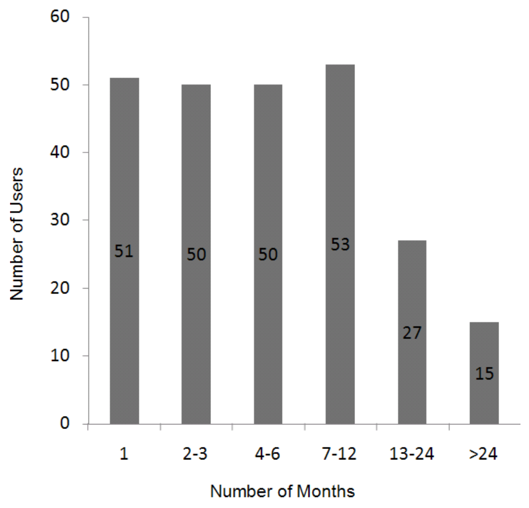


Figure 10: For each user matching H1, the duration between the first relevant Bugzilla activity and the first code commit.

nomenon for the studied Eclipse products. A small percentage of the users (6%) took more than two years before performing the first commit; among these two users took more than five years.

Finally, we examined the negative instances of Hypothesis H2 among the Bugzilla users—the cases where a Bugzilla user did not become a committer in a project in spite of committing code to other projects. We found 727 such instances, which accounted for for 6% of the 10549 Bugzilla users (Figure 3).

4.3 Discussion

Threats to external validity arise when the observed results cannot be generalized to other experimental setups. In our evaluation, we studied the Eclipse projects only; moreover, we had to eliminate many projects from the origin set of 219 projects. Therefore, we cannot conclude how our observations might generalize to the excluded Eclipse projects and other open-source projects. Further experimentation with other projects is required to evaluate how our results may generalize, which we intend to do in future research.

Threats to internal validity arise when factors affect the dependent variables (the data described in Section 4.2) without the researchers' knowledge. In our study, such factors are errors in the implementation. Moreover, some of our heuristics could have introduced inaccuracies in the results. First, we used heuristics to match Eclipse projects with Bugzilla products. We manually verified all the computed mappings. Second, we used a heuristic to approximate the cut-off date, and because of inaccuracies, some core/non-core committers may have been classified incorrectly. Finally, the name-matching heuristics could be another source of inaccuracy: they may have caused some names to be identified as matched that do not, in fact, match. To mitigate this threat, we manually verified a large sample (approximately 18%) of the computed name mappings. This threat is further mitigated by the fact that, among the five heuristics, NH4 and NH5 are most likely to compute incorrect matches because they perform substring matching instead of exact (equality) matching. However, these two heuristics together computed only one mapping. Thus, the name matching heuristics are not a significant threat to internal validity.

Our hypotheses are formulated on the premise that activities in the code and bug repositories can be indicative of the establishment of trust. One could question that this potentially leads to construct threats to validity (which arise when the measures do not adequately capture the concepts they purport to measure). However, contributions via such repositories is widely accepted as a factor that determines the establishment of trust, as evidenced by the Eclipse charter normative guidance (see Section 2), and the investigation of similar hypotheses in other studies (*e.g.,* [2, 6]).

Ideally, an investigation of a causal relationship between a factor and an outcome, should study both instances where the outcome occurs (positive instances) and where it does not occur (negative instances). We performed such an investigation of Hypotheses H1 and H2. However, for H3, we did not investigate the occurrence of the negative instances—*i.e.,* the extent to which Bugzilla users who do not become non-core committers belong to the same organization as a core committer. The limitation that employment information for the Eclipse Bugzilla users is not publicly available prevented us from performing this investigation.

5. RELATED WORK

Other researchers have studied the phenomenon of developer initiation as code committers [2, 6, 15].

Krogh and colleagues [15] studied the developer-initiation process in the Freenet project. They collected data via interviews, extracting email exchanges on developers mailing list, and analyzing the code-commit history. They classified the participants into three categories: joiner (a person who has joined the mailing list but does not have commit privileges), newcomer (someone who has just started committing changes), and developer (who is contributing core code to the project). Similar to our finding, they found that actions, such as offering bug fixes, are common among contributors who eventually become committers.

Bird and colleagues [2] performed a similar quantitative study of the Apache, PostGres, and Python projects. They considered the duration between an individual's first involvement in a project's mailing list and the first code contribution, and analyzed three determining factors: (1) knowledge and skill level as demonstrated by patch submissions, (2) individual reputation based on responses and collaboration on mailing lists, and (3) commitment as demonstrated by continued participation. They found that (2) applied to all the three projects, whereas (1) and (3) were partially supported.

Ducheneaut [6] studied the Python project to explore the interactions that developers have as they transition from being newcomers to code committers to core developers. He used a combination of an ethnographic study and mining email and code databases, and found that raising bug reports and providing solutions to the bugs enable an individual to get commit privileges. However, further interactions with the right people are needed for an individual to rise in the ranks to become a core developer of the project.

There are several distinguishing aspects of our work from these studies. First, we study two additional socio-technical factors that these studies did not consider: our second and third hypotheses have not been investigated by other researchers. Second, we performed the study on the Eclipse projects, which are some of the most popular and successful open-source projects, and the developer-initiation phenomenon has not been studied on these projects. Third, our methodology differs from the other studies. Specifically, the key feature of our study—the classification of the user population into four categories—which enables the investigation of interesting research questions about these classes of users, is novel.

Other researchers have studied developers' motivations behind participating in open-source projects. For example, Ghosh [7] performed a survey of developers involved in open-source projects to understand their motivations for contributing, such as skill building, reputation building, and monetary considerations. Roberts and colleagues [13] studied the Apache developers to understand the motivations for participation, and the interrelationships between motivational factors and developer performance.

The effects of different factors on the success of an open-source project has also been a topic of several studies (*e.g.,* [9, 10, 11]). Long [10] considered different factors, such as the number of core developers, the number of messages in the mailing lists, and the numbers of bug reports and patch submissions. On a corpus of 300 projects from SourceForge.NET, they found all these factors to correlate positively with success. Mockus and colleagues [11] studied the Apache and Mozilla projects to understand the success factors, and suggested seven hypotheses for evaluation on successful open-source projects. Koch [9] analyzed the CVS history of projects on SourceForge.NET to study the growth rate of open-source projects. He concluded that projects with high user participation show a super-linear growth rate, projects with medium participation exhibit decreasing growth rate, and small projects show linear growth. However, even in projects with a super-linear growth rate, most of the user population performs supporting activities, such as bug submissions and fixes—a much smaller user population is involved in core development. In our current evaluation, we have not studied correlations between the initiation of code committers and the success of an open-source project.

Other research that has investigated aspects of developer participation in open-source projects includes studies of developer demographics (*e.g.,* [5]) and evolution of developer participation in terms of specialization, generalization, and movement from periphery to core (*e.g.,* [4]).

6. CONCLUSIONS AND FUTURE WORK

In this paper, we studied a subset of the Eclipse suite of products to understand the factors that influence a contributor's initiation as a code committer in an open-source project. The key to the initiation is the establishment of trust and credibility. Therefore, we formulated and evaluated three hypotheses about how the trust could be developed: through demonstrated knowledge and skill, measured by contributions to the bug repository (Hypothesis H1), via code contributions to related projects (Hypothesis H2), and based on the employing organization (Hypothesis H3). To evaluate the hypotheses, we presented an approach based on automated mining of bug and code repositories and other publicly available information sources. One of the key steps of our approach classifies the developer population into four categories, which enables the investigation of the hypotheses.

Our results indicate that H1 is applicable most often—for the products that we studied, 51% of the committers satisfied H1. H2 and H3 were applicable, respectively, for 16% and 38% of the committers. For 21% of the committers, both H1 and H3 were applicable. Thus, for the Eclipse projects, demonstration of skill through activity in the bug repository seems to be the most significant factor for the granting of committer status. However, social factors, such as shared employment with a core committer, can also influence the granting of committer privileges.

There are several directions for future research, which we intend to pursue. Ducheneaut [6] studied how the connectedness of potential committers to core team members increases over time; we plan to perform a similar investigation of the Eclipse projects. To study whether our results generalize, we will perform additional experimentation using popular open-source projects, such as Apache and Mozilla.

Our hypotheses did not hold for 27% of the non-core committers. Therefore, we will investigate additional hypotheses, focusing on the influence of socio-technical factors on developers' initiation as code committers. Toward this goal, we will investigate how user information can be augmented via analysis of profiles and networks at social-networking sites, such as LinkedIn and Facebook.

ACKNOWLEDGEMENTS

We thank Pankaj Dhoolia and Debdoot Mukherjee for discussions related to the ideas presented in this paper. The anonymous reviewers gave helpful comments, which have improved the presentation of the work.

7. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*, pages 361–370, 2006.

[2] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu. Open borders? Immigration in open source projects. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 6–6, 2007.

[3] C. Bird, A. Gourley, P. T. Devanbu, M. Gertz, and A. Swaminathan. Mining email social networks. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 137–143, 2006.

[4] J. M. R. Costa, F. W. Santana, and C. R. B. De Souza. Understanding open source developers' evolution using transflow. In *Proceedings of the 15th International Conference on Groupware: Design, Implementation, and Use*, pages 65–78, 2009.

[5] B. J. Dempsey, D. Weiss, P. Jones, and J. Greenberg. Who is an open source software developer? *Communications of the ACM*, 45(2):67–72, February 2002.

[6] N. Ducheneaut. Socialization in an open source software community: A socio-technical analysis. *Comput. Supported Coop. Work*, 14:323–368, August 2005.

[7] R. A. Ghosh. Understanding free software developers: Findings from the FLOSS study. In S. Hissam and K. Lakhani, editors, *Perspectives on Free and Open Source Software*. June 2005.

[8] C. Jensen and W. Scacchi. Role Migration and Advancement Processes in OSSD Projects: A Comparative Case Study. In *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[9] S. Koch. Software evolution in open source project—A large-scale investigation. *Journal of Software Maintenance and Evolution: Research and Practice*, 19:361–382, November 2007.

[10] J. Long. Understanding the role of core developers in open source development. *Journal of Information, Information Technology, and Organizations*, 1:75–85, 2006.

[11] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, July 2002.

[12] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33:31–88, March 2001.

[13] J. A. Roberts, I.-H. Hann, and S. A. Slaughter. Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the apache projects. *Management Science*, 52(7):984–999, July.

[14] D. Stewart. Social Status in an Open-Source Community. *American Sociological Review*, 70(5):823–842, October 2005.

[15] G. von Krogh, S. Spaeth, and K. R. Lakhani. Community, joining, and specialization in open source software innovation: A case study. *Research Policy*, 32(7):1217–1241, July 2003.