Delft University of Technology
Software Engineering Research Group
Technical Report Series

An Exploratory Study of the Pull-based Software Development Model

Georgios Gousios, Martin Pinzger, and Arie van Deursen

Report TUD-SERG-2013-010

TUD-SERG-2013-010

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
`http://www.se.ewi.tudelft.nl/techreports/`

For more information about the Software Engineering Research Group:
`http://www.se.ewi.tudelft.nl/`

© copyright 2013, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

An Exploratory Study of the Pull-based Software Development Model

Georgios Gousios
Delft University of Technology
Delft, The Netherlands
G.Gousios@tudelft.nl

Martin Pinzger
University of Klagenfurt
Klagenfurt, Austria
martin.pinzger@aau.at

Arie van Deursen
Delft University of Technology
Delft, The Netherlands
Arie.vandeursen@tudelft.nl

ABSTRACT
The advent of distributed version control systems has led to the development of a new paradigm for distributed software development; instead of pushing changes to a central repository, developers pull them from other repositories and merge them locally. Various code hosting sites, notably Github, have tapped on the opportunity to facilitate pull-based development by offering workflow support tools, such as code reviewing systems and integrated issue trackers. In this work, we explore how pull-based software development works, first on the GHTorrent corpus and then on a carefully selected sample of 88 projects. We find that the pull request model offers fast turnaround, increased opportunities for community engagement and significantly decreased time to incorporate contributions. Moreover, we show that a relatively small number of factors affect both the decision to merge a pull request and the time to process it.

Categories and Subject Descriptors
D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Version control*; D.2.9 [**Software Engineering**]: Management—*Programming teams*

General Terms
Management

Keywords
pull-based development, pull request, distributed software development

1. INTRODUCTION
Pull-based development is an emerging paradigm for distributed software development. As more developers appreciate isolated development and branching [9], more projects, both closed source and, especially, open source, are being migrated to code hosting sites such as Github and Bitbucket with support for pull-based development [2]. A unique characteristic of such sites is that they allow any user to fork any public repository. The clone creates a public project that belongs to the user that cloned it, so the user can modify the repository without being part of the development team. What is more important is that they automate the selective contribution of commits from the clone to the source through pull requests.

Pull requests as a distributed development model in general, and as implemented by Github in particular, form a new method for collaborating on distributed software development. The novelty lays on the decoupling of the development effort from the decision to incorporate the results of the development in the code base. By separating the concerns of building artifacts and incorporating changes, work is cleanly distributed among a core team which is responsible to perform the merges and a peripheral team that submits, often occasional, changes to be considered for merging.

Previous work has identified the processes of collaboration in distributed development through patch submission and acceptance [24, 6, 33]. There are many similarities to the way pull requests work; for example, identical work team structures emerge, while pull requests go through a similar assessment process. What pull requests offer in addition is process automation and centralization of information. With pull requests, the code does not have to leave the revision control system, and therefore it can be versioned across repositories, while authorship information is effortlessly maintained. Communication is context-specific, being rooted on a single pull request. Moreover, the review mechanism that Github incorporates has the additional effect of improving awareness [12]; core developers can access in an efficient way all information that relates to a pull request and solicit the opinions of the community ("crowdsource") about the merging decision.

A distributed development workflow is effective if pull requests are eventually accepted, and it is efficient if the time this takes is as short as possible. Advancing our insight in the effectiveness and efficiency of pull request handling is of direct interest to contributors and developers alike. The goal of this work is to investigate pull request usage and to analyze the factors that affect the efficiency of the pull-based software development model. Specifically, the questions we are trying to answer are:

RQ1 How popular is the pull-based development model among users of distributed version control systems?

RQ2 What factors affect the decision to merge a pull request?

RQ3 What factors affect the time required to process a pull request, until it is merged?

Our study is based on data from the Github collaborative develop-

ment forge, as made available through the GHTorrent project [17]. Using it, we first explore the use of pull requests across all projects in Github. We then examine 88 carefully selected Ruby, Java and Scala projects (in total, 37,492 pull requests), and identify, using machine learning tools, common factors that affect pull request lifetime and merging. The results show that pull request *acceptance* can be predicted with high accuracy (> 90%), while it is mostly dependent on on the parts of the system affected by the pull request. On the other hand, pull request *merge time* can be predicted with reasonable accuracy (> 70%), while it is mostly dependent on the ensuing discussion, the size of the project and its test coverage. Both results can be utilized by developers to submit better pull requests and project managers to prioritize pull requests, while it opens new opportunities for research, such as building automated tools for pull request triaging.

2. BACKGROUND

2.1 Pull-based development

Since their appearance in 2001, distributed version control systems (DVCS), notably Git, have revolutionized the way distributed software development is carried out. Driven by pragmatic needs, DVCSs were designed from scratch to work as advanced patch management systems, rather than versioned file systems, the then dominant version control paradigm. In most DVCSs, a file is an ordered set of changes, the serial application of which leads to the current file state. The changesets can originate from a local filesystem or a remote host; tools facilitate the acquisition and application of changesets on a local mirror. The distributed nature of DVCS enables a pull-based development model, where changes are offered to a project repository through a network of project forks; it is up to the repository owner to accept or reject the incoming pull requests.

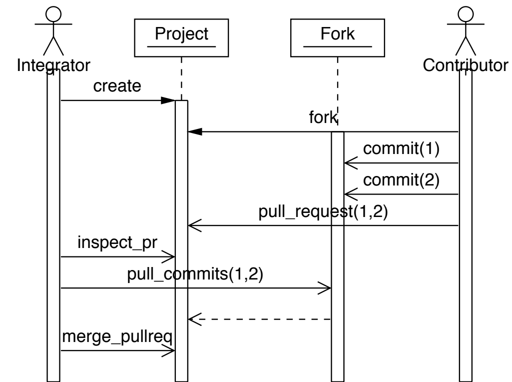
The development models afforded by DVCSs are a superset of those in centralized version control environments [31, 8]. With respect to receiving and processing external contributions, the following strategies can be employed:

Shared resporitory Developers share a common repository, with read and write permissions. To work, they clone it locally, modify its contents, potentially introducing new branches, and push their changes back to the central one. To cope with multiple versions and multiple developers, larger projects usually adopt a *branching model* i.e., an organized way to accept and test contributions before those are merged to the main development branch [9].

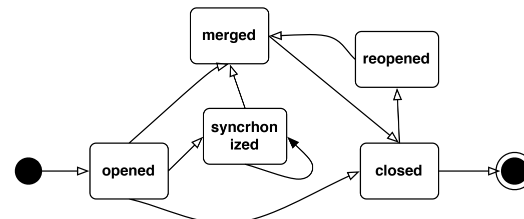
Pull requests The project's main repository is not shared among developers; instead, developers clone ("fork") the repository and make their changes independent of each other. When a set of changes is ready to be submitted to the main repository, they create a *pull request*, which specifies a local branch and a list of commits to be merged with a branch in the main repository. The main repository owner is responsible to inspect the changes and pull them to the project's master branch. If changes are considered unsatisfactory, more changes may be requested; in that case, developers need to update their pull requests with more commits. Furthermore, as pull requests only specify branches from which certain commits can be pulled, there is nothing that forbids their use in the shared repository approach. An overview of the pull request process can be seen in Figure 1(a).

2.2 Pull requests on Github

Github supports all types of distributed development outlined above; however, pull requests receive special treatment. The site is tuned



(a) The pull request process.



(b) The different states of a pull request on Github.

Figure 1: Processing pull requests

to allow easy forking of projects by external users, while automating the generation of pull requests through automatic comparison of project branches. Similarly to Git-based pull requests, a Github pull request contains a list of commits to be merged. The list of commits is automatically updated when newer commits have been added to the forked repository after the pull request has been created. Each Github pull request has an attached implicit state machine (see Figure 1(b)), which is automatically updated by Github as users manipulate the pull request.

By default, pull requests are submitted to the base repository for review. As a result of the discussion, pull requests can be updated with new commits or be closed as redundant or uninteresting. To merge a pull request, a user must be part of the main project team. The versatility of the Git tool suite enables pull requests to be merged in various ways, presented below sorted by the amount of preservation of the original source code properties:

Through Github facilities. Github can automatically verify whether a pull request can be merged without conflicts to the base repository. When a merge is requested, Github will automatically apply the commits in the pull request and record the merge event. All authorship and history information is maintained in the merged commits.

Using Git merge. When a pull request cannot be applied cleanly or when project-related policies do not permit automatic merging, a pull request can be merged using plain Git utilities, using the following techniques:

- *Branch merging:* The remote branch containing the pull request commits is added as a source to a local repository. The

remote branch is merged to a local upstream branch, which is then pushed to the central repository or published for further pulling by other developers. Both history and authorship information are maintained, but Github cannot detect the merge in order to record a merge event.

- *Cherry-picking:* Instead of merging all commits, the merger picks specific commits from the remote branch, which then applies to the upstream branch. The commit unique identifier changes, so exact history cannot be maintained, but authorship is preserved.

A technique that complements both of the above is *commit squashing*: when the full history is not of interest to the project, several consecutive commits are merged into a single one on the pull request branch, which can then be merged or cherry-picked to the upstream branch. In this case, the author of the commit can be different from the person that applied the commit.

Committing the patch. The merger creates a textual difference between the upstream and the pull request branch, which then applies to the upstream branch. Both history and authorship information are lost.

3. EXPERIMENTAL DESIGN

In this section, we present the research approach and datasets we use to answer the research questions.

To answer our research questions, we distinguish two phases. The first phase primarily aims at answering RQ1, addressing the use of the pull-based development model among users of distributed version control systems. In the second phase, we use the lessons learned from answering RQ1 to understand which factors affect the decision and time required to merge.

3.1 Github Data

For both phases, we use data obtained through the Github API. In particular, we do this by means of the GHTorrent dataset [17]. GHTorrent is an off-line mirror of the data offered through the Github API. The Github API data come in two forms; a streaming data flow lists events, such as forking or creating pull requests, happening on repositories in real time, while a static view contains the current state of entities. To obtain references to the roots of the static view entities, the GHTorrent project follows the event stream. From there, it applies a recursive dependency-based parsing approach to yield all data offered through the API. The data is stored in unprocessed format, in a MongoDB database, while metadata is extracted and stored in a MySQL relational database. The GHTorrent dataset covers a broad range of development activities on Github, including pull requests and issues. Up to February 2013, more than a million pull requests from more that a hundred thousand projects have been collected.

3.2 Phase I: Pull Request Usage

To asses the popularity of the pull-based development model (RQ1), we collect descriptive statistics on the use of pull request in Github. In particular, we investigate such questions as how many projects actually make use of pull requests, how many of the projects are original repositories (versus, e.g., clones), and what the relation is to Github's issue tracking facilities. The outcomes are in Section 4.

3.3 Phase II: Pull Request Acceptance

The insights obtained from the analysis for RQ1 can be used to address RQ2 and RQ3.

Answering RQ2 and RQ3 first of all requires a dedicated dataset of projects that have a sufficiently long history of using pull requests. This dataset is described in Section 5.

Given this dataset, we answer RQ2 and RQ3 by determining a set of suitable candidate features by consulting related work in the fields of patch submission and bug triaging. Then, we clean it up through cross-correlation analysis to obtain a set of features with maximum predictive power (Section 6). Using the extracted data for the extracted features, we perform a detailed statistical analysis of pull request characteristics (Section 7).

Using the identified features, we build prediction models, one per research question, which we use to train a set of classification algorithms to retrieve the dominant ones. Prior to running the classification algorithms, we labeled each pull request with a binary attribute; in the case of the merge decision classification task, the attribute signifies whether the pull request has been merged. For the merge time task, we first filter out pull requests that have not been merged and then split the remaining data points in two bins, slow and fast, using the mean time to merge, so that we get two groups of equal number of data points.

At a high level, the process to retrieve the dominant features for both problems consists of two steps: i) we run each dataset through 4 well known classification algorithms, namely Random Forests (`randomforest`), Support Vector Machines (`svm`), Binary Logistic Regression (`binlogregr`) and Naïve Bayes (`naivebayes`), and ii) we select the best classifier and apply a classifier-specific process to extract the feature importance.

To evaluate the classification performance, we used the Accuracy (ACC) and Area Under the receiver operating characteristic Curve (AUC) metrics. We chose those as opposed to the more popular Precision (PREC) and Recall (REC) ones, following advice in reference [23]. To select the appropriate classification algorithm, we run a 10-fold random selection cross-validation and aggregate the mean values for each classification metric.[1] At each iteration, the algorithm randomly samples 10,000 data points from the whole dataset, train a classifier with 90% percent of the input and use it to predict the remaining 10%. The 10-fold run results also allowed us to evaluate the metric stability across runs. We did not perform any additional tuning to the classification algorithms (Section 8).

We used R to implement and run the experiments.

4. DESCRIPTIVE STATISTICS OF PULL REQUEST USAGE ON GITHUB

Github is a very popular development site. As of February 2013, Github reports more than 5 million repositories and 3 million users. However, not all those projects are active: in 2012, the GHTorrent dataset captured events initiated by (approximately) 1,470,200 users affecting 2,607,550 repositories. The majority of registered repositories are forks of other repositories, special repositories hosting

[1] For the `randomforest` classifier, cross validation is considered redundant [10]. Nevertheless, we did it for consistency across our experiment. The stability of the results across cross validation runs confirm the redundancy of cross-validation.

user web pages (named as `<name>.github.com`), program configuration files (`dotfiles`) and temporary repositories for evaluating Git (`try_git`). In the GHTorrent dataset, less than half (1,177,000 or 45%) of the active repositories are original repositories. From those, 587,800 (or 28% in total) received a commit in 2012.

Pull requests are enabled by default on all repositories opened on Github. On a monthly basis, pull request usage is increasing (Figure 2(a)). From the set of original repositories as described above, 92,600 received at least one pull request in 2012. During the same period, 525,900 original repositories used the shared repository approach, having received commits by more than one developers and no pull requests. The situation is similar during the first two months of 2013; 42,400 repositories received a pull request while 222,967 used the shared repository approach exclusively. While pull request usage is increasing overall, they are not the most popular approach for distributed software development yet.

For those projects that received pull requests in 2012, the mean number of pull requests per project is relatively low at 7.1 (percentiles: 5%: 1, 95%: 21); however, the distribution of the number of pull requests in projects is highly skewed, as shown in Figure 2(b). From the pull requests that have been opened in 2012, 73,07% have been merged, thereby indicating that pull requests in principle can work as a medium for obtaining external contributions. Moreover, even though one might expect that it is the well known projects that receive most pull requests, this is not very well supported by our data: the Spearman rank correlation between the number of watchers of a project and the number of pull requests it has received is $\rho = 0.42$ ($p < 0.001, n = 75011$).

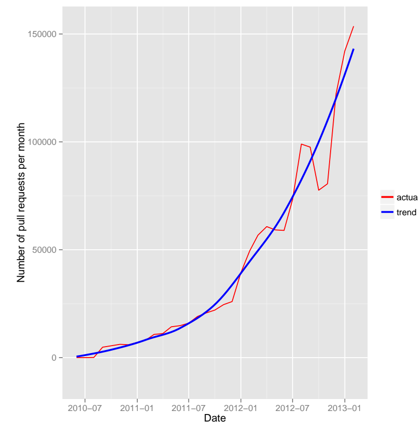
Reviews on a pull request can either target the pull as whole or the individual commits, thereby resembling a code review. Even though any Github user can participate to the review process, usually it is the project community members that do so: only 0.011% of pull request comments come from users that have not committed to the project repository. Pull requests receive comments quite frequently: on average, each pull request receives 12 (quantiles: 5%: 2, 95%: 35) discussion comments.

Issues and pull requests are dual on Github; for each pull request, an issue is opened automatically. Commits can also be attached to issues to convert them to pull requests (albeit with external tools). This duality enables project administrators to treat pull requests as work items, which can be managed using the same facilities used for issues. Across projects that received pull requests in 2012, 35% also received a bug report (not pull-request based) on the Github issue tracker, indicating a strong use of Github's collaboration facilities by both the project and the project community.

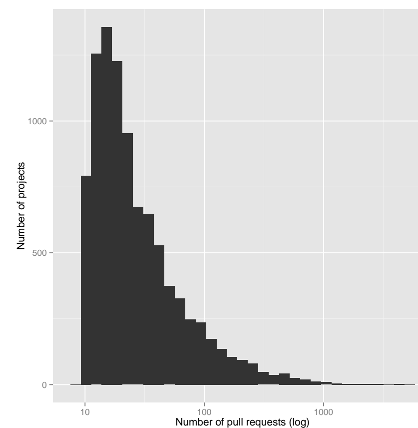
RQ1: 14% of repositories are using pull requests on Github. Pull request usage is increasing, even though the proportion of repositories using them is relatively stable.

5. DATA: PULL REQUEST PROJECT SAMPLE

While an overview of pull request activity can be safely extracted from the GHTorrent dataset, several limitations do not allow us to apply our detailed analysis across all projects. Specifically, projects worth analyzing have a history longer than one year, and it is important to retrieve all information available during the project's life.



(a) Number of pull requests per month



(b) Histogram of pull request frequencies for projects with more than 10 pull requests.

Figure 2: Pull requests on Github

To make the data collection and subsequent analysis practical, we restrict our input data to two lists of projects, a handpicked and a random one. Both lists contain 50 projects of different sizes and languages. To build them, we use the following inclusion criteria:

- Projects should have more than 10 pull requests in total. This is to filter out toy projects.
- Projects should include tests. To measure the effect of testing on pull request acceptance, we could only use projects that include tests which we could measure reliably. For that, we exploited the convention-based project layout in the Ruby (Gem), Java and Scala (both Maven) language ecosystems, so our project selection was limited to those languages.
- Projects should have a committer count larger than the main team member count, to ensure that the project is open to external contributions and that pull requests are indeed used by developers outside the project.

The handpicked list was created by browsing Github's most starred projects per language for projects that met the criteria. The random

list was created by performing a random selection in the GHTorrent database. In cases were the randomly selected projects did not match the criteria (the existence of tests could not be evaluated automatically) or overlapped with the handpicked list, the projects where replaced with other random projects.

After selection, the datasets where merged, the full history (including pull requests, issues and commits) of the included projects was downloaded and features were extracted by querying the GHTorrent database and analysing each project's Git directory. To compensate for pull requests that are merged outside Github, we examined whether at least one of the commits associated with the pull request appears in the list of commits of the target project. In case commit squashing or patch-based merge has occurred, the merge could not be detected, and therefore the pull request was marked as unmerged. After creating the data files, we investigated cases where the pull request merge ratio was significantly less (in some cases, as small as 10%) than the one we calculated across Github (73%), as this might mean that non-authorship preserving merges occurred (see Section 4). This way, we filtered out 12 projects, which we did not replace.

The final dataset on which we run the prediction experiments consisted of 88 projects (43 Ruby, 34 Java, 11 Scala) and 37,452 data points (pull requests).

6. FEATURE SELECTION

The feature selection was based on prior work in the areas of patch submission and acceptance [25, 5, 33, 4], bug triaging [1, 15] and also on the semi-structured interviews of Github developers in Pham et al. [27]. The selected features are split in three categories:

Pull request impact. These features attempt to quantify the impact of the pull request on the affected code base. When examining external code contributions, the size of the patch is affecting both acceptance and acceptance time [33]. There are various metrics to determine the size of a patch that have been used by researchers: code churn [25, 28], changed files [25] and number of commits [13]. In the particular case of pull requests, developers reported that tests in a pull request increases their confidence to merge it [27]. To investigate this, we split the churn feature to two features, namely `src_churn` and `test_churn`.

Project characteristics. These features quantify how receptive to pull requests the project is. If the project's process is open to external contributions, then we expect to see an increased ratio of external contributors over team members. The project's size may be a detrimental factor to the speed of processing a pull request, as its impact may be more difficult to assess. Also, incoming changes tend to cluster over time (the "yesterday's weather" change pattern [16]), so it is natural to assume that pull requests affecting a part of the system that is under active development will be more likely to merge. Testing plays a role in speed of processing; according to [27], projects struggling with constant flux of contributors use testing, manual or preferably automated, a safety net to handle contributions from unknown developers.

Developer. Developer-based features quantify the influence that the person that created the pull request has on the decision to merge it and the time to process it. In particular, the developer that created the patch has been shown to influence the patch acceptance decision [21]. To abstract the results across projects with different developers, we include features that quantify the developer's track record, namely the number of previous pull requests and their acceptance rate; the former has been indicated as a strong indicator of pull request quality [27]. Bird et al. [6], presented evidence that social reputation has an impact on whether a patch will be merged; in our dataset, the number of followers on Github is an interesting proxy for reputation.

All features are calculated at the time a pull request has been closed or merged, to evaluate the effect of intermediate updates to the pull request as a result of the ensuing discussion. Features that contain a temporal dimension in their calculation (e.g., `team_size` or `commits_on_files_touched`) are calculated in a time window of $t - 3$ months, where t is the time the pull request has been closed.

The initial selection contained 23 features. To check whether the selected features are independent enough, and therefore have strong explanatory power, we conducted a pair-wise correlation analysis using the Spearman rank correlation (ρ) metric across all features. We set a threshold of $\rho = 0.8$, above which we eliminated features that are strongly correlated. Using this cutoff, we removed 2 features, `asserts_per_kloc` and `test_cases_per_kloc` as they where very strongly correlated ($\rho > 0.92$) with the included `test_lines_per_1000_lines` feature. We also removed features that could not be calculated reliably at the time a pull request was done (`followers` and `stars`).[2] Finally, we merged similar features (i.e. `doc_files` and `src_files` were merged to `files_changed`).

The post processing phase left us with 13 features, which can be seen in Table 1. The cross-correlation analysis result for the selected features can be seen in Figure 3. In general, very few features are correlated at a value $\rho > 0.2$, while only two, `src_churn` and `files_changed`, are strongly correlated at $\rho = 0.71$. While the correlation is strong, it is below our threshold and it is not definite; therefore we do not remove either feature from the dataset. All results are statistically significant ($n = 37,492, p < 0.001$).

7. PULL REQUEST CHARACTERISTICS

() In this section, we analyze the extracted dataset and provide insights by relating the selected features to our two research questions.

Lifetime of pull requests. After being submitted, pull requests can be in two states: merged or closed (and therefore not-merged). [3] In our dataset, most pull requests (mean: 74.5%) are getting merged. This result is slightly higher than the overall we calculated for Github, but this can be attributed to the fact that the dataset was cleaned from projects for which we could not detect merges accurately.

For merged pull requests, an important property is the time required to process and merge them. The time to merge distribution is highly skewed, with the great majority of merges happening very fast. Measured in days, 95% of the pull requests are merged in 19, 90% in 8 and 80% in only 2.8 days. An interesting observation is that there are many (8,525 or 30% of our sample) pull requests

[2] The Github API does not report a timestamp on when a user followed another user, or when a user stared a project, so it is impossible to know how many followers a user had or stars a project had at a specific time instance.

[3] Intermediate states shown in Figure 1(b) are added by the Github pull request handling process.

Feature	Description and Justification
Pull Request Impact	
`num_commits`	Number of commits in the pull request. A bigger pull request should be slower to examine and therefore to merge.
`src_churn`	Number of lines changed (added + deleted) by the pull request. The more lines changed, the longer a pull request should take to be reviewed.
`test_churn`	Number of test lines changed in the pull request. Pull requests that include tests should be easier to merge, as testing would increase confidence for the merger.
`files_changed`	Number of files touched by the pull request. An indication of the impact of the pull request.
`num_comments`	The total number of comments (discussion and code review). More pull code review comments may indicate a rigorous process, therefore slowing down acceptance.
Project Characteristics	
`sloc`	Executable lines of code at pull request merge time. The bigger the project, the more difficult would be to assess the impact of a pull request.
`team_size`	Number of active core team members during the last 3 months prior the pull request creation. A big team will be faster to process a pull request as each team member will have less work load to handle and will be more specialized.
`perc_external_contribs`	The ratio of commits from external members over core team members in the last 3 months prior to pull request creation. A higher ratio indicates a more open process, which may lead to more accepted pull requests.
`commits_on_files_touched`	Number of total commits on files touched by the pull request 3 months before the pull request creation time. If the pull request touches files in a hotspot, it will be merged faster.
`test_lines_per_1000_lines`	A proxy for the project's test coverage. The more well tested a project is, the faster a pull request could be merged, as applying it and testing its impact would be faster.
Developer	
`prev_pullreqs`	Number of pull requests submitted by a specific developer, prior to the examined pull request. The more pull requests, the better the developer is known to the project.
`requester_succ_rate`	The percentage of the developer's pull requests that have been merged up to the creation of the examined pull request. A high success rate indicates a developer trusted by the project.
`main_team_member`	Whether the developer belongs to the main repository team. Pull requests from main team members should be faster to accept and merge due to increased confidence.

Table 1: Selected features and justification.

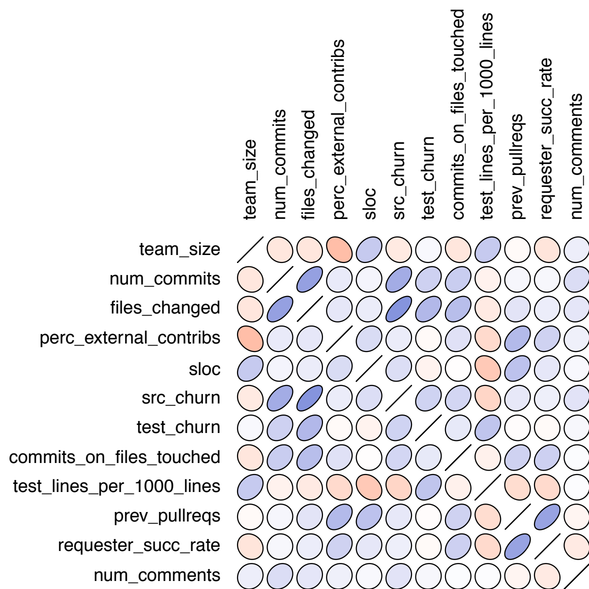


Figure 3: Cross correlation matrix (Spearman) for selected features. The more elliptical a point is, the strongest the correlation. A right slant indicates a positive correlation.

which are merged in under one hour; the majority of such pull requests (60%) come from core team members, while their source code churn is significantly lower than that of the remaining pull requests. If we compare the time to process pull requests that are being merged against those that are not, we can see (Figure 4(a)) that pull requests that have been merged are closed much faster (mean: 6,921 minutes) than unmerged (mean: 39,008 minutes) pull requests. The results of an unpaired Mann-Witney test ($p < 0.001$) showed that this difference is statistically significant, with a relatively significant effect size (Cliff's $\delta : 0.35$). This means that pull requests are either processed fast or left lingering for long before they are closed.

Based on these observations, a question that emerges is whether the pull requests originating from main team members are treated faster than those from external contributors. To answer it, we performed an unpaired Mann-Witney test among the times to merge pull requests from each group. The result is that while the two groups differ in a statistically significant manner ($n_1 = 14316, n_2 = 12463, p < 0.001$), the apparent difference is very small (Cliff's $\delta : -0.11$, see also Figure 4(b)). This means that merged pull requests received no special treatment, irrespective whether they came from core team members or from the community.

On a per project basis, if we calculate the mean time to merge a pull request, we see that in the majority of projects (72%), the time to merge a pull request is less than 7 days. The mean time to merge a pull request per project is not correlated with the project's size, as indicated by a low Spearman correlation value ($\rho = -0.23$) or the project's test coverage ($\rho = 0.34$). Moreover, projects are not getting faster at pull request processing by processing more pull requests; the correlation between the average time to merge and the number of pull requests is weak ($\rho = -0.33, n = 88, p < 0.01$).

80% of pull requests are merged in less than 3 days, while 30% are merged within one hour. Pull requests are treated the same irrespective of their origin (project team or community).

Sizes of pull requests.

A pull request bundles together a set of commits; the number of commits on a pull request is generally less than 10 (95% percentile: 11, 90% percentile: 6, 80% percentile: 3), with a median of 1. The number of files that are changed by a pull request is generally less than 20 (95% percentile: 36, 90% percentile: 17, 80% percentile: 7), with median number of 2. The number of total lines changed by pull requests is on average less than 500 (95% percentile: 1227, 90% percentile: 497, 80% percentile: 168) with a median number of 20.

Except from the project's source code, pull requests also modify test code. In our sample, 33% of the pull requests included modifications in test code, while 4% modified test code exclusively. Of the pull requests that included tests, 72% were merged, which is similar to the average. The presence of tests in a pull request does not seem to affect the merge time either: an unpaired Mann-Witney test shows that while there is a difference in the means of the pull request merge time for pull requests that include tests ($tests = 8831, no_tests = 19061, p < 0.001$), the effect size is very small ($\delta = 0.07$). This result contradicts the findings by Pham et al. [27], where interviewed developers identified the presence of tests as a major factor for the acceptance of pull requests.

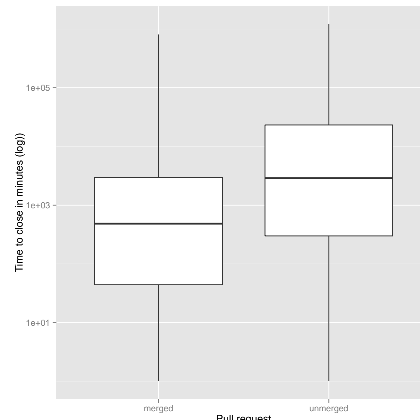
Including test code does not help pull requests to be processed faster.

Discussion and code review. Once a pull request has been submitted, it is open for discussion until it is merged or closed. The discussion is usually brief: 95% of pull requests receive 10 comments or less (80% less than 4 comments). The number of comments in the discussion is neither correlated with whether a pull request will be merged nor with the life time of the pull request.

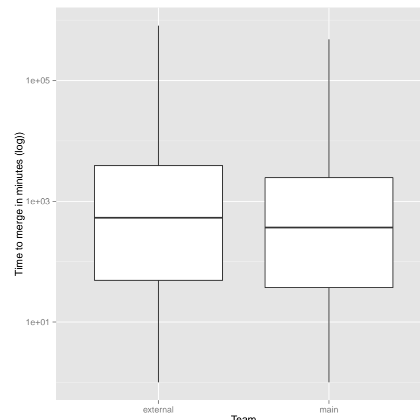
Any Github user can participate to the discussion of any pull request. Usually, the discussion occurs between core team members trying to understand the changes introduced by the pull request and community members (often, the pull request creator) that explain it. Figure 4(c) presents the distribution of external commenters and comments to pull request discussion across projects in our samples. We observe that in most projects, more than half of the commenters are community members. This is not true however for the number of comments; in most projects the majority of the comments come from core team members. One might think that the bigger the percentage of external commenters on pull requests the more open the project is and therefore the highest the percentage of external contributions; a Spearman correlation test quickly indicates that this is not true ($\rho = 0.22, n = 88, p < 0.05$).

8. RESULTS

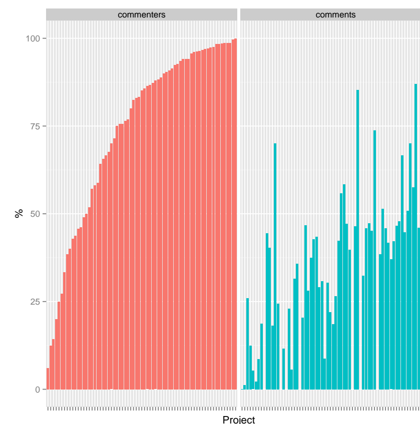
We run the classification processes according to the protocol specified in Section 3.3. The data comprises of 37,492 and 27,892 pull requests for the merge decision and merge time classification task, respectively. Based on the results presented in Table 2, we selected the `randomforest` classification algorithm for both our experiments. For the mergetime experiment, `randomforest` achieved an AUC of 0.73, with a prior probability of 50%. For the



(a) Time to close pull requests.



(b) Time to merge pull requests for internal vs external team members.



(c) Percentage of external commenters and comments per project. Bars are sorted by percentage of external commenters. Project names omitted for clarity.

Figure 4: Dataset characteristics.

classifier	AUC	ACC	PREC	REC
mergedecision task($n = 37,492$)				
binlogregr	0.73	0.63	0.87	0.58
naivebayes	0.69	0.61	0.85	0.57
randomforest	0.94	0.87	0.93	0.93
svm	0.52	0.48	0.72	0.48
merge time task($n = 27,892$)				
binlogregr	0.60	0.55	0.56	0.57
naivebayes	0.58	0.54	0.54	0.57
randomforest	0.73	0.61	0.65	0.70
svm	0.54	0.49	0.45	0.45

Table 2: Classifier performance for the merge decision and merge time classification tasks. The results are means of 10-fold random selection cross validation (sample size 10,000), with 90% train and 10% test data.

mergedecision experiment, the prior probability was 74% which allowed the algorithm to achieve near perfect scores. In both cases, the stability of the AUC metric across folds was good (mergetime: $\sigma_{auc} = 0.019$, mergedecision: $\sigma_{auc} = 0.008$).

To extract the features that are important for each classification task, we used the process suggested in reference [14]. Specifically, we run the algorithm 50 times on the full dataset for each experiment, using a large number of generated trees (2000) and trying 5 random variables per split. We then used the mean across 50 runs of the Mean Decrease in Accuracy metric, as reported by the R implementation of the random forest algorithm, to evaluate the importance of each feature. The results can be seen in Figure 5.

Finally, to validate our feature selection, we rerun the 10-fold cross-validation process with increasing number of predictor features starting from the most important one per case. In each iteration step, we add to the model the next most important feature. We stop the process when the mean AUC metric is within 2% from the value in Table 2 for each task. The selected set of features should then be enough to predict the classification outcome with reasonable accuracy, and therefore can be described as important.

For the mergedecision task, the feature importance result is dominated by the `commits_on_files_touched` feature. By re-running the cross validation process, we conclude that it suffices to use the features `commits_on_files_touched`, `sloc` and `files_changed` to predict whether a pull request will be merged (AUC: 0.93, ACC: 0.86). Therefore, we can conclude that the decision to merge a pull request is affected by whether it touches an actively developed part of the system (a variation of the "yesterday's weather" hypothesis), how large the project's source code base is and how many files the pull request changes.

> *RQ2: The three main factors that affect the decision to merge a pull request are: i) How active the area affect by the pull request has been recently ii) The size of the project iii) The number of files changed by the pull request.*

For the mergetime task there is again a dominating feature, but the difference is not as pronounced as in the previous case. However, if we re-run the cross-validation process we see that the `num_comments` along with the `sloc` and the `test_lines_per_1000_lines` are enough to enable `randomforest` to reach similar performance levels (AUC: 0.74, ACC: 0.65), as with the full feature set.

From the results, we can conclude that the discussion comments and the size of the project play a significant role on how fast pull requests are processed.

> *RQ3: The three main factors that affect the time to merge a pull request are: i) The number of discussion comments ii) The size of the project iii) The project's test coverage.*

9. DISCUSSION

9.1 The pull-based development model

Development turnover. One of the promises of the pull request model is fast development turnover, i.e., the time between the submission of a pull request and its acceptance in the project's main repository. In various studies of the patch submission process in projects such as Apache and Mozilla, the researchers found that the time to commit 50% of the contributions to the main project repository ranges from a few hours [30] to less than 3 days [33, 4]. Our findings show that the majority (80%) of pull requests are merged within 3 days, while a very significant number (30%) are merged within one hour (independent of project size). These numbers are significantly faster indicating that pull requests are more efficient than traditional email-based patches. What's more important is that it is project-related factors that affect the turnover time, rather than characteristics of the pull request. This means that it is mostly up to the project to tune its processes (notably, reviewing and testing) for faster turnover.

> *Pull requests are faster to merge than email-based patches. Projects can tune their reviewing and testing processes for faster turnover.*

Attracting contributions. Pham et al. [27], mention that pull requests make casual contributions straightforward through a mechanism often referred to as "drive-by commits". As the relative cost to fork a repository is negligible on Github (54% of the repositories are forks), it is not uncommon for developers to fork other repositories to perform casual commits, such as fixes to spelling mistakes or indentation issues. In addition, Github provides web based editors for various file formats, using which any user can edit a file in another repository; Behind the scenes, Github will fork the repository and ask the user to create a pull request to the original one. Such commits might be identified as pull requests that contain a single commit from users that are not yet part of the project's community. Even under this naïve definition, 7% of pull requests in 2012 can be classified as drive by commits. Moreover, 3.5% of the forks were created for the sole purpose of creating a drive-by commit. More work needs to be done for the accurate definition and assessment of the implications of drive-by commits, which we defer for future work.

Crowd sourcing the code review. An important part of the contribution process to an open source project is the review of the provided code. Rigby and German [29], report that 80% of the core team members are also participating in the code reviews for patches, a number that is also in line with earlier findings by Mockus et al. [24]. In our dataset, we found that *all* core team members across all projects have participated in at least one discussion in a pull request. Moreover, we found that in most projects in our dataset, the community discussing pull requests is actually bigger than the project team members, even though only in few cases does the community contribute more to the discussion overall.

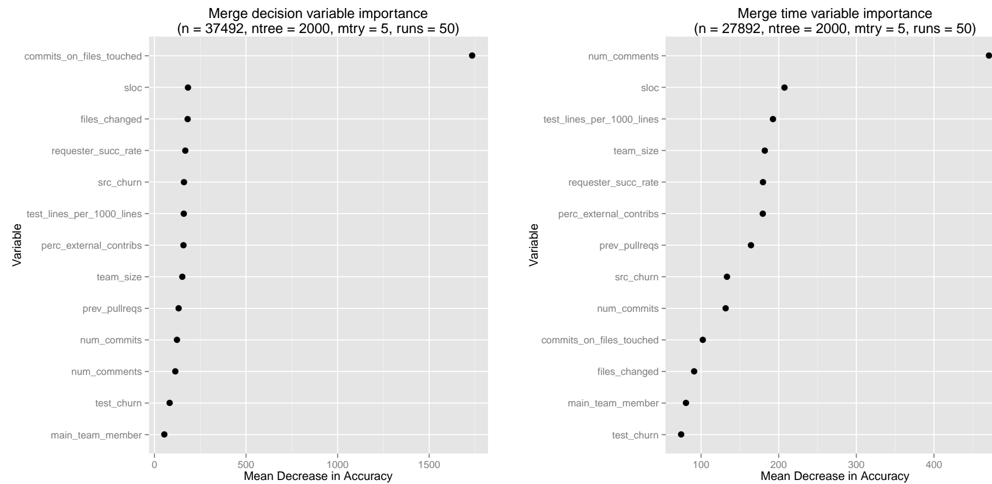


Figure 5: Random forest feature importance for predicting merge decision (a) and merge time (b)

Pull requests can help involve project community to the code review process.

Democratizing development. One of the key findings of this work is that pull requests are not treated differently based on their origin; both core team members and external developers have equal chances to get their pull request accepted within the same time boundaries. Indeed, even the classification models we built assign to the corresponding feature the least importance. In our opinion, this is a radical change in the way open source development is being carried out. Before pull requests, most projects employed membership promotion strategies [20] to promote interested third party developers to the core team. With pull requests, developers can contribute to any repository, without loss of authorship information. The changes that those contributions will get accepted are higher with pull requests; in our sample, and also across Github, more than 70% of external contributions are merged (40% in other studies [29, 33]). Specialized sites such as Ohloh and CoderWall track developer activity and help developers advertise their expertise. We believe that the democratization of the development effort will lead to a substantially stronger shared ecosystem; this remains to be verified in further studies.

9.2 Implications

Contributors. Prospective project contributors will want their contributions to be accepted. Our research shows that pull requests that affect parts of the project that have been changed often lately (are "hot") are very likely to get merged. Also, most pull requests that include test code are accepted, while 80% of the merged pull requests modify three or less files and include patches less than 100 lines long. Therefore, our advice to contributors seeking to add a particular feature or fix a bug is to "keep it short". If the contribution's purpose is to make the contributor known to the project community, it is also beneficial to affect a project area that is hot.

Project Owners. The primary job of the project owner, or owning team, is to evaluate a list of pull requests and decide whether to apply them or not. To make sure pull requests are processed on time, the obvious strategy would be to enforce limits on the discussion as it does not seem to affect the decision to merge a pull request significantly and only adds time overhead; however, this may have adverse effects on the quality of the submitted contributions. A more comprehensive strategy would include a clear articulation of what is expected from a pull request, for example tests or localized changes, on a prominent location in the project's Github page.[4] Moreover, a project owner can invest in a comprehensive test suite; this will improve the time to process a pull request, especially if used with an automated continuous integration system.

A direct application of our results is the construction of tools to help project owners to prioritize their work; since we can predict with very high accuracy whether a pull request will be merged or not, a potential tool might suggest which pull requests can be merged without further examination by the time they arrive. Other tools might examine the quality of the pull request at the contributor's site, and based on the base repository's profile, would provide suggestions for improvement (e.g., more tests, documentation).

9.3 Threats to validity

Internal validity. Our statistical analysis uses random forests as a way to identify and rank cross-factor importance on two response variables. While this is a valid approach [14], we have yet to find other studies in empirical software engineering that follow it. Moreover, the classification scores in the mergetime case are not perfect, so feature ranking may not be exactly the same given a different dataset. Further work is needed on validating the models on data from different sources (e.g., manual pull request handling in the Linux kernel) or projects in different languages.

To analyze the projects, we extracted data from i) the GHTorrent relational database ii) The GHTorrent raw database iii) each project's Git repository. Differences in the data abstraction afforded by each data source may lead to different results in the following cases: i) Intra-branch merges: GHTorrent currently does not record the source and target branch names that are affected by a pull request. Therefore, it is not possible to use the heuristics presented in Sec-

[4] Surprisingly, a quick investigation revealed that no project in our sample had such information in the top-level README.md file.

tion 5 to identify a merged branch, if the project does not use Github facilities to merge. In those cases, we report the pull request as non-merged. In our dataset, 810 (or 2% in total) pull requests are label as non-merged and intra-repository; some of them may actually be merged. ii) Number of files and commits on touched files: The commits reported in a pull request also contain commits that merge branches, which the developer may have merged prior to performing his changes. These commits may contain several files not related to the pull request itself, which in turn affects our results. Therefore, we filtered out those commits, but this may not reflect the contents of certain pull requests.

External validity. In our study, we used merged data from several projects. The statistical analysis treated all projects as equal, even though differences do exist. For example, the larger project in our dataset, Ruby on Rails, has more than 5,000 pull requests while the smaller one, titan, just 24. While we believe that the uniform treatment of the samples led to more robust results in the classification experiment, variations in pull request handling among projects with smaller core teams may be ironed out. The fact that we performed random selection cross-validation (instead of the more common sliding window version) and obtained very stable prediction results is, nevertheless, encouraging.

9.4 On Replication

During the execution of this study, we invested significant effort to make it replicable. This study has been conducted using the GHTorrent dataset and custom-built Ruby and R tools. We share the tools, original data sources and extracted data files on the Github repository `gousiosg/pullreqs`.[5] The execution of all tools is scripted and it is possible to run it in sequence automatically. We invite other researchers to use the tools and data to replicate the study or create new studies based on it.

10. RELATED WORK

Software development can be distributed across various dimensions [18]; among them, physical distribution distributes programming tasks across people collaborating remotely, while temporal distribution distributes tasks to people working in different time zones. Distribution of development activities has been initially thought to hinder collaboration [19, 3], even though subsequent studies have shown that it does not pose significant threats to project quality [32, 26, 7]. Access to online collaboration tools such as version control systems and bug databases have been identified as a necessary requirement for collaborative software development [11]. Moreover, distributed collaboration on software artifacts can be facilitated through awareness building tools [12, 22].

Arguably, the first study of DVCS systems as input for research was done by Bird et. al in [8]. One finding related to our work is that maintaining authorship information leads to better identification of the developer's contributions. Bird and Zimmermann [9] investigated the use of branches in DVCSs (in Section 2, we refer to this DVCS use as "shared repository") and found that excessive use of branching may have a measurable, but minimal, effect on the project's time planning. On the other hand, Barr et al. [2] find that branches offer developers increased isolation, even if they are working on inter-related tasks. Finally, Shihab et al. [31] investigate the effect of branching on software quality; they find that misalignment of branching structure and organizational structure is associated with higher post-release failure rates.

[5]Note to reviewers: To be available online after acceptance

This work builds upon a long line of work on patch submission and acceptance. In reference [24], Mockus et al. presented one of the first studies of how developers interact on large open source projects in terms of bug reporting. Bird et al [5] introduced tools to detect patch submission and acceptance in open source projects. Weißgerber et al. presented an analysis of patch submission, where they find that small patches are processed faster and have higher change to be accepted into the repository. Baysal et al. [4] find that 47% of the total patches make it into the source code repository, a number much lower than our finding for pull requests (74%).

11. CONCLUSION

We have presented an empirical investigation of how the pull-based development model works in practice. We explored its use on the Github source code hosting site and then focused on the factors that affect pull request acceptance and processing time. This work makes the following contributions:

- A statistical analysis of pull request usage on Github. We find increasing usage of pull requests through time.
- A statistical analysis of several characteristics of pull requests. We find that pull requests are treated equally irrespective of whether they originate from the project's main team or the community, that projects are not getting faster at processing pull requests and that the inclusion of tests in a pull request does not directly affect whether it will be merged or not.
- The main factors that affect pull request acceptance and processing time.
- A carefully constructed dataset and a statistical toolkit for performing analysis of pull request usage.

This work is the first to explore the emerging paradigm of pull-based distributed software development. As such, while comprehensive, is not complete. Further research on the field can include the role and consequences of drive-by commits, the formation of teams and management hierarchies in a seemingly flat workspace, the role of testing in assessing pull request impact, and replications in more rigid work environments (e.g., companies).

Acknowledgements

The authors would like to thank Efthimia Aivaloglou for her help with optimizing SQL queries and Panos Louridas for reviewing the statistical analysis parts of the paper. This work is partially supported by Marie Curie IEF 298930 — SEFUNC.

12. REFERENCES

[1] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *ICSE '06*, pages 361–370, New York, NY, USA, 2006. ACM.

[2] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu. Cohesive and isolated development with branches. In *FASE' 12*. Springer, 2012.

[3] R. Battin, R. Crocker, J. Kreidler, and K. Subramanian. Leveraging resources in global software development. *Software, IEEE*, 18(2):70 –77, mar/apr 2001.

[4] O. Baysal, R. Holmes, and M. W. Godfrey. Mining usage data and development artifacts. In *9th IEEE Working Conference on Mining Software Repositories (MSR), 2012*, pages 98–107. IEEE, 2012.

[5] C. Bird, A. Gourley, and P. Devanbu. Detecting patch submission and acceptance in oss projects. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 26, Washington, DC, USA, 2007. IEEE Computer Society.

[6] C. Bird, A. Gourley, P. Devanbu, A. Swaminathan, and G. Hsu. Open borders? Immigration in open source projects. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 6, Washington, DC, USA, 2007. IEEE Computer Society.

[7] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. Does distributed development affect software quality?: an empirical case study of Windows Vista. *Commun. ACM*, 52(8):85–93, Aug. 2009.

[8] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining Git. In *MSR '09: Proceedings of the 6th IEEE Intl. Working Conference on Mining Software Repositories*, pages 1–10, Vancouver, Canada, 2009.

[9] C. Bird and T. Zimmermann. Assessing the value of branches with what-if analysis. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 45:1–45:11, New York, NY, USA, 2012. ACM.

[10] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[11] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, CSCW '06, pages 353–362, New York, NY, USA, 2006. ACM.

[12] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb. Social coding in Github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, CSCW '12, pages 1277–1286, New York, NY, USA, 2012. ACM.

[13] B. Fluri, M. Wursch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[14] R. Genuer, J.-M. Poggi, and C. Tuleau-Malot. Variable selection using random forests. *Pattern Recognition Letters*, 31(14):2225 – 2236, 2010.

[15] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 52–56, New York, NY, USA, 2010. ACM.

[16] T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: guiding early reverse engineering efforts by summarizing the evolution of changes. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 40 – 49, sept. 2004.

[17] G. Gousios and D. Spinellis. GHTorrent: GitHub's data from a firehose. In *MSR '12: Proceedings of the 9th Working Conference on Mining Software Repositories*, pages 12–21. IEEE, June 2012.

[18] D. Gumm. Distribution dimensions in software development projects: A taxonomy. *Software, IEEE*, 23(5):45 –51, Sept. 2006.

[19] J. D. Herbsleb and R. E. Grinter. Architectures, coordination, and distance: Conway's law and beyond. *IEEE Softw.*, 16(5):63–70, Sept. 1999.

[20] C. Jensen and W. Scacchi. Role migration and advancement processes in OSSD projects: A comparative case study. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 364–374, Washington, DC, USA, 2007. IEEE Computer Society.

[21] G. Jeong, S. Kim, T. Zimmermann, and K. Yi. Improving code review by predicting reviewers and acceptance of patches. *Research on Software Analysis for Error-free Computing Center Tech-Memo (ROSAEC MEMO)*, 2009.

[22] M. Lanza, L. Hattori, and A. Guzzi. Supporting collaboration awareness with real-time visualization of development activity. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 202 –211, march 2010.

[23] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Trans. Softw. Eng.*, 34(4):485–496, July 2008.

[24] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.

[25] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of ICSE '05*, pages 284–292, New York, NY, USA, 2005. ACM.

[26] T. Nguyen, T. Wolf, and D. Damian. Global software development and delay: Does distance still matter? In *IEEE International Conference on Global Software Engineering, 2008.*, pages 45 –54, aug. 2008.

[27] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider. Creating a shared understanding of testing culture on a social coding site. In *Proceedings of the 35th International Conference on Software Engineering (to appear)*, 2013.

[28] J. Ratzinger, M. Pinzger, and H. Gall. EQ-mine: predicting short-term defects for software evolution. In *Proceedings of FASE'07*, pages 12–26, Berlin, Heidelberg, 2007. Springer-Verlag.

[29] P. C. Rigby and D. M. German. A preliminary examination of code review processes in open source projects. *University of Victoria, Canada, Tech. Rep. DCS-305-IR*, 2006.

[30] P. C. Rigby, D. M. German, and M.-A. Storey. Open source software peer review practices: a case study of the Apache server. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 541–550, New York, NY, USA, 2008. ACM.

[31] E. Shihab, C. Bird, and T. Zimmermann. The effect of branching strategies on software quality. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '12, pages 301–310, New York, NY, USA, 2012. ACM.

[32] D. Spinelis. Global software development in the FreeBSD project. In *International Workshop on Global Software Development for the Practitioner*, pages 73–79. ACM Press, May 2006.

[33] P. Weißgerber, D. Neu, and S. Diehl. Small patches get in! In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 67–76, New York, NY, USA, 2008. ACM.

TUD-SERG-2013-010
ISSN 1872-5392