HAL
archives-ouvertes.fr

Impact of Developer Turnover on Quality in Open-Source Software

Matthieu Foucault, Marc Palyart, Xavier Blanc, Gail Murphy, Jean-Rémy Falleri

To cite this version:

Matthieu Foucault, Marc Palyart, Xavier Blanc, Gail Murphy, Jean-Rémy Falleri. Impact of Developer Turnover on Quality in Open-Source Software. Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Sep 2015, Bergamo, Italy. <10.1145/2786805.2786870>. <hal-01186466>

HAL Id: hal-01186466

https://hal.archives-ouvertes.fr/hal-01186466

Submitted on 27 Aug 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Impact of Developer Turnover on Quality in Open-Source Software

Matthieu Foucault
U. of Bordeaux, LaBRI, France
mfoucaul@labri.fr

Marc Palyart
UBC, Canada
mpalyart@cs.ubc.ca

Xavier Blanc
U. of Bordeaux, LaBRI, France
xblanc@labri.fr

Gail C. Murphy
UBC, Canada
murphy@cs.ubc.ca

Jean-Rémy Falleri
U. of Bordeaux, LaBRI, France
falleri@labri.fr

ABSTRACT

Turnover is the phenomenon of continuous influx and retreat of human resources in a team. Despite being well-studied in many settings, turnover has not been characterized for open-source software projects. We study the source code repositories of five open-source projects to characterize patterns of turnover and to determine the effects of turnover on software quality. We define the base concepts of both external and internal turnover, which are the mobility of developers in and out of a project, and the mobility of developers inside a project, respectively. We provide a qualitative analysis of turnover patterns. We also found, in a quantitative analysis, that the activity of external newcomers negatively impact software quality.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.8 [**Software Engineering**]: Metrics—*process metrics*

Keywords

Mining software repositories, qualitative analysis, software metrics

1. INTRODUCTION

Throughout the evolution of a project, the team contributing to it evolves, with collaborators joining, leaving, or changing their role in the project. This phenomenon of continuous influx and retreat of human resources is called *turnover*. Turnover has been studied in managerial science and human-computer interaction research, with several theories regarding its impact. The most common theory holds that turnover has a negative impact on performance and on the quality of the work, due to a loss of experience [22]. Other theories suggest that turnover has (1) a positive impact since the most dissatisfied members leave the team, and that only the most motivated ones stay in it [27], (2) helps renew experience and knowledge on the team [40], and (3) increases social interactions [10].

In the software development context, developer turnover has been analyzed by Mockus [30] on one industrial project. He found that developers leaving the project had a negative impact on quality but that new members had no effect on it. Our work extends these findings made on an industrial software project by looking at five large open-source software projects. These projects are interesting to study given their extensive use and low barriers to entry and exit for collaborators [15].

To study turnover in open-source, we introduce activity metrics that measure external and internal turnover. By splitting a software project into different modules, we are able to measure not only the arrivals and departures of developers from the project (i.e. external turnover), but also the movement of developers within the project (i.e. internal turnover).

Based on the concepts we define in this paper, we quantify the level of turnover, both external and internal, in open-source software projects. We quantify turnover by measuring the amount of changes performed in the source code by newcomers or leavers, instead of measuring the actual number of developers joining or leaving, as there is a great disparity between developers in open-source projects. We then perform an empirical study on 5 large open-source projects (Angular.JS, Ansible, Jenkins, JQuery and Rails) to provide insights on the relationship among developer turnover and software quality, where quality was measured based on the density of bug-fixing commits. The extraction process for bug-fixing commits is performed manually, to reduce the risk of errors produced by automatic approaches [5, 7, 21], thus limiting the number of projects that can be considered in this paper.

We provide the following contributions, for the five open-source projects mentioned above:

- We provide a curated set of bugs.
- We provide metrics to measure turnover.
- We show the importance of the turnover phenomenon in open-source projects.
- We observe several trends of internal and external turnover.
- We show that there is a relationship between turnover and quality of software modules.

This paper is structured as follows: Section 2 presents the theory and related work. Turnover metrics are defined in

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3675-8/15/08...
http://dx.doi.org/10.1145/2786805.2786870

Section 3. Our research questions are detailed in Section 4 and the methodology we used to build our dataset in Section 5. Our results are then presented in Section 6. Section 7 presents an overview of the main threats to the validity of these results, and finally, Section 8 concludes and presents trails for future work. We produced a replication package which allows to reproduce and extend the results presented in this study. This package, which has been successfully evaluated by the Replication Packages Evaluation Committee and found to meet expectations, is presented Section 10.

2. THEORY & RELATED WORK

As the literature contains different and sometimes contradictory opinions on turnover, we first describe all its possible interpretations. We then present existing work on turnover in collaborative communities and finally research specific to software development.

2.1 Turnover Perception

Member turnover, initially defined as the rate at which individuals leave a project, can be extended to all the changes made to the development team of a project. These modifications of the team can be either external, (i.e., a member leaves or joins the team) or internal (i.e., a member changes her role in the team). Distinct theories regarding the impact of turnover, whether it is external or internal, suggest that it has both positive and negative aspects on a team.

2.1.1 External Turnover

The most common vision holds that external turnover negatively impacts employee performance [22, 43]. Departures lead to a loss of experience and knowledge, but also disrupt the social network and environment of those who remain [4, 11]. Moreover, it induces devoting resources and time to recruit and train new employees.

A second vision considers turnover as a good opportunity for organizations, as leavers are those most dissatisfied with the current organization, and those who remain enjoy better conditions and performance [27].

A last perspective sees moderate levels of turnover as the best organizational performance [3]. When there is no turnover, experience and knowledge are not renewed, and become obsolete and parochial [40]. Introduction of new people is a solution to overcome this situation, as their vision is less established and less redundant with respect to the knowledge possessed by the current team.

2.1.2 Internal Turnover

Internal turnover was defined in traditional organizations as the number of employees who changed function within an organization [20]. Motivations behind such actions are opportunities for career moves to increase income and autonomy as well as getting new responsibilities and expressing new skills [41]. Kanter et al. pointed out that members had lower aspirations and involvements in their work when mobility was blocked [26]. Thus, internal mobility is commonly supported to maintain members commitment to the organization.

2.2 Turnover in Collaborative Platforms

Turnover has been studied in online communities and collaborative platforms where participants are free to enter or leave at any moment without any cost. In the English Wikipedia, high turnover is even the norm with sixty percent of editors contributing only for a single day [32]. Ransbotham et al. suggested that collaboration success can be reached thanks to moderate levels of turnover [36], provided that the level of novel knowledge exceeds the loss of existing knowledge held by departing people. Similarly, Dabbish et al. discovered that membership turnover might bring fresh levels of activity and liveliness in a community which leads to increased participation [10]. Inversely, Qin et al. observed that departures of WikiProjects contributors has a negative effect on the community and causes social capital losses [35].

2.3 Turnover in Software Development

Developer turnover in open-source software projects was studied mainly to understand developers motivations to contribute. Yu et al. suggested that personal expectation plays a role in project retention, and that turnover is partially explained by dissatisfaction [45]. Hynninen et al. conducted a survey with developers and suggested that their departures from a project can be a manifestation of low organizational commitment [23]. A study from Schilling et al. unveiled that the level of development experience and knowledge is strongly associated with retention [38]. According to Sharma et al., past activity, age and size of a project as well as developer tenures are important predictors of turnover [39]. These observations are consistent with other classical theoretical models related to job satisfaction [44].

Measures of knowledge loss were suggested by Izquierdo-Cortazar et al [24]. These measures include the evolution of *orphan* lines of code lastly edited by a developer who left the team. They showed that while in some projects, developers devote efforts to maintain code introduced by former developers, in others, they seek to eliminate such code. Robles et al. designed a methodology to compute generations of joining and leaving developers [37]. Finally, Fronza et al. propose a wordle to visualize the level of cooperation of a team and mitigate the knowledge loss due to turnover [17].

Hall et al. conducted a survey with practitioners to unveil that turnover may be related to project success, but however did not define turnover metrics computable by analyzing the history of the project [19]. Mockus found that while departures of members impact the software quality because of the loss of knowledge and experience, newcomers are not responsible for an increase of defects, possibly because they are not assigned to important changes [30]. Mockus also found a relationship between turnover and productivity in commercial projects [29].

Mens et al. explored developer turnover in the GNOME ecosystem with concepts and metrics similar as the ones we use in this paper [28]. They looked at developer turnover at a coarser grain: in their study, internal turnover refers to the mobility of developers between projects of the GNOME ecosystem, while external turnover in their case was associated to developers entering or leaving the GNOME ecosystem. Our study differs from theirs as it we look at a finer granularity: we measure mobility of developer between modules of a project, and in and out of a project. In their study they sought for possible patterns of developer turnover, with the conclusion that this is a highly project-specific phenomenon. They did not, seek for a relationship between turnover and code quality.

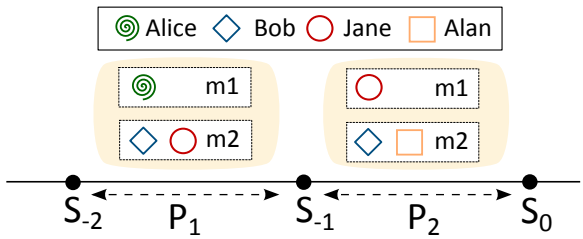

Figure 1: Example of fictive software project containing two modules.

3. TURNOVER METRICS

A software project can have many kinds of turnover. To be able to study different aspects of turnover, we introduce five metrics that can be computed from the source code history of a software project.

3.1 Setup and Requirements

In order to compute turnover metrics, we need to define the periods over which turnover will be computed, as well as how contributors are identified.

3.1.1 Period Selection

We compute developer turnover by comparing the contributors of software modules in two consecutive time periods: P_1 and P_2. These two periods are therefore delimited by three snapshots of the project history: S_0, S_{-1} and S_{-2} such that P_1 is delimited by S_{-2} and S_{-1} and that P_2 is delimited by S_{-1} and S_0 (see Figure 1).

In practice, S_0 is the snapshot for which we want to compute turnover metrics. The selection of the two other snapshots can be based on different approaches. One could consider either the prior releases of the software, snapshots such that periods P_1 and P_2 have the same duration, or snapshots such that periods P_1 and P_2 have the same overall activity in the repository. We study in Section 5 the impact of these choices on turnover computation.

3.1.2 Software Modules and Contributors

Developer turnover is relative to the software project's structure. A developer who has only worked on a few modules in the system that suddenly contributes to more, should be considered as new, or inexperienced, as she moves to new parts of the code base.

We therefore consider that a software project is composed of a finite set M of software modules developed by a finite set of developers who submit their code modifications by sending commits to a shared code repository. Each module is defined by a finite set of source code files. When a developer modifies one of the files of a software module by committing her work, she is contributing to that module. A developer contributes to the software project as soon as she contributes once to any module of the software.

To illustrate all our definitions, we rely on the fictitious software depicted in Figure 1, which is composed of two modules developed over two periods P_1 and P_2. A total of four developers participated to this software between the S_{-2} and S_0 snapshots.

Given a module m, $D_{m,P}$ is the set of developers who made at least one contribution to m during the period P. We obtain with our example D_{m2,P_1} = {Bob, Jane} and D_{m2,P_2} = {Bob, Alan}.

D_t is the set of developers who made at least one contribution to the software during the period t, i.e. $D_P = \bigcup_m^M D_{m,P}$. From our example, we have D_{P_1} = {Alice, Bob, Jane} and D_{P_2} = {Bob, Jane, Alan}.

3.2 Turnover Actors and Metrics

We now provide formal definitions for the sets of developers involved in turnover, and the metrics associated to them. We consider two kinds of developer turnover: external and internal turnover. The developers involved in each kind of turnover are considered to be either newcomers or leavers. Finally, we define stayers, i.e. the developers contributing to both studied periods.

We consider as newcomers the developers who joined the team of a module in the period P_2, whereas leavers are the developers who left the team of a module within the period P_1. This difference between the periods is due to the fact that the intent of our metrics is to evaluate the impact of turnover on the quality of the software at the S_0 snapshot. Thus, newcomers of the P_2 period may influence its quality as their first contributions on a module were between S_{-1} and S_0, and leavers of the P_1 period may influence its quality as the loss of knowledge their departure induce will be perceptible after they left, i.e., after the S_{-1} snapshot.

3.2.1 External Turnover

External turnover refers to the movement of developer in and out of a project.

External newcomers of a module m are the developers who contributed to the module between S_{-1} and S_0, but did not contribute to any module of the project between S_{-2} and S_{-1} (i.e., during the P_1 period). The set of external newcomers is noted EN_{m,P_1,P_2} and is computed as follows:

$$EN_{m,P_1,P_2} = D_{m,P_2} - D_{P_1}$$

In Figure 1, we observe that Alan is a newcomer in $m2$, and that he did not work on any module during P_1. He is therefore an external newcomer, and thus EN_{m2,P_1,P_2} = {Alan}.

External leavers of a module m refer to developers who worked on the module during P_1 but did not contribute to the project at all in P_2. The set of external leavers is noted EL_{m,P_1,P_2} and is computed as follows:

$$EL_{m,P_1,P_2} = D_{m,P_1} - D_{P_2}$$

We observe that only Alice contributed to $m1$ during P_1 but was inactive on the project in P_2. Consequently, EL_{m1,P_1,P_2} = {Alice}.

3.2.2 Internal Turnover

Internal turnover refers to movements of developers inside a project. Even though some developers contribute to a project in both periods P_1 and P_2, they may not work on the same modules in the two periods.

Internal newcomers are the developers who contributed to m in P_2, but not in P_1. However, they contributed to at least one other module than m in this period. They are noted IN_{m,P_1,P_2} and are computed as follows:

$$IN_{m,P_1,P_2} = (D_{m,P_2} - D_{m,P_1}) \cap D_{P_1}$$

Following the previous illustrations, we obtain here $IN_{m1,P_1,P_2} = \{\text{Jane}\}$ and $IN_{m2,P_1,P_2} = \emptyset$.

Internal leavers refer to developers who ceased to contribute to a module m but are still active in the project. This set is noted IL_{m,P_1,P_2} and is computed as follows:

$$IL_{m,P_1,P_2} = (D_{m,P_1} - D_{m,P_2}) \cap D_{P_2}$$

We observe that only Jane modified $m1$ during P_2 but not in P_1, while working on $m2$ during P_1. Consequently, $IL_{m2,P_1,P_2} = \{\text{Jane}\}$.

3.2.3 Stayers

Finally, stayers are the developers who contributed to a module m in both P_1 and P_2. We define the set of stayers for a given module as:

$$St_{m,P_1,P_2} = D_{m,P_1} \cap D_{m,P_2}$$

3.2.4 Metric Definitions

The intention of our metrics is to quantify the impact that the different turnover actors may have on a module's quality at the snapshot S_0 of the project. Due to the large inequalities in the involvement of developers in open-source projects, we cannot quantify turnover by counting the number (or ratio) of developer in each of the categories defined above. Filtering the developers by considering only core or paid contributors is not a viable alternative either. Indeed, peripheral developers as a group still produce a significant amount of contributions, and ignoring these contributions may significantly impact our measurements. Therefore, to measure the impact that each category of turnover actors have on the source code, we use the activity of developers, i.e., the amount of source code they produce.

For a given module m, developer d and period t, we define $A_{m,d,t}$ as the activity of the developer, which we measure using the code churn, i.e. the number of lines of code added or deleted by can be measured with the number of file modifications she performed on the module, or the code churn (i.e., the total number of lines added or deleted) of such modifications. In this paper we only present results obtained using the code churn as an activity measure. However, results obtained with the number of modifications are similar, and are available online (see Section 9).

The five metrics we define are the internal and external leavers activity (ILA and ELA, resp.), the internal and external newcomers ratio (INA and ENA, resp.), and the stayers activity (SA):

$$ILA_{m,P_1,P_2} = \sum\nolimits_{d \in IL_{m,P_1,P_2}} A_{m,d,P_1}$$

$$ELA_{m,P_1,P_2} = \sum\nolimits_{d \in EL_{m,P_1,P_2}} A_{m,d,P_1}$$

$$INA_{m,P_1,P_2} = \sum\nolimits_{d \in IN_{m,P_1,P_2}} A_{m,d,P_2}$$

$$ENA_{m,P_1,P_2} = \sum\nolimits_{d \in EN_{m,P_1,P_2}} A_{m,d,P_2},$$

$$StA_{m,P_1,P_2} = \sum\nolimits_{d \in St_{m,P_1,P_2}} avg(A_{m,d,P_1}, A_{m,d,P_2})$$

4. RESEARCH QUESTIONS

To the best of our knowledge we found no previous study that looked at trends of developer turnover in open-source software projects. Hence the first objective of our study is to seek for such trends, starting with a global view of turnover at the project level, and then focusing on developer turnover on module thanks to the metrics previously defined.

More formally, we seek to answer the following two research questions:

RQ1 Using the concepts of external newcomers and leavers at the project level, is turnover an important phenomenon (in terms of number of developers involved) in open-source software projects?

RQ2 Looking deeply into the project at the module level, is there any patterns regarding the contributions of persistent, internal and external developers?

By answering the aforementioned research questions, we provide an overview of developer turnover both at the project and at the module levels. We then go further by exploring the relationship between developer turnover at the module level and software quality, which we measure based on bug-fix information.

We then answer the following research question:

RQ3 Using the turnover metrics at the module level, is there any relationship with the quality of the software modules?

5. DATASET CONSTRUCTION

Although many automatic techniques are often used to build large datasets, they are all imprecise to a certain extent. Instead of having a dataset with dozens of project containing approximate measures, we chose to focus on the reliability of the information extracted from the dataset. In particular, to answer our research questions, our dataset must meet several requirements:

- The author of each contribution must be clearly identified.
- The source code of the project must be organized into modules.
- A measure of quality must be available for each module.

Each of these criteria is addressed in current research, and software engineering researchers are still developing techniques to extract reliable information from software repositories, as we detail below.

5.1 Authors Identification

Centralized VCS.

The first issue regarding the identification of authors is related to the version control system (VCS) used by the project. In centralized VCSs such as Subversion, a developer must enter her credentials to commit her code to the central repository. Given the large number of contributors to open-source projects, assigning credentials to each of them would be unwieldy, and contributions are therefore submitted via patches, and applied by core developers who have credentials for the repository.

This issue is fixed by the use of decentralized VCSs such as Git, which are able to distinguish the original author of a commit and the developer who added it to the main repository (i.e., the committer) [6]. However, automatically selecting a large number of Git repositories (from hosting

platforms such as GitHub) would not be a suitable process in our case as a non negligible amount of large Git repositories are simply mirrors of Subversion repositories. Well known examples of such repositories include the `gcc` compiler project, or most of the projects hosted by the Apache Software Foundation (eg. the `httpd` server). Moreover, even if a project currently uses Git as a VCS, it may not have been so for all its development history. It is not uncommon for a project, especially older projects, to migrate its code base from one VCS to another throughout its history. This is the case of two projects selected in our dataset, Rails and Jenkins, which originally used Subversion and then migrated to Git. We manually searched commit messages for contents such as "Patch sent by Alice" to determine if at one point these projects were still using Subversion or if they did migrate to Git, and only include the history subsequent to this migration in our analyses.

Identity Merging.

Even when the identity of each contribution's author is reliable, it is possible that a single developer has several identities in the VCS, because of typos, changes in the configuration of the Git client, or a change of email address for instance. This issue is addressed by identity merging, for which Goeminne and Mens address a comprehensive review [18]. Following their recommendations, we use a semi-automatic process which is based on their simple algorithm which has a very high recall. To counter the low precision of the algorithm, we manually review the results of the identity merge algorithm and remove false positive merges.

5.2 Quality Measurement

Our study aims to evaluate the quality of project' modules for a given snapshot. In most software engineering studies the quality of software projects is assessed by looking at the number of bugs fixed by the developers.

Bug Fix Identification.

In order to measure the amount of these bugs, the state-of-the-art technique used in studies mining software repositories consists in parsing the commit messages, looking for the identifier of a bug stored in the project's bugtracker (e.g., "Bug #42") [46]. However, recent work raised concerns regarding the precision and recall of this automatic process, due to the misclassification of issues in the bugtrackers, or imprecision of algorithms linking bugs to source code [21, 5, 7].

Some approaches remove these concerns by only considering information stored in the VCS, and assume that the number of bug-fixing commits is a fair representation of the actual number of bugs within a software module. Unfortunately, to the best of our knowledge, no automatic approach has a satisfactory precision to produce reliable statistics. For instance, among the best automatic approaches, the ones developed by Tian et al. [42] and Mockus et al. [31] have a precision of only 53% and 61%, respectively, in the evaluated benchmarks, which in our case would have unpredictable effect on the number of bug-fixing commits identified, and would be a non-negligible bias to our study.

As we did not find a suitable automatic approach we chose to manually analyze commits to constitute our dataset to the detriment of the number of projects that we were able to include in it. Our manual approach therefore aims to identify commits that are true bugfixes.

Maintenance Branches.

To have measures which are representative of the quality of the code at a given snapshot or release, we need to isolate post-release bugfixes from development bugfixes. Post-release bugfixes for a snapshot S_0 are commits that fix a bug which was in the project's code at the snapshot S_0, while development bugfixes performed after the snapshot S_0 may have been introduced between the snapshot S_0 and the time of the bugfix. If the development history of a project is linear (i.e. if all the commits are performed on a single branch), isolating one category of commits from the other may be cumbersome and imprecise. Therefore, another constraint is added to the projects to include in our dataset: the release S_0 must have a dedicated maintenance branch, sometimes called long time support (or LTS) branch, where the only commits performed in it aim to improve the code quality of the release S_0. These maintenance branches differ from development branches. They usually do not contain new features. The operations performed in such branches are mainly bug-fixing, documentation, optimizations, or compatibility updates related to third party dependencies (e.g., the 2.3.x maintenance branch of Rails contains updates related to new versions of the Ruby programming language). Moreover, we restrict our search to maintenance branches where no commit was performed for the past six months, in order to have branches where most of the bugs were had time to get fixed.

Bug Fix Classification.

Our definition of a bug-fixing commit includes any semantic changes to the source code which fixes an unwanted behavior. The type of bugs considered includes any arithmetic or logic bug (e.g., division by zero, infinite loops, etc.), resource bugs (e.g., null pointer exceptions, buffer overflows, etc.), multi-threading issues such as deadlocks or race conditions, interfacing bugs (e.g., wrong usage of a particular API, incorrect protocol implementation or assumptions of a particular platform, etc), security vulnerabilities, as well as misunderstood requirements and design flaws.

We identified bug-fixing commits manually, discarding commits where new features are implemented. We choose to ignore commits where performance optimizations are performed, as we consider performance issues as a different aspect of code quality. Moreover, we also ignore commits that resolve compatibility issues due to the evolution of a third-party dependency, as these bugfixes are not due to the lack of quality of the changed code, but to the modification of an external requirement. Finally, it occurs that bug-fixing commits are lated discarded by the developers due to a regression introduced by the bugfix. In such cases, the developers perform a "revert" operation of such commits, and we ignore both the "revert" and the "reverted" commits.

We consider that bug-fixing commits are atomic, in the way that we do not consider the possibility that a bug-fixing commit may in fact include two bug-fixes. Moreover, if a bug-fixing commits affects two modules, the number of bug-fixing commits will be incremented in both modules.

5.3 Code Modularization

In this study, we use metrics that target software modules. Breaking a software system into modules is known to be a hard task that requires some subjective choices [33]. We consider two heuristics for determining software modules, such as its organization within files and directories or the co-change activity. We present here the different sets of modules based on these heuristics.

5.3.1 Using the Directory Structure

The first modularization approach we consider is based on the directory structure of the system, in which software modules are defined to be either a file or a directory, with the possibility to include or not its subdirectories. We chose not to simply extract a modularization based on the directory structure, instead we manually inspected the directory structure of each project to select a suitable level of granularity so that a module includes similar features, based on file and directories names, and on the information found in projects configuration files. To overcome the bias of having a single judge for the module decomposition, we asked three members of our research group (three PhD students in software engineering) to provide, for each of the five projects in the corpus, a list of software modules. The three judges then met to merge their results. They agreed on the granularity of most of the modules of projects such as JQuery, Angular.JS and Ansible, while agreement on Jenkins and Rails was initially reached by only two judges, the third one having chosen a coarser granularity. As the decisions made by the judges may be different than the developers of the projects, we tried to contact their core developers to confirm our decompositions, using the official mailing lists and/or IRC channels of each project. Unfortunately, we did not obtained any answers.

5.3.2 Using the Co-change Activity

The second modularization technique we use considers that source code files that are changed together (i.e. in the same commit) belong to the same module, regardless of the directory structure of the project. We use an automated process that consists in building the co-change graph of the project, which is a weighted, undirected graph where each vertex is a source code file of the project, and the weight of an edge is equal to the number on commits where both files were modified together. To determine the modules, we used two algorithms aiming at building communities in a graph [9, 34]. Both algorithms produced a relatively low number of modules (less than ten) in the projects developed in Javascript (Angular.JS and JQuery), which is due to the fact that Javascript projects tend to have fewer, larger files compared to projects in languages such as Java. Therefore, these decomposition allow to produce statistical results on only three projects. As the results obtained with this modularization algorithms are similar to the ones obtained with the manual decomposition based on directories structure, they are not presented in this paper. However, they are available in our additional results online (see Section 9).

5.4 Periods Selection

The computation of turnover metrics for a snapshot S_0 relies on the choice of two periods P_1 and P_2 (Figure 1).

To choose a suitable size for the periods P_1 and P_2, we measured the impact of these periods on the sets of turnover actors (i.e. internal and external leavers and newcomers). The length of the periods P_1 and P_2 may impact the resulting sets of actors, especially if the periods are too short, in which case we may consider as newcomers or leavers developers who stopped contributing to the project for a period of time before re-starting. To assess the impact of this choice we have tested four configurations for the lengths of the periods: one release-based configuration where S_0, S_{-1} and S_{-2} are three following releases of the project, and three time-based configurations where P_1 and P_2 both last for 1, 3 and 6 months.

Using $|P_1| = |P_2|$ may limit our vision in the past. This may for example result in considering some developers as newcomers because they were inactive for sometime, but the length of P_1 is not sufficient to see their previous contributions. On the other hand, if we looked at the whole history of the project to check whether developers are newcomers or leavers, we may consider as stayers developers who did not contribute to the project for several years. To quantify the impact of the length of P_1 and P_2, we compute two versions of each turnover set:

- A version with *limited visibility*, where $|P_1| = |P_2|$.
- A version with *full visibility*, where:
  - S_{-2} = is the beginning of the Git repository when computing the sets of newcomers.
  - S_0 is the most recent release available in the project when computing the sets of leavers.

To decide which period size is suitable for our analyses, we chose to measure the similarity between sets of turnover actors computed with limited and full visibility, using the Sorensen-Dice quotient of similarity, which is equal to 1 when two sets are identical, and 0 when they are disjoint [12]. The selected period size is the first period size where the median Dice coefficient is, for all projects and actors sets, above a threshold of 0.75.

The distributions of Dice coefficients obtained for each project are available online (See Section 9). For each period size $|P|$, project and category of turnover actors (e.g., external newcomers), we have a distribution of Dice coefficient, as we computed one Dice coefficient for each module. These distribution show that, with a period of one month, several sets of developers have large differences between limited and full visibility, the worst case being with Angular.JS where sets of external newcomers computed with limited visibility have no intersection with sets computed with full visibility. With $|P| = 3$ months, the distributions are closer to a dice coefficient of 1, but there are still cases where the median Dice coefficient is below the threshold of 0.75, especially with internal turnover. With $|P| = 6$ months, most of the sets of turnover actors are identical whether we use limited or full visibility. Only few modules have a dice coefficient of zero, and the median Dice coefficient for all projects and categories of turnover actors is above the threshold of 0.75 The release period configuration is not stable as the length of time between two releases depends on the roadmap of the project and on the features that are developed.

Therefore, we chose to use the 6 months period for the remainder of our analysis: all the results presented in this paper consider that $|P_1| = |P_2| = 6$ months.

5.5 Resulting Dataset

Our dataset, listed in Table 1 includes five projects, written in four different programming languages. The selected

Table 1: The projects included in our dataset.

Project	Language	Release (S_0)	#Bugfixes	LoC	#Modules
Angular.JS	JavaScript	1.0.0	147	11,041	26
Ansible	Python	1.5.0	62	50,553	29
Jenkins	Java	1.509	74	79,774	60
JQuery	Javascript	1.8.0	46	5,306	23
Rails	Ruby	2.3.2	390	33,919	46

releases are minor releases (i.e., no breaking changes have been performed in the selected development period) in Ansible, JQuery, and Rails. They are major release in Angular.JS and Jenkins. The selected releases are, with the exception of the one in Ansible, considered to be long term supported (LTS) releases. For these LTS releases, bug-fixing commits are backported from the main development branch even after subsequent releases are available. In Ansible, although the maintenance of the 1.5.x releases stopped a couple of week before the availability of the 1.6.0 release, it was performed simultaneously with the development of the 1.6.0 release. This dataset is available online (see Section 9) and can be reused for future studies.

6. RESULTS

6.1 Turnover at the Project Level (RQ1)

In order to characterize developer turnover at the project level we look at the number of external newcomers, external leavers and stayers during the life of each project.

Developers Volatility.

Since we defined $|P_1| = |P_2| = 6$ months we compute the different sets of actors by starting with S_0= 12 months after the earliest version of the project when we know that Git was used as a VCS, and move S_0 toward the end of the project by steps of two weeks. The resulting numbers are presented in Figure 2.

We can observe two types of phases during the life of a project. The first phase that we call the "enthusiastic" phase can only be seen in Angular.JS and Ansible since we are missing the beginning of the other projects as we excluded from the study the period when they were using SVN. During the "enthusiastic" phase (2011-2014 for Angular.JS and 2013-06/2014 for Ansible) the number of newcomers is constantly superior to the number of leavers. At some point projects switch to the second phase that we call the "alternating" phase where either the number of newcomers or leavers is higher than the other one.

In all projects, the number of newcomers and leavers is quite high. Throughout the histories of these projects, at least 80% of developers are either newcomers or leavers. Overall this confirms that turnover in open-source software projects is an important phenomenon.

Stayers Conversion and Motivations.

The number of stayers increase mainly during the "enthusiastic" phase and stay fairly stable during the "alternating" phase. To further understand the evolution of the population of stayers we use the notion of conversion rate that is usually found in marketing. In our case the conversion rate represents the proportion of newcomers that the project was able to keep long enough so they could become stayers. It is equal to the number of developers who were at least once stayer divided by the number of developers in the whole history of the project we look at. The conversion rates for each project are between 8% (Ansible) and 19% (Jenkins and JQuery). Even if it is not in the same proportion for each project we observed that only a low ratio of newcomers become stayers.

To better understand what make developers stay in their project we looked at the top 10 stayers of each projects: developers who were in the stayers set the highest number of times over the project history. We searched their Github and LinkedIn profiles as well as their personal web pages to understand their motivation. We found four categories:

- Developers who are paid by the company that develops the project. For example 7 out of the top 10 stayers of Angular.JS work at Google which maintains the framework.
- Developers who are paid by a company that use the project for their business. It is the case 6 times in the top 10 stayers of Ansible.
- Developers who are consultants on the technology developed within the project. For example 5 out of the top 10 stayers of Rails are consultants.
- Developers who contribute on their spare time without direct or indirect financial interest. Out of the 50 top stayers we looked at only 2 fit that category.

In conjunction to these categories developers were sometimes also the initial creators of the project (6 developers out of 50).

6.2 Patterns of Contributions (RQ2)

The visualizations in Figure 3 represents the turnover metrics[1] computed with $|P_1| = |P_2| = 6$ months and where the S_0 snapshots are the releases mentioned in Table 1 (these releases are also indicated in Figure 2 via vertical lines). We use these visualizations to observe the different patterns of contributions.

In Angular.JS, most of the activity is due to stayer or external leavers. The high amount of external leavers activity is in fact due to the contributions of a single developer, a major contributor who was inactive in the six months prior to the release of Angular.JS 1.0.0.

In Ansible, all categories of developers have similar levels of activity, and all contributed to a wide range of modules. This differs from other projects, especially for external newcomers: all but one module has external newcomers, and these developers often have an important activity. We looked more closely at the module where external newcomers were the most active, which is the module containing "cloud" plugins for Ansible. Among the newcomers making the most contributions, one was hired at Ansible, Inc., and two worked at Rackspace, a managed cloud computing company, and developed an Ansible plugin for the Rackspace cloud storage. These developers were most probably paid to do their contributions, which explains this high level of activity, not present with most of the newcomers.

The last three projects, Jenkins, JQuery and Rails, exhibit the same patterns. In these three projects, internal newcomers are active in most of the modules, while exter-

[1]Figure 3 also contains information related to bugfixes, which are discussed in the next research question.

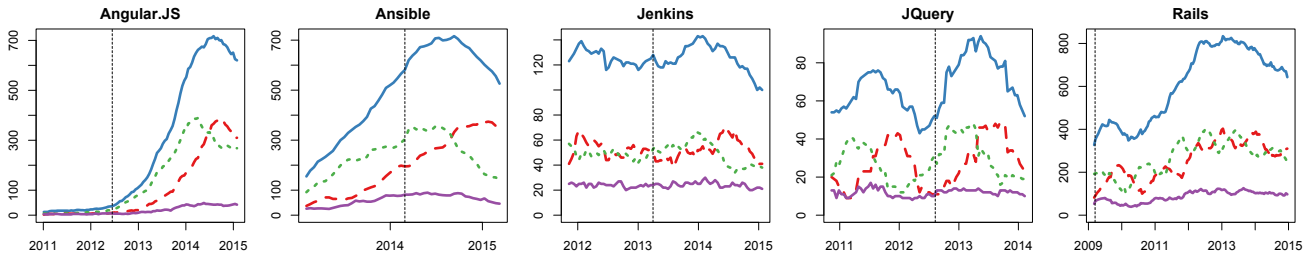

Figure 2: Evolution of developer turnover. The plain blue line (on top) represents the total number of developers, the plain purple line (on the bottom) the number of stayers, the green dotted line the number of external newcomers and the red dashed line the number of external leavers.

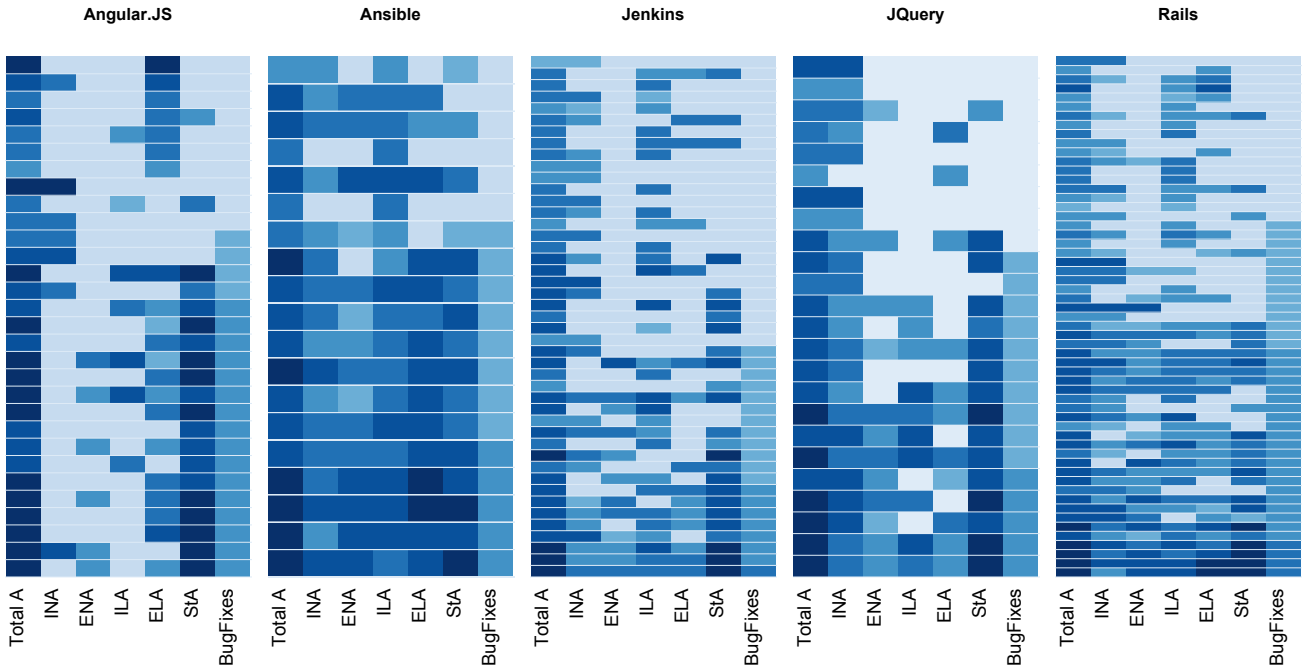

Figure 3: Visualization of developers activity and the quantity of bugfixes for each module. Each horizontal line of blocks represents a module. The darker the color, the higher the metric value.

nal newcomers and leavers are more focused, and do not contribute to more than half of the modules.

Overall there is no module that was changed exclusively by external newcomers. In all the projects of our corpus, the external newcomers always contributed to modules with either internal internal or permanent developers.

6.3 Developer Turnover and Software Module Quality (RQ3)

To answer our third research question, related to the relationship between module turnover and software quality, we use the bug-related information extracted from our dataset (same configuration as RQ2 for P_1 and P_2). We perform Spearman correlation tests between each turnover metric and our quality metric. The quality metric we use is the density of bugfixes per module (i.e., the number of commits that fixed bugs divided by the size of the module). These bugfixes are extracted from the maintenance branch associated to S_0, meaning that there is a high probability that they indeed fix defects that occur in S_0.

Table 2 presents the results of these correlation tests for each project and metric. Correlation coefficients vary from -1 to 1, which corresponds to a perfect negative and positive correlation, respectively. Also, a correlation coefficient of 0 reveals an absence of correlation. We used bootstrap with the BCa statistic to compute 95% confidence intervals of the correlation coefficients [13, 14]. If both ends of a confidence interval are either positive or negative (results highlighted in bold), this means that there is a strong probability that there is a positive or negative correlation, respectively, between the turnover metric and the density of bug-fixing commits.

To have a deeper understanding of the observed correlations, Figure 3 presents a graphical visualization of the turnover metrics and the number of bugfixes. In that Figure, each project is presented by a matrix where each column represents a metric, and each line represents a component of the project. The cell of the matrix then represents the value of the corresponding metrics and darker colors represent higher values.

Table 2: Spearman correlation coefficients between turnover metrics and the density of bug-fixing commits per module. Confidence intervals are computed using bootstrap.

Project	INA	ILA	ENA	ELA	StA	Overall Activity
Angular.JS	[-0.41 , 0.37]	[-0.36 , 0.25]	**[0.12 , 0.69]**	[-0.56 , 0.16]	**[0.23 , 0.83]**	[-0.16 , 0.7]
Ansible	[-0.27 , 0.73]	[-0.31 , 0.65]	[-0.21 , 0.68]	[-0.3 , 0.7]	[-0.15 , 0.76]	[-0.27 , 0.75]
Jenkins	[-0.3 , 0.28]	[-0.18 , 0.42]	**[0.3 , 0.75]**	[-0.05 , 0.51]	**[0.05 , 0.63]**	[-0.01 , 0.6]
JQuery	[-0.1 , 0.69]	**[0.13 , 0.81]**	[-0.02 , 0.73]	[-0.4 , 0.44]	**[0.09 , 0.84]**	**[0.14 , 0.8]**
Rails	[-0.01 , 0.52]	[-0.24 , 0.3]	**[0.09 , 0.57]**	[-0.23 , 0.3]	**[0.14 , 0.58]**	**[0.03 , 0.51]**

The most important information in the results presented in Table 2 is that there is a positive correlation between the External Newcomer Activity and the density of bugfixes. Almost all of the projects exhibit a quite strong correlation. Only Ansible exhibits a weak correlation but, looking at Figure 3, this is certainly due to the fact that external newcomers contributed to almost all of the components, even to the ones that were not the target of bugfixes. This is consistent with the theories exposed in Section 2, which suggest that external turnover has a negative effect on the quality of a team's work. External Leavers Activity on the other hand do not show any statistically significant correlation with bugfix density in Table 2, and the two columns seem completely independent in Figure 3.

Although Table 2 shows three statistically significant correlations between the Internal Leaver and Newcomer Activity and bugfixes, their interpretation when looking at Figure 3 is unclear. As discussed with the previous research question, internal newcomers contribute to the majority of the modules, even the ones without any bugfix.

Finally, as expected, there is a correlation between the activity of persistent developers and the density of bugfixes. This then raises the question of the relative importance of the turnover metrics regarding the software quality, and especially for the External Newcomer Activity (ENA) metrics, as there is no correlation of the internal turnover metrics. To measure how important is ENA we therefore built multiple linear regression models including other metrics, such as the size of modules or the number of developers who contributed to it. Unfortunately, it did not produce exploitable results, due to the low R-squared of the resulting models, and multicollinearity issues exposed by high variance inflation factors of the predictors. We therefore cannot provide sound answer to that point.

7. THREATS TO VALIDITY

The validity of the results presented above is exposed to several threats that we present here.

7.1 External Validity

The generalization of the results is our first concern. On one hand, we selected projects that use different programming languages and that have hundreds of developers. On the other hand, the study was performed on only five projects that were manually selected. To overcome this threat, further studies have to be performed, to confirm and improve the findings presented in this paper. A barrier to achieve these studies is to build curated datasets, following the requirements presented in Section 5.

7.2 Internal Validity

Our metrics assume that the only way developers contribute to a project is by modifying its source code. This is an approximation, as developers can modify other files such as build and documentation files. A project is not confined to its version control system: other types of repositories, such as bug-tracking system or mailing lists, might reveal that some developers considered as newcomers or leavers might be in fact persistent contributors of the project. Turnover metrics based on multiple kinds of repositories are left for future work.

It should be noted that our results should not be interpreted as if external leavers and newcomers developers introduced more bugs than internal. We do not provide or have any information on who introduce bugs because there is, to the best of our knowledge, no reliable algorithm that can identify the author of a bug.

We did not find a reliable way to identify developers with push rights to the repositories. Hence, we could not determine the impact of this feature on the different patterns of turnover. However it should be noted that the projects in our dataset mainly follow a pull-request workflow (a popular approach on Github). With this workflow, even if a developer has push rights she will create a pull-request in the project when making a contribution to benefit from the review mechanism. Thus, except for the developers in charge of merging the pull-requests the other core members do not need to have push rights.

7.3 Construct Validity

In addition, we identify several threats to construct validity from the previous study. On GitHub, developers can submit pull requests, so that the project leaders, who have write permission on the repository, can add their contributions to the project. As the identity of the initial author is maintained through the *pull* operation, she is identifiable even though she does not have access to the main repository. However, as shown in [25] it may happen that a developer discussed with a pull request author to agree on its acceptance. Even though this developer spent time to fix or improve the pull request content, all the credits will go to the pull request author. This may also introduce a bias in the results.

Identifying software modules in a project is not a straightforward task and might be subject to interpretation. Since we could not get the confirmation from the different development teams some modules in our decomposition might be split or merged in comparison to what the development teams would have defined.

Related to the same threat the quality of the software architecture can have an impact on the metrics. Retention

could appear higher in well modularized systems than poorly designed systems where one fix might require changes to many modules. The fact that the results produced with the decomposition based on co-change activity overlaps strongly with the manual decomposition based on directories shows that the impact is negligible for this dataset.

We deliberatively did not rely on the information provided by bugtrackers, as several studies showed that their use can introduce an important bias [21]. The drawback of our technique is that the number of bug-fixing commits may not reveal the actual number of bugs that appeared in the software modules. There may exists bugs that are tedious to fix and remain to be resolved. In addition, the manual analysis has some limits due to the subjective evaluation to decide whether or not a commit is a bug-fixing commit. We only went through a maintenance branch to collect such commits for each project, although it potentially exists bug-fixing commits from the main development branch that have not been backported to the maintenance branch. Finally some bug-fixing commits may fix bugs that were not introduced in the current release but in one of the older releases.

8. CONCLUSION AND FUTURE WORK

In this paper, we propose and investigate metrics to measure turnover in open-source software projects. Our metrics measure how the structure of a group of developers is changing, both internally and externally, for a given period of a software project. We used these metrics on five open-source projects with two objectives: to observe the turnover phenomenon, and to evaluate its relationship with software quality.

We observed that the five open-source projects in our corpus, chosen because of their popularity and success, have a high turnover. This observation disagrees with the conclusions of Hall et al. that recommend to control turnover to improve the success of industrial projects [19]. Our results then suggest that turnover and success may have a different relationship in open-source projects.

Looking at the module level, we show some very interesting turnover patterns. These patterns reveal that the projects of our corpus act differently regarding turnover. For instance, in some projects modules receive contributions only by internal developers with no contribution from stayers. These patterns also show that in all projects external newcomers always work with either permanent or internal developers, who hopefully supervise them. Such an observation opens the room for rules or guidelines that will define how newcomers should be supervised, and how they should contribute to modules of a project [8].

We also found that external turnover has a negative impact on the quality of the modules. This result is consistent with theories that suggest that external turnover has a negative effect on the quality of a team's work. However, it differs from the ones of Mockus [30], as in our case newcomers have a relationship with quality and leavers do not have such relationship, while it was the opposite in Mockus' study. On the other hand, internal turnover has almost no effect. Our observations therefore do not confirm the theories that suggest that internal turnover is beneficial. These findings can be reused by researchers when using software metrics based on the activity of developers on the source code: as the activity of external newcomers has a stronger relationship with quality than the activity of other categories of developers, this may be the only activity worth considering.

Finally, our study and findings lead the way for many kinds of future work:

- Our findings are based on observations made on software modules, with manual observation of patterns and using correlation between the density of bug-fixing commits and the activity of the different categories of turnover actors. As there are no related work that performed such observations on open-source projects, our study needs to be replicated on more projects, which is facilitated by our replication package (Section 10).
- The main limitation of our metrics is the fact that they require a selection of periods. In particular, we shown that the length of the chosen periods has a major impact on the measures, and we therefore provide some insights showing that a time period is adequate in the case of open-source project, with good results with 6 months periods. We then plan to overcome this limitation by developing continuous metrics for turnover, where the discretization of the history is not necessary.
- Our results regarding turnover patterns suggest that the observed patterns are impacted by the motivation of developers, which mainly depends on the fact that they are paid or not. This hypothesis can be evaluated only if it is possible to distinguish paid contributors from volunteers. We then plan to identify the employee of the developers, and then to analyze its relationship with turnover metrics.
- Independently of whether developers are paid or volunteers, they may be core member of the projects, and thus have a higher retention level than other developers, as well as a higher level of activity in the project. Identifying core members of a project may help us understanding the impact of developer turnover on software quality.

9. AUXILIARY MATERIAL

Due to the space constraint of this paper, part of our results are available online [2]. This page includes results regarding the period selection, as well as more detailed versions of Figure 2 and Figure 3.

10. REPLICATION PACKAGE

The dataset built using the methodology presented in Section 5, the code necessary to extract the metrics, as well as additional results are available online, in a replication package that has been successfully evaluated by the Replication Packages Evaluation Committee and found to meet expectations [2]. We describe here the technical aspects of this package, the data produced by the executed software and the installation process of the replication package.

10.1 The Diggit Tool

The software of our replication package relies on the Diggit tool, which supports analysis of Git repositories and which helps manage the analysis process [1]. Diggit manages a set of Git repositories. On each repository, Diggit applies several user specified *analyses*. When all user-specified analyses have been applied, Diggit applies global analyses (called *joins* in the tool) that use the results of all the previously applied analyses to produce final results. Diggit is used within a *diggit folder*, which contains various configuration files, including the list of Git repositories to clone and analyse, the

list of analyses and joins to perform, and additional information that may be used by analyses. This tool is developed in Ruby by two of the authors of this paper; we used version 2.0.2.

10.2 Package Installation and Usage

The replication package is distributed as a VirtualBox virtual machine image. This image is based on a minimal version of a Linux Ubuntu on which only the requirements to replicate the study were installed. The list of commands required to install our replication package from a fresh install of a minimal Ubuntu is also available online.

The package consists of a set of diggit analyses, that are all loaded in a diggit folder in the VM image (this folder can also be generated with a script). The first step of the replication is to clone the five Git repositories to be analyzed using the `dgit clone` command. Then, the `dgit analyses perform` command allows to run, for each of the cloned repositories, the following analyses:

- Extraction of the number of lines of code of each file at release S_0, which is used to compute bug-fixing commits density.
- Computation of the number of lines of code and the number of bug-fixing commits of each module (the list of modules is stored in a configuration file).
- Computation of the activity of each developer on each module.
- Computation of the activity of developers at the project level.

The data produced for each repository is then aggregated by a global analysis that uses the R programming language and produces all the results presented in this paper and in the additional results available online.

10.3 Replication Data

Our replication package also provides data that can be reused for future studies. It includes the information described in Section 5 which is given as input to the analyses described above, and activity information, which can be reused to compute other metrics than developer turnover (such as code ownership for instance [16]).

For each repository, the data provided as input of the analyses is the following:

- The author renaming information.
- The lists of modules extracted with the different modularization techniques.
- The commit id of the S_0 release.
- The commit ids of the bug-fixing commits performed in S_0 release's maintenance branch.

This data is stored in a single JSON file and thus can be easily reused.

Besides the final results provided by the global analysis that are presented in this paper, each analysis produces intermediary results that are stored in a *MongoDB* database. The data stored in this database consist in a monthly measure of activity (code churn) for each developer and module, and each month prior to the S_0 release up to the start of the Git history of the repository.

11. REFERENCES

[1] The diggit git repository analysis tool. `https://github.com/jrfaller/diggit`. Accessed: 2015-07-15.

[2] Replication package - impact of developer turnover on quality in open-source software. `http://se.labri.fr/a/FSE15-foucault`. Accessed: 2015-07-15.

[3] M. A. Abelson and B. D. Baysinger. Optimal and dysfunctional turnover: Toward an organizational level model. *Academy of Management Review*, 9(2):331–341, Apr. 1984.

[4] L. Argote and D. Epple. Learning curves in manufacturing. *Science*, 247(4945):920–924, Feb. 1990.

[5] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering (ESEC/FSE)*, page 121–130, 2009.

[6] C. Bird, P. Rigby, E. Barr, D. Hamilton, D. German, and P. Devanbu. The promises and perils of mining git. In *6th IEEE International Working Conference on Mining Software Repositories, 2009. MSR '09*, pages 1 –10, May 2009.

[7] T. F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Réveillère. Empirical evaluation of bug linking. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013)*, pages 1–10, Mar. 2013.

[8] G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. Who is going to mentor newcomers in open source projects? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 44, 2012.

[9] A. Clauset, M. E. Newman, and C. Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[10] L. Dabbish, R. Farzan, R. Kraut, and T. Postmes. Fresh faces in the crowd: Turnover, identity, and commitment in online groups. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, page 245–248. ACM, 2012.

[11] G. G. Dess and J. D. Shaw. Voluntary turnover, social capital, and organizational performance. *Academy of Management Review*, 26(3):446–456, July 2001.

[12] L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297, July 1945.

[13] B. Efron. Bootstrap methods: another look at the jackknife. *The Annals of Statistics*, page 1–26, 1979.

[14] B. Efron. Better bootstrap confidence intervals. *Journal of the American Statistical Association*, 82(397):171–185, 1987.

[15] K. Fogel. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, first edition, Feb. 2013. http://www.producingoss.com/.

[16] M. Foucault, J.-R. Falleri, and X. Blanc. Code ownership in open-source software. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, page 39:1–39:9. ACM, 2014.

[17] I. Fronza, A. Janes, A. Sillitti, G. Succi, and S. Trebeschi. Cooperation wordle using pre-attentive processing techniques. In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 57–64, May 2013.

[18] M. Goeminne and T. Mens. A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 78(8):971–986, 2013.

[19] T. Hall, S. Beecham, J. Verner, and D. Wilson. The impact of staff turnover on software projects: The importance of understanding what makes software practitioners tick. In *Proceedings of the 2008 ACM SIGMIS CPR Conference on Computer Personnel Doctoral Consortium and Research*, SIGMIS CPR '08, page 30–39. ACM, 2008.

[20] D. S. Hamermesh, W. H. J. Hassink, and J. C. v. Ours. Job turnover and labor turnover: A taxonomy of employment dynamics. Open Access publications from Tilburg University 12-86873, Tilburg University, 1996.

[21] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, page 392–401, 2013.

[22] M. A. Huselid. The impact of human resource management practices on turnover, productivity, and corporate financial performance. *Academy of Management Journal*, 38(3):635–672, June 1995.

[23] P. Hynninen, A. Piri, and T. Niinimaki. Off-site commitment and voluntary turnover in GSD projects. In *2010 5th IEEE International Conference on Global Software Engineering (ICGSE)*, pages 145–154, Aug. 2010.

[24] D. Izquierdo-Cortazar. Relationship between orphaning and productivity in evolution and GIMP. 2008.

[25] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, page 92–101. ACM, 2014.

[26] R. M. Kanter. The impact of hierarchical structures on the work behavior of women and men. *Social Problems*, 23(4):415–430, Apr. 1976.

[27] D. Krackhardt and L. W. Porter. When friends leave: A structural analysis of the relationship between turnover and stayers' attitudes. *Administrative Science Quarterly*, 30(2):242–61, Jan. 1985.

[28] T. Mens, M. Claes, P. Grosjean, and A. Serebrenik. Studying evolving software ecosystems based on ecological models. In *Evolving Software Systems*, pages 297–326. Springer Berlin Heidelberg, Jan. 2014.

[29] A. Mockus. Succession: Measuring transfer of code and developer productivity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 67–77. IEEE Computer Society, 2009.

[30] A. Mockus. Organizational volatility and its effects on software defects. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, page 117–126. ACM, 2010.

[31] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, ICSM '00, page 120–. IEEE Computer Society, 2000.

[32] K. Panciera, A. Halfaker, and L. Terveen. Wikipedians are born, not made: A study of power editors on wikipedia. In *Proceedings of the ACM 2009 International Conference on Supporting Group Work*, GROUP '09, page 51–60. ACM, 2009.

[33] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[34] P. Pons and M. Latapy. Computing communities in large networks using random walks. In *Computer and Information Sciences - ISCIS 2005*, number 3733 in Lecture Notes in Computer Science, pages 284–293. Springer Berlin Heidelberg, 2005.

[35] X. Qin, M. Salter-Townshend, and P. Cunningham. Exploring the relationship between membership turnover and productivity in online communities. 2014.

[36] S. Ransbotham and G. C. Kane. Online communities: Explaining rises and falls from grace in wikipedia. *MIS Q.*, 35(3):613–628, Sept. 2011.

[37] G. Robles and J. M. Gonzalez-Barahona. Contributor turnover in libre software projects. In *Open Source Systems*, number 203 in IFIP International Federation for Information Processing, pages 273–286. Springer US, Jan. 2006.

[38] A. Schilling, S. Laumer, and T. Weitzel. Who will remain? an evaluation of actual person-job and person-team fit to predict developer retention in FLOSS projects. In *2012 45th Hawaii International Conference on System Science (HICSS)*, pages 3446–3455, Jan. 2012.

[39] P. N. Sharma, J. Hulland, and S. Daniel. Examining turnover in open source software projects using logistic hierarchical linear modeling approach. In *Open Source Systems: Long-Term Sustainability*, number 378 in IFIP Advances in Information and Communication Technology, pages 331–337. Springer Berlin Heidelberg, Jan. 2012.

[40] J. D. Shaw, N. Gupta, and J. E. Delery. Alternative conceptualizations of the relationship between voluntary turnover and organizational performance. *Academy of Management Journal*, 48(1):50–68, Feb. 2005.

[41] J. D. Thompson. Organizations in action: Social science bases of administrative theory. SSRN Scholarly Paper ID 1496215, Social Science Research Network, 1967.

[42] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *Software Engineering (ICSE), 2012 34th International Conference on*, page 386–396, 2012.

[43] Z. Ton and R. S. Huckman. Managing the impact of employee turnover on performance: The role of process conformance. Jan. 2008.

[44] S. G. Westlund and J. C. Hannon. Retaining talent: Assessing job satisfaction facets most significantly related to software developer turnover intentions.

Journal of Information Technology Management, 19(4):1–15, 2008.

[45] Y. Yu, A. Benlian, and T. Hess. An empirical study of volunteer members' perceived turnover in open source software projects. In *2012 45th Hawaii International Conference on System Science (HICSS)*, pages 3396–3405, Jan. 2012.

[46] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *International Workshop on Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007*, page 9, May 2007.