How the Evolution of Emerging Collaborations Relates to Code Changes: An Empirical Study

Sebastiano Panichella¹, Gerardo Canfora¹, Massimiliano Di Penta¹, Rocco Oliveto²
¹Department of Engineering, University of Sannio, Benevento, Italy
²Department of Bioscience and Territory, University of Molise, Pesche (IS), Italy

ABSTRACT
Developers contributing to open source projects spontaneously group into “emerging” teams, reflected by messages exchanged over mailing lists, issue trackers and other communication means. Previous studies suggested that such teams somewhat mirror the software modularity. This paper empirically investigates how, when a project evolves, emerging teams re-organize themselves—e.g., by splitting or merging. We relate the evolution of teams to the files they change, to investigate whether teams split to work on cohesive groups of files. Results of this study—conducted on the evolution history of four open source projects, namely Apache httpd, Eclipse JDT, Netbeans, and Samba—provide indications of what happens in the project when teams reorganize. Specifically, we found that emerging team splits imply working on more cohesive groups of files and emerging team merges imply working on groups of files that are cohesive from structural perspective. Such indications serve to better understand the evolution of software projects. More important, the observation of how emerging teams change can serve to suggest software remodularization actions.

Categories and Subject Descriptors
D.2.9 [Software Engineering]: Management—Programming teams.

General Terms
Experimentation, Human Factors

Keywords
Developers’ Communication, Open Source Projects, Mining Software Repositories, Empirical Studies.

1. INTRODUCTION
The organization of developers into teams is crucial for the success of software projects. In industrial projects, teams are often defined and staffed by project managers, that group people based on the needs of a particular task, and on their availability, skills, and attitude to work together. Dynamics are different in open source projects [11] involving developers spread across the world and working in different time zones, often communicating using electronic means such as mailing lists. In essence, developers participating in open source projects are not staffed into teams by project managers. Moreover, the way they collaborate depends on the structure of the open source project, i.e., whether the project is of “cathedral” or “bazaar” type [28]. Generally speaking, developers spontaneously group themselves into “emerging teams”, that can be recognized by observing the developers’ communication network and how developers change source code files. In the rest of the paper, we use such a definition of team.

Bird et al. [6] analyzed social networks built on mailing list communication, and found a high correlation relationship between the level of email activity and the level of activity in source code development. Later on, Bird et al. [7] found that there is a causal consequence between the modularity of a software project and the way developers group into teams. They also found that developers belonging to the same sub-community share a larger proportion of files than developers belonging to different sub-communities.

When a software project evolves, the way emerging teams are formed and operate may change. This is because during its lifetime a project undergoes different kinds of changes, requiring the contribution of different, and possibly new, people. As pointed out by Hong et al. [21], emerging teams reorganization often happens in correspondence to new project releases.

Aim of the paper. Stemming from the above considerations, this paper investigates how emerging teams evolve in open source software projects as people focus on different technical activities, i.e., code-changes. By analyzing how people collaborate through mailing lists and issue trackers, and what files they modify, we investigate whether emerging teams evolve with the aim of working on more cohesive groups of files. Figure 1 provides an overview of our analyses: on the one side, we identify emerging teams in different time periods following software releases, and map teams related to subsequent releases/periods. Then, we analyze what files these teams change in the versioning system, and analyze how the cohesiveness of such files changes when teams split or merge.

Study Overview. The study has been conducted on the evolution history—consisting of data from versioning sys-
tems, mailing lists and issue tracker—of four open source projects, namely Apache httpd\textsuperscript{1}, Eclipse JDT\textsuperscript{2}, Netbeans\textsuperscript{3}, and Samba\textsuperscript{4}. Results of the study indicate that, when emerging teams merge and split, we observe a significant increase of the cohesiveness—measured both structurally and semantically—of files on which such teams work. In other words, developers reorganize themselves in order to work on more cohesive groups of files.

**Paper Contributions.** The contributions of this paper can be exploited in different ways. On the one hand, the natural (re)grouping of people in the project communication might suggest the need to re-structure the software design and/or source code organization. On the other hand, the observation of how the project communication evolve with respect to the source code structure could help project leaders to better monitor and understand the project activities, as well as to provide them recommendations on how and when restaffing project teams.

**Structure of the paper.** Section 2 provides the empirical study definition, formulates the research questions the study is going to address, and details the data extraction and analysis method. Section 3 reports and discusses the empirical study results. Threats to validity are discussed in Section 4, while Section 5 relates the paper to the existing literature. Finally, Section 6 concludes the paper and outlines directions for future work.

2. **STUDY DESCRIPTION**

The goal of this study is to investigate how emerging teams of developers evolve over time in open source software projects. The purpose is to understand how such evolution relates to the cohesiveness of files people change over time. The perspective of the study is of researchers interested to observe the relationship between developers’ communication and activities in software projects. Such observation can be used to devise recommenders for software project managers.

The context consists of the source code history, mailing lists and issue trackers of four open source projects, namely Apache httpd, Eclipse JDT, Netbeans, and Samba. Apache httpd (in the following abbreviated as Apache) is an open-source HTTP server, Eclipse-JDT and Netbeans are Java Integrated Development Environments (IDE), and Samba is a cross-operating system layer for printer and file sharing. We have chosen these four projects to have a reasonable variety in terms of size and application domains: two of them, Apache httpd and Samba, are network applications, while the other two are really the same kind of system, i.e. Java IDE. The latter choice is also motivated to allow comparing projects that are pretty similar in terms of features offered. Also, we had to choose projects for which both mailing lists and issue trackers were available. Table 1 reports key information about the four projects and, above all, of their mailing lists, issue trackers, and versioning systems, namely project URL, time period analyzed, size range in KLOC, list of considered releases, number of mailing list contributors ($Mc$), number of issue tracker contributors ($Ic$), and number of authors indicated in the versioning systems ($Au$). Note that the versioning system adopted for the analyzed systems, i.e. Git, provides explicit information for authors, other than just for committers, although in many cases authors and committers match. In addition, the table reports the intersections $Mc \cap Au$, $Ic \cap Au$ and $(Mc \cup Ic) \cap Au$. Note that the latter is of particular interest for our study, in that it tells us how many authors communicated through the two communication means considered. In particular, the percentage of the authors that participated in discussions in mailing lists and/or issue trackers range between 68% and 80%. As the work of Guzzi et al. [20] pointed out, sometimes core developers’ participation in mailing lists is very low. In our cases, it is of less than 30% of authors for Eclipse JDT and Netbeans. For this reason, we decided to consider in our study both sources of information.

2.1 **Research Questions**

The study aims at addressing the following research questions:

- **RQ1:** *How do emerging collaborations change across software releases?* This research question has a merely exploratory nature, and poses the basis for the subsequent one. It aims at observing how emerging teams evolve, i.e., to what extent these teams merge, split, and recombine. Also, we analyze the proportion of developers that join the project, those that are inactive in a given time period, and those leaving the project, to understand whether different projects exhibit different dynamics in terms of team reorganization.

- **RQ2:** *How does the evolution of emerging collaboration relate to the cohesiveness of files changed by emerging teams?* This is the core research question of the paper, and aims at investigating whether a team split or merger is reflected by change activity being done in more (or less) cohesive groups of files. Previous work [8] suggested that such a structure reflects the system architecture. Such information can provide multiple indications. If a team works on a set of unrelated files, in some circumstances it could be useful to split a team into smaller teams working on cohesive files.

As it will be detailed in Section 2.2, we measure the cohesiveness of files modified by emerging teams using both structural and semantic measures, which have

\[\text{http://httpd.apache.org} \quad \text{http://www.eclipse.org/jdt} \quad \text{https://netbeans.org} \quad \text{www.samba.org}\]
been found to capture different, complementary aspects of software modularity [3].

2.2 Data Extraction Process

To address these research questions, we need two data sources: (i) developers’ communication (extracted from mailing lists and issue trackers), that we use to identify emerging collaboration teams; and (ii) change history, extracted from versioning systems, that is used to determine which files developers work on over time. In the following, we detail how such data sources are used to address the two research questions.

2.2.1 Step 1: Extracting the Communication Network

To analyze the evolution of the emerging teams, we build communication graphs using data from mailing lists and issue trackers.

Mailing list contributors and their social interactions are extracted from messages by identifying the sender name and email address and, whenever available, the recipient/reply-to name and email address. Since messages can be sent to a mailing list rather than to single addresses, we assume that (i) Tom and John communicate if there is a direct message with Tom as sender and John as recipient or vice versa, or (ii) if John answers, even if replying to the mailing list, to a thread initiated by Tom, or which Tom has previously replied to.

Issue tracker contributors and their social interactions are extracted considering the comments reported in each bug by all the participants. More precisely, we build the issue tracker network using an approach similar to what done by Hong et al. [21], i.e., developers who made comments on the same bugs (discussions) are connected.

When building the contributors’ network considering both mailing lists and issues trackers, we need to unify the names belonging to the same person, e.g., using the full first name or only its initials, using sometimes middle name. We apply a semi-automatic approach to unify all these names. In particular, we use an approach similar to the one proposed by Bird et al. [6] that we also applied to analyze mailing lists for different research [9, 10]. Briefly speaking, the approach merges together, where there is no possible ambiguity, cases where for example the first name is represented with initial or is full stated, or where the middle name is omitted. For example John Kennedy, J. Kennedy and J. F. Kennedy are considered to be the same person. Also, the approach uses heuristics to match names to email addresses. For example Sebastiano Panichella is mapped onto spanichella@unisannio.it, to panicella@unisannio.it or to Sebastiano.Panichella@unisannio.it.

Since the approach illustrated above is not 100% accurate (i.e., it can produce some wrong mappings) nor complete (i.e., it may miss some mappings), the obtained mapping was manually validated and links were fixed where needed. With the aim to reduce the noise generated by such automatic approach by Bird et al. we check and complete this initial mapping manually (to avoid the problem of the identity merging, i.e. the fact that the same person may have different identities/names). Overall, such a manual analysis allowed to add about 15% missing links with respect to the initial (automatic) mapping, while no false positives were detected.

Similarly, we map contributor names to committer IDs, as we want to focus our attention to discussions involving developers only, and not all contributors. This task is straightforward in cases where the committer ID is a full name and/or an email, e.g., in projects using git (as in our study). Also in this case the mapping was (partially, with same percentage of integration) complemented manually. Of course, by doing this, we are excluding the activity of some developers not discussing over mailing lists or issue trackers (see the bottom part of Table 1).

We are interested to study the evolution of teams, therefore we build communication networks related to different time periods, corresponding to the intervals between two subsequent releases. As in some cases releases are frequently issued, analyzing only few months of communication would not be representative of the emerging groups. Hence, we analyze the period between one release and the next one such that the time interval between them is of at least one year. In the following, we refer to $r_k$ the network concerning the communication between release $r_k$ and release $r_{k+1}$.

2.2.2 Step 2: Identifying Emerging Teams

To identify emerging teams, we apply clustering algorithms on the social networks extracted in Step 1, using the intensity of the communication between two developers—i.e., the number of emails exchanged—as an indicator of the connection’s strength.

To cluster developers, it is important to choose a suitable clustering algorithm, and, if the algorithm requires it, to determine the number of clusters. Previous work [6] used k-means as clustering algorithm. However, one issue of k-means is that it does not allow one item to be placed in more than one cluster. However, when identifying emerging teams, it is realistic to consider that a developer may primarily work for a team, but that she can also contribute to other teams. To this aim, we use a fuzzy alternative to k-means, i.e., the fuzzy C-Means [5] which, other than assigning a developer to a team, also provides the membership score $m_{i,j}$, i.e., the likelihood that a developer $d_i$ can belong to a team $t_j$. Specifically, we used the $cmeans$ function of the e1071 package available in the R [27] statistical environment.

After having analyzed the distributions of likelihoods we need to identify which teams the developer contributes to. To this aim, we define a threshold, named Membership Score Threshold ($MST$). If $m_{i,j} \geq MST$, then we assume that $d_i$ contributes to $t_j$, while we consider the contribution negligible if $m_{i,j} < MST$. In addition, we distinguish the primary team of a developer (the one with the highest $m_{i,j}$) from the other teams to which she contributes.

In our study, we analyzed the values of $m_{i,j}$ for all four projects and found that it is above 0.5 for the primary teams, whereas it is very low (and always < 0.3) for teams where a developer had an occasional or negligible participation. In conclusion, observation of such values suggested us to set $MST$ equal to 0.3.

To determine the number of clusters $k$, we compute the Silhouette coefficient introduced by Kaufman and Rousseeuw [22]. Let us consider a clustering obtained for a given $k$. For the observation $i$ (in our case a developer) let $a(i)$ be the average distance to the other points (developers) in its cluster (the team), and $b(i)$ the average distance to points in the nearest cluster (a different team). Then the Silhouette coefficient is
statistics is defined as:

\[ s(i) = \frac{b(i) - a(i)}{\max(a(i), b(i))} \]

For different values of \( k \), we will obtain different values of \( s(i) \). Kaufman and Rousseeuw suggested choosing the optimal number of clusters as the value maximizing the average \( s(i) \) over the dataset. In literature, it is assumed that the Silhouette curve knee indicates the appropriate number of clusters. In our study, we obtained \( k = 10 \) for Apache httpd and for Samba, \( k = 7 \) for Eclipse JDT, and \( k = 11 \) for Netbeans. It is important to note that the different projects obtain different values of \( k \) because of the different communication network density and size of the different projects.

2.2.3 Step 3: Mapping Teams Across Different Time Intervals

After having identified the emerging teams, we compare the teams identified in two subsequent time intervals \( r_k \) and \( r_{k+1} \), and try to build a mapping between them. Specifically:

- a team \( t_j \) of \( r_k \) splits if two or more teams of \( r_{k+1} \) contain subgroups of \( t_j \). We define a subgroup as a set of at least two developers belonging to a group. On the one hand, we did not impose a threshold greater than two because, as it will be clearer from the results, groups are often small. Also, this will allow us to observe how small sub-communities move. As it will be shown in Section 3, this frequently occurs in smaller projects. On the other hand, we did not consider singletons moving from a group to another as a case of group splitting (most of the group remained the same), but rather as a physiological phenomenon not really related to a massive work re-organization.

- vice versa, we detect a merger if subgroups of two teams \( t_i \) and \( t_j \) of \( r_k \) belong to the same team in \( r_{k+1} \).

- finally, one team survives between \( r_k \) and \( r_{k+1} \) if at least one of its subgroups remains in \( r_{k+1} \) and no other subgroup merges with it.

2.2.4 Step 4: Analyze Changes Performed by the Emerging Teams

This step analyzes to what extent developers belonging to the same emerging team work on a cohesive set of files. As explained before, we observe such a cohesiveness from both a structural perspective and conceptual (semantic) perspective.

As a structural measure, we rely on the Modularity Quality (MQ) defined by Mancoridis et al. [24]. MQ has been widely used in the context of software remodularization, for example as a fitness function for search-based remodularization algorithms [25].

Formally, MQ is defined as follows:

\[ MQ = \left\{ \begin{array}{ll} \frac{1}{2} \sum_{i=1}^{k} A_i - \frac{1}{N_i^2} \sum_{i,j=1}^{k} E_{i,j} & \text{if } k > 1 \\ A_1 & \text{if } k = 1 \end{array} \right. \]

where intra-connectivity measurement \( A_i \) of cluster \( i \) consisting of \( N_i \) components and \( \mu_i \) intra-edge dependencies is \( A_i = \frac{\mu_i}{N_i} \). Moreover, \( E_{i,j} \) is the inter-connectivity between the \( i^{th} \) and \( j^{th} \) clusters. Formally, the inter-connectivity \( E_{i,j} \) between clusters \( i \) and \( j \) consisting of \( N_i \) and \( N_j \) components, respectively, with \( \alpha_{i,j} \) inter-edge dependencies is defined as:

\[ E_{i,j} = \left\{ \begin{array}{ll} \alpha_{i,j} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{array} \right. \]

In our context, we compute the MQ for an emerging team by considering the set of files modified by the team members. As also done in previous studies [25], we assume a correspondence between modules and system directories. That is, Figure 2 illustrates an example calculation of MQ for an emerging team of three developers, which changed files belonging to two different modules.

After having computed the MQ, we observe how it changes when teams split and merge over the observed evolution period.

Note that previous work by Bird et al. [7] measured the cohesiveness of files modified by an emerging team using activity focus instead of MQ. The activity focus [7] is defined as the average directory distance between files changed by developers belonging to the same team, and compared it with the average directory distance of files changed by developers belonging to different teams. We use MQ because, differently from activity focus, it also accounts for dependencies between files (besides how they are organized in directories). Nevertheless, we also checked how activity focus changes when teams split and merge, and found consistent results with MQ.

As a complement to the MQ, we use the Conceptual Coupling Between Classes (CCBC) defined by Poshyvanyk et al. [26]. The CCBC is based on the semantic information cap-
MQ = \frac{2/9 + 1/9 - 1/18}{2} = 0.14

Intra-connectivity Inter-connectivity

Figure 2: Modularization Quality (MQ) computation for files modified by an emerging team.

where \( CCBC \) and \( CCM \) are similar, i.e., they have similar responsibilities. The definition of \( CCBC \) requires the introduction of a lower-level measure [26]: the Conceptual Coupling Between Methods (CCM). To measure \( CCM \), Latent Semantic Indexing (LSI) is used to represent each method as a real-valued vector that spans a space defined by the vocabulary extracted from the code. The conceptual coupling between two methods \( m_i \) and \( m_j \) is then calculated as the cosine of the angle between their corresponding vectors [2]:

\[
CCM(m_i, m_j) = \frac{\overrightarrow{m_i} \cdot \overrightarrow{m_j}}{||m_i|| \cdot ||m_j||}
\]

where \( \overrightarrow{m_i} \) and \( \overrightarrow{m_j} \) are the vectors corresponding to the methods \( m_i \) and \( m_j \), respectively, and \( ||x|| \) represents the Euclidean norm of the vector \( x \) [2]. Thus, the higher the value of \( CCM \) the higher the similarity between two methods. Clearly, \( CCM \) depends on the consistency of naming used in the source code and comments.

Now we can define the conceptual coupling between two classes \( c_i \) and \( c_j \) as:

\[
CCBC(c_i, c_j) = \sum_{m_k \in c_i} \sum_{m_k \in c_j} CCM(m_k, m_k) / |c_i| \times |c_j|
\]

where \( |c_i| \) (|c_j|) is the number of methods in \( c_i \) (\( c_j \)). Thus, \( CCBC(c_i, c_j) \) is the average of the coupling between all unordered pairs of methods from class \( c_i \) and class \( c_j \). The definition of this measure ensures that \( CCBC \) is symmetrical, i.e.,

\[
CCBC(c_i, c_j) = CCBC(c_j, c_i)
\]

It is worthwhile to point out that, while for structural modularity we relied on a measure capturing both cohesion and coupling information (\( MQ \), for semantic modularity we only relied on coupling. This is because semantic cohesion and coupling measures provide a similarity indicator, and cannot be directly used to build an index equivalent to MQ, unless imposing over them some arbitrary thresholds. For this reason, we preferred to just rely on the \( CCBC \) that, as shown in a previous study [3], provides a good indication of software modularity, sometimes even better than structural measures.

In our study, we tried to verify whether developers teams that split and merge between two releases re-organize the work in more ‘semantically’ related files. Thus, we compute the values of conceptual coupling for an emerging team by considering the set of files modified by the team members, and then we observe how it changes when teams split and merge.

2.3 Analysis Method

To address RQ1, given two releases \( r_k \) and \( r_{k+1} \), we report the number and percentage of (i) teams in \( r_k \) that survive in \( r_{k+1} \), (ii) teams in \( r_k \) that split in \( r_{k+1} \), and (iii) mergers occurring in \( r_{k+1} \) from (sub) teams of \( r_k \). Note that the number of mergers also considers sub-groups derived from the splits in \( r_k \). That is, two groups \( t_1, t_2 \) of \( r_k \) can split in \( t_1', t_2' \) as well as \( t_1'' \) and \( t_2'' \) merge (two splits occur). Then, in \( r_{k+1} \), \( t_1', t_2' \) as well as \( t_1'' \) and \( t_2'' \) merge (two mergers occur).

To address RQ2, we compute the average \( MQ \) and \( CCBC \) for files modified by each team before and after splits and mergers, and compare it by means of boxplots, and appropriate statistical tests, i.e., Wilcoxon rank sum paired test [15]. Specifically, we test the null hypothesis \( H_{0a} \) : there is no significant difference between the average \( MQ \) \( (CCBC) \) before and after split, and the null hypothesis \( H_{0b} \) : there is no significant difference between the average \( MQ \) \( (CCBC) \) before and after mergers. Then, we estimate the magnitude of the variation using Cliff’s Delta (\( d \)) [19], a non-parametric effect size measure for ordinal data, which indicates the magnitude of the effect of the main treatment on the dependent variables. The effect size ranges in the interval \([-1, 1]\) and is considered small for \( 0.148 \leq d < 0.33 \), medium for \( 0.33 \leq d < 0.474 \), and large for \( d \geq 0.474 \) [14].

2.4 Replication Package

To favor the study replicability, a package with the following artifacts is available: (i) raw communication from the mailing lists and issue trackers, (ii) communication graphs obtained from mailing lists and issue trackers, (iii) parsed Git history logs, (iv) working data sets (i.e., spreadsheets) from which the statistics used to address the RQs were computed. The replication package can be downloaded from www.rcost.unisannio.it/mdipenta/team-evol.tgz

3. EMPIRICAL STUDY RESULTS

This section describes the results achieved aiming at providing answers to the research questions formulated in Section 2.

3.1 RQ1: How do emerging collaborations change across software releases?

Table 2 reports—for each pair of releases \( r_k \) and \( r_{k+1} \) considered in our study—the number of teams identified in the discussions following the two releases, as well as the number of teams that split in release \( r_k \), merged in release \( r_{k+1} \), that survived in \( r_{k+1} \), totally disappeared, and the new teams
that emerged in \( r_{k+1} \). Details about developers’ involvement in the various releases are shown in Table 3 which reports—for each pair of releases—the number of developers that (i) were inactive during that interval, i.e., did not participate to the discussion nor they made any code change, (ii) joined the project, i.e., during that period they participated to the discussion and performed code changes, or, (iii) likely left the project, i.e., from that period until the end of our observation they never participated in the discussion nor made changes again.

The analysis of results reveals that, even if the number of teams across releases is quite stable, there is a continuous reorganization of teams during the evolution of a project. In all four projects, at least 50% of the teams, on average, survived from release \( r_k \) to \( r_{k+1} \) (for Samba and Netbeans this value is above 62%). This means that in the new release at least half of the developers tend to work in the same teams, while the remaining ones re-organize their work.

There is also a substantial number of teams that disappeared from release \( r_k \) to release \( r_{k+1} \). This phenomenon is particularly evident for Eclipse JDT and Apache. Specifically, for Apache and Eclipse JDT the percentage of teams identified in \( r_k \) that are not present anymore in release \( r_{k+1} \) is around 50%. This means that several teams completely crumble from one release to another, i.e., some developers join different teams or leave (definitively or temporarily) the project (see Table 3). It can be noticed that, for Apache and Eclipse JDT, the high number of teams that disappeared is balanced by a comparable number of survived teams (with few new emerging teams). Specifically, for such projects about 50% of the teams in release \( r_{k+1} \) is completely new, i.e., composed of developers belonging to different teams or of newcomers, as can be noticed in Table 2. However, if we consider all the projects in this study, the number of teams disappeared in a project is very often much lower of the number of survived teams. For all the projects, the number of teams that disappear in a version \( r_k \) and the number of teams that emerge in \( r_{k+1} \) is much lower, suggesting that a part of the existing teams tend to recombine.

To provide a qualitative evidence of the above observations, we inspected the source code change history and mails for the above identified teams. Figure 3 shows a group of

<table>
<thead>
<tr>
<th>Release ( r_k )</th>
<th>Teams in ( r_k )</th>
<th>Teams in ( r_{k+1} )</th>
<th>Teams disappeared in ( r_{k+1} )</th>
<th>Teams survived in ( r_{k+1} )</th>
<th>New teams in ( r_{k+1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache 2.0 to 2.2.0</td>
<td>9</td>
<td>10</td>
<td>5 (50%)</td>
<td>4 (40%)</td>
<td>1 (10%)</td>
</tr>
<tr>
<td>Apache 2.2.0 to 2.2.4</td>
<td>10</td>
<td>9</td>
<td>3 (30%)</td>
<td>2 (20%)</td>
<td>2 (20%)</td>
</tr>
<tr>
<td>Apache 2.2.4 to 2.2.12</td>
<td>9</td>
<td>8</td>
<td>4 (44%)</td>
<td>2 (22%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Apache 2.2.12 to 2.2.4</td>
<td>8</td>
<td>8</td>
<td>4 (50%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Eclipse JDT 3.0 to 3.4</td>
<td>7</td>
<td>7</td>
<td>2 (27%)</td>
<td>1 (14%)</td>
<td>1 (14%)</td>
</tr>
<tr>
<td>Eclipse JDT 3.4 to 3.6</td>
<td>7</td>
<td>6</td>
<td>3 (43%)</td>
<td>1 (14%)</td>
<td>1 (14%)</td>
</tr>
<tr>
<td>Eclipse JDT 3.6 to 4.2</td>
<td>6</td>
<td>6</td>
<td>3 (50%)</td>
<td>1 (17%)</td>
<td>1 (17%)</td>
</tr>
</tbody>
</table>

**Figure 3:** A stable groups of developers (in blue) that joined different teams during software evolution.

Table 2: Evolution of teams across software releases.

<table>
<thead>
<tr>
<th>Release ( r_k )</th>
<th>Teams in ( r_k )</th>
<th>Teams in ( r_{k+1} )</th>
<th>Teams disappeared in ( r_{k+1} )</th>
<th>Teams survived in ( r_{k+1} )</th>
<th>New teams in ( r_{k+1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache 2.0 to 2.2.0</td>
<td>9</td>
<td>10</td>
<td>5 (50%)</td>
<td>4 (40%)</td>
<td>1 (10%)</td>
</tr>
<tr>
<td>Apache 2.2.0 to 2.2.4</td>
<td>10</td>
<td>9</td>
<td>3 (30%)</td>
<td>2 (20%)</td>
<td>2 (20%)</td>
</tr>
<tr>
<td>Apache 2.2.4 to 2.2.12</td>
<td>9</td>
<td>8</td>
<td>4 (44%)</td>
<td>2 (22%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Apache 2.2.12 to 2.2.4</td>
<td>8</td>
<td>8</td>
<td>4 (50%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Eclipse JDT 3.0 to 3.4</td>
<td>7</td>
<td>7</td>
<td>2 (27%)</td>
<td>1 (14%)</td>
<td>1 (14%)</td>
</tr>
<tr>
<td>Eclipse JDT 3.4 to 3.6</td>
<td>7</td>
<td>6</td>
<td>3 (43%)</td>
<td>1 (14%)</td>
<td>1 (14%)</td>
</tr>
<tr>
<td>Eclipse JDT 3.6 to 4.2</td>
<td>6</td>
<td>6</td>
<td>3 (50%)</td>
<td>1 (17%)</td>
<td>1 (17%)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Release ( r_k )</th>
<th>Teams in ( r_k )</th>
<th>Teams in ( r_{k+1} )</th>
<th>Teams disappeared in ( r_{k+1} )</th>
<th>Teams survived in ( r_{k+1} )</th>
<th>New teams in ( r_{k+1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache 2.0 to 2.2.0</td>
<td>9</td>
<td>10</td>
<td>5 (50%)</td>
<td>4 (40%)</td>
<td>1 (10%)</td>
</tr>
<tr>
<td>Apache 2.2.0 to 2.2.4</td>
<td>10</td>
<td>9</td>
<td>3 (30%)</td>
<td>2 (20%)</td>
<td>2 (20%)</td>
</tr>
<tr>
<td>Apache 2.2.4 to 2.2.12</td>
<td>9</td>
<td>8</td>
<td>4 (44%)</td>
<td>2 (22%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Apache 2.2.12 to 2.2.4</td>
<td>8</td>
<td>8</td>
<td>4 (50%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Eclipse JDT 3.0 to 3.4</td>
<td>7</td>
<td>7</td>
<td>2 (27%)</td>
<td>1 (14%)</td>
<td>1 (14%)</td>
</tr>
<tr>
<td>Eclipse JDT 3.4 to 3.6</td>
<td>7</td>
<td>6</td>
<td>3 (43%)</td>
<td>1 (14%)</td>
<td>1 (14%)</td>
</tr>
<tr>
<td>Eclipse JDT 3.6 to 4.2</td>
<td>6</td>
<td>6</td>
<td>3 (50%)</td>
<td>1 (17%)</td>
<td>1 (17%)</td>
</tr>
</tbody>
</table>

**Table 3:** Inactive (IN), new (N), and developers that likely left the project (DL).

<table>
<thead>
<tr>
<th>Release ( r_k )</th>
<th>Teams in ( r_k )</th>
<th>Teams in ( r_{k+1} )</th>
<th>Teams disappeared in ( r_{k+1} )</th>
<th>Teams survived in ( r_{k+1} )</th>
<th>New teams in ( r_{k+1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache 2.0 to 2.2.0</td>
<td>9</td>
<td>10</td>
<td>5 (50%)</td>
<td>4 (40%)</td>
<td>1 (10%)</td>
</tr>
<tr>
<td>Apache 2.2.0 to 2.2.4</td>
<td>10</td>
<td>9</td>
<td>3 (30%)</td>
<td>2 (20%)</td>
<td>2 (20%)</td>
</tr>
<tr>
<td>Apache 2.2.4 to 2.2.12</td>
<td>9</td>
<td>8</td>
<td>4 (44%)</td>
<td>2 (22%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Apache 2.2.12 to 2.2.4</td>
<td>8</td>
<td>8</td>
<td>4 (50%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Eclipse JDT 3.0 to 3.4</td>
<td>7</td>
<td>7</td>
<td>2 (27%)</td>
<td>1 (14%)</td>
<td>1 (14%)</td>
</tr>
<tr>
<td>Eclipse JDT 3.4 to 3.6</td>
<td>7</td>
<td>6</td>
<td>3 (43%)</td>
<td>1 (14%)</td>
<td>1 (14%)</td>
</tr>
<tr>
<td>Eclipse JDT 3.6 to 4.2</td>
<td>6</td>
<td>6</td>
<td>3 (50%)</td>
<td>1 (17%)</td>
<td>1 (17%)</td>
</tr>
</tbody>
</table>
nine Netbeans developers (highlighted in blue) that joined different teams in three subsequent releases (2.3, 3.0.20, and 3.0.25). As we can notice such developers always worked together, albeit moving across different teams, i.e., involving other people in different releases.

The obtained results allow us to distill a first lesson learned:

L1. Teams continuously evolve during the evolution of a software system. We observed that groups of people build stable working relations (at least 50% of the teams survive between two subsequent releases), and occasionally work with others.

3.2 RQ2: How does the evolution of emerging collaboration relate to the cohesiveness of files changed by emerging teams?

Figure 4 shows boxplots of how MQ and CCBC vary after teams split. Table 4 complements Figure 4 with results of Wilcoxon test and Cliff’s $d$ effect size values. Results for all four projects indicate that when teams splits, MQ and CCBC significantly increase with a large effect size. Therefore for both metrics we can reject the null hypothesis $H_0$.

For the mergers, Figure 5 shows boxplots of MQ and CCBC variations, while Table 5 reports results of Wilcoxon test and Cliff’s $d$ effect size values. In this case, we did not observe a significant increase of MQ (except for Apache). Hence, it is not possible to reject $H_0$ for what concerns MQ. In the case of team merger one would expect that the MQ could even decrease, because the focus of the new team will likely be broader. However, not only this does not happen, but also MQ slightly increases. Also, we can observe how the CCBC always significantly increases (hence we can reject $H_0$ in this case), although the effect size is medium for Netbeans and Samba, and small in the other cases. A possible interpretation to the results obtained for the mergers is that the previous teams join their force to work on a
new, focused task.

To provide qualitative explanations to the above results, we inspected source code changes and mails for the above identified teams. Figure 6 depicts a scenario where two Samba teams merge between release 2.3 and 3.0.20. The analysis of the file changed by each team revealed that:

- **Team 1** is a testing team, i.e., a team of developers that is called very often to test some new components or features. The developers of this team, between the various commitments, mainly worked on the testsuite folder of the Samba project. Specifically, they wrote test suites for the modules libsmbclient, smbd, and nsswitch.

- **Team 2** is a development team. The analysis of the change logs highlights that the development effort of this team is mainly focused on the modules auth, smbd, nsswitch, smbd and auth.

This suggests that in Samba 2.3 there is an evident effort in the development of test cases, that required a group of developers to work mainly on testing activities.

In addition, from the analysis of test cases developed by Team 1, we can derive a "latent" relationship between Team 1 and Team 2, i.e., the testing team provided test cases for some modules developed by Team 2.

In release 3.0.20, Team 1 and Team 2 merged. As a consequence of such a merger, the values of both MQ and CCBC increased. Specifically, the MQ increased by 73%, from an average of 0.71 to 1.23. The CCBC increased by 53%, from an average of 0.43 to 0.66. We expect that this variation depends on the fact that developers of the two past teams in the new release focus their effort in files that are structurally and semantically more related. The analysis of changes performed by the new team indicated less activity on the development of test suites, likely because such test suites were mostly produced in the previous release, hence they only needed to be repaired where necessary. In addition, the files changed by the new emerging team in release 3.0.20 revealed that the developers focused the development activity on the module heimdal. In release 3 Samba used the external Kerberos authentication. Starting from such a release the developers worked on the integration with Heimdal (an implementation of Kerberos 5), integration that was fully available only in release 4. In this case the reorganization of teams is triggered by the need to deal with changes in the project design and implementation.

We found cases in which, when teams split, one of the metrics (CCBC) significantly increased, while the other (MQ) did not, or vice versa. Specifically, let us to consider the Apache httpd project, where there are few cases in which emerging teams that are obtained by a split of a previous emerging team (from the previous release) had an increase of the resulting CCBC with a corresponding decrease of MQ. For example, as it can be seen from Figure 7, from release 2.2.4 to release 2.2.12, a team of five developers (Bryan, Lars, Mads, Mark, Sander) is split into two smaller teams of two (Bryan and Sanders) and three developers (Lars, Mads and Mark) respectively. The team of release 2.2.4 is mainly focused on some sub-modules of os (e.g., os/win2, os/unix, os/os2) and os (e.g., modules/proxy, modules/test, modules/database). In the new release, the team splits into two sub-teams. Both teams continue to work on sub-modules of os, however separating their activities, i.e., the first team works on other modules of modules/ (e.g., modules/proxy, modules/test, modules/database) and the second on support/win32, docs/manual and docs/conf modules. At the same time, both teams also work on some os modules. It happens that the modules modified by each of these two teams are somewhat decoupled (therefore, the MQ decreases by 45%, from an initial MQ of 0.29 to an average MQ of 0.20 between the two teams). However, the conceptual cohesion is high, in that such modules share several concepts. For example, we found an increase of shared common words between the os, support, docs modules, terms mainly related to code changes for support windows platform (winapi, windowsStation and so on). As a consequence, the CCBC value increases of more than times its value (from an initial CCBC of 0.36 to a CCBC of 0.86).

In summary, the above observations allow us to distill a second lesson:

L2. Both MQ and CCBC increase when teams split, while MQ does not significantly changes and CCBC increases when teams merge. This means that team reorganization relates to work on more (structurally and semantically) cohesive sets of files.

4. THREATS TO VALIDITY

This section discusses the threats to validity of the study reported in this paper, categorized into threats to construct,
conclusion, internal, and external validity.

Threats to construct validity concern the relationship between theory and observation. Previous work in this area used either mailing lists [7] or issue trackers [21] to identify the developers’ communication network. To mitigate the risk of missing links, in this paper we consider both sources of information. However, we are aware that we did not ask the original developers to validate their identified teams. Also, although communication in open source projects is usually asynchronous and occurring through electronic channels [11], we are aware that written communication through mailing lists can constitute a partial view of the overall communication, as also reported in a paper by Aranda and Vena-lia [1]. However, such a problem is more crucial in industrial projects than in OSS projects where emails constitute the premier mean of communication for developers working around the world.

As explained in Section 2.2, we mitigated imprecision and incompleteness of the mapping and unification of mailing lists and issue trackers contributor names, and of their mapping onto author IDs available in the versioning system, by performing a manual validation. Besides that, it is important to note that we considered as authors only those mentioned in the versioning system. However, we cannot exclude that code changes were authored by somebody not mentioned there.

Threats to conclusion validity concern the relationship between treatment and outcome. Where appropriate—i.e., for RQ2—we use appropriate (non parametric) statistical tests and effect size measures to support our findings. In other cases, we analyze the phenomenon by using descriptive statistics.

Threats to internal validity concern factors that can affect the results. The study is purely observational and we cannot claim a causation between the observed phenomenon, e.g., team splits/mergers and increase of MQ and CCBC as observed in RQ2. However, we qualitatively support such finding by some manual analysis of source code changes and of emails exchanged between developers.

Threats to external validity concern the generalization of findings. In Section 2 we have motivated the choice of the four open source projects considered in this paper. However, the obtained findings have a validity confined to considered OSS projects, and above all cannot be directly extended to some commercial projects where the face-to-face communication is more than often used in place of written communication. Having said that, many companies are increasingly using written communication in issue trackers or mailing lists as a way to record the decisions taken in a project, hence in such situations what described in this paper is still applicable.

5. RELATED WORK

An open source community can be considered as a complex, self-organizing system. Specifically, such systems are typically comprised of large numbers of locally interacting elements, where developers are the main components in this network [4]. This has motivated research effort in mining the social networking among developers. Xu et al. [32] use social network analysis to study the whole developer community at SourceForge. When building such a network, two developers are considered socially related if they participate in the same project. The obtained developer network is a scale free network. Wagstrom et al. [31] build a developer network by considering several sources, including blogs, email lists and networking web sites. The analysis derived from such a network is used to simulate how contributors joined and left projects. Lopez-Fernandez et al. [23] use information derived from CVS to build a developer network as well as a module network. For the former network, there is a link between two developers if they commit the same file. As for the module network, a social link between two modules exists if there is at least one developer that worked on both. Both networks are small-world networks.

Guzzi et al. [20] quantitatively and qualitatively analyze a sample of 506 email threads from the development mailing list of Apache Lucene. Their study reveals that core developers participate in less than 75% of the threads, and in only about 35% of the threads implementation details are discussed. This motivated us to integrate, in our study, data from mailing lists with data from issue trackers.

Capra and Wassermann [11] characterize—by means of a survey involving 70 projects—management activities in both commercial and open source organizations. They analyze mechanisms for communication and collaboration, as the in-
ternal organizational structure of the projects. In the investigated open source projects, communication occurs through asynchronous tools such as mailing lists, although IRC channels are also used, and physical meetings are sometimes organized. Also, they found that the organizational structure of some open source projects is not much different from those of some closed source projects.

Crowston and Howison [16] use co-occurrence of developers on bug reports as indicators of a social link. They observe that development teams vary widely in their communications centralization, from projects completely centered on one developer (usually smaller projects) to projects that are highly decentralized (usually larger projects). Such a finding inspired the study by Bird et al. [7] that analyze the relationship between communications structure and code modularity. Their results show that the sub communities identified using communication information are significantly connected with collaboration behavior. In our study, we analyze how the connectedness between communication structure and code cohesiveness evolve, although as explained in Section 2 we use different measures ($MQ$ and $CCBC$) than what used by Bird et al. (activity focus).

Hong et al. [21] compare developers’ network events with events occurring in traditional social networks such as Facebook or Twitter. They find that, while events in traditional social networks follow a power law, this does not happen in developers’ networks. They also find that the developers’ community structure tends to stabilize over time, and that the reorganization of developers teams reflect the transition into new releases. Our study shares with Hong et al. [21] the analysis of how emerging teams of developers re-organize themselves over time. However, we specifically related such a reorganization with the cohesiveness of files they modify, measured in terms of $MQ$ and $CCBC$.

Previous studies have related communication among developers to software quality [13] and developers’ productivity [12]. Zhou and Mockus [33] use developer communications in order to compute the relative sociality of a developer as a ratio between the size of her communication network and the number of tasks she participates in. A study conducted on three industrial and three open source projects indicated that the sociality level of a project in a particular moment influences the likelihood for newcomers to become long-term contributors. The authors found that it is more likely that this happens when the project sociality is low, because senior developers have more time to train newcomers.

Singh [29] analyzes over 4,000 projects from SourceForge to understand how the relations among developers influence development activities. His results show that “small-world communities”—i.e., projects where there are small clusters of developers discussing with each other—are a factor for the project success. We share with this work the importance of developers’ discussion during development and the fact that such a discussion, possibly, occurs in small clusters related to specific components or to specific issues to be handled.

Surian et al. [30] mine collaboration patterns from a large developers’ network in SourceForge. They find that not all developers are connected each other, and that there are many collaboration clusters. Their findings are consistent with those of Singh et al. [29]; that is, the small-world phenomenon also exists in SourceForge, especially when developers in a network are separated, on average, by approximately 6 hops.

The methodology used in this paper to analyze the evolution of teams might complement all these approaches. In particular, the analysis of the communication network can be exploited not only to identify roles in developer teams but also to identify how teams evolve during software evolution. The evolution of the teams can be analyzed by a project manager in order to monitor the “project landscape” [17] and facilitate the integration of newcomers.

6. CONCLUSION AND FUTURE WORK

Differently from commercial projects, where teams are often designed by project managers, open source projects are characterized by spontaneous collaborations, that result in emerging teams, often reflected by frequent communication through asynchronous channels such as e-mails and communications in issue trackers.

This paper investigated, by analyzing software repositories of four open source projects—namely Apache httpd, Eclipse JDT, Netbeans and Samba—how such emerging teams evolve over time, and to what extent such evolution relates with the developers’ activity on source code. Results of the study indicate that:

- Emerging teams tend to recombine over time. We often observed groups (e.g., of two or more developers) having a stable collaboration over time, although involving each time different other people, i.e., moving between different emerging teams.

- Team splits correspond to a significantly higher cohesiveness of the files teams modify; specifically, we observed a significant increase of both modularization quality and conceptual coupling between classes. The qualitative analysis indicated that the new emerging teams formed after a split work on closer files than the original ones.

- Team mergers do not imply a decrease of structural and conceptual cohesiveness. On the contrary, we found that in general $MQ$ does not change, or it slightly improves, while the $CCBC$ still increases. Qualitative analysis suggested that teams merged when they were working on related files and specific circumstances lead developers of one or both teams to change their activities. This is for example the case when developers having worked in a period over test cases join their forces with those working on production code as there is less need to work on tests. In other cases, team reorganization can be dictated by changes in the system design and/or implementation.

The obtained results shed the light on the collaboration dynamics in open source projects, highlighting how such dynamics can be influenced by various factors, that could be the attitude of small teams of developers to work together, or changes in specific project needs, e.g., performing testing vs. development activities.

Future work aims at combining information from emerging team evolution with other sources of information with the aim of building better recommenders for software re-modularization. Also, we plan to extend the empirical investigation to a large number and variety of open source projects.
REFERENCES


