Computing and Informatics, Vol. 34, 2015, 678–724

EVOLVING THE SYSTEM'S CORE: A CASE STUDY ON THE IDENTIFICATION AND CHARACTERIZATION OF KEY DEVELOPERS IN APACHE ANT

Gustavo Ansaldi Oliva, José Teodoro da Silva
Marco Aurélio Gerosa

Department of Computer Science
University of São Paulo (USP)
São Paulo, Brazil
e-mail: {goliva, jteodoro, gerosa}@ime.usp.br

Francisco Werther Silva Santana, Cláudia Maria Lima Werner

Department of Computer Science
Federal University of Rio de Janeiro (UFRJ)
Rio de Janeiro, Brazil
e-mail: {fwsantana, werner}@cos.ufrj.br

Cleidson Ronald Botelho de Souza

Department of Computer Science
Federal University of Pará (UFPA)
Pará, Brazil
&
Vale Technological Institute (ITV – DS)
Pará, Brazil
e-mail: cleidson.desouza@acm.org

Kleverton Carlos Macedo de Oliveira

Department of Computer Science
Federal University of Pará (UFPA)
Pará, Brazil
e-mail: kleverton.macedo@gmail.com

Abstract. Software systems usually include a limited number of important classes that a large number of other classes depend upon. These files comprise the technical core of a software system. Evolving this core is naturally difficult and requires caution, since inappropriate changes may induce side and ripple effects. Indeed, despite the rhetoric about openness in free/libre open source software, modifications to the technical core are usually accomplished by a small set of contributors known as key developers. Automatically identifying key developers is relevant for a number of reasons, such as supporting the recruitment of specialists, assigning tutors to newcomers, and estimating the longevity likelihood of projects. Moreover, little is known about how key developers evolve and about the social skills that support them in their technical tasks. In this paper, we describe a case study involving the Apache Ant project. Our goal was to identify key developers and characterize them in terms of their social activity and contributions. We conceived, implemented, and applied a method to identify key developers. Such method identified four individuals, who were the main target of our investigation. We built communication networks from mailing list data and a coordination requirements network from their development traces. The analysis of these two networks indicated that three key developers socialized more than the others. They acted as bridges connecting other developers and communicated with almost everyone they were supposed to. The other key developer showed a very distinct behavior, as he participated very rarely in the mailing list. We also analyzed key developers' contributions and found some patterns. Core commits and non-core commits were often interleaved and key developers also contributed to peripheral portions of the system. Finally, we observed that the set of key developers was indistinguishable from the set of top committers. We expect this characterization to foster the definition of key developers' profiles that take into account their social activities and contribution characteristics.

Keywords: Key developers, core developers, free/libre open source software, socio-technical analysis, mining software repositories, case study, apache ant

Mathematics Subject Classification 2010: 68N01, 68P20, 62H30

1 INTRODUCTION

The volatility of requirements and technologies impose a constant pressure for changes in software systems. The Lehman's Laws of Software Evolution [29] and agile methods [6] account for changes as something intrinsic to the nature of a useful software [8]. It is not hard to notice that changes are actually intrinsic to a lot of things. Indeed, the pre-Socratic Greek philosopher Heraclitus once said that *one cannot step into the same river twice* [45].

However, changing and evolving software systems is far from being a trivial task. In the particular context of object-oriented programming (OOP), software

structure is implemented as a set of classes that depend on one another. The resulting dependency graph is known as the *technical network* of a software system. As software evolves, a specific subset of classes begins to assume a very important role by having many other classes depending on them (either directly or indirectly). We say that such important classes form the *core of the technical network* of a software system. Changing this core is often difficult and risky, since inappropriate modifications may induce side and ripple effects [44, 2, 20]. Accumulating bad modifications to the core ends up violating architectural rules and breaking encapsulation, which might ultimately lead to software design degradation [21, 32, 39, 42].

Despite the challenges of changing the technical core, software projects frequently have a small subset of developers who actually overcome such challenges. In fact, even in free/libre open source software (FLOSS) projects with a regular influx of participants, this same phenomenon occurs [34]. These special developers are known as *key developers* (a.k.a. *core developers*). Achieving the status of key developer also grants one with social standing and identity in the community [28]. They also tend to have a crucial role in conflict resolution and leadership establishment processes [15]. However, even though both academia and practitioners recognize their importance, the literature lacks a proper definition of their profile. There is a variety of questions to be answered. How are key developers different from the others? How can they be automatically identified in the dynamic context of FLOSS development? Do key developers contribute to the core only? How to know whether a key developer grew his expertise from the project (over time) or brought it from past experiences? What are the social skills that support key developers doing their technical work? Do they socialize more than others? Do they act as *brokers*? After all, what it takes to become a key developer? We believe that identifying and characterizing the many profiles of FLOSS contributors is an essential step towards a better understanding of how FLOSS communities operate and maintain vibrancy. For instance, merely knowing the number of key developers might indicate the longevity likelihood of software systems. As an illustrative example, the development of the popular Linux image processing software GIMP halted for about 20 months right after its two creators graduated from Berkeley, started full-time employment, and no longer had time to be involved in the project [57].

In this paper, we are interested in identifying key developers and characterizing them from a socio-technical perspective, focusing on the investigation of how these developers communicate, coordinate their tasks, and contribute to the project. More specifically, we conducted a descriptive case study involving the Apache Ant[1] project. Firstly, we applied an approach to identify the set of key developers based on the kinds of artifacts that developers modified over time. Afterwards, using Social Network Analysis (SNA) techniques [56, 36], we investigated how the key developers communicated by building and analyzing a social network from mailing list data and

[1] `http://ant.apache.org`

investigating whether they were central in such network. In a software development context, task dependencies drive the need to coordinate work activities [13]. Hence, we built a coordination requirements network and analyzed whether key developers were central in such network. Furthermore, we investigated whether key developers had a high socio-technical congruence [12] by evaluating the social activities that actually occurred (given by the mailing list network) and those that should have occurred (given by the coordination requirements network). Finally, we characterized key developers in terms of their contribution to the project. In particular, we analyzed not only their contribution volume, but also the kinds of artifacts they modified.

This paper extends our previous study [40] by:

1. providing a deeper investigation of how key developers contribute (e.g., how core commits are distributed over time and the kinds of artifacts that key developers work on),
2. considering different techniques to build the mailing list communication network, and
3. further investigating the role of key developers in the communication network (e.g., by employing other SNA metrics).

The motivation and supporting concepts of this study are also presented in more details.

The rest of this paper is organized as follows. In Section 2, we present the fundamental concepts and theory supporting this study. In Section 3, we motivate the identification and characterization of key developers and state our research questions. In Section 4, we describe the research design, including details about the case study and the supporting tools we developed. In Section 5, we present the results we obtained from the case study. In Section 6, we discuss the aforementioned results. In Section 7, we present and discuss related work. Finally, in Section 8, we state our conclusions and plans for future work.

2 BACKGROUND

2.1 Key Developers

Despite the lack of a precise definition, it is a consensus that key developers are the ones responsible for making decisions and conducting the project's development. In turn, peripheral developers tend to marginally contribute to the project and be less committed to it. In this paper, we call key developers the set of developers who evolve the technical core of a software system. The technical core is extracted from the technical network.

2.2 Technical Networks

Technical networks describe software systems from a structural perspective. In the context of OOP, such networks depict how classes depend on one another. Clustering techniques may be used to determine how packages depend on others, providing a high-level view of the system. In the following, we describe two techniques we used in our study.

2.2.1 Call-Graph

A *call-graph* is a directed graph that represents calling relationships between subroutines in a computer program. A *static call-graph* is a special kind of call-graph obtained by means of static analysis of the source code. In the context of OOP, a static call-graph denotes calling relationships between operations. In this graph, a directed edge indicates that the source operation makes use of the results of the target operation. We highlight that the source operation is always a class operation, while the target operation can be a class operation or an interface operation. The latter case denotes a call dependency that leverages polymorphism.

2.2.2 Logical Dependencies Graph

A visionary work of Ball et al. [4] introduced the idea that version control systems (VCSs) contain a significant amount of data exploitable in systems evolution studies. One year later, Gall et al. [22] introduced the idea of logical dependencies (a.k.a. logical coupling), which they defined as "observed identical change behavior or different elements during system evolution." In other words, a logical dependency refers to an evolutionary relationship established among artifacts that have been frequently changed together during a specific timeframe.

A source code file can be viewed as representing a "bundle" of technical decisions [12]. If a certain modification request (bug fix, new feature, etc.) can be implemented by changing only one file, then it provides no evidence of any dependency among files. However, when a modification request results in changes to more than one file, then it can be assumed that decisions about the change of one file depend in some way on the decisions made about changes to other files involved in implementing the modification request [12]. The more a set of files changes together, the more they are evolutionarily connected.

Unlike static analysis, logical dependencies analysis spots dependencies between any kind of artifact that composes a system, including source code, configuration files, and documentation. Such analysis is usually performed by parsing and analyzing the commit logs of a VCS. The logical dependencies graph is a specific graph in which nodes refer to artifacts, and edges refer to logical dependencies connecting these artifacts. More information about logical dependencies can be found in [4, 23, 61, 62, 17, 41, 38].

2.3 Social Networks

Social Network Analysis (SNA) focuses on the analysis of relationships among social entities [56]. In this field, social structures are often modeled by a graph whose nodes represent social entities (e.g., an individual or group) and edges represent a certain relationship or tie between two of such entities. In the context of software development, social networks are often built to represent how developers interact and communicate. Examples include building social networks to model developers communication extracted from messages in mailing lists or comments in issue trackers.

2.3.1 Social Network Metrics

Several metrics were conceived to have indications of the importance of a certain individual in a social network. Figure 1 depicts the metrics we used in this paper.

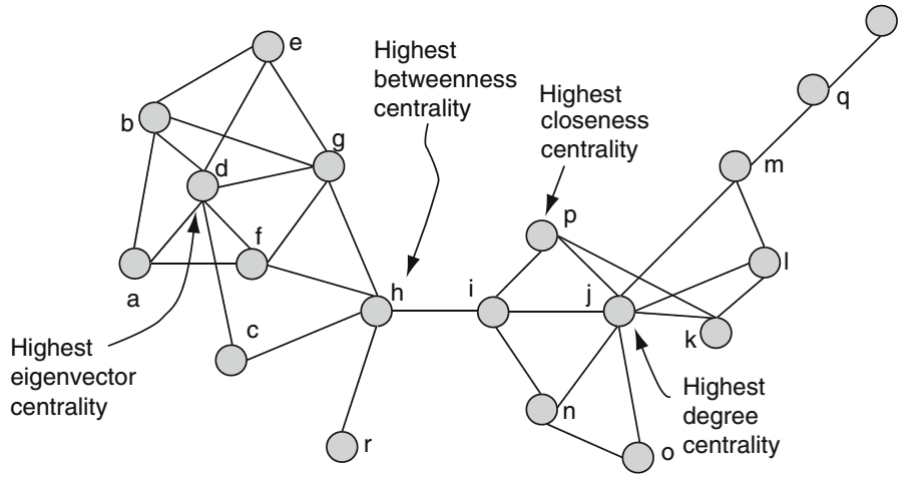


Figure 1: Social network metrics (adapted from [43])

In the following, we present the rationale behind each metric and how they are calculated:

Degree Centrality. Degree centrality is defined as the number of ties that a node has [56]. Nodes with high degree centrality have higher probability of receiving and/or transmitting whatever information flows in the network, i.e., they influence the nodes in their neighborhood [1]. Degree centrality is a *local measure*, since only the connections of a node with its neighbors are taken into account to evaluate its importance.

Eigenvector Centrality. The Eigenvector centrality is a natural extension to the notion of Degree Centrality: each node awards "one centrality point" for every neighbor it has and Eigenvector Centrality gives each node a score proportional to the sum of the scores of its neighbors. The key idea is that a node's importance is increased when it has connections to other nodes that are themselves important [56, 36]. Eigenvector Centrality can be computed by an iterative degree calculation procedure known as the *accelerated power method* [25].

Algorithm 1: Eigenvector Centrality computation via Accelerated Power Method

Input : An adjacency matrix $A_{i,j}$, where $A_{i,j} = 1$ if the i^{th} node is adjacent to the j^{th} node, and $Ai, j = 0$ otherwise
Output: Eigenvector centrality value for all graph nodes

1 Set $C_E(v_i) = 1$ for all i ;
2 Compute $C^*_E(v_i) = \sum_j A_{i,j} * C_E(v_j)$;
3 Set λ equal to the square root of the sum of squares of each $C^*_E(v_i)$;
4 Set $C_E(v_i) = C^*_E(v_i)/\lambda$ for all i ;
5 Repeat lines 2 to 4 until λ stops changing ;

Betweenness Centrality. Nodes that occur on many shortest paths (a.k.a. geodesic distance) between other vertices have higher betweenness than those that do not. Hence, betweenness centrality evaluates the degree of control a node has over the information flowing in the network. Messages sent through the network frequently pass through these nodes, i.e., they act as "brokers". Betweenness Centrality is given by the following equation:

$$C_B(v) = \left(\sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \right) / [(n-1) * (n-2)],$$

where σ_{st} is the number of shortest paths from s to t, and $\sigma_{st}(v)$ is the number of shortest paths from s to t that pass through a vertex v.

Closeness Centrality. The vertex closeness refers to the geodesic distance between a vertex v and all other vertices reachable from it. Closeness measures how close a node is located with respect to every other node in the network. Nodes with high closeness are able to reach most or all other nodes in the network through geodesic paths. Closeness Centrality is given by the following equation:

$$C_C(v) = \frac{|J_v|/(n-1)}{\sum_{t \in J_v} d_G(v,t)/|J_v|}$$

where J_v is the set of vertices reachable from v (a.k.a influence range of v) and $d_G(v,t)$ is the length of the geodesic path from v to t.

2.3.2 Coordination Requirements Network and Socio-Technical Congruence

Organizations often cope with complex tasks by first dividing them into smaller interdependent work units and then assigning such units to teams. In this context, coordination among teams arises as a response to such interdependent work units [31]. Cataldo et al. [13, 12, 11] conceived an approach to elicit coordination requirements. More specifically, his approach tackles the following problem: given a particular set of dependencies among tasks, identify which set of individuals should coordinate their activities.

Cataldo's approach relies on two sets of relationships (Figure 2). The first set is called *Task Assignments* (Ta) and defines which individuals are working on which tasks. This set is represented by a matrix where each cell $[i, j]$ indicates that the developer i was assigned to the task j. In the context of software development, this set might be built upon the set of files modified by each developer on a modification request or throughout the development of a software release. The second set of relationships is called *Task Dependencies* (Td) and defines the interdependencies between tasks. This set is also represented by a matrix where each cell $[i, j]$ (or $[j, i]$) indicates whether tasks i and j are interdependent. In the context of software development, this set might be built upon either the set of structural (syntactic) dependencies or the set of co-changes (logical dependencies). Cataldo tested both types of dependencies and concluded that co-changes provided a more accurate representation of the most relevant product dependencies in software development projects [12]. In the particular case of co-changes, the diagonal of Td indicates the total number of times the source code files were changed during a certain development period. In turn, off-diagonal cells indicate the number of times the two files were changed together.

Once Task Assignments (Ta) and Task Dependencies (Td) matrices are built, coordination requirements are ready to be determined. Multiplying Ta by Td results in a people by task matrix that represents the extent to which a particular worker should be aware of tasks that are interdependent to those that he or she is responsible for [12]. Multiplying the $Ta * Td$ product by the transpose of Ta results in a people by people matrix where a cell $[i, j]$ represents the extent to which person i works on tasks that share dependencies with the tasks worked on by person j [12]. In other words, this last matrix represents the Coordination Requirements (Cr), or the extent to which each pair of people needs to coordinate their work. When calculating Td using co-changes, the resulting Cr matrix is symmetric (2).

The term *socio-technical congruence* was coined by Cataldo et al. [12]. It refers to the match between the coordination requirements and the actual coordination activities carried out by workers. This concept builds mainly on the idea of "fit" from the organizational theory literature [9]. In practical terms, given a certain Cr matrix, it is possible to compare it to an Actual Coordination (Ca) matrix representing the coordination activities that took place. This last matrix can be built based on data from mailing lists or issue trackers (or both). Congruence is computed

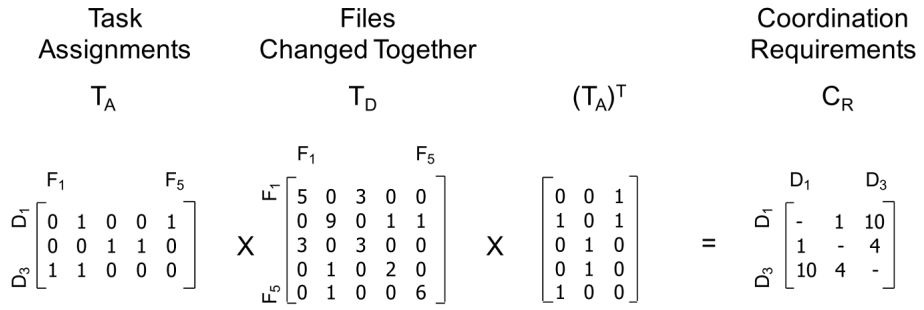


Figure 2: Illustrative example of coordination requirements calculation (extracted from [10])

as the proportion of coordination activities that actually occurred (represented by the *Ca* matrix) relative to the total number of coordination activities that should have taken place (represented by the *Cr* matrix). For instance, if the *Cr* matrix shows that 10 pairs should coordinate, and of these, only 5 pairs show coordination activities in the *Ca* matrix, then the congruence is 0.5.

3 MOTIVATION AND RESEARCH QUESTIONS

3.1 Identifying Key Developers

In this study, we were interested in identifying key developers, i.e., the ones who evolve the technical core of a software system. There are several reasons motivating such identification. Firstly, an individual might be interested in contacting a key developer and it might be the case that the projects' website is not updated (due to the high turnover of developers in FLOSS [60], for example) or does not include such information. An example would include a certain project needing one expert for a specific activity and trying to search for key developers in related projects (e.g., from similar domain). Secondly, key developers might serve as tutors to newcomers and help them become familiar with the project's landscape [16]. Finally, (iii) projects might reach their end of life just because their set of key developers decided to stop contributing to the project. Thus, the number of current active key developers may indicate the longevity of the project. Indeed, longevity expectation is often an important factor driving the selection of one among similar FLOSS projects and libraries.

However, identifying key developers is not a trivial task. It requires identifying the technical core of a software system at different points in time and determining the extent to which each developer helped to evolve such core. This leads to our first research question:

RQ 1: How to identify key developers?

3.2 Characterizing Key Developers from a Social Network Analysis Perspective

Social interaction within software development is acknowledged as an important aspect and has been the subject of a series of studies [14, 7, 19, 52, 11]. Given the role of key developers in technical tasks (e.g., maintenance and evolution of the technical core) and social aspects (e.g., development of a shared understanding of the system architecture, conflict resolution, leadership establishment, and others [15]), we are interested in analyzing how exactly these developers behave in terms of communication and coordination. Do key developers socialize more than other developers? Do they act as brokers? Do they better fulfill their coordination requirements (socio-technical congruence) when compared to others? This leads us to our second research question:

RQ 2: What is the participation of key developers in terms of communication and coordination within the project?

3.3 Characterizing Key Developers According to Their Contribution

Besides characterizing key developers from a SNA perspective, we also intend to characterize them according to their contributions to the project. Firstly, how key developers reach this important status? Do they always make core commits or do they reach this status progressively by first doing some peripheral commits and then changing the core more frequently? Such an investigation may unveil whether key developers build their skills and expertise during the course of the project or bring it from past experiences. Besides that, do key developers commit more? For instance, while conducting case studies involving the Apache Server and the Mozilla web browser, Mockus et al. [34] hypothesized that "open source developments have a core of developers who control the code base, and create approximately 80 % or more of the new functionality". Finally, we are interested in exploring whether key developers are specialists who contribute only to specific modules of the system or are generalists who contribute to different modules. Do key developers have similar patterns of contribution? This leads to our last research question:

RQ 3: What are the characteristics of the contributions of key developers?

4 RESEARCH DESIGN

In order to answer our research questions, we adopted a case study as our research method. A case study is a well-established empirical method aimed at investigating contemporary phenomena in their natural context [58]. More specifically, we conducted a descriptive case study with retrospective data collection [48]. Our case study sought to portray the characteristics of key developers by leveraging the

project's available historical information. In contrast to embedded case studies, where multiple units of analysis are studied within a case, our case study is essentially holistic, i.e., the case is studied "as a whole". In a nutshell, we focused on a particular release of a FLOSS project and gathered different types of information from it. Using a series of supporting tools (Section 4.1), we identified key developers (RQ1), characterized them according to SNA techniques (RQ2), and characterized their contributions (RQ3). In the next subsections, we introduce the design of this study by presenting the supporting tools used, the rationale for choosing the case, and the main steps followed for the identification and analysis of the key developers.

4.1 Supporting Tools

Empirical studies that mine software repositories usually demand extensive tool support due to the large amount and complexity of data to be collected, processed, and analyzed [48]. Given the different data sources required in this study, we employed and developed a variety of tools: XFlow [49], JDX[2], MMX[3], and Jung[4].

XFlow. XFlow is an extensible open source tool we developed developed. Its main goal is to support empirical software evolution analyses by considering both social and technical aspects. By bringing together these two views, the tool aims to support exploratory and descriptive case studies that call for a deeper understanding of software evolution aspects. In this study, XFlow was employed to calculate the coordination requirements network [13] and build treemaps [51].

JDX. Java Dependency eXtractor (JDX) is a Java library we developed to extract dependencies and compute the call-graph from Java code. The library relies on the robust Java Development Tools Core (JDT Core) library, which is the Eclipse IDE incremental compiler. As a desirable consequence, JDX is able to handle Java source code in its plain form. This facilitates studies that involve processing large amounts of code mined from VCSs.

MMX. Mail Message eXtractor (MMX) is a tool we developed to retrieve messages from mailing lists and compute social networks. It parses raw mailing-list archives (mboxes) and uses the header of emails to couple messages, email addresses, and threads. MMX was employed to retrieve message threads from the Ant developers' mailing list[5] and compute the communication network.

[2] `https://github.com/joseteodoro/JDX`
[3] `https://code.google.com/p/message-extractor/`
[4] `http://jung.sourceforge.net/`
[5] `http://ant.apache.org/mail/dev/`

Jung. Java Universal Network/Graph Framework is a Java library that provides a common and extensible language for modeling, analyzing, and visualizing data that can be represented as a graph or network. Jung was employed to compute network properties, such as the eigenvector centrality of nodes.

4.2 The Case Selection and Initial Data Collection

For the case study, we decided to focus on the analysis of a timeframe corresponding to the development of a particular FLOSS project release. The goal was to minimize influencing factors, since different releases may be developed using different processes/methods and may last for different periods of time. We looked for a particular project/release that met the following criteria:

1. source code hosted on a Subversion (SVN) repository with anonymous read access;
2. availability of information about the development activities (change logs and communication traces), and
3. a large active development team.

The first requirement was due to constraints of the tools we developed/used. The second one was raised because we needed historical development information to conduct the socio-technical analysis. Finally, the requirement to have a large active development team took place because we needed enough social data to answer our research questions and we were interested in identifying key developers as a subset of a large group.

We relied on data from the Ohloh platform[6] to decide about the size of the development team for our study. Ohloh is a social coding platform that monitors a huge number of open source projects by collecting information about their licenses, number of commits, number of developers, total lines of code, main programming languages, etc. By the time this paper was written, the Ohloh platform monitored 656 761 open source projects and characterized team size according to the data depicted in Table 1. Such characterization is based on a statistical analysis that the platform does on the number of active developers from the last twelve months.

After inspecting a series of FLOSS projects, we decided to analyze *Apache Ant*. This project is one of the most popular open source tools for automating software build processes. Differently from *make* and other shell-based build tools, Ant is written in Java and provides extensibility points via the implementation of Java classes. Ant operates on a user-provided XML configuration file that describes the build processes. Apache Ant is hosted by the Apache Software Foundation (ASF), which is a non-profit organization that encompasses nearly a hundred distinguishing FLOSS projects that cover a wide range of technologies and

[6] Ohloh by BLACK DUCK – `http://www.ohloh.net/`.

Number of Active Developers	Team Characterization
0	No recent development activity
1	Only a single active developer
2 or 3	Small development team
4 to 6	Average size development team
7 to 27	Large, active development team
28 or more	Very large, active development team

Table 1: Characterization of team size according to the Ohloh platform

address several problems from diverse domains. Examples of ASF projects include Apache HTTP Server, Geronimo, Cassandra, Lucene, Hadoop, Maven, and Struts.

We picked the development period of release 1.7, which started on December 19, 2003 and ended on December 13, 2006 (roughly 3 years). Such release met all preestablished criteria. Its code was hosted on a Subversion repository, information about development activities was available in commit logs and mailing lists, and there was a large team of active developers. To identify the number of active developers, XFlow was employed to parse the commit logs from the project's repository. We focused on the mainstream development portion of the project. More specifically, only commits with files from the trunk branch ("ant/core/trunk/src/main/") were considered. Besides that, commits having no java files were discarded. Applying these filters resulted in the identification of 1 834 commits done by 16 developers. Figure 3 depicts the number of active contributors in the first day of each month of the picked development period. We deem a certain developer as active in the first day of month m if the period encompassing his first and last commits in the project includes the first day of m. Such period is constrained to the 3-year development period we chose. In Figure 3, red bars denote *a single active developer*, yellow bars denote an *average-size development team*, and blue bars denote a *very large, active development team*. Such classification was done based on the data from the Ohloh platform (Table 1).

Figure 3 reveals that the Ant project had an *average size development team* during the first year of the analyzed period. Afterwards, the project had mostly a *large, active development team* (with peaks of 11 active developers).

4.3 Addressing RQ1: Identifying Key Developers

As we were interested in characterizing key developers, our first step was to discover which developers actually worked on the core source code artifacts of the Apache Ant project. This investigation required finding both the core of the technical network and the particular developers that worked on such core. We applied the following algorithm (described in pseudocode) for the 1834 commits of the case study:

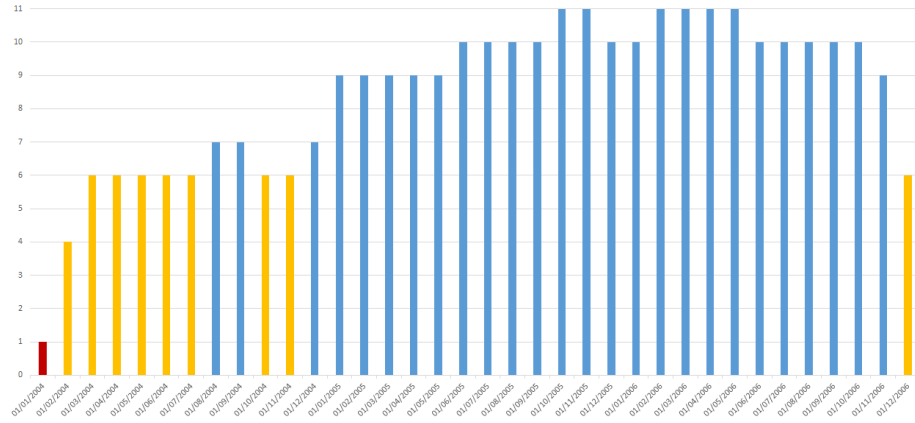


Figure 3: Number of active developers in the beginning of each month

Algorithm 2: Determining Key Developers

Input : The list of commits during the period
Output: The list of key developers

```
1.  coreCommits <-- new Empty List();
2.  for each commit do {
3.    code <-- checkoutCodeFromSVN(commit.revisionNumber);
4.    technicalNetwork <-- JDX.calculateClusteredCallGraph(code);
5.    technicalNetworkCore <-- obtainCore(callGraph, egv);
6.    if (isCoreCommit(commit, technicalNetworkCore)){
7.      coreCommits.add(commit);
8.    }
9.  }
10. keyDevelopers <-- determineKeyDevs(coreCommits);
```

For a given commit, we checked out its corresponding code from Subversion (line 3). Then, we used JDX to calculate the project's static call-graph (line 4). JDX clusters the method nodes belonging to the same Java file, which results in a graph in which nodes represent the Java files and the edges represent their calling relationships. We then employed the eigenvector centrality metric and a quartile analysis to find the core of the technical network produced in the prior step (line 5). We recall that such metric embodies the notion that a node's importance in a network increases by having connections to other vertices that are themselves important [36]. Thus, we assume that a source code file becomes more important by having connections to other source code files that are themselves important. The files that had a centrality score equal to or larger than the third quartile (Q3) were deemed as core. Then, we determined whether a commit was core or pe-

ripheral (line 6). In this study, we considered that core commits were those that contained at least one source code file that belonged to the core of the technical network. Those commits showed that the developer in question was actually able to modify the core of the project (even if the developer changed a single core file). Finally, after calculating the list of core commits, we determined the key developers (line 10). We accomplished that by computing the absolute number of core commits per developer and performing a quartile analysis on such distribution. Developers whose number of core commits was equal to or larger than Q3 were deemed as key. We chose the absolute number of core commits because it determines the frequency with which each developer contributed to the technical core. In this sense, we consider that developers who produce few core commits (even if that accounts for all their commits) cannot be deemed as key developers.

4.4 Addressing RQ2: Characterizing Key Developers from a Social Network Analysis Perspective

This subsection describes how we characterized developers from a social network analysis perspective. In summary, we first built communication networks from the project's mailing list using three different approaches. After that, we analyzed these networks using SNA metrics. Next, we calculated the coordination requirements network and found its core. Finally, we calculated the socio-technical congruence of each key developer.

4.4.1 Building the Communication Networks

To characterize key developers from a social network analysis perspective, we built the communication network from the projects' mailing list. In the literature, the construction of a communication network from mailing lists is done in different ways depending on the purpose of the study at hand. Hence, we decided to evaluate how the key developers fit into communication networks calculated according to three different strategies (Figure 4), namely: *Prior* [26], *FirstAndPrior*, and *TransposeAndTimes* [56, 3]. In the following, we describe each of these strategies.

Prior. This method results in an undirected graph where vertices are developers and edges link developers. An edge links developers a and b when either a directly replies to b or b directly replies to a (Figure 4 a)). We rely on the Response-ID and Message-ID fields from the email headers to create these connections.

FirstAndPrior. As the previous strategy, this one creates edges based on the Message-ID and Response-ID fields. However, it additionally creates links between the developer who initiates the thread and all other participants (Figure 4 b)). A reasonable interpretation is that all developers in the thread are

indirectly responding to the first message. This method is biased towards the developers who started a thread, since those initiators are always linked to those who responded them.

TransposeAndTimes. This method links together everyone that participated in the same thread (Figure 4 c)). To compute this graph, a Developer × Thread matrix is created. In such matrix, the cell $[i, j]$ is assigned the value one when developer i participated in thread j (i.e., the developer sent a message in this thread). After this matrix is completed, we multiply it by its transposed version to obtain a Developer x Developer matrix. The output of the method is built on top of this last matrix: links are created between developers i and j when the matrix cell $[i, j]$ has a value greater than one.

We collected and parsed data from the developers' mailing list[7] using the MMX tool we developed. We highlight that we collected messages from the release development period only (i.e., from December 19, 2003 to December 13, 2006). Afterwards, we built communication networks according to the three aforementioned strategies.

4.4.2 Analyzing the Communication Networks Using SNA Metrics

We analyzed the communication networks using SNA metrics (Section 2.3.1). The goal of this analysis was to better understand how key developers behave in terms of communication. We employed Degree Centrality and Eigenvector Centrality to investigate whether key developers socialized more than others did. Next, we employed Betweenness Centrality to assess if key developers acted as brokers that end up bridging other developers. Finally, we used Closeness Centrality to analyze the communication distance to the other developers. In all three analyses, we distinguished high centrality values by performing a quartile analysis and determining those that were higher than or equal to the third quartile. All SNA metrics were calculated using Jung 4.1.

4.4.3 Building the Coordination Requirements Network and Determining Its Core

The coordination requirements network depicts the set of other developers that a certain developer should coordinate his/her work with. These requirements arise as a consequence of developers working on interdependent artifacts. We built the project's coordination requirements network by applying the method introduced by Cataldo et al. [13, 12, 11] (Section 2.3.2). We used XFlow to build the *Ta* and *Td* matrices. *Ta* matrix denoted which files were changed by each developer over the release development period. *Td* matrix was built considering the co-changes that occurred during the same period. As co-changes are very sensitive to commits with crosscutting changes [41] (such as changing license or fix-

[7] `http://mail-archives.apache.org/mod_mbox/ant-dev/`

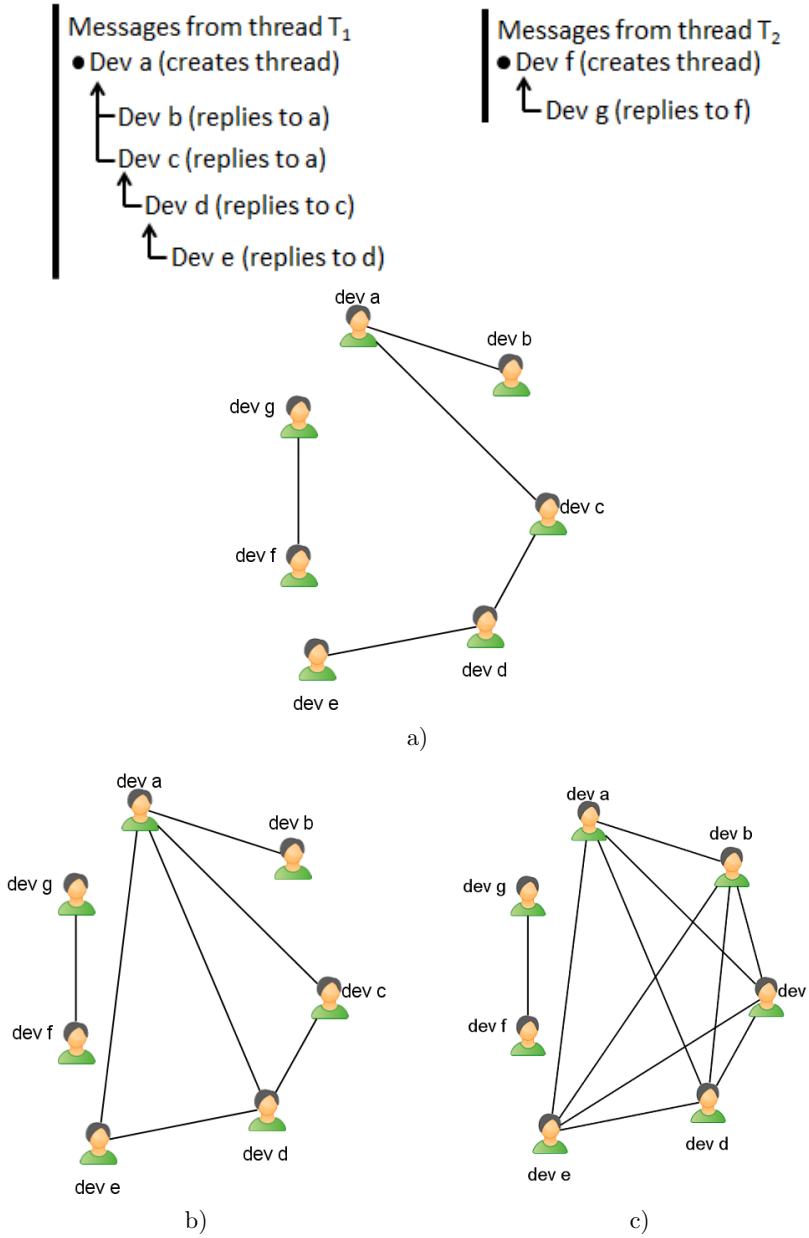


Figure 4: Multiple methods to define a communication network based on an email thread: a) Prior, b) FirstAndPrior, c) TransposeAndTimes

ing code style), we decided to filter out such commits. We accomplished that by determining the *number of files per commit* distribution and finding the outliers. More precisely, we ignored those commits in which the number of files was above $(Q3 + 1.5 * IQR)$, where Q3 indicates the third quartile and IQR indicates the interquartile range.

With the coordination requirements network in hands, we determined its core. Analogously to the case of the technical network, we proceeded by calculating the Eigenvector Centrality of each node and employing a quartile analysis. This way, we were able to investigate the extent to which key developers were supposed to coordinate their work with other developers.

4.4.4 Determining the Socio-Technical Congruence

Inspired by the measure of socio-technical congruence defined by Cataldo et al. [13, 12, 11] (Section 2.3.2), we computed the proportion of social activity that actually occurred (given by a communication network) relative to the social activity that should have occurred (given by the coordination requirements network) *for each developer*. Since we calculated the communication network according to three different strategies, we also calculated three congruence lists. For each list, we performed the same quartile analysis of previous scenarios: congruence values that were equal to or higher than the third quartile were deemed as high. This analysis revealed whether key developers have a high socio-technical congruence.

4.5 Addressing RQ3: Characterizing Key Developers' Contributions

After having characterized key developers from a SNA perspective, we started to investigate their contributions. First, we investigated how interleaved their core and non-core commits were and how core commits were distributed over time. The goal was to better understand how key developers reached this status, i.e., were they always key developers or did they reach this status progressively? Afterwards, using XFlow, we computed the list of top contributors, i.e., those developers that made most part of the commits. More precisely, we determined the list of top committers by analyzing the distribution of commits per developer. The goal was to check if key developers were also top committers. In particular, as we stated in Section 3.3, Mockus et al. [34] hypothesized that "open source developments have a core of developers who control the code base, and create approximately 80 % or more of the new functionality". Since we were not able to differentiate new features from bug fixes, we tested a more general hypothesis: we verified whether key developers did 80 % or more of the total number of commits. Finally, we also intended to investigate whether key developers tended to be specialists or generalists. For this purpose, we leveraged the treemap visualization provided by XFlow and analyzed how dispersed their contributions were. Treemap is a compact visualization method that uses nested rectangles to display information with hierarchical charac-

teristics [51]. In our case, the treemaps operate on all directories and Java files inside the *"ant/core/trunk/src/main"* path. In particular, leaf rectangles represented files and non-leaf rectangles represented folders. We built a treemap for each key developer and colored the non-leaf rectangles (source code files) that he or she worked on.

5 RESULTS

5.1 Identification of Key Developers

As mentioned before, we computed the technical network at the time of each commit and calculated the core of such network, identifying if the commit contained a core modification. After that, we calculated the number of core modifications made by each developer. In Table 2, we depict the results we obtained:

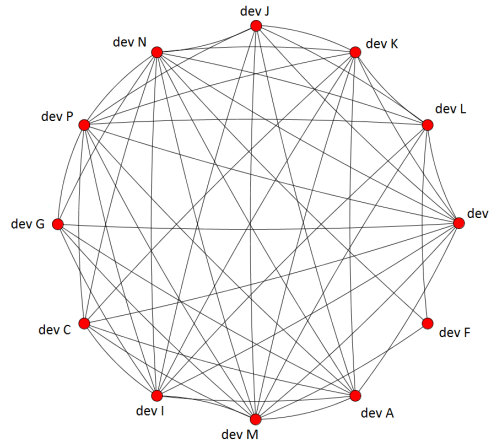
Developer	Number of Core Commits	Delta	Key Developer
dev A	0	0	NO
dev B	0	0	NO
dev C	1	1	NO
dev D	2	1	NO
dev E	3	1	NO
dev F	3	0	NO
dev G	5	2	NO
dev H	7	2	NO
dev I	18	11	NO
dev J	28	10	NO
dev K	31	3	NO
dev L	58	27	NO
dev M	89	31	YES
dev N	147	58	YES
dev O	182	35	YES
dev P	232	50	YES

Table 2: Developers and associated number of core modifications to the system

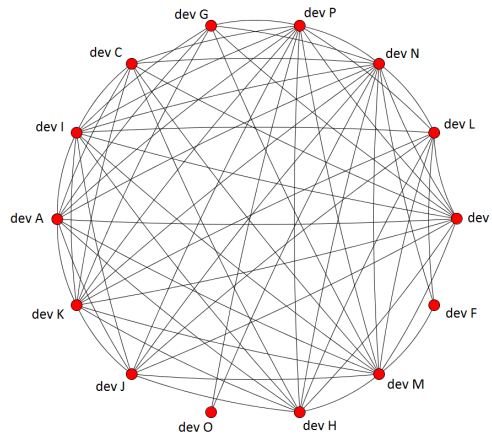
We sorted the developers according to the number of core commits they performed. The third column of the table (delta) shows the difference between the number of commits of a developer and his predecessor. The total number of core commits was 806, which represents 44 % of all commits. The third quartile of the number of core commits per developer was 81.25. Four developers made more than 81 core commits, namely: dev M, dev N, dev O, and dev P. Interestingly, this set of 4 key developers was responsible for 81 % of all core commits. Hence, the Pareto Principle holds for this project.

5.2 Characterizing Key Developers from a Social Network Analysis Perspective

We used MMX to compute the three different communication networks of the project. Figure 5 depicts the results we obtained in the form of graphs in which vertices represent developers and edges represent the connections among them. The graphs are shown using a circular layout because it makes it easier to determine the density of the networks and the number of connections a certain node has.

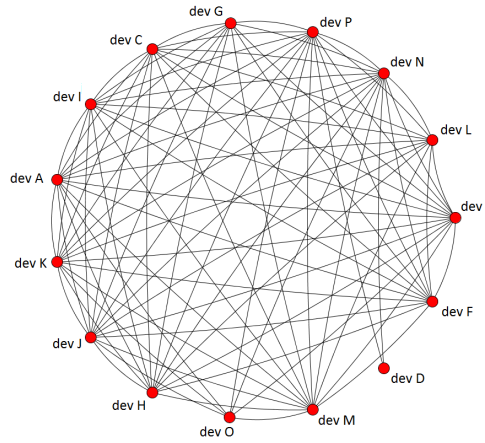


a)



b)

The rationale behind the construction of each communication network led to networks of different sizes. The communication network built using *Prior* had



c)

Figure 5: Communication networks of Apache Ant calculated using a) Prior, b) FirstAndPrior, and c) TransposeAndTimes strategies

12 developers and 51 connections. The communication network built using *FirstAndPrior* had 14 developers and 64 connections. The communication network built using *TransposeAndTimes* was the largest one with 15 developers and 86 connections. In the next subsections, we present the results of analyzing these social networks.

5.2.1 Analyzing the Communication Networks Using SNA Metrics

We first wanted to understand whether key developers socialize more. We operationalized that by inspecting the communication networks built from the project's mailing list. To this end, we calculated the Degree Centrality and the Eigenvector Centrality metrics for the three networks. These two metrics essentially inform how coupled nodes are in a network. The results are depicted in Table 3, where cells in green denote values that are higher than or equal to the third quartile of each column.

Even though the two centralities inform coupling from different perspectives (local vs global) and the communication networks were built using different strategies, the results did not change. The developers *dev M*, *dev P*, and *dev N* had high scores in all cases, which indicates that they socialized more than others did. Interestingly, *dev O* had a very distinct behavior: he did not socialize much (he did not even appear in some of the networks).

Subsequently, we investigated whether key developers acted as brokers (or bridges) in the communication networks. To this end, we analyzed the three networks using Betweenness Centrality. The results are depicted in Table 4.

	Degree			**Eigenvector**		
	P	FP	TT	P	FP	TT
dev E	10	11	13	0.098	0.086	0.076
dev F	3	3	12	0.029	0.023	0.070
dev D			2			0.012
dev M	11	12	13	0.108	0.094	0.076
dev O		2	7		0.016	0.041
dev H		10	12		0.078	0.070
dev J	8	9	13	0.078	0.070	0.076
dev K	9	11	13	0.088	0.086	0.076
dev A	9	10	13	0.088	0.078	0.076
dev I	10	11	12	0.098	0.086	0.070
dev C	7	8	12	0.069	0.063	0.070
dev G	6	7	11	0.059	0.055	0.064
dev P	10	12	14	0.098	0.094	0.081
dev N	11	13	14	0.108	0.102	0.081
dev L	8	9	11	0.078	0.070	0.064
Q3	10	11	13	0.098	0.088	0.076
SUM	102	128	172	1.000	1.000	1.000

Table 3: Do key developers socialize more?

	Betweennes		
	P	FP	TT
dev E	1.152	0.988	0.948
dev F	0.000	0.000	0.091
dev D			0.000
dev M	4.319	4.488	0.948
dev O		0.000	0.000
dev H		0.393	0.091
dev J	0.143	0.125	0.948
dev K	0.452	0.988	0.948
dev A	0.643	0.554	0.948
dev I	1.152	0.988	0.091
dev C	0.000	0.000	0.091
dev G	0.000	0.000	0.000
dev P	1.152	5.988	6.948
dev N	4.319	10.488	6.948
dev L	1.667	2.000	0.000
Q3	1.538	2.622	0.948
SUM	15.000	27.000	19.000

Table 4: Do key developers act as brokers/bridges?

Again, key developers had a distinct participation. For all three networks, *dev M*, *dev P*, and *dev N* had distinctly high centrality scores (except for *dev P* in the *Prior* network). Therefore, we conclude that these key developers are not only connected to many others (either directly or indirectly), but also serve as bridges between other developers. The key developer *dev O* had zero betweenness for *FirstAndPrior* and *TransposeAndTimes*.

Finally, we wanted to investigate how close the key developers were to the other developers. The closer they are, the easier it is for them to reach other developers (and vice-versa). To this end, we calculated the Closeness Centrality metric for each of the three communication networks. The results are depicted in Table 5.

	Closeness		
	P	FP	TT
dev E	0.917	0.867	0.933
dev F	0.579	0.565	0.875
dev D			0.538
dev M	1.000	0.929	0.933
dev O		0.542	0.667
dev H		0.813	0.875
dev J	0.786	0.765	0.933
dev K	0.846	0.867	0.933
dev A	0.846	0.813	0.933
dev I	0.917	0.867	0.875
dev C	0.733	0.722	0.875
dev G	0.688	0.684	0.824
dev P	0.917	0.929	1.000
dev N	1.000	1.000	1.000
dev L	0.786	0.765	0.824
Q3	0.917	0.882	0.933
SUM	10.014	11.125	13.019

Table 5: Are key developers close to other developers?

The results we obtained put key developers in the spotlight. Developers *dev M*, *dev P*, and *dev N* had a centrality score higher than the third quartile. Developer *dev O* had the lowest closeness score in the *FirstAndPrior* network and the second lowest score in the *TransposeAndTimes* network.

5.2.2 The Core of the Coordination Requirements Network

We used XFlow to calculate the coordination requirements network (Figure 6). Each vertex represents a developer and each edge maps two developers that were supposed to coordinate their efforts because the artifacts they changed were interdependent

from an evolutionary point of view. The coordination requirements network we obtained was dense, with 16 developers and 102 connections.

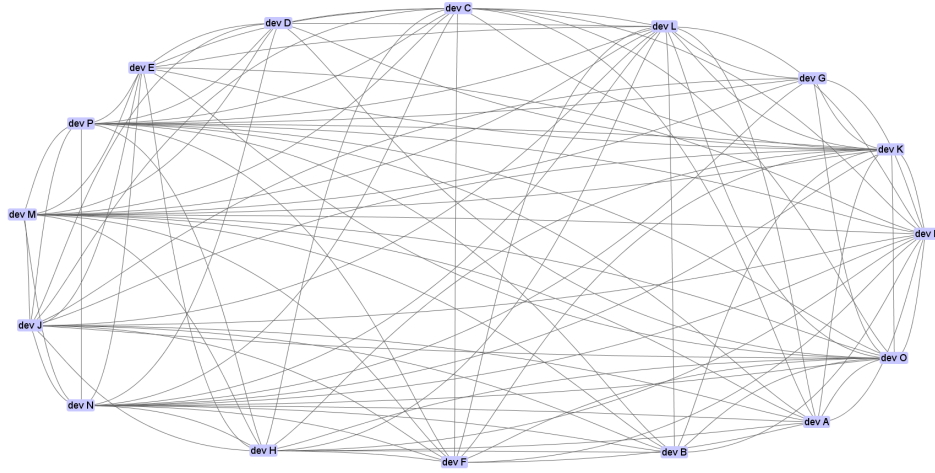


Figure 6: The coordination requirements network

We further analyzed this network by determining its core and checking whether key developers belonged to it. The goal was to better understand the extent to which key developers were supposed to coordinate their work with other developers. The results we obtained are depicted in Table 6, where green cells denote the developers in the core.

The results indicate that all key developers were in the core of the coordination requirements network: *dev P*, *dev N*, *dev O*, *dev M*. Indeed, these four developers had connections to all other developers in the network.

5.2.3 Socio-Technical Congruence

We computed the socio-technical congruence of each developer. Figure 7 depicts the results we obtained. The data show that the interval of congruence values is large (ranging from 0 % to 100 %). We also performed a quartile analysis to identify developers with high congruence. The results we obtained were the following:

High Congruence for Prior: *dev N*, *dev M*, *dev E*, *dev I*

High Congruence for FirstAndPrior: *dev N*, *dev M*, *dev E*, *dev P*,

High Congruence for TransposeAndTimes: *dev N*, *dev E*, *dev P*, *dev A*, *dev F*

5.3 Characterizing Key Developers' Contributions

We investigated how interleaved core and non-core commits were for each key developer. The results are depicted in Figure 8. In this figure, each single vertical bar

	Eigenvector Centrality Score
dev P	0.074
dev N	0.074
dev M	0.074
dev K	0.074
dev C	0.064
dev O	0.074
dev D	0.044
dev H	0.064
dev G	0.039
dev J	0.074
dev L	0.074
dev F	0.054
dev B	0.044
dev A	0.054
dev E	0.054
dev I	0.069
Q3	0.074
SUM	1

Table 6: Developers in the core of the coordination requirements network

denotes the activity of one week. If a developer only produces core commits during a week, then the bar representing that week becomes totally green. Analogously, if a developer produces only non-core commits during a week, then the bar becomes totally red. If half of the commits are core, then the bar becomes half green and half red.

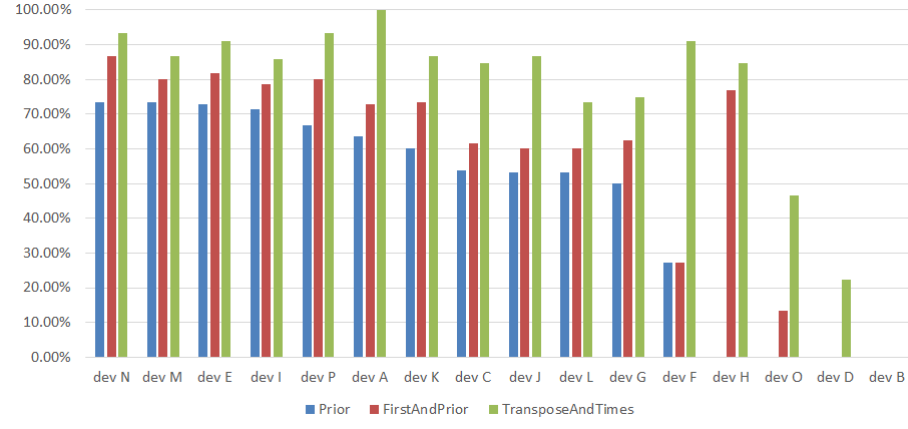


Figure 7: Socio-technical congruence of the developers

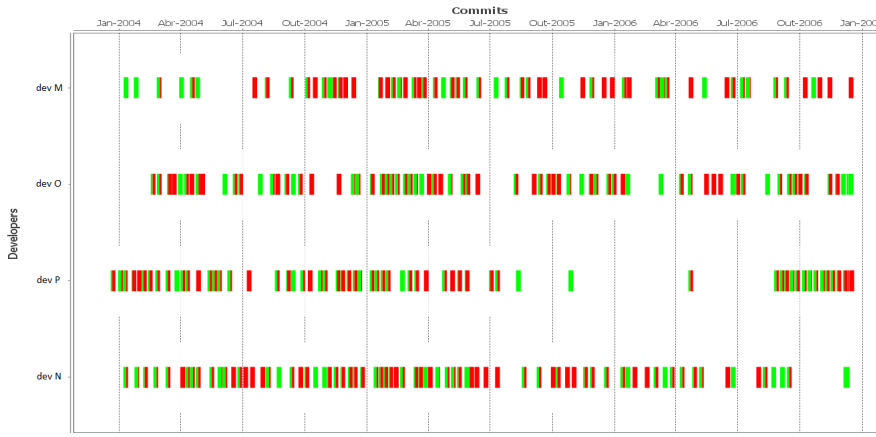


Figure 8: Key developers: Interleaving of core and peripheral commits

Apart from the first contributions of *dev M*, we can clearly notice that core and non-core commits were very much interleaved for all key developers. This shows that these developers were very important for the project over the whole release development period. This picture also suggests that the proportion of core commits was very stable for each key developer. We further investigated this phenomenon by calculating, for each key developer, the cumulative count of core and non-core commits, as well as the cumulative percentage of core commits. The results are depicted in Figure 9.

The key developer *dev M* had higher ratio of core commits in his first contributions. However, starting from his 46^{th} commit, the number of non-core commits dominated (Figure 9 a)). From his 64^{th} commit onwards, the ratio of core commits stabilized at the 35 % to 42 % range (Figure 9 b)). The key developer *dev N* also had a higher ratio of core commits in his first contributions. Although it took just a little bit longer, the turnover also ended up occurring (Figure 9 c)). The number of non-core commits also grew at a little bit higher pace. However, from his 262^{nd} commit onwards, the ratio of core commits stabilized at the 38 % to 41 % range (Figure 9 d)). The key developer *dev O* had much more interleaved commits. However, as in the previous cases, the turnover also occurred at some point (Figure 9 e)). Starting from his 66^{th} commit, the rate of core commits stabilized at the 45 % to 53 % range with a descending trend at the very end (Figure 9 f)). Finally, the key developer *dev P* had a very stable contribution pattern. Differently from others, his ratio of non-core commits was often higher (except for very rare circumstances that happened in the beginning) (Figure 9 g)). Right from his 72^{th} commit onwards, his ratio of core commits stabilized at the 41 % to 47 % range (Figure 9 h)). Given the temporal dimension provided (Figure 8), it is possible to notice that this developer produced very few commits from the middle of July, 2005 until the middle of July, 2006 (one year period).

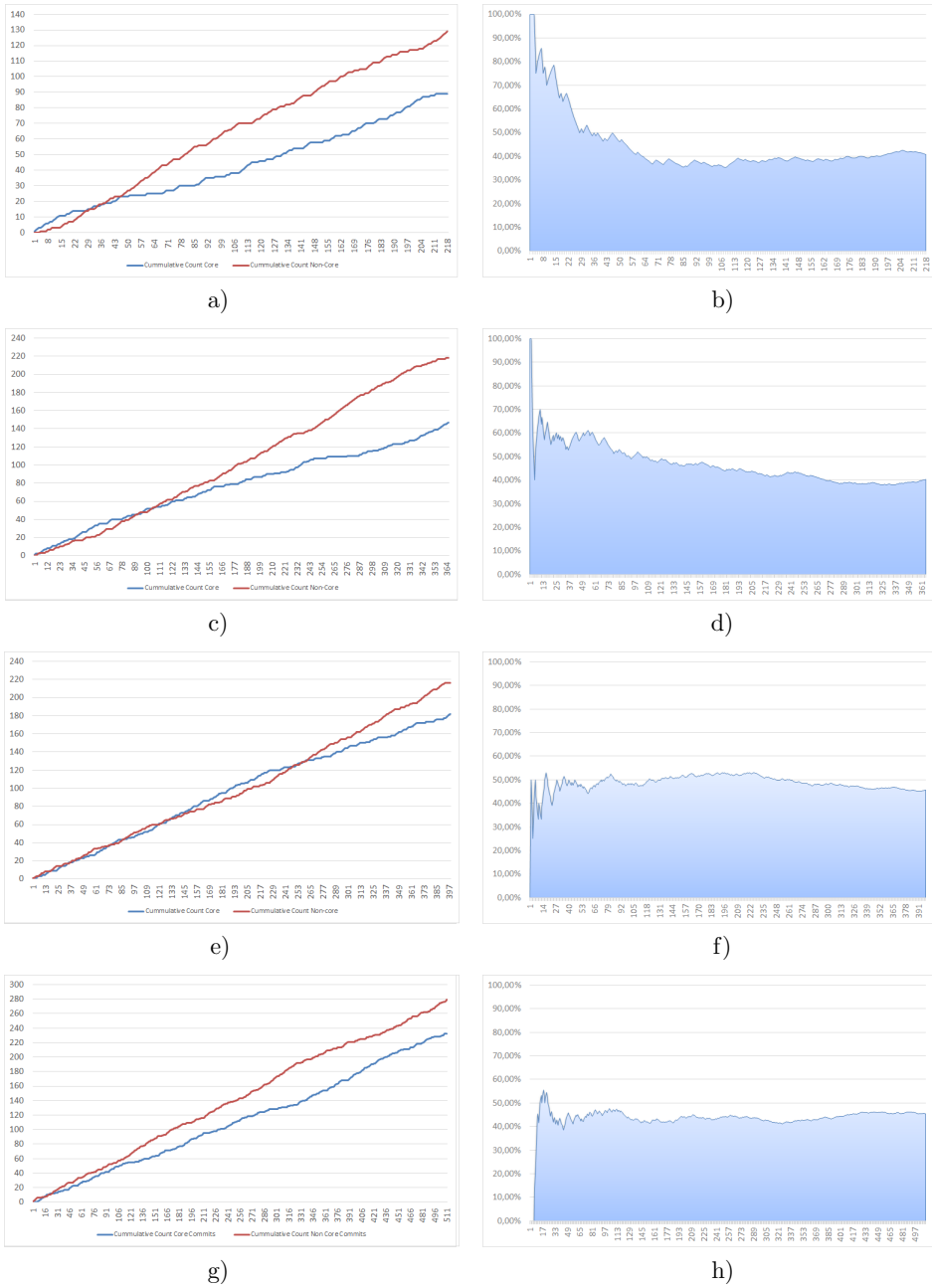


Figure 9: Core commits vs. non-core commits (left) and cumulative percentage of core commits (right): *dev M* a), b), *dev N* c), d), *dev O* e), f), *dev P* g), h)

Following this analysis, we employed XFlow and calculated the set of top contributors. Figure 10 depicts the cumulative percentage of the number of commits. According to the data, the four key developers (25 % of them) were responsible for 81 % of the commits. Therefore, we conclude that the operational version of Mockus' hypothesis we defined indeed holds for this release of the Ant project.

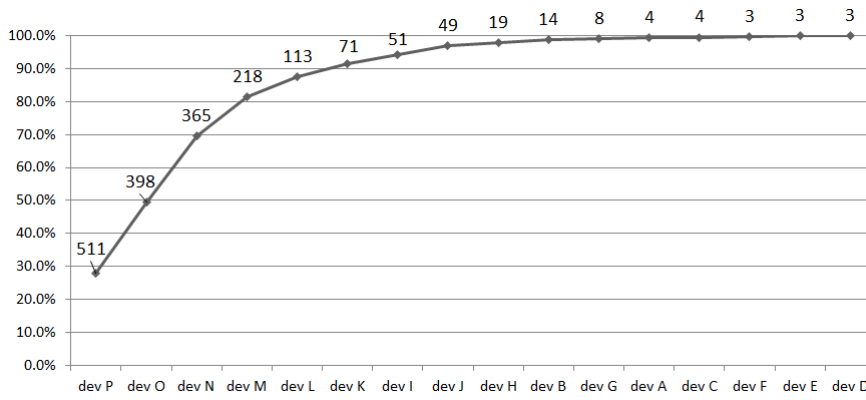
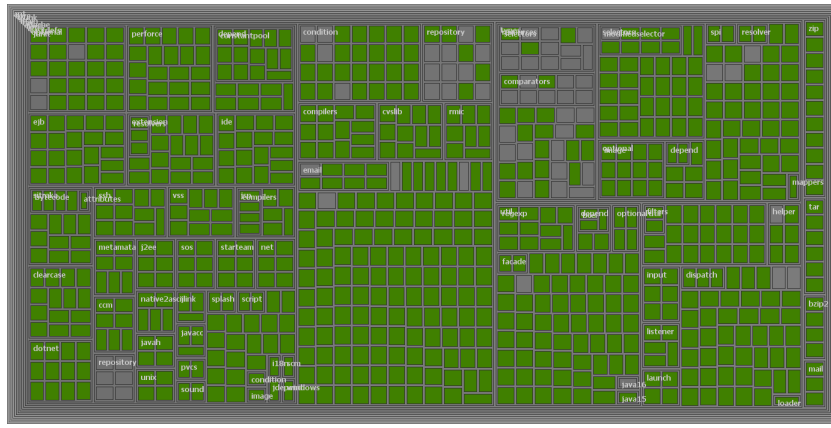
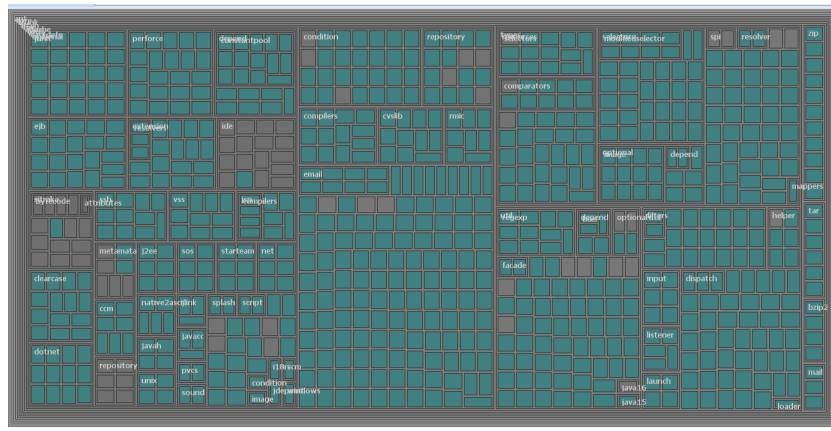


Figure 10: Cumulative percentage of the number of commits

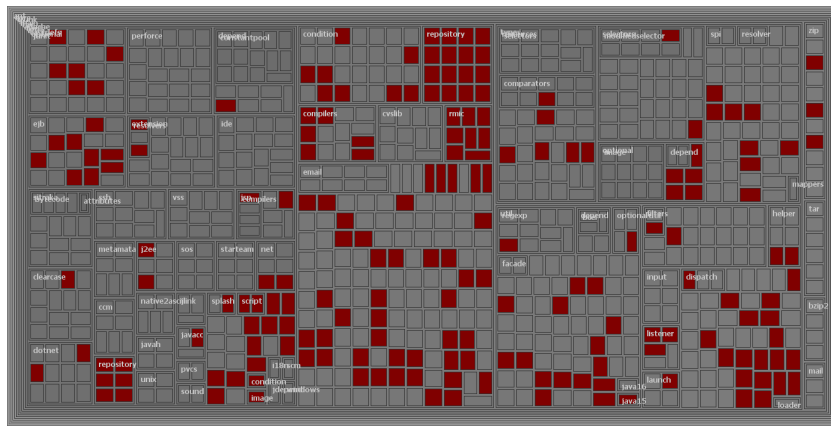
Finally, to discover the focus of the key developers' contributions, we built treemaps. Each treemap depicts the files that a certain developer worked on. Figure 11 shows the treemap of each key developer. The first treemap (Figure 11 a)) depicts the files that *dev P* worked on. His treemap is almost full-colored, which indicates that he worked on many classes of the project during the studied release. Interestingly, he did not touch any file in the repository folder (bottom left portion of the figure). The second treemap (Figure 11 b)) belongs to *dev N*. This treemap is also very colored and shows that the developer worked on many files. Again, no files from the repository folder were touched. The third treemap (Figure 11 c)) belongs to *dev M*. Although his treemap is not as colored as the previous ones, it is possible to notice that his contributions are spread all over the system. In particular, differently from the two previous key developers, he worked on all files from the repository folder. Finally, the last treemap (Figure 11 d)) depicts the files that *dev O* touched. Although *dev O* did more commits than *dev N*, his treemap is much less colored. In this sense, his contributions are more focused than *dev N's*. In particular, his contributions seem to be a little bit more concentrated on the right-hand side of the treemap. However, it is also true that he worked on many different classes of the system.


perforce
condition
repository
comparators
compilers
cvslib
rmic
ejb
ide
email
vss
metamata
j2ee
sos
starteam
net
clearcase
native2ascii
splash
script
ccm
javah
javacc
dotnet
repository
pvcs
unix
sound
image
facade
input
dispatch
listener
launch
java16
java15
loader
zip
mappers
tar
helper
bzip2
mail

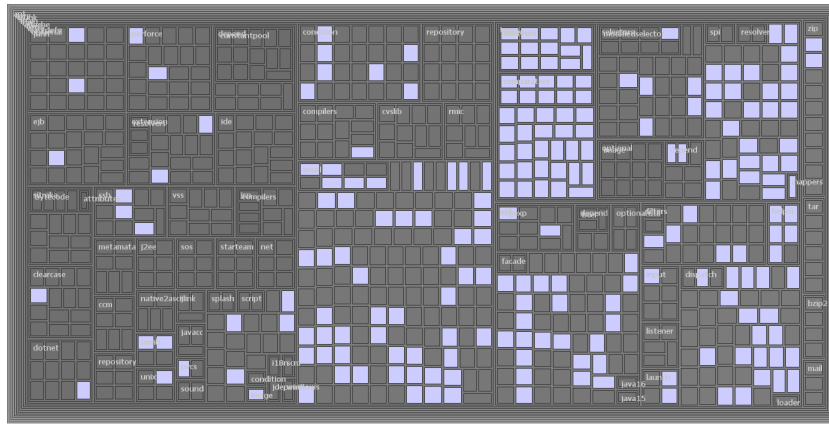
a)


perforce
condition
repository
comparators
compilers
cvslib
rmic
ejb
ide
email
vss
metamata
j2ee
sos
starteam
net
clearcase
native2ascii
splash
script
ccm
javah
javacc
dotnet
repository
pvcs
unix
sound
image
facade
input
dispatch
listener
launch
java16
java15
loader
zip
mappers
tar
helper
bzip2
mail

b)


perforce
condition
repository
comparators
compilers
cvslib
rmic
ejb
ide
email
vss
metamata
j2ee
sos
starteam
net
clearcase
native2ascii
splash
script
ccm
javah
javacc
dotnet
repository
pvcs
unix
sound
image
facade
input
dispatch
listener
launch
java16
java15
loader
zip
mappers
tar
helper
bzip2
mail

c)



d)

Figure 11: Contribution treemaps: a) *dev P*, b) *dev N*, c) *dev M*, and d) *dev O*

5.4 Summary of Findings

We summarized our findings in Table 7. This table depicts the list of key developers, developers in the core of the communication networks (Eigenvector Centrality analysis), developers in the core of the coordination requirements network, developers with high socio-technical congruence, and top contributors.

We identified four key developers, namely: *dev P*, *dev N*, *dev O*, and *dev M*. Although the core of the communication networks was different, three of the key developers appeared in all of them. This indicates that most key developers were also very active in the developers' mailing list during the analyzed period. Interestingly, while the core of *Prior* and *TransposeAndTimes* communication networks included other developers, *FirstAndPrior* had the best match with the set of key developers. Moreover, although the *FirstAndPrior* network has a larger number of links than *Prior*, its core is smaller.

In relation to the core of the coordination requirements network, all key developers belonged to it (although it included three more developers). This means that key developers indeed had to coordinate their efforts with a large number of people. We also computed the socio-technical congruence by comparing the communication networks with the coordination requirements network. Two key developers (*dev N* and *dev M*) had a high congruence when considering Prior, three key developers (*dev P*, *dev N*, and *dev M*) had a high congruence when considering *FirstAndPrior*, and two key developers (*dev P* and *dev N*) had a high congruence when considering *TransposeAndTimes*. The congruence calculated based on *FirstAndPrior* showed the best match with key developers. The key developer *dev O* did not appear in any of the calculated congruencies. Curiously, the congruencies C1 and C3 had a very small intersection (only *devs N* and *E* appeared in both). Interestingly,

Developer	Key Developer	Important in Communication Network			Core of Coordination Requirements Network	High Congruence			Top Contributors
		P	FP	TT		C1	C2	C3	
dev P	✔	✔	✔	✔	✔		✔	✔	✔
dev N	✔	✔	✔	✔	✔	✔	✔	✔	✔
dev O	✔				✔				✔
dev M	✔	✔	✔	✔	✔	✔	✔		✔
dev L					✔				
dev J				✔	✔				
dev K				✔	✔				
dev E		✔		✔		✔	✔	✔	
dev H									
dev G									
dev C									
dev F								✔	
dev I		✔				✔			
dev A				✔				✔	
dev D									
dev B									

Table 7: Summary of findings

the sets of key developers and top contributors were identical. In fact, by taking a closer look at the contribution volumes, we conclude that the set of key developers also heavily contributed to the peripheral areas of the technical network. Finally, despite the various networks we calculated and analyses we performed, five developers did not show up in any of them, namely: *dev H*, *dev G*, *dev C*, *dev D*, and *dev B*.

Finally, we notice that *dev N* had a mark for each criterion included in Table 7. Furthermore, *dev P* and *dev M* also missed one socio-technical congruence evaluation each. In other words, the rows corresponding to these developers are much more filled than those associated with other developers. In turn, the developer *dev O* exhibited a very different behavior, since he was a key developer and top committer who participated very rarely in the mailing list.

6 DISCUSSION

In the following, we discuss our research questions in light of the results we obtained. We conclude the section by discussing threats to the validity of the results.

6.1 RQ 1: How to Identify Key Developers?

To identify key developers, we conceived the method described in Section 4.3. In our case study, 4 developers were responsible for 81 % of all core commits. This corroborates the hypothesis that a small number of developers becomes responsible for the technical core of a system.

It is worth to note that we automated the whole process of identifying key developers in a way that it is possible to reuse it for any Java project. Given the motivation presented in Section 3.1, we believe this methodological and technological support can be leveraged for various scientific and practical purposes.

We faced some challenges from the technical point of view to operationalize our method, since it required processing 1834 code snapshots and calculating the associated core of each corresponding technical network. To that end, we had to build our own tool (JDX) for calculating dependencies from Java code. We needed to develop a tool to operate directly on source code due to the difficulties associated with compiling and building automatically each version of a software. We also needed to develop a tool for extracting different social networks from the mailing list.

On the other hand, challenges still exist for replicating our study considering a large number of projects. Key developers and the social networks are identified on a delimited timeframe basis, such as a release. Using releases as the unit of analysis requires gathering the release dates of all projects, which might not be available in the project's documentation. In addition, different projects have different release cycle policies. In particular, releases may vary from short ones (e.g., Firefox) to long ones (e.g., our case study). Moreover, the same project might even change its own release cycle policy over time. Apache Ant, for example, has both very short and very long releases. These facts suggest that the specific timeframes to be analyzed should be carefully chosen on a case basis. Difficulties also come from the fact that some projects make extensive use of branching. In those cases, analyzing only the trunk would provide misleading results. Therefore, doing a large scale study would require developing a rationale for determining whether branches need to be considered, as well as a way to group the results from different branches.

6.2 RQ 2: What Is the Participation of Key Developers in Terms of Communication and Coordination within the Project?

The first step required building the communication network from the mailing-list archive of the project. In particular, we built communication networks according to three different strategies: *Prior*, *FirstAndPrior*, and *TransposeAndTimes*. Analyzing such network with SNA metrics showed interesting patterns. Three of four key developers were very active in the mailing lists, having high scores for different centrality measures. Given the specific set of metrics we considered, we concluded that those three key developers socialize more than other developers, act

as bridges linking other developers, and were socially close to them. However, the key developer *dev O* had a very distinct behavior. He communicated very rarely in the mailing list and did not appear in the communication network built using *Prior*.

Calculating the coordination requirements for the project resulted in a dense network. This indicated that developers shared many interdependent tasks. We highlight that all four key developers were in the core of this network. Indeed, key developers were connected to all other developers. In the next step, we calculated the socio-technical congruence of each developer. As we had three communication networks, we obtained three different associated scores for each developer. With the exception of *dev O*, each key developer had at least two high scores. As expected, *dev O* had a low congruence in all three cases.

In summary, these results show that three of the four key developers often communicate to others (in the mailing list) and have high socio-technical congruence. A key developer did not exhibit this behavior, but performed a large number of core commits. Cataldo et al. [13] stated that *most productive workers reach higher levels of congruence than the less productive ones*. In our case, this did not happen with one of the key developers. As Cataldo's study was conducted with *industrial* projects, further investigation is necessary under open source settings.

6.3 RQ 3: What Are the Characteristics of the Contributions of the Key Developers?

We investigated how key developers reached this status. The results of the analysis of their commits unveiled interesting patterns. Firstly, we noticed that core and non-core commits were often interleaved. This indicates that all key developers brought their skills and expertise from contributing to previous releases of the system (or even other systems). A further examination also revealed that, for each developer, the proportion of non-core commits eventually became higher than that of core commits. Also, by the end of the release development period, key developers had similar ratios of core commits (ranging from 40 % to 45 %). Hence, we concluded that key developers exhibited similar contribution behaviors in terms of the frequency that they touched the technical core.

Afterwards, we determined the set of top contributors by calculating the number of commits made by each developer. Key developers were the top four committers, being responsible for 81 % of all commits. This showed that the operational version of Mockus' hypothesis we defined held for the case we studied. We further examined the situation by investigating how dispersed their contributions were. In general, all key developers contributed to many different parts of the system. In particular, *dev P* and *dev N* had almost full-colored treemaps. At the same time, neither of them changed any files from a particular package called "repository". Later on, we observed that only *dev M* touched that package. Furthermore, even though *dev O* had a low socio-technical congruence, he was able to make changes to different packages of the system. On the other hand, we hypothesize that his socio-technical

congruence indeed played a role in his contributions. We discovered that, although he committed more than *dev N*, his treemap was much less colored, indicating that his contributions were more focused than *dev N*'s. In other words, we hypothesize that the low participation of *dev O* in the communication network posed difficulties for him to touch certain portions of the code. In general, this last analysis showed that the key developers not only contributed to the core of the system, but also to peripheral areas.

6.4 Threats to Validity

Some factors may have influenced the validity of our study. A common practice in FLOSS development concerns the submission of patches by external developers. As these developers do not have permission to commit their fixes to the projects' version control system, their contributions are often committed by one of the regular project developers – and SVN does not record the original author. As a result, this may have introduced some noise in the data used to calculate key developers. Furthermore, we gathered communication data just from the mailing list, so any communication outside such list was not considered.

The methods we employed to build the communication networks also incur some limitations. The *Prior* method links the author of a message to its direct repliers, thus assuming that each replier reads at least the last available message. The method *FirstAndPrior* is less restrictive, as it also links repliers to the author that initiated the message thread. This method thus assumes that repliers also read the message that started the thread. This may lead to bias in projects which designate specific members to start discussion threads in the mailing list, since links will always be created between thread responders and the thread initiator. The *TransposeAndTimes* is the less restrictive method, since it links all thread contributors to each other. This method models the assumption that each thread contributor reads all messages before writing its own. In summary, each of the three methods models a particular kind of participant behavior and can be seen as complementary. Interestingly, our results revealed that most key developers could be deemed communicative regardless of the particular way in which the communication network was built.

The adoption of the Eigenvector Centrality metric to define the core of a network might have affected our findings. We believe that this measure captures a behavior that seems adequate to our analysis, but we acknowledge that other approaches (e.g. k-core or islands) would likely provide different results. Regarding the method we used to define key developers, we initially tried a different approach that considered a commit to be core when at least half of its files belonged to the technical core. The list of key developers we obtained in that case was identical to the one presented in this paper. Hence, we decided to keep the simpler version of the method.

The inclusive nature of the algorithm we employed to calculate the coordination requirements network may have generated a network that is denser than it should be. This may have also leaded to a core that is also larger than it should be.

In this study, we adopted a quartile analysis approach to support the identification of key developers and the networks' core. This strategy enabled us to compare distributions under the same settings. However, it is possible that alternative statistical techniques would lead us to different conclusions. In addition, our analysis method assumes that all studied networks possess a core structure, which might not hold true for all projects. Finally, our analysis focused on data from two repositories only: the mailing list and the version control system. As a consequence, we may have missed empirical evidence that could be found in other repositories or in other releases of the project.

In relation to the external validity, since we studied a single release of a project, we cannot state that these results remain valid for other projects or even for the entire Apache Ant project. In fact, threats to the generalizability of this study are inherent to the nature of the employed research design. McGrath [33] states that no research method can satisfy adequately the following three dimensions at the same time: generalizability, realism, and precision. In particular, case studies naturally maximize realism, but seldom satisfy generalizability (since they involve a small number of non-randomly selected situations) or precision (because there is a low level of control over influencing factors). Hence, we leverage the realism of our results and conclusions.

7 RELATED WORK

7.1 Identification of Key Developers

A number of previous studies investigated different approaches to determine key developers. Crowston et al. [15] investigated three specific approaches, namely

1. the list of contributors officially named as developers,
2. the most frequent contributors, and
3. a social network analysis of the developers' interaction patterns.

By applying these approaches to the bug fixing interactions of 116 SourceForge projects, the authors concluded that each approach identified different individuals as key developers. As in our paper, the results suggest that the group of key developers in FLOSS projects corresponds to only a small fraction of the total number of contributors. Similarly to our approach, Jergensen et al. [28] determined the set of key developers based on their contributions to the system's core. However, while we determined the technical core using call-graphs, Jergensen and colleagues determined it using logical dependencies (co-changes). They computed the Eigenvector

Centrality of each artifact included in the technical network and calculated the centrality of each commit as the mean of the centrality scores of the files it comprises. From this, they compute an overall code centrality score for each developer, which refers to the sum of the centralities of their commits. Thus, in their approach, a developer can become prominent (key) by either making many commits to files with low or medium centralities, or by making fewer commits with high centrality scores. Our approach is more pragmatic, as we consider that a commit is core when it includes at least one artifact belonging to the technical core (i.e., an artifact with high Eigenvector Centrality score). In this sense, our notion of key developer is slightly different from theirs. We consider that developers become prominent (key) only when they frequently modify the technical core. While this study and Jergensen's rely on Eigenvector Centrality to determine the technical core, different approaches have been suggested in the literature. For instance, MacCormack et al. [30] identify core-periphery structures in software systems based on two metrics, namely Fan-In Visibility (FIV) and Fan-Out Visibility (FOV). FIV corresponds to the number of other components that a specific component transitively depends upon. FOV, in turn, corresponds to the number of other components that transitively depends on a specific component. Core components are those that have both FIV and FOV above 50 % of the maximum value of these metrics. A discussion of other approaches for defining the technical core of software systems can be found in [50]. Zhang et al. [59] defined key developers as those who regularly contributed to the project and participated in the mailing lists. They studied the ArgoUML project and determined the set of key developers using SNA metrics. They calculated the precision and recall of their method using an oracle. The oracle was initially defined based on their definition of key developers and was then refined with the help of ArgoUML's project leader. According to Zhang et al., different metrics had similar performance, and they were able to identify more than 60 % of the key developers. In a later study [60], the same authors extended the original work by adding bipartite networks that link developers to email topics. They achieved the best performance in the identification of key developers when using bipartite networks and degree centrality. In a broader context, Arroyo et al. [43] state that few methods exist to determine *sets* of key players in social networks. They present such methods, discuss them, and propose a new one based on entropy measures. Their approach selects the set of nodes that produces the largest change in connectivity entropy when removed from a graph.

7.2 Characterizing Key Developers from a SNA Perspective

Other studies have focused on investigating the characteristics and behavior of key developers from a social network analysis perspective. Bird et al. [7] found a strong relationship between the number of messages sent and the number of different people who respond to them. The authors also found that the level of activity in the source code is a strong indicator of the social status of a developer. Indeed, in our case study we found that all key developers were top contributors. In addition, we found

that three of four key developers were often active in the mailing list, which might contribute to their social status within the community.

Hung et al. [27] built a social network based on the paths (directories) included in the change-set of commits (Figure 12). After that, they calculated the Closeness Centrality of each node and determined how core they were. They also proposed to classify developers according to the model proposed by Ye and Kishida [57], which includes the following FLOSS roles: project leader, core member, active developer, peripheral developer, bug fixer, bug reporter, reader, and passive user.

$$
\begin{aligned}
G_d &= \{V_d, E_d\} \\
V_d &= \{d | d \text{ is a developer}\} \\
E_d &= \{(d_1, d_2) | \exists \text{ path } p \text{ s.t } d_1 \in D_p \text{ and } d_2 \in D_p\} \\
D_p &= \{d | \text{developer } d \text{ has modified path } p\}
\end{aligned}
$$

Figure 12: Building a social network based on commit history (adapted from [27])

In a broader context, Oezbek et al. [37] investigated the patterns of interaction among the core and peripheral sets of developers to check the validity of the "onion model" [35]. After building social networks based on mailing lists data from 11 FLOSS projects of different domains, the authors observed that the core holds a disproportionately large share of communication with the periphery. They also observe that members of the core not only show a particular intense participation, but also appears to have a qualitatively different role as well. However, such hypothesis remains to be investigated. The authors also concluded that the transition of individual mailing list participants towards higher participation is qualitatively discontinuous. Finally, in the domain of online communities, researchers have studied ways to foster content contributions [54].

7.3 Characterizing Key Developers According to Their Contribution

Terceiro et al. [55] investigated the relationship between code structural complexity and the participation level of developers (dichotomized as core and peripheral). By relying on previous studies of Robles et al. [46, 47], the authors split the entire studied period in 20 periods of equal duration, and for each period, they considered the 20 % top committers to be the core team. They found out that core developers make changes to the source code without introducing as much structural complexity as the peripheral developers. Moreover, core developers also remove more structural complexity than peripheral developers do. Geldenhuys [24] studied the claim that 20 % of the participants in FLOSS projects often contribute 80 % of the work. In their investigation, they considered 9 FLOSS projects of different domains and sizes, and found that 80 % of all commits were done by 3.1 % to 8.9 % of the developers. The actual number of developers that did 80 % of all commits also ranged from 1 to 327. These results pose doubts on the hypothesis stated by

Mockus et al.: *"... if this core group uses only informal, ad hoc means of coordinating their work, it will be no larger than 10–15 people"*. In contrast, our study corroborates Mockus' hypothesis, since the four key developers we identified performed 81 % of all commits. Furthermore, differently from the Geldenhuys's study, we determine key developers based on how frequently developers contributed to the core.

7.4 Other Studies

De Souza et al. [18] investigated the ways in which development processes are somehow inscribed into software artifacts. The authors hypothesized that when developers shift from the periphery to the core of the code authorship social network, a distinct phenomenon occurs: developers initially contribute with code that performs some functionality by calling others' code and, as these developers become more important, their code start to be called by other developers. In particular, De Souza and colleagues showed a periphery to core shift within the MegaMek project, and a core to periphery shift (opposite effect) within the Apache Ant project. Finally, regarding the foundations of our study, researchers and practitioners have long recognized the relationship between the architecture of a software system and the coordination effort required to evolve such a system. For instance, it has been shown that the performance of software developers is related to how well they align their coordination efforts with the existing technical dependencies in the software architecture, both at the team level [53] and at the individual level [10]. Indeed, misalignment between these aspects is seen as a possible explanation for breakdowns in software development projects [5].

8 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a descriptive case study involving a release of the Apache Ant project. We aimed to identify and characterize *key developers*, i.e., those developers that evolve the technical core of the system. Given the dynamics of FLOSS development and the associated high turnover of developers, it becomes difficult to distinguish the set of key developers from the outside. Consequently, making this information explicit in the project's website and keeping it updated can be difficult. For instance, Zhang et al. [60] reported the extreme case of ArgoUML, which had contributions from 1,100 developers during a period of 8 years. At the same time, recognizing key developers and their characteristics (skills, social profile, etc.) can be useful for a number of reasons, such as for recruiting specialists, assigning tutors to newcomers, and determining the longevity likelihood of the project. Characterizing key developers is particularly useful to the maintenance of healthy FLOSS communities. For instance, determining the expertise of key developers can be useful to reason about what would happen if those developers left the project. The example provided by Ye et al. [57] is emblematic: the

development of the now popular GIMP image processing tool froze for about 20 months because their two creators graduated, started full-time employment, and no longer had time to contribute to the project. In a long term perspective, we expect that characterizing key developers will help researchers understand the requirements and the process that developers often undergo in order to become key ones.

We addressed three research questions in our case study: *How to identify key developers?* (RQ 1), *What is the participation of key developers in terms of communication and coordination within the project?* (RQ 2), and *What are the characteristics of the contributions of the key developers?* (RQ 3). We tackled RQ 1 by conceiving, implementing, and applying a robust method for identifying key developers (Section 4.3). The method accounts for the evolutive nature of software and extracts the technical core of Java systems directly from source code. Applying the method resulted in the identification of four key developers. After that, we addressed RQ 2 by investigating two kinds of networks: a *communication network* built from mailing list data and a *coordination requirements network* [13, 12, 11] built from co-changes in the source code files. We built the communication networks according to three different strategies and we found in all cases that three of the key developers socialized more than other developers, acted as bridges connecting other developers, and were close to them in the social structure. Furthermore, we found that these same developers also had a high socio-technical congruence [13, 12, 11]. At the same time, we noticed that a key developer did not communicate often in the mailing list and had a low socio-technical congruence. This was somewhat counter-intuitive, as we concluded that this developer was able to evolve the technical core without talking to others in the mailing list. We then answered RQ 3 by analyzing how frequently key developers touched the technical core. We discovered that their core and non-core commits were often very interleaved, which provided evidence that they brought their skills and expertise from previous experiences. We also analyzed the volume and dispersion of key developers' contributions. In terms of their contribution volume, we found evidence that the set of key developers was identical to the set of top committers. Besides that, we noticed that key developers not only contributed to the core, but also to peripheral areas of the system.

As future work, we believe that applying our analysis to different FLOSS projects would help to verify whether key developers characteristics are similar to those we reported. Furthermore, extending our analysis framework with statistical information regarding the number of messages sent and replied in the mailing list could provide additional insights. Considering other sources of information (such as issue tracking systems) would likely provide additional insight into key developers' characteristics as well. Employing qualitative research would also enrich the outcomes of this study. The final step would be to conduct large scale examinations involving a considerable number of projects. This way, it would be possible to better determine the different profiles of key developers. However, a series of challenges need to be tackled beforehand in order to make this study fea-

sible, such as mapping email addresses (aliases) to individuals in an automated way.

Acknowledgments

We thank Steve Abrams for his insightful contribution to the design of this research. Gustavo Oliva received an individual grant from the CHOReOS EC FP7 project. Marco Gerosa and Claudia Werner received individual grants from CNPq. Cleidson de Souza was supported by FAPESPA through "Edital Universal No. 167 003/2008". Francisco Santana received an individual grant from CAPES. A prior version of this study was published in the proceedings of the 18th International Conference on Collaboration and Technology (CRIWG 2012). We are thankful for all the feedback obtained during the submission and presentation of that paper.

REFERENCES

[1] Abraham, A.—Hassanien, A.-E.—Snášel, V.: Computational Social Network Analysis: Trends, Tools and Research Advances, 1st ed., Springer Publishing Company, Incorporated, 2009.

[2] Arnold, R. S.: Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[3] Balieiro, M. A.—de Júnior, S. F. S.—de Souza, C. R. B.: Facilitating Social Network Studies of FLOSS Using the OSSNetwork Environment. In: Russo, B., Damiani, E., Hissam, S. A., Lundell, B., Succi, G. (Eds.): Open Source Development, Communities and Quality, IFIP 20th World Computer Congress, Working Group 2.3 on Open Source Software (OSS 2008), September 7–10, 2008, Milano, Italy, IFIP, Springer, 2008, Vol. 275, pp. 343–350.

[4] Ball, T.—Adam, J.-M. K.—Harvey, A. P.—Siy, P.: If Your Version Control System Could Talk... ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering, March 1997.

[5] Bass, M.—Mikulovic, V.—Bass, L.—Herbsleb, J.—Cataldo, M.: Architectural Misalignment: An Experience Report. Proceedings of the Sixth Working IEEE/IFIP Conference on Software Architecture (WICSA '07), Washington, DC, USA, IEEE Computer Society, 2007, pp. 17.

[6] Beck, K.—Andres, C.: Extreme Programming Explained: Embrace Change. Second ed., Addison-Wesley Professional, 2004.

[7] Bird, C.—Gourley, A.—Devanbu, P.—Gertz, M.—Swaminathan, A.: Mining Email Social Networks. Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06), New York, NY, USA, ACM, 2006, pp. 137–143.

[8] Booch, G.—Maksimchuk, R. A.—Engel, M. W.—Young, B. J.—Conallen, J.—Houston, K. A.: Object-Oriented Analysis and Design with Applications. Third ed., Addison-Wesley Professional, 2007.

[9] Burton, R. M.—Obel, B.: Strategic Organizational Diagnosis and Design: The Dynamics of Fit (Information and Organization Design Series). Third ed., Springer, 2003.

[10] Cataldo, M.: Dependencies in Geographically Distributed Software Development: Overcoming the Limits of Modularity. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2007, AAI3292617.

[11] Cataldo, M.—Herbsleb, J.: Coordination Breakdowns and Their Impact on Development Productivity and Software Failures. Software Engineering, IEEE Transactions, Vol. 39, 2013, No. 3, pp. 343–360.

[12] Cataldo, M.—Herbsleb, J. D.—Carley, K. M.: Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity. Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '08), New York, NY, USA, ACM, 2008, pp. 2–11.

[13] Cataldo, M.—Wagstrom, P.—Herbsleb, J. D.—Carley, K. M.: Identification of Coordination Requirements: Implications for the Design of Collaboration and Awareness Tools. In: Hinds, P. J., Martin, D. (Eds.): CSCW, ACM, 2006, pp. 353–362.

[14] Conway, M.: How Do Committees Invent? Datamation, Vol. 14, 1968, No. 4, pp. 28–31.

[15] Crowston, K.—Wei, K.—Li, Q.—Howison, J.: Core and Periphery in Free/Libre and Open Source Software Team Communications. Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS '06), Washington, DC, USA, 2006, IEEE Computer Society, 2006, Vol. 06, p. 118.1.

[16] Dagenais, B.—Ossher, H.—Bellamy, R. K. E.—Robillard, M. P.—de Vries, J. P.: Moving into a New Software Project Landscape. Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10), New York, NY, USA, ACM, 2010, Vol. 1, pp. 275–284.

[17] D'Ambros, M.—Lanza, M.—Lungu, M.: Visualizing Co-Change Information with the Evolution Radar. IEEE Trans. Software Eng., Vol. 35, 2009, No. 5, pp. 720–735.

[18] de Souza, C.—Froehlich, J.—Dourish, P.: Seeking the Source: Software Source Code as a Social and Technical Artifact. Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work (GROUP '05), ACM, 2005, pp. 197–206.

[19] de Souza, C. R.—Quirk, S.—Trainer, E.—Redmiles, D. F.: Supporting Collaborative Software Development Through the Visualization of Socio-Technical Dependencies. Proceedings of the 2007 International ACM Conference on Supporting Group Work (GROUP '07), ACM, 2007, pp. 147–156.

[20] de Souza, C. R. B.—Redmiles, D. F.: An Empirical Study of Software Developers' Management of Dependencies and Changes. Proceedings of the 30th International Conference on Software Engineering (ICSE '08), New York, NY, USA, ACM, 2008, pp. 241–250.

[21] Foote, B.—Yoder, J. W.: Big Ball of Mud. In: Harrison, N., Foote, B., Rohnert, H. (Eds.): Pattern Languages of Program Design, Addison-Wesley Professional, 2000, Vol. 4, Ch. 29, pp. 654–692.

[22] Gall, H.—Hajek, K.—Jazayeri, M.: Detection of Logical Coupling Based on Product Release History. Proceedings of the International Conference on Software Maintenance (ICSM '98), Washington, DC, USA, IEEE Computer Society, 1998, pp. 190.

[23] Gall, H.—Jazayeri, M.—Krajewski, J.: CVS Release History Data for Detecting Logical Couplings. Proceedings of the 6th International Workshop on Principles of Software Evolution, Washington, DC, USA, 2003, IEEE Computer Society, pp. 13.

[24] Geldenhuys, J.: Finding the Core Developers. Proceedings of the 2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA '10), Washington, DC, USA, IEEE Computer Society, 2010, pp. 447–450.

[25] Hotelling, H.: Simplified Calculation of Principal Components. Psychometrika, Vol. 1, 1936, pp. 27–35, DOI 10.1007/BF02287921.

[26] Howison, J.—Inoue, K.—Crowston, K.: Social Dynamics of Free and Open Source Team Communications. In: Damiani, E., Fitzgerald, B., Scacchi, W., Scotto, M., Succi, G. (Eds.): Open Source Systems, 2006, IFIP, Springer, 2006, Vol. 203, pp. 319–330.

[27] Huang, S.-K.—Liu, K.-M.: Mining Version Histories to Verify the Learning Process of Legitimate Peripheral Participants. Proceedings of the 2005 International Workshop on Mining Software Repositories (MSR '05), New York, NY, USA, ACM, 2005, pp. 1–5.

[28] Jergensen, C.—Sarma, A.—Wagstrom, P.: The Onion Patch: Migration in Open Source Ecosystems. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11), New York, NY, USA, ACM, 2011, pp. 70–80.

[29] Lehman, M.—Perry, D.—Ramil, J.—Turski, W.—Wernick, P.: Metrics and Laws of Software Evolution – The Nineties View. Proceedings IEEE International Software Metrics Symposium (METRICS '97), Los Alamitos CA, 1997, IEEE Computer Society Press, pp. 20–32.

[30] Maccormack, A.: The Architecture of Complex Systems: Do "Core-Periphery" Structures Dominate? Academy of Management Proceedings 2010, Vol. 1, 2010, pp. 1–6.

[31] March, J. G.—Simon, H. A.: Organizations. 2nd ed., Wiley-Blackwell, 1993.

[32] Martin, R. C.—Martin, M.: Agile Principles, Patterns, and Practices in C#. First ed. Prentice Hall, 2006.

[33] McGrath, J. E.: Dilemmatics: The study of Research Choices and Dilemmas. American Behavioral Scientist, Vol. 25, 1981, No. 2, pp. 179–210.

[34] Mockus, A.—Fielding, R. T.—Herbsleb, J. D.: Two Case Studies of Open Source Software Development: Apache and Mozilla. ACM Trans. Softw. Eng. Methodol., Vol. 11, July 2002, pp. 309–346.

[35] Nakakoji, K.—Yamamoto, Y.—Nishinaka, Y.—Kishida, K.—Ye, Y.: Evolution Patterns of Open-Source Software Systems and Communities. Proceedings of

the International Workshop on Principles of Software Evolution (IWPSE '02), New York, NY, USA, ACM, 2002, pp. 76–85.

[36] Newman, M.: Networks: An Introduction. First ed. Oxford University Press, 2010.

[37] Oezbek, C.—Prechelt, L.—Thiel, F.: The Onion has Cancer: Some Social Network Analysis Visualizations of Open Source Project Communication. Proceedings of the 3rd International Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development (FLOSS '10), New York, NY, USA, ACM, 2010, pp. 5–10.

[38] Oliva, G. A.—Gerosa, M. A.: On the Interplay Between Structural and Logical Dependencies in Open-Source Software. Proceedings of the 2011 25th Brazilian Symposium on Software Engineering (SBES '11), Washington, DC, USA, IEEE Computer Society, 2011, pp. 144–153.

[39] Oliva, G. A.—Gerosa, M. A.: IVAR: A Conceptual Framework for Dependency Management. Proceedings of the IX Workshop on Modern Software Maintenance (WMSWM '12), 2012.

[40] Oliva, G. A.—Santana, F. W.—de Oliveira, K. C. M.—de Souza, C. R. B.—Gerosa, M. A.: Characterizing Key Developers: A Case Study with Apache Ant. Proceedings of the 18th International Conference on Collaboration and Technology (CRIWG '12), Springer-Verlag, Berlin, Heidelberg, 2012, pp. 97–112.

[41] Oliva, G. A.—Santana, F. W.—Gerosa, M. A.—de Souza, C. R.: Towards a Classification of Logical Dependencies Origins: A Case Study. Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution (IWPSEEVOL '11), New York, NY, USA, ACM, 2011, pp. 31–40.

[42] Oliva, G. A.—Steinmacher, I.—Wiese, I.—Gerosa, M. A.: What Can Commit Metadata Tell Us about Design Degradation? Proceedings of the 2013 International Workshop on Principles of Software Evolution (IWPSE 2013), New York, NY, USA, ACM, 2013, pp. 18–27.

[43] Ortiz-Arroyo, D.: Discovering Sets of Key Players in Social Networks. In: Abraham, A., Hassanien, A.-E., Snášel, V. (Eds.): Computational Social Network Analysis, Computer Communications and Networks. Springer London, 2010, pp. 27–47.

[44] Parnas, D. L.: On the Criteria to Be Used in Decomposing Systems into Modules. Commun. ACM, Vol. 15, 1972, pp. 1053–1058.

[45] Plato: Cratylus. Parmenides. Greater Hippias. Lesser Hippias. Loeb Classical Library, 1926. Translated by Harold North Fowler.

[46] Robles, G.—Gonzalez-Barahona, J.: Contributor Turnover in Libre Software Projects. In: Damiani, E. Fitzgerald, B., Scacchi, W., Scotto, M., Succi, G. (Eds.): Open Source Systems, IFIP International Federation for Information Processing, Springer US, 2006, Vol. 203, pp. 273–286.

[47] Robles, G.—Gonzalez-Barahona, J. M.—Herraiz, I.: Evolution of the Core Team of Developers in Libre Software Projects. Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories (MSR '09), Washington, DC, USA, IEEE Computer Society, 2009, pp. 167–170.

[48] Robson, C.: Real World Research, Second ed., John Wiley & Sons, 2002.

[49] Santana, F.—Oliva, G.—de Souza, C. R. B.—Gerosa, M. A.: Xflow: An Extensible Tool for Empirical Analysis of Software Systems Evolution. Proceedings of the VIII Experimental Software Engineering Latin American Workshop (ESELAW '11), 2011.

[50] Santana, F.—Oliva, G. A.—Gerosa, M. A.—Souza, C. R. B. D.: Understanding Complex Software Ecosystems: The Role of Core-Periphery Identification. The Future of Collaborative Software Development, FutureCSD @ CSCW 2012. Downloadable from `http://lapessc.ime.usp.br`, 2012.

[51] Shneiderman, B.: Tree Visualization with Tree-Maps: 2-D Space-Filling Approach. ACM Trans. Graph., Vol. 11, 1992, No. 1, pp. 92–99.

[52] Souza, C. R.—Redmiles, D. F.: On the Roles of Apis in the Coordination of Collaborative Software Development. Comput. Supported Coop. Work, Vol. 18, 2009, No. 5-6, pp. 445–475.

[53] Staudenmayer, N. A.: Managing Multiple Interdependencies in Large Scale Software Development Projects. Ph.D. thesis, Alfred P. Sloan School of Management – Massachusetts Institute of Technology (MIT), 1997, AAI0598584.

[54] Tedjamulia, S. J. J.—Dean, D. L.—Olsen, D. R.—Albrecht, C. C.: Motivating Content Contributions to Online Communities: Toward a More Comprehensive Theory. Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS '05), Washington, DC, USA, IEEE Computer Society, 2005, Vol. 07, pp. 193.2.

[55] Terceiro, A.—Rios, L. R.—Chavez, C.: An Empirical Study on the Structural Complexity Introduced by Core and Peripheral Developers in Free Software Projects. Proceedings of the 2010 Brazilian Symposium on Software Engineering (SBES '10), Washington, DC, USA, IEEE Computer Society, 2010, pp. 21–29.

[56] Wasserman, S.—Faust, K.: Social Network Analysis: Methods and Applications. 1st ed., Cambridge University Press, 1994.

[57] Ye, Y.—Kishida, K.: Toward an Understanding of the Motivation Open Source Software Developers. Proceedings of the 25th International Conference on Software Engineering (ICSE '03), Washington, DC, USA, IEEE Computer Society, 2003, pp. 419–429.

[58] Yin, R. K.: Case Study Research: Design and Methods. Third ed., Sage Publications, 2003.

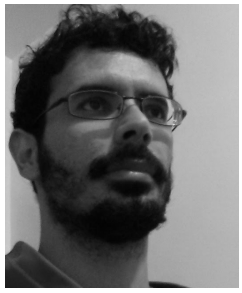
[59] Zhang, W.—Yang, Y.—Wang, Q.: Network Analysis of OSS Evolution: An Empirical Study on ArgoUML Project. Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution (IWPSE-EVOL '11), New York, NY, USA, ACM, 2011, pp. 71–80.

[60] Zhang, W.—Yang, Y.—Wang, Q.: An Empirical Study on Identifying Core Developers Using Network Analysis. Proceedings of the 2nd International Workshop on Evidential Assessment of Software Technologies (EAST '12), New York, NY, USA, ACM, 2012, pp. 43–48.

[61] Zimmermann, T.—Diehl, S.—Zeller, A.: How History Justifies System Architecture (or Not). Proceedings of Sixth International Workshop on Principles of Software Evolution, 2003, pp. 73–83.

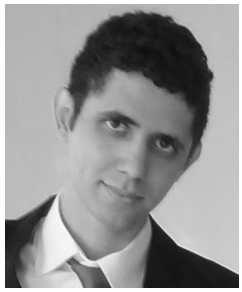
[62] Zimmermann, T.—Weissgerber, P.—Diehl, S.—Zeller, A.: Mining Version Histories to Guide Software Changes. IEEE Trans. Softw. Eng., Vol. 31, 2005, pp. 429–445.

Gustavo Ansaldi Oliva received his master's degree in computer science from the University of São Paulo (USP) in Brazil under the supervision of Marco Gerosa. Currently, he is a Ph.D. student at the same university still under the supervision of Marco. His expertise is on software engineering and he has done research in the fields of mining software repositories (MSR), software maintenance and evolution, open-source software development, and service-oriented computing. In particular, he received grants from HP Brazil and the European Commission for developing novel change impact analysis mechanisms for service compositions. In industry, he worked as a software developer at IBM Brazil for more than 3 years. In his Ph.D., he is researching better ways to identify logical (evolutionary) dependencies from version control systems.



José Teodoro da Silva received a bachelor's degree in computer science from the Federal Technological University of Paraná (UTFPR). Since 2012, he is a master's degree student at the University of São Paulo (USP) under the supervision of Marco Gerosa. He currently works with data mining, fault prediction, and social networks.

Marco Aurélio Gerosa is Associate Professor in the Computer Science Department at the University of São Paulo (USP), Brazil. His research lies in the intersection between software engineering and social computing, focusing on the fields of empirical software engineering, mining software repositories, software evolution, and social dimensions of software development. He has received the productivity grant from the Brazilian Council for Scientific and Technological Development. In addition to his research, he also coordinates award-winning open source projects.



Francisco Werther Silva de Santana Jr. received a bachelor's degree in computer science from the Federal University of Pará (UFPA). He is currently a master's student at the Graduate School of Engineering of the Federal University of Rio de Janeiro (COPPE – UFRJ) under the supervision of Cláudia Werner. His research interests are computer-supported collaborative work (CSCW), human-computer interaction, and software visualization.

Cláudia Maria Lima Werner is Associate Professor in the Computer Science Department at COPPE – UFRJ, Brazil. Her areas of interest include software reuse, software development environments, and component-based development. She received her Ph.D. in computer science in 1992 from COPPE – UFRJ. She has over 100 technical papers published in international conferences and journals. She is a member of the program committee for ICSR (International Conference on Software Reuse), SPLC (International Software Product Line Conference), and several Latin-American conferences. She is a member of SBC (the Brazilian Society of Computer Science), and is currently the leader of the Software Engineering group at COPPE/UFRJ. She is a co-editor of the Journal of Software Engineering Research and Development (JSERD), Springer.

Cleidson Ronald Botelho de Souza is Senior Researcher at the Vale Institute of Technology and a faculty at the Federal University of Pará (UFPA). His research interests are in the intersection between software engineering and computer-supported cooperative work. He received his Ph.D. in information and computer science in 2005 from University of California, Irvine. From 1998 to 2010 he worked as an Associate Professor at the Federal University of Pará, Brazil, while from 2010 till 2012 he was a Research Scientist at IBM Research – Brazil. He is a member of the program committee of several conferences both in software engineering and in collaborative systems. He is an affiliated member of the Brazilian Academy of Science since 2014.

Kleverton Carlos Macedo de Oliveira received a bachelor's degree in computer science from the Federal University of Pará (UFPA). Currently, he is a programmer at the IClass Consultoria company in Brazil. He has experience in the area of software development methods and software visualization.