University of Passau

Department of Informatics and Mathematics

Bachelor Thesis

The Influence of Developer Roles on Contributions to Open-Source Software Projects

Author:

Christian Hechtl

April 07, 2018

Advisors:

Prof. Dr.-Ing. Sven Apel

Chair of Software Engineering I

Claus Hunsen

Chair of Software Engineering I

Thomas Bock

Chair of Software Engineering I

Hechtl, Christian:
The Influence of Developer Roles on Contributions to Open-Source Software Projects
Bachelor Thesis, University of Passau, 2018.

Abstract

It is not generally known, how the fact that a developer is a core or a peripheral developer, influences the outcome of the developer's contribution to Open-Source Software (OSS) projects. To investigate this we replicate a empirical study by Bosu et al. [BC14], which finds that core developers have to wait shorter for a first feedback on their contribution and the completion of the review process, as well. Moreover, they find that core developers have a higher code-acceptance rate. To replicate these results, we extract data from three OSS projects and build social developer networks based on the mail interactions between the developers. The projects we analyze are: Jailhouse, BusyBox and the Apache HTTP Server project. We then classify the developers in core and peripheral groups and analyze their code contributions.

We are not able to confirm the results of the original study. We find that core developers do not have a shorter first-feedback and review interval, than peripheral developers. Moreover, we find that the code-acceptance rate, indeed seems to be higher for core developers, but since the results are not statistically significant we can not confirm it overall. Furthermore, we find that the number of patch revisions a developer needs until the patch is accepted is not lower for core developers than for peripheral developers.

Contents

List of Figures

List of Tables

1 Introduction

One of the main reasons that Open-Source Software (OSS) projects thrive and gain more and more importance in the world of software development, are voluntary contributions to them. Without such contributions, OSS development would, most likely, not be possible. One of the main motivations for a developer, to decide to contribute to an OSS project, is to gain a good reputation among the other developers [LW05]. Bosu et al. [BC14] state that it is not known if and how a higher reputation, of a developer, influences the outcome of the developer's code-review requests.

Our motivation, to study this topic, is that knowing what influences ones contributions to an OSS project positively or negatively can help developers. It can be kind of a guideline for new developers, which tells them how they can participate successfully. Moreover it can help developers that have been working on projects for a long time, to increase their productivity and success.

For these reasons, the goal of this thesis is to find out, in what way developer roles influence contributions to OSS projects. To do so, we replicate an empirical study by Bosu et al. [BC14] on this topic with a slightly different approach and different case studies. The original study investigates several factors of OSS contributions, such as contribution success, review interval, feedback time, and the number of patch revisions needed for a successful contribution. They investigate these factors separate for core and peripheral developers. Their main findings are that the contributions of core developers are usually more successful, get accepted faster, and that core developers have to wait a shorter amount of time for a first feedback on their contributions. In this thesis we take a look at the same factors regarding OSS contributions as the original study, but use different techniques to extract the necessary data, classify the developers into core and peripheral, and calculate the results.

We use three different case studies for this thesis, which all use a mailing-list as contribution tool. These case studies are: Jailhouse, BusyBox, and the Apache HTTP Server Project (httpd). The base data, we need, is extracted from version control data and mailing-list archives. The main things, we use, are the lists

of commits and mails for the three projects. Using this data we build developer networks with mail communications as connections between developers.

Once the networks are built, we use them to classify the developers into core and peripheral groups. To do so, we use different network measures. Once this classification is done, we can analyze the code contributions separated for core and peripheral developers.

Another important thing, we need, is a mapping of patches on the projects' mailing-lists to their respective commits within the repository. We need this to be able to calculate the percentage of successful contributions as well as the time from submission to acceptance.

We are not able to confirm the original study. Our results lead us to reject all of the four hypotheses. The original study finds that core developers have a shorter first-feedback interval than peripheral developers. We find that this is not the case. In fact, the results for JAILHOUSE even indicate that the complete opposite is the case. The next thing the original study finds is that core developers have a shorter review interval than peripheral ones. We can not confirm these results either. Moreover, Bosu et al. find that core developers have a higher code-acceptance rate than peripheral developers. In our thesis we can neither accept nor reject this hypothesis, since more than half of our results militate in favor of this. But since our results are not distinct, we find them to be inconclusive. The last hypothesis centers on the number of patch revisions a developer needs to send until the patch is accepted. The original study finds their results, for this topic, to be inconclusive. We, on the other hand, have to refuse the hypothesis based on our data.

The rest of this thesis is structured as follows: In Chapter 2, we give an overview of topics concerning the topic of this thesis. We present details about the OSS development and social-network analysis. Furthermore, we present related work. In Chapter 3, we present the approach and details of the original study. Subsequently we explain the goal and hypotheses of the thesis and explain the detailed approach we use to answer the hypotheses. Chapter 4 is all about the results of our analyses. We present the outcomes and discuss their meaning. In Chapter 5, we give an overview of threats that may affect the validity of the outcomes of this thesis. Finally, in Chapter 6, we summarize our thesis and give prospects on future work.

2 Background

In this chapter we give background information about the topics that are relevant for the topic of the thesis. These information include OSS development and social-network analysis. Moreover, we give an overview of related work.

2.1 OSS Development

OSS development is a very important type of software development. It gives people the chance to use high quality software for free and even contribute to projects that they are passionate about. These facts might lead to more motivated collaborators, because a lot of them are working on the project for fun in their spare time.

To understand the goal of this thesis, we first have to explain how OSS development works and explain the different developer roles.

2.1.1 The Patch-Submit-Review Process

Since there are usually a lot of people contributing to an OSS project, there has to be some kind of mechanism which regulates the code that makes it into the project. In closed and mostly commercial projects, the quality of the code is usually assured by code inspections or walkthroughs. This basically means that the code gets checked by or presented to co-workers or supervisors. For OSS, Asundi et al. [AJ07] describe the Patch-Review-Submit process as equivalent to the review practices in closed projects.

Collaborators of OSS projects are usually scattered around the globe, which makes a personal meeting nearly impossible. So, in contrast to closed, commercial projects, there has to be a tool for the review of new code without a direct personal interaction. There are two main types of such contribution tools: *patch-based* and *pull-request-based* tools [ZZM16]. Since we only work with projects that use a mailing-list, which counts as a patch-based tool, we only explain this type of contribution tool.

To contribute to such a project, one has to first get a copy of the code-base. This usually works by cloning the repository of the project. Subsequently, the code

changes are made or new code is inserted into the private copy of the project. These changes are called a *patch*. Following its creation, this patch has to be published, for it to be reviewed. This happens by submitting it to the projects mailing-list. Now the review begins. Other developers comment on the patch or request changes by replying to the mail. Subsequently, the creator of the patch has to adapt it until all reviewers are satisfied. Such a resubmitted patch is called a *patch revision*. The collection of all patch revisions is called the set of patch revisions. Once the patch has been approved, it can be integrated into the main project by a developer with the necessary rights.[AJ07] The whole process of a patch-submit-review process via a mailing-list is shown in Figure 2.1.

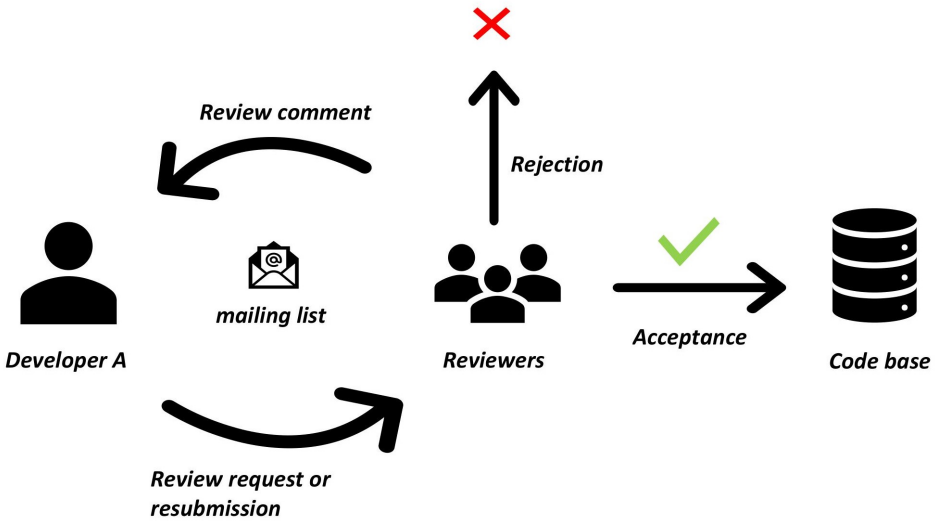


Figure 2.1: Patch-review-submit process. Developer A originates a patch and submits it via the mailing-list to the reviewers. They either request changes, accept the patch into the code base or reject it completely.

2.1.2 Developer Roles

Since we are investigating the influence of developer roles on the outcomes of OSS contributions, one has to know what these roles are. Nakakoji et al. [NYN+02] define eight different roles within the community of OSS projects. These roles range from simple users of the software, through developers to the project leader. Since this thesis is concerned with contributions to such projects, we only look at the active developer groups.

These actively developing groups are often categorized into two groups. A small group of people that are responsible for the majority of the workload and a significantly larger group responsible for few and irregular contributions [CWLH06, MFH02]. The developers within the first group are usually called *core developers* and the members of the latter group are simultaneously called *peripheral developers* [CWLH06].

Core developers are usually people that are involved in the project for a long time and have a deep knowledge of the software. Asundi et al. [AJ07] describe these

developers as gate-keepers of the project, since they are often strongly involved in the decision whether a patch makes it into the code base. Sometimes they even have a distinct status within the project, like the "maintainer" status in the LINUX KERNEL project.

Peripheral developers, on the other hand, are responsible for a significantly smaller part of the workload within the project, even though the peripheral group is clearly larger than the core group. They are usually only involved with small changes or bug fixes in the software [CWLH06].

2.2 Social-Network Analysis

Networks are another important construct for this thesis. A network is a theoretical structure, that represents connections between people or things. The analysis of such constructs is applied in many different fields, like electrical engineering, project planning, biology, and many more [BE05]. Social networks usually describe such networks on the basis of interactions, of some kind, between people. In computer science, these people are usually developers and the connection among them is mostly extracted from their relationships, communications or common activities within a software project. These networks are called *developer networks* [LLFH09], which we describe in Section 2.2.1.

Such networks are usually represented by graphs. The formal notation is $G = (V, E)$ with V being the set of vertices and E the set of edges. Edges are used to link vertices and therefore the set of edges is described as $E \subseteq V \times V$. Such graphs can either be directed or undirected. In directed graphs, the edges between two nodes have one vertex as start and one as target. In formal terms, this means $\langle u, x\rangle \neq \langle v, x\rangle$ with the vertices $u, v \in V$ and $x \in E$ the edges between them. In undirected graphs, edges do not have a direction, which means that one edge is enough to describe the edges from u to v and reversed. Graphs can also be simplified. This means that all edges, that exist between two vertices and have the same direction, are unified into one single edge, which makes a graph clearer in terms of clarity, since it decreases the number of edges. [BE05]

2.2.1 Developer Networks

Developer networks are, as described in Section 2.2, constructs that describe communications, relationships or mutual activities between developers of software projects. These networks can be the basis for studies on the social structures in such projects. Joblin [Job17] identifies two different types of developer networks: *coordination networks* and *communication networks*.

Coordination networks are mainly constructed from data that is extracted from the projects' version-control system (VCS). One of the most important data sources are the commits of the project, since the most used coordination network is a co-change network. This means that the connections between developers represent the fact that both worked on the same file in the project. [Job17]

Communication networks, on the other hand, are constructed from communication data between the developers. The data is usually extracted from the main communication channel of the developers. For OSS projects, the main communication

channel is often a mailing-list. Since these mailing-lists are, in the most cases, publicly accessible, they are often used in various studies of social structures in OSS projects. We use such networks in this thesis.

2.2.2 Centrality Metrics

Once the developer network is built, the next task is to identify core and peripheral developers, as described in Section 2.1.2. One possibility for this is to determine the degree of centrality for every developer within the network. To do so, one can use different centrality metrics. Since there are a lot of such metrics, we only describe the two we use in this thesis: degree centrality and eigenvector centrality.

The *degree centrality* has to be calculated in different ways for directed and undirected networks. In the case of undirected networks, the degree of a vertex is simply the number of edges that are connected with it. This is called the *total degree* of the vertex. When working with a directed network,the total degree is split into the *in-degree* and the *out-degree*, but the overall total degree stays the same. The in-degree is the number of edges that start at the vertex and out-degree the number of edges that end at the vertex. The notation of the degree for a vertex v is $deg(v)$ and the higher this degree is, the higher is the centrality of the vertex. To illustrate the degree centrality we show a small example network in Figure 2.2. The degree values are 1 for the outer, white vertices and 4 for the inner, colored ones. Degree centrality is highly dependent on the neighbors of the vertex and therefore is a local centrality. [JAHM17, BE05]

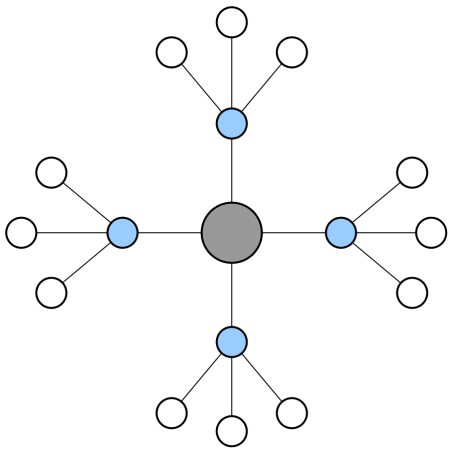


Figure 2.2: Small network with which the degree and eigenvector centrality are visualized. The vertices with the highest degrees are the colored ones because all of them are connected to 4 edges. As for eigenvector centrality, the gray vertex in the middle has the highest value, because all of its neighbors have a degree of 4 but these neighbors are only connected to 3 other vertices with degree 4 and one with degree 1. This figure is taken from [Job17] Figure 2.4.

While degree centrality describes the centrality of a vertex based on its local neighborhood, *eigenvector centrality* describes the centrality of the vertex within this neighborhood. This means that the centrality value of the vertex is determined by the centrality of the vertices that are directly connected to it. The higher their value

is, the higher is the value of the vertex. The vertex with the highest eigenvector centrality in Figure 2.2 is the gray one in the middle, because it is connected to 4 vertices with a degree of 4. These vertices, however, are only connected to three others with a degree of 1 and one with a degree of 4, which makes the gray one the most important in the neighborhood. The eigenvector centrality value x_i for each vertex i is calculated with the following formula:

$$x_i = \frac{1}{\lambda} \sum_{j \in N(i)} x_j \tag{2.1}$$

$N(i)$ is here the collection of all neighbors of i and λ is a proportionality constant [JAHM17, BE05].

2.2.3 Hierarchy

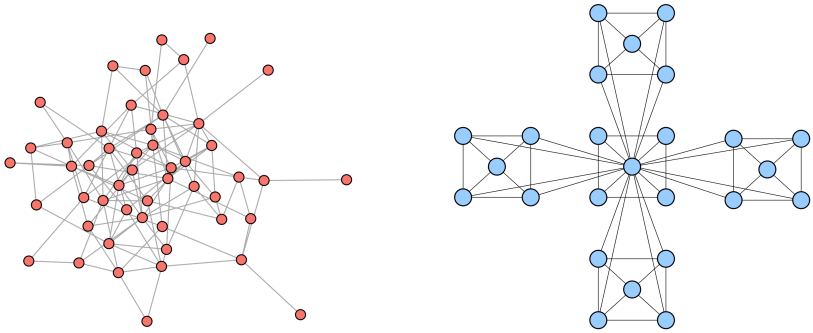
Beyond the centrality metrics we describe in the section above, there are also other, more complex metrics that can be used to determine the core-periphery structure. One of these metrics is the *hierarchy*. Joblin et al. [JAHM17] describe hierarchy as a concept that describes how local groups, within a network, are organized relative to each other. It is dependent on two other metrics: the *clustering coefficient*, which we describe in this section, and the node degree, which we have explained in Section 2.2.2. [RB03]

The clustering coefficient is a metric that describes how strong a vertex is fixed within a *cluster* of vertices. Such a cluster is a set of vertices that are strongly connected among each other. For example, in Figure 2.3 on the right side, the different conglomerates of vertices are clusters because they are all connected to each other. The clustering coefficient gives a quantitative measure of the likelihood that the neighbors of a vertex are connected to each other as well [JAM17]. The clustering coefficient c_i for a vertex i is defined as

$$c_i = \frac{2n_i}{k_i(k_i - 1)}, \tag{2.2}$$

with k_i being the number of edges that are connected to i and n_i being the number of edges between these neighbors. [BLM+06]

Once the values for the clustering coefficient and the vertex degree are determined, we can take a look at the hierarchy. As mentioned before, this metric is described as a interaction of the two values. In Figure 2.3, two networks are shown. The left one is a random network without any hierarchy. The right one is hierarchically organized. The higher the degree and the lower the clustering coefficient are, the higher is the vertex in the hierarchy. In Figure 2.3 the top vertex in this hierarchy is the one in the middle, because it is connected to many other vertices not only in its small cluster but among all the clusters. [Job17]



Random Network Hierarchical Network

Figure 2.3: A random, unorganized network on the left and a hierarchical network on the right. The hierarchical network is organized in clusters, which are connected to each other and thus all together build a larger cluster. The top of the hierarchy is represented by the vertex in the middle because it exhibits a high degree and a low clustering coefficient. The figure is taken from [Job17] Figure 2.9.

2.2.4 Core-Periphery Detection

In the end, the three metrics described in the two sections above, only form the basis for the core-periphery detection of social networks. Through these metrics, the developers get assigned values. With the help of these, the classification can be done.

One of the most used approaches to do so, is a classification due to a threshold at 80%. Originally, this method is designed for the classification of developers with the help of their commit count. The threshold for the core detection is here at the 80% percentile and developers with commit counts above this threshold are considered to be in the core group [CWLH06]. We use a similar approach, although we do not use the commit count but the values from the network metrics above. Furthermore, Crowston et al. [CWLH06] detect that 20% of the developers do 80% of the work.

2.3 Related Work

There is a broad spectrum of studies around the contributions to OSS projects and these projects themselves. In this section we present a few of them that mainly deal with the success of such contributions. The study that is most related to this thesis is the study by Bosu et al. [BC14]. They investigate the influence of developer reputation on the outcomes of code contributions to OSS projects. Since we replicate the study in this thesis, we present it in detail in Section 3.1.

Weißgerber et al. [WND08] investigate the influence of the patch size on the outcome of the contributions, regarding the success of it and the time it needs to be accepted. They find that smaller patches indeed have a higher rate of success in the projects that they investigate. As for the acceptance time, they conclude that the patch size does not seem to have a notable influence. Another study that deals with this research field is a study by Jiang et al. [JAG13]. They investigate, how contributions to the Linux kernel can be influenced. They find that the success of contributions is influenced by many factors, like developer experience or the maturity of the patch.

As for the time it needs to get accepted, they, again, identify several different factors. Some of these factors are developer experience and the choice of reviewers. The last study we want to present is a study on the Apache HTTP Server project by Rigby et al. [RG06]. They find that the most of the commits are done by the core group, which we describe in Section 2.1.2. Furthermore they find that the acceptance time of patches for this project is rather short and therefore conclude that the patches are, very likely, very small.

Another related topic is the classification of developers. Joblin et al. [JAHM17] compare count-based and network based metrics that can be used for the classification of developers into core and peripheral groups. Count-based metrics are, for example, the commit count and the mail count of a developer, while network-based metrics are more complex, like the three metrics we have described in the Sections 2.2.2 and 2.2.3. Joblin et al. find that the classifications based on network-based metrics, agree more with the perception of developers, but that they are not in fact better than count-based metrics.

3 Approach and Hypotheses

In this chapter, we present our research approach and state the concrete hypotheses for our study. Since this is a replication study, we have to present the replicated study by Bosu et al. [BC14] first. From now on, we call their paper *original study*. Subsequent to the presentation of their research questions, approach, and result, we state our hypotheses for this thesis. The last part of this chapter is about our detailed approach to investigate the hypotheses, including data extraction and processing.

3.1 Original Study

According to Lakhani et al. [LW05], one of the main goals of OSS developers is to gain a good reputation. Bosu et al. [BC14] state that it is not clear in what way a good reputation of developers affects their contributions to the project. To investigate this question, they define the objective of the study to be identifying how the reputation affects the outcome of a developer's review requests in an OSS project.

3.1.1 Hypotheses

The first thing we have to take a look at are the hypotheses Bosu et al. [BC14] formulate. The high-level hypothesis is stated as objective of the study in the section above. Since reputation is something that is not easy to measure, they use the core-periphery structure of an OSS project as proxy for the reputation with the assumption that core developers have a higher reputation. For this reason they pose the following four hypotheses.

The first thing they take a look at is the *first-feedback interval* of a review request. They define it as the amount of time that elapses from the submission of the request until the first answer by a reviewer. The hypothesis they pose here is that the first-feedback interval is shorter for core developers than for peripheral developers. They base this on reasons like that core developers should know what reviewers are most appropriate for the submission and that their core position may lead reviewers to a

faster and less detailed review. Another reason might be that reviewers prioritize the review of a core developer's submission due to their prior good relationship.

The next object of investigation of the original study is the review interval. It is defined as the time span from the submission of the request until the end of the review process [RGS08]. The hypothesis they provide surrounding this interval is that core developers have a shorter review interval than peripheral developers. They state that this should be the case for similar reasons as the ones for the first-feedback interval.

The third hypothesis centers on the code-acceptance rate. This is the ratio of eventually accepted patches among all of the submitted ones by a developer. Core developers should be more familiar with the code base and the coding guidelines of a project. For this reason, the third hypothesis is that core developers have a higher code acceptance rate than peripheral developers.

Last but not least, the fourth object for investigating the high-level hypothesis is the number of patch revisions per review request. When a reviewer identifies some kind of problem with a patch, the originator has to rework it and submit the new patch revision again. The number of such resubmissions until the end of the review process is the number of patch revisions. Bosu et al. call a patch a patchset and therefore use number of patchsets instead of number of patch revisions, like we do. To avoid confusion, we use patch revisions in our thesis. For the same reason as for the code acceptance rate, core developers should be able to produce higher-quality patches and, thus, need less reworks. For that reason, they hypothesize that the number of patch revisions is lower for core developers than for peripheral developers.

3.1.2 Data Extraction

To investigate the hypotheses from Section 3.1.1, Bosu et al. [BC14] use eight different OSS projects. These are Chromium OS, ITK/VTK, LibreOffice, Omap-Zoom, OpenStack, OVirt, Qt Project, and Typo3. All of these projects use Gerrit[1], which is a code-review tool that captures all important data surrounding a submission to the project. So to get these information, they develop a miner similar to Mukadam et al. [MBR13], which is able to extract core-review data from Gerrit. With this, they extract all completed code review requests from the eight projects. Subsequently, they removed all automated answers to a request as they only want to work with comments by real reviewers.

3.1.3 Network Construction

The next step toward the study results is the construction of social networks based on the extracted data. They use this data to calculate the number of interactions between two developers regarding the review of patches. Having calculated the number of interactions, they build undirected networks with developers as vertices. The weight of the edges between them is the calculated number of interactions.

[1] https://www.gerritcodereview.com/

3.1.4 Core Detection

Once the network is built, the developers can be classified into core and peripheral groups. The approach that Bosu et al. [BC14] develop for that reason is called *Core Identification using K-means (CIK)*. They use the centrality of a developer as a proxy for the core classification. For that reason, centrality metrics, like the ones we describe in Section 2.2.2, form the basis of the approach. Besides the degree and eigenvector centrality, they use *betweenness*, *closeness*, and *PageRank* centrality. Moreover they use *eccentricity*. They combine all of these metrics by using the K-means clustering algorithm [Mac67].

Betweenness centrality is a metric that describes how many shortest paths in the graph, traverse the vertex. The higher this value is, the higher is the importance of the vertex in the network. Closeness centrality is a metric that describes how close a vertex is to the other vertices in the network and the higher this number is, the faster can a developer reach the entire network, which makes the developer more important. PageRank is a modification of the eigenvector centrality, which we have described in Section 2.2.2. This metric is a indicator of how much influence the developer has over the network. The last of the metrics is eccentricity, which describes how far away the farthest vertex in the network is. A low eccentricity is an indication of a developers centrality, as a central developer should have ties to most of the other developers. [BC14]

Table 3.1 contains an excerpt of the data they extract and process using the CIK approach. The groups of core developers are here in all cases significantly smaller than the peripheral groups but, in most cases, core developers do a lot more work.

Project	# of dev	# of core dev	# of peripheral dev	commits by the core	reviews by the core
ChromiumOS	642	79	533	64.7%	72.5%
ITK/VTK	244	19	225	57.0%	77.2%
LibreOffice	207	20	187	37.6%	88.0%
OmapZoom	642	34	608	34.3%	60.2%
OpenStack	1880	128	1752	53.6%	66.0%
OVirt	193	20	173	51.3%	61.1%
Qt Project	888	63	825	55.9%	66.1%
Typo3	387	30	357	56.3%	71.0%

Table 3.1: Excerpt of general project data taken from Table 2 in the study by Bosu et al. [BC14]. *Developers* is abbreviated with *dev*.

3.1.5 Results

Subsequent to the data extraction, the network construction, and the classification of developers into core and peripheral, which we have presented in the sections above, Bosu et al. [BC14] calculate and evaluate their results. We present the results for each of the four hyptoheses from Section 3.1.1 in this section.

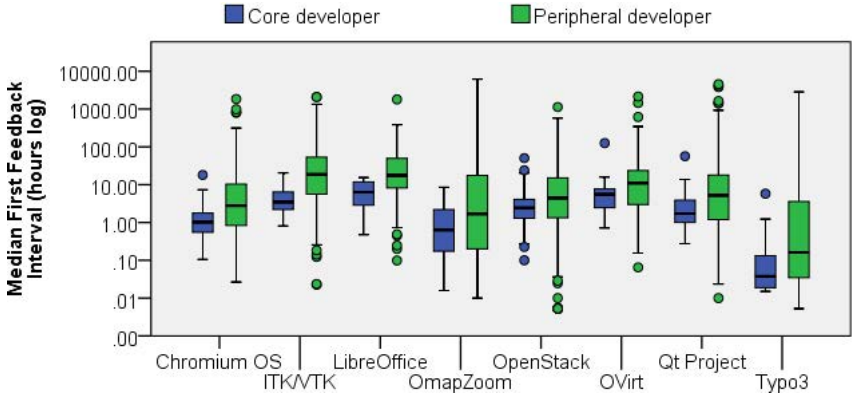


Figure 3.1: Result plot for the first-feedback interval hypothesis of the original study. It shows that core developers have a significantly shorter interval than peripheral developers. This figure is taken from Figure 3 of the original study [BC14].

The first results we present are the ones for the first-feedback interval. We show the results in Figure 3.1. They show that the time span from submission until the first real review comment are 1.8 to 6 times lower for core developers than for peripheral developers. These results support the first hypothesis and thus they accept it.

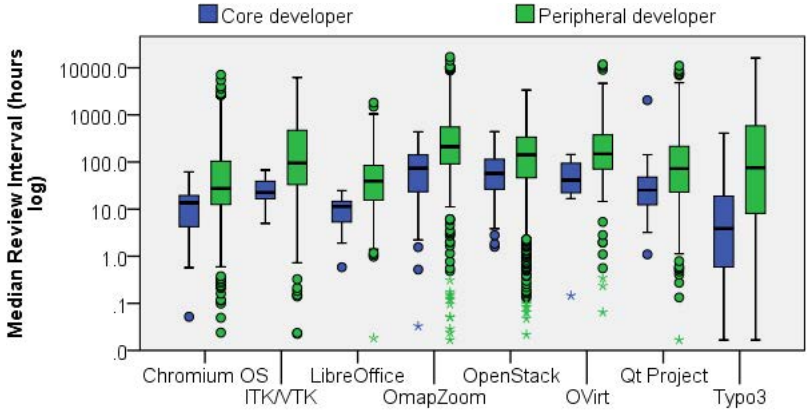


Figure 3.2: Result plot for the review interval hypothesis of the original study. It shows that core developers have a significantly shorter interval than peripheral developers. This figure is taken from Figure 5 of the original study [BC14].

The next hypothesis is the one about the review interval. The median review intervals for core and peripheral developers are shown in Figure 3.2. These values show that core developers have a significantly lower review interval by a factor of 2 to 19 times. This supports the second hypothesis.

The third hypothesis deals with the code-acceptance rate of core and peripheral developers. The acceptance rates for the different case studies are shown in Figure 3.3. The displayed results show that the acceptance rate of core developers is significantly higher than the acceptance rate of peripheral developers. This, again, supports the hypothesis.

The last of the hypotheses we have presented in Section 3.1.1 is about the number of patch revisions a developer has to submit until the patch is accepted. The average

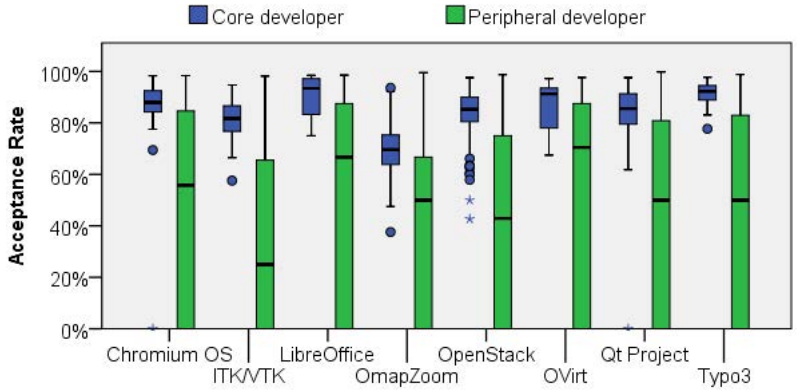


Figure 3.3: Result plot for the code acceptance rate hypothesis of the original study. It shows that core developers have a significantly higher rate than peripheral developers. This figure is taken from Figure 7 of the original study [BC14].

number of patch revisions for core and peripheral developers is shown in Figure 3.4. These results show, unlike the other three hypotheses, no significant difference between core and peripheral developers, in most of the case studies. In fact, the number of patch revisions is higher for the core developers of all the projects, but the results are not statistically significant for six of the eight projects. For that reason, the hypothesis is overall inconclusive.

Based on these results, although the fourth hypothesis is inconclusive, they accept the high-level hypothesis. For that reason, they prove that reputation does indeed have an influence on the outcomes of code-review requests in OSS projects.

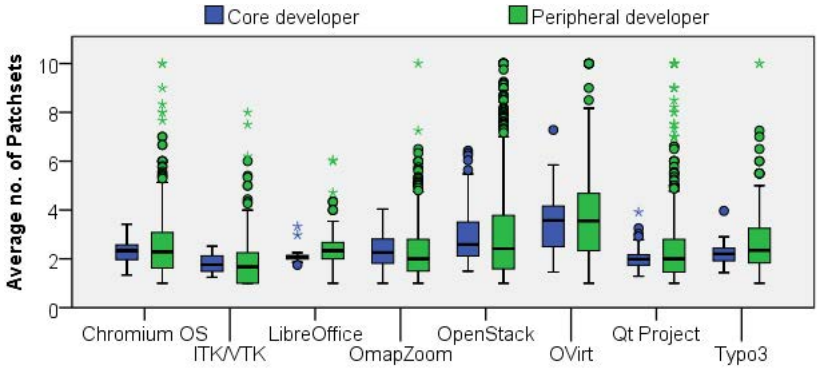


Figure 3.4: Result plot for the number of patch revisions hypothesis of the original study. The results for this hypothesis are overall inconclusive. This figure is taken from Figure 9 of the original study [BC14].

3.2 Research Question and Hypotheses

In this thesis, we want to replicate the original study by Bosu et al. [BC14], which we have described in detail in Section 3.1. For this reason, we state a similar research question: *How do developer roles influence contributions to OSS projects?* To answer our research question we pose the same four hypotheses as the original study. We present these hypotheses in the following subsections.

3.2.1 First-Feedback Interval

Just like Bosu et al. [BC14], we define the first-feedback interval as the amount of time from submission of the contribution on the mailing list to the first answer on it. Core developers should have to wait shorter for a first answer, because they usually have a good relationship with other developers. This could lead these other developers to prioritizing the review of core developers' contributions. Furthermore, core developers should be able to tell who is most appropriate for the review of their contribution and, thus, can address them personally. For this reason we pose the following hypothesis:

Hypothesis H1. *Core developers have a shorter first-feedback interval than peripheral developers.*

3.2.2 Review Interval

Another important measure to consider when looking at contributions to OSS projects, is the time from submission until acceptance. Core developers should be able to get their contributions accepted faster than peripheral developers. The reasons for that are similar to the reasons we propose for the first-feedback interval. Therefore we pose the second hypothesis:

Hypothesis H2. *Core developers have a shorter review interval than peripheral developers.*

3.2.3 Code-Acceptance Rate

One of the most used factors to measure the success of a developer within an OSS project is the code-acceptance rate. This describes the rate of code contributions that get accepted into the project among all the submitted ones. Since core developers usually have a deeper knowledge of the project and things like coding guidelines, they should be able to produce acceptable code more often. For this reason, we pose the next hypothesis:

Hypothesis H3. *Core developers have a higher code-acceptance rate than peripheral developers.*

3.2.4 Number of Patch Revisions

The last measure we take a look at is the number of patch revisions a developer needs for one contribution to be accepted. Every time a reviewer requests changes for a contribution and the originator reworks and resubmits it, the number of revisions is increased by one. So, the higher this number is, the more issues are there with the contribution. Since core developers, like stated before, should have a better knowledge of coding guidelines and the project itself, they should be able to produce higher quality code. Therefore we pose the last hypothesis:

Hypothesis H4. *Core developers need less patch revisions until the contribution gets accepted.*

3.3 Approach

To answer the research question and the four hypotheses that we have presented in Section 3.2, we extract the data of our case studies, construct social networks, classify the developers into core and peripheral, and analyze all according to the hypotheses. We present the detailed approach in the following subsections.

3.3.1 Case Studies

The first thing we need, to investigate the objective of the thesis, are case studies. We choose three OSS projects that use a mailing list as contribution tool.The first project is JAILHOUSE[2], which is a partitioning hypervisor based on Linux. The second project is BUSYBOX[3]. This is a tool that combines a lot of small UNIX utilities into one small executable. The third OSS project we use is the APACHE HTTP SERVER (HTTPD)[4] project. This is a open-source HTTP server for modern operating systems like Windows or Linux. On the basis of these three OSS projects we want to investigate the research question.

3.3.2 Data Extraction

Project	# commits	# mails	time frame
JAILHOUSE	1,786	5,919	2013-10-20 - 2017-01-28
BUSYBOX	14,259	42,013	1999-10-05 - 2016-10-27
HTTPD	29,671	54,921	1996-07-03 - 2017-02-25

Table 3.2: Number of the extracted commits and mails for each project and the time frame the data are extracted from.

To analyze the specifics of the case studies, we need to extract data from them. The three main things we need here are a list of all commits to the project, a list of all the authors, and a list of all mails in the mailing list. We perform this extraction with the help of CODEFACE[5] and a companion tool called CODEFACE-EXTRACTION[6]. CODEFACE is a tool, developed by *Siemens*, that can extract data from multiple data sources surrounding a software project [JMA+15]. The commits and authors, we need, are extracted directly from the git repository of the project. The mails on the mailing list are extracted from publicly accessible `mbox` archives, which we have downloaded from gmane[7].

Once the data is extracted, CODEFACE saves them in a MYSQL database. We extract the data using the CODEFACE-EXTRACTION. This tool saves the lists of commits, authors, and mails into `CSV` files.

In Table 3.2, we present an overview of how many mails and commits are extracted from the different projects and the time frame that the mails got sent in and the

[2]https://github.com/siemens/jailhouse
[3]https://busybox.net/
[4]https://httpd.apache.org/
[5]https://siemens.github.io/codeface/#/home
[6]https://github.com/se-passau/codeface-extraction
[7]http://www.gmane.org

commits got done in. One can see that HTTPD is the largest project, with 29,671 commits, followed by BusyBox, with 14,259 commits. Jailhouse brings up the rear with just 1,786 commits.

3.3.3 Network Construction

Subsequent to the extraction of the necessary data, we construct developer networks, which we have introduced in Section 2.2.1. We use a R library that implements a lot of necessary functionalities surrounding the construction and analysis of networks. We call this tool Network Library from now on.

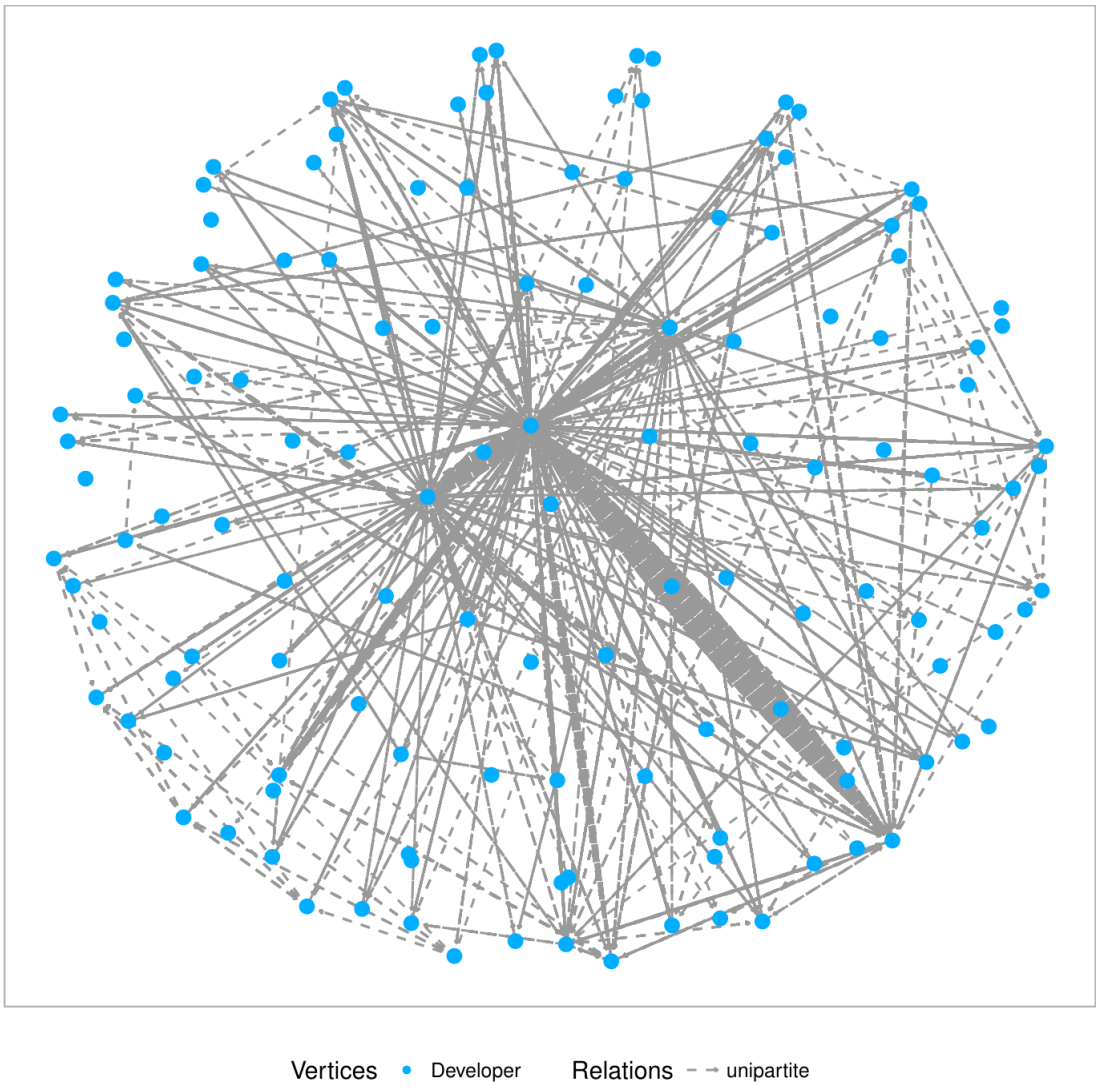


Figure 3.5: Network plot of Jailhouse. One can clearly see the vertices with the highest degrees, since they are nested in big accumulations of edges.

The main part of this Network Library is separated into two classes: The `ProjectData` and the `NetworkBuilder` class. The first of the two classes handles all tasks surrounding the base data for the respective analysis, like reading the data from CSV files. The `NetworkBuilder` on the other hand, is responsible for the actual

network construction. These classes get supported by two configuration classes: The `ProjectConf` and the `NetworkConf`. In the `ProjectConf` object are all necessary values for the current analysis, like the project name. The `NetworkConf` handles all configuration parameters surrounding the construction of the network. This includes things like the relation between developers, whether or not the network is directed or the list of attributes that should be added to the edges.

For this thesis, we build directed developer networks with mail interactions as relation between them. These networks form the basis for our core-periphery-structure detection. We show one of these networks in Figure 3.5. This is the network for JAILHOUSE. One can clearly see the vertices with the most edges in the middle. The vertex with the highest number of edges here is *Jan Kiszka*, who is the top core developer within the degree-core-classification. The networks of the other two case studies look similar to this but are too large, for them to be properly displayed.

3.3.4 Core-Periphery Detection

After the construction of networks, we use them to classify the developers into core and peripheral groups. For that reason, the first thing we need to do, is to apply the two centrality metrics, which we have described in Section 2.2.2, and the hierarchy metric from Section 2.2.3.

The values that these metrics assign to the developers in the network, form the basis for the classification. We use the method, that we have described in Section 2.2.2. We do not use some sort of unification metric between the three metrics, like Bosu et al. [BC14] use the K-means clustering algorithm. For that reason, we process the core-detection for every one of our three network metrics separately and evaluate them separately, as well. To do so, we calculate the thresholds at the 80% percentile for the current metric. Subsequently we assign every developer that has a metric value above the threshold to the core group and the others to the peripheral group. The developers seem to be doing 80% of the work and that is the most used incentive for a developer to be a core developer, like we have described in Section 2.2.4.

To calculate the agreement between the three classification methods, we calculate the Cohen's kappa. The Cohen's kappa is calculated with the following formula:

$$\kappa = \frac{p_o - p_c}{1 - p_c} \tag{3.1}$$

In this equation, p_o represents the number of times that two classifications agree, divided by the number of developers. The value of p_c is the expected probability of agreement, when the developers get classified randomly but with the same proportion of the core and the peripheral group. [JAHM17, LK77]

3.3.5 Mapping Patches and Commits

The last step toward the analysis of our hypotheses, is the mapping of patches on the mailing list to the respective commit, they lead to. To do so, we use PaStA[8], a tool for the analysis of patch stacks. Since we only use the mailbox analysis of the tool, we do not get into detail about what a patch stack is or in what way it is analyzed. [RLM16]

The first step of the mailbox analysis is to extract the mails from, in our case, `mbox` archives and store them in a folder structure according to the years of their sending. Subsequently, the mails get cached and the ones containing patches detected. Following this step, one has to compare the patches on the mailing list against each other semi-automatically. This is done by setting the thresholds that represent the resemblance between two patches from hand and look at the results above this this threshold. Once the right threshold is found, by lowering it more and more until only false positives are displayed, PaStA maps the pairs above it together. These mappings represent different versions of the same patch on the mailing list. Subsequent to this first mapping, PaStA caches the commits in the repository of the current project. In the end, one has to do the same mapping for commits and patches as for patches among each other. The results are then saved in a file, which we refer to as pasta data. We read this data from the file and process it with the help of the Network Library. The reason that we do this mapping is that it is the only way to find out whether or not a patch is successful in the end and how much time it needed to be accepted.

3.3.6 Approach for the Analysis of the Hypotheses

Having done all the steps from the sections above, we can analyze the hypotheses from Section 3.2 in detail. Therefore, we use `R`-scripts. We present the approach for evaluating our four hypotheses in this subsection.

For the investigation of H1, we need to calculate the average first-feedback interval per developer. First, we iterate over the classified lists of core and peripheral developers, developer by developer. Then we iterate over the patch-revision sets that contain at least one patch by the developer. We then extract the earliest patch within this set, that was submitted by the developer. Subsequently, we extract all the mail-threads that contain one of the patches in the set and find the earliest mail after the previously extracted patch, that was not sent by the developer. In the end we calculate the time difference in hours between the earliest patch and the earliest answer. Once this is done, we calculate the mean of these intervals.

For the analysis of H2, we analyze the review interval. To calculate this, we iterate over the developers in the previously described way. Then, we look at every successful set of patch revisions, that contain at least one patch by the developer. Subsequently, we extract the earliest patch by the developer within the set of patch revisions and find the earliest commit that is associated with the set of patch revisions. Finally, we calculate time difference between the earliest patch and the earliest commit and calculate the mean value of all these intervals for a developer, in the end.

[8] https://github.com/lfd/PaStA

For the investigation of H3, we need to to calculate the code-acceptance rate for every developer. To do so, we take the classified lists of developers and the list of mails of the project. Thereafter, we iterate over the developers per classification metric, separated for core and peripheral developers, and find the number of patch-revision sets that contain at least one patch by the developer. Then we extract the number of these patch-revision sets that are eventually merged. Finally, we divide the number of successful patch-revision sets by the total number of patch-revision sets of the developer and get the code-acceptance rate as result.

For the analysis of H4, we calculate the average number of patch revisions for each developer. The first step to calculate this is, again, to take the classified lists of core and peripheral developers and iterate over them separated by developer role and classification metric. We then look at every successful set of patch revisions that contain at least one patch by the developer and calculate the mean number of patches that all of these sets contain.

For the statistical evaluation of our results, we use Wilcoxon Mann-Whitney tests. This test elaborates whether two distributions of values are from the same or from different distributions. We use alternative hypotheses for our analyses, because we want to prove that the distribution of the values of core developers is either greater or less than the one for peripheral developers. For the evaluation of H1, H2, and H4 we use the alternative hypothesis that the core distribution is less than the peripheral distribution. For H3, we use the alternative hypothesis that the core distribution is greater than the peripheral distribution.

4 Evaluation and Results

In this chapter, we present the results of our core-periphery classifications, the results of the Cohen's kappa calculation and the results of the analysis with PaStA. Moreover, we evaluate the results of the analyses of our hypotheses. For this, we present violin plots for the results of the different case studies and classifications and statistically evaluate the results by using Wilcoxon-Mann-Whitney tests. Subsequently, we discuss all the results in contrast to the results of the original study.

4.1 Overview on Data Preparation

In this first section, we present the results of the core-periphery classifications, including the Cohen's kappa among them. Furthermore, we present the results of the mapping of patches on the mailing-list to commits in the project's repository.

4.1.1 Classifications

We explain how we classify the developers of our three case studies on the base of three different network metrics in Section 3.3.4. We show the results of these classifications in Table 4.1. There we show that the number of core developers for degree and eigenvector centrality are, at least, similar for every case study. The number of core developers in the hierarchy classification is significantly lower and the overall number of developers seems to be lower as well. This is due to the fact that developers that do not belong in the hierarchy, do not appear here. These developers are represented by loose vertices in the network. These can be seen in Figure 3.5, for example. These vertices do not exhibit a clustering coefficient and can therefore not be classified with the hierarchy metric. We elaborate this further in Section 4.3.1.

Next, we calculated Cohen's kappa among the three classification methods to determine the agreement of them. We show the results in Table 4.2. The values indicate that the agreement of degree and eigenvector centrality is quite high. The agreement of the two centrality classifications and the hierarchy classification, on

Project	Degree		Eigenvector		Hierarchy	
	core	peripheral	core	peripheral	core	peripheral
Jailhouse	3	128	1	130	2	67
BusyBox	81	2,655	78	2,658	3	1,608
httpd	61	2,086	45	2,102	4	1,158

Table 4.1: Numbers of core and peripheral developers per classification metric and case study.

Classification	Degree	Eigenvector	Hierarchy
Degree	-	0.81	0.11
Eigenvector	0.81	-	0.12
Hierarchy	0.11	0.12	-

Table 4.2: Cohen's kappa for the three core-peripheral classifications, indicating the agreement among the different classifications.

the other hand, is very low. These results are similar to the results Joblin et al. have found in their study on core-periphery detection [JAHM17].

One topic we need to address here is that the overall number of developers is lower, when the projects get classified with the hierarchy metric, than for the other two classifications. This is due to the fact that we include all developers that are somehow included in the projects into our networks. This means that there are also some developers that did never send a mail on the mailing-list, which leads them to be loose vertices without edges in the network. This is no problem for degree centrality and eigenvector centrality, since these vertices are just classified as the lowermost developers in the peripheral group. But since they are not connected to any other vertices and thus do not have a clustering coefficient, the hierarchy metric can not classify these loose vertices. This leads to them being eliminated during the classification process. We do not bother to re-include them, as they do not send any patches anyway and therefore have no influence on the results of the hypotheses.

4.1.2 PaStA Mapping

Project	# patches	# revisions sets	# successful revision sets
Jailhouse	2,035	1,428	1,040
BusyBox	2,229	1,924	786
httpd	839	750	39

Table 4.3: Analysis results of PaStA.

Next, we present the mapping results of the mailing-list analysis with PaStA, as outlined in Section 3.3.3. We present in Table 4.3, the number of mails that contain patches, the number of patch-revision sets, and the number of successful patch-revision sets. The data indicates that most of the submissions to Jailhouse eventually get accepted (∼73%). The data for BusyBox shows that around 40% of

the patch submissions get accepted eventually. For HTTPD, the acceptance rate is only at around 5%, which is a very low rate. This could have multiple reasons. We elaborate this further within the discussion part in Section 4.3.1.

4.2 Results

In the following, we present the detailed results regarding our hypotheses. Furthermore, we decide whether we accept the respective hypothesis for every case study based on the statistical results or not. Moreover, we present the result plots for the different hypotheses, case studies, and classifications. We do not present the mean values for the first-feedback interval, the review interval, the code-acceptance rate, and the number of patch revisions, in numerical form. We do this, because these values are not very relevant for the results of the analyses. But we do mark these mean values within the plots. They are painted as the red dot in the violin plots.

The significance level of our statistical results is at 5%. So every p-value in the following sections, which is lower than 0.05, marks a statistically significant result.

4.2.1 Hypothesis H1: First-Feedback Interval

The first results we present are for the first-feedback interval. As stated in Section 3.2.1, we hypothesize that core developers get a faster first feedback on their submissions to an OSS project. The statistical results with the result values of the Wilcoxon Mann-Whitney test W and the statistical significance levels p are presented in Table 4.4.

	Degree		Eigenvector		Hierarchy	
Project	W	p-value	W	p-value	W	p-value
JAILHOUSE	75	>0.99	26	0.96	40	>0.99
BUSYBOX	7,813	0.55	6,937	0.34	338	0.65
HTTPD	1,381	0.54	1,169	0.58	135	0.32

Table 4.4: This table shows the statistical results of the analysis of Hypothesis H1. W and p mark the results for the Wilcoxon-Mann-Whitney tests, where p is the more important value since it describes the statistical significance.

The results for JAILHOUSE show that the first-feedback interval seems to be lower for peripheral developers than for core developers. The most outstanding result here is the one for the classification with degree centrality. It shows a significance level of over 99%, which could indicate that the first-feedback interval is actually higher for core developers than for peripheral developers, as evident in Figure 4.1. Since we do not perform the Wilcoxon Mann-Whitney test with the opposite alternative, we can not definitely confirm this. In Figure 4.1, the violin plot for core developers is clearly positioned higher. The results for the other two classifications show similar results. The plots for these two classifications are shown in the Figures A.1 and A.2.

The results for the analysis of BUSYBOX do not support our hypothesis either. The p-values in Table 4.4 are not below 0.05, which indicates that the results are

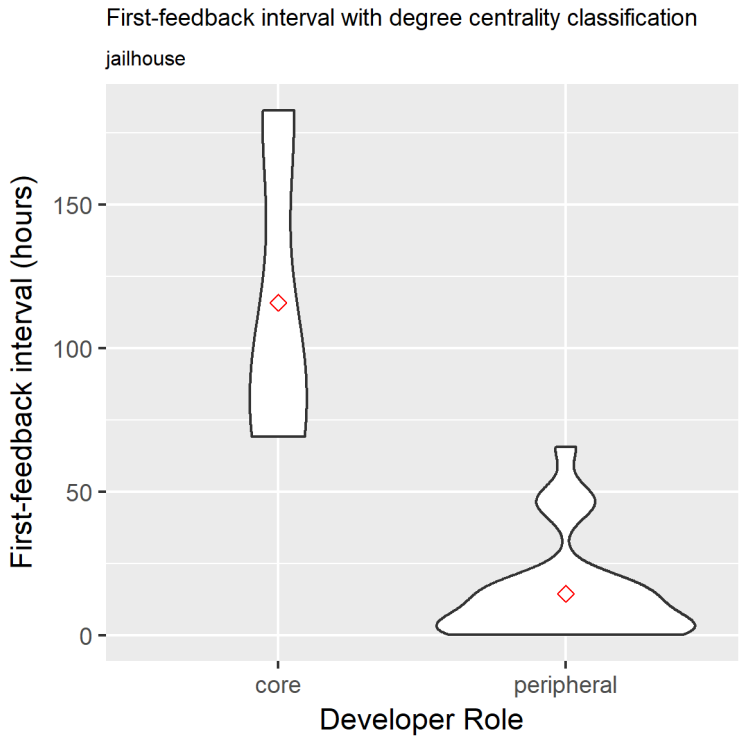


Figure 4.1: Violin plots for the first-feedback intervals of core and peripheral developers in the JAILHOUSE project. The classification metric is degree centrality.

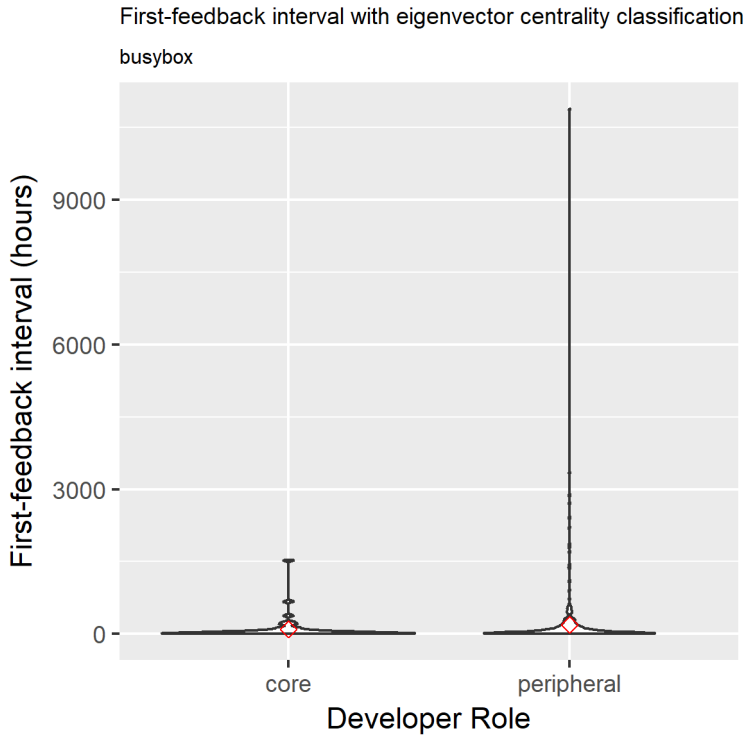


Figure 4.2: Violin plots for the first-feedback intervals of core and peripheral developers in the BUSYBOX project. The classification metric is degree centrality.

not significant for our hypothesis. Since the plots for the classification with degree centrality and the one with eigenvector centrality show basically the same, we show only one of them, presented in Figure 4.2. Here, we see that the first-feedback intervals for core and peripheral developers are both very low, but very similar in terms of hours. Furthermore, we see that some peripheral developers have a way higher first-feedback interval than the average, which is not the case for core developers. The plot for the classification with hierarchy is shown in Figure A.3.

The results for HTTPD are similar to the results for BUSYBOX. Just like for BUSYBOX, the results for HTTPD do not show statistical significance. The plot of the classification with hierarchy, which we show in Figure 4.3, looks similar to the plots of BUSYBOX. We see again that the the first-feedback interval for most core and peripheral developers is very low, but the outliers among the peripheral developers have to wait significantly longer for a first feedback than the rest. The plots for the classifications with degree and eigenvector centrality show similar findings. We show these plots in the Figures A.4 and A.5.

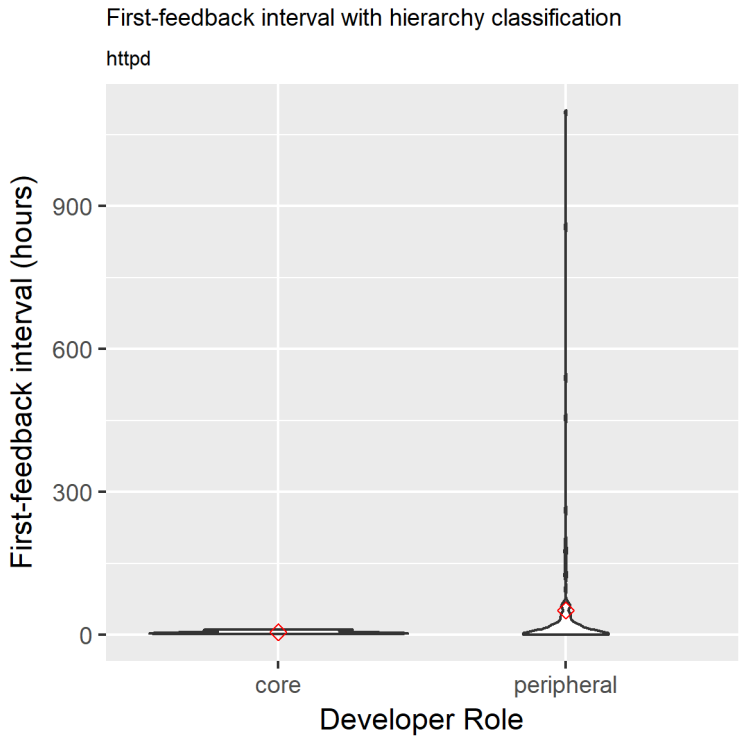


Figure 4.3: Violin plots for the first-feedback intervals of core and peripheral developers in the HTTPD project. The classification metric is hierarchy.

In conclusion, we **reject** Hypothesis H1, since none of the results support it. Some of the results even indicate that the exact opposite of H1 could be the case.

4.2.2 Hypothesis H2: Review Interval

	Degree		Eigenvector		Hierarchy	
Project	W	p-value	W	p-value	W	p-value
JAILHOUSE	47	0.99	16	0.85	22	0.90
BUSYBOX	2,401	0.80	2,300	0.74	96	0.22
HTTPD	33	0.14	36	0.24	5	0.06

Table 4.5: This table shows the statistical results of the analysis of Hypothesis H2. W and p mark the results for the Wilcoxon-Mann-Whitney tests, where p is the more important value since it describes the statistical significance.

In this next section, we present the results for Hypothesis H2, which centers on the review interval of core and peripheral developers. As stated in Section 3.2.2, we hypothesize that the review interval is shorter for core developers than for peripheral developers.

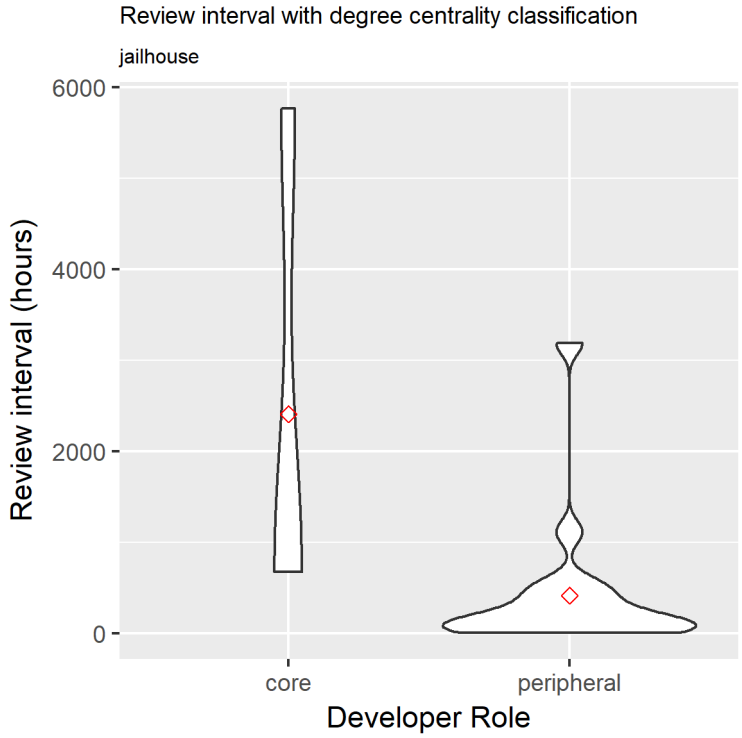


Figure 4.4: Violin plots for the review intervals of core and peripheral developers in the JAILHOUSE project. The classification metric is degree centrality.

The first case study we take a look at is JAILHOUSE. The results for this project do not support our hypothesis. This is evident from the p-values listed in Table 4.5. Since these values are not below 0.05, there is no statistical significance in the results for our hypothesis. The values for the classification with degree centrality even indicate that the opposite hypothesis may be true. This could show that the review intervals of core developers are higher than the ones of peripheral developers.

We show the plot to support this in Figure 4.4. The results for the classifications with eigenvector centrality and hierarchy show that the mean value is basically on the same level for core and peripheral developers, but that some outliers of the peripheral developers have a way higher review interval. Since the plots for these two classifications very much look alike, we only present one of them in Figure A.6.

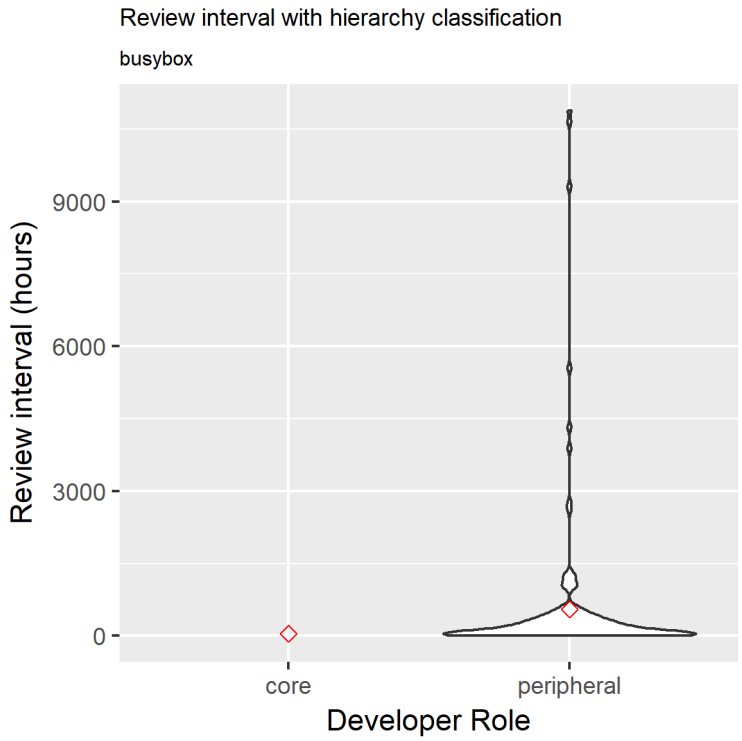


Figure 4.5: Violin plots for the review intervals of core and peripheral developers in the BusyBox project. The classification metric is hierarchy.

The next case study is BusyBox. Here, we find that the two core developers indeed have a shorter review interval than peripheral developers, when the developers get classified using the hierarchy. We show the violin plot in Figure 4.5. However, the results do not show statistical significance, which can be seen in Table 4.5. The results for the classifications with degree and eigenvector centrality, on the other hand, show that the review interval is similar for core and peripheral developers. The only difference is that the outliers of the peripheral developers have to wait longer for the completion of the review process than the outliers of the core developers. The plots for the two classifications do look alike and for that reason, we present only one of them in Figure A.7.

The results for our third case study, HTTPD, do show that core developers have to wait less time until their contribution finishes the review process. But the results show no statistical significance. The plot for the classification with hierarchy is shown in Figure 4.6, where we can see that the review intervals seem to be shorter for core developers than for peripheral developers. The plots for the other two classifications are again very similar and for that reason, we only show one of them in Figure A.8. Here, we can see that the plot for core developers is indeed posi-

tioned lower than the one of peripheral developers. But since there is no statistical significance in the results, we can not accept the hypothesis here.

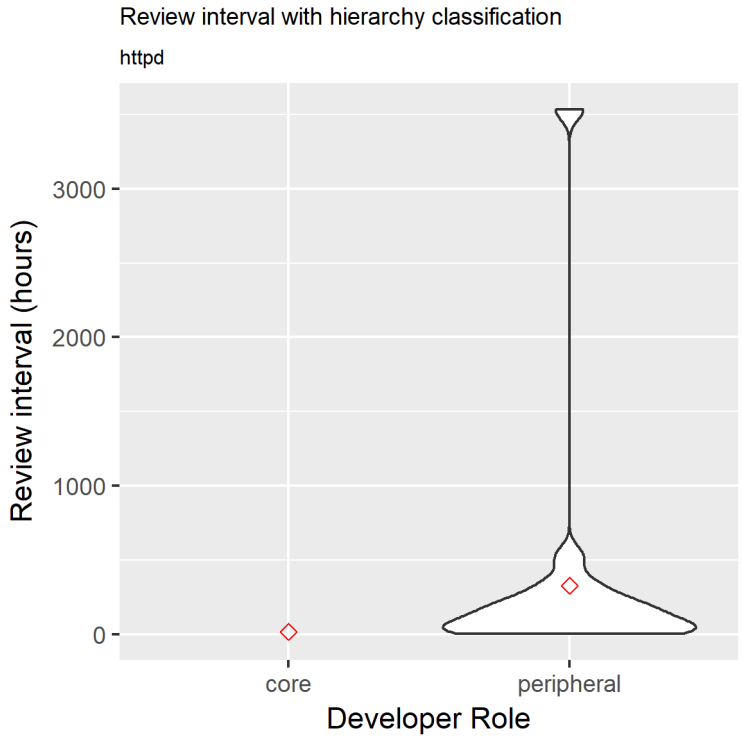


Figure 4.6: Violin plots for the review intervals of core and peripheral developers in the HTTPD project. The classification metric is hierarchy.

Two of our three case studies show results that could support our hypothesis, but they show no statistical significance. Furthermore, there are even results that support the opposite hypothesis, meaning that core developers have to wait longer than peripheral developers. For these reasons, we **reject** Hypothesis H2.

4.2.3 Hypothesis H3: Code-Acceptance Rate

The third results we present are the results for the code-acceptance hypothesis that we pose in Section 3.2.3. The statistical results are shown in Table 4.3.

	Degree		Eigenvector		Hierarchy	
Project	W	p-value	W	p-value	W	p-value
JAILHOUSE	44	0.33	10	0.69	26	0.26
BUSYBOX	10,957	0.04	10,624	<0.01	457	0.12
HTTPD	2,246	0.03	1,904	0.02	288	0.02

Table 4.6: This table shows the statistical results of the analysis of Hypothesis H3. W and p mark the results for the Wilcoxon-Mann-Whitney tests, where p is the more important value since it describes the statistical significance.

Figure 4.7: Violin plots for the code-acceptance rates of core and peripheral developers in the JAILHOUSE project. The classification metric is degree centrality.

Figure 4.8: Violin plots for the code-acceptance rates of core and peripheral developers in the BUSYBOX project. The classification metric is eigenvector centrality.

The results for JAILHOUSE show that the mean code-acceptance rate is higher for core developers than for peripheral developers in two of the three classifications. The classification with eigenvector centrality shows a higher acceptance rate for peripheral developers. This might be due to the fact that only one developer is classified as core. We show the plot for the code-acceptance rates of the classification with degree centrality in Figure 4.7. We clearly see that the distribution of acceptance rates is positioned higher than the one of peripheral developers. The plots for the classification with eigenvector centrality and hierarchy are shown in the Figures A.9 and A.10. As for statistical significance, none of the results exhibit p-values less than 0.05, as shown in Table 4.6.

The results for BUSYBOX show that the code-acceptance rate of core developers is higher or equal for all three classifications. But only the classifications with degree and eigenvector centrality show statistical significance. The distribution of the acceptance rates with eigenvector centrality as classification metric, is shown in Figure 4.8. We can see that most of the peripheral developers have a acceptance rate of under 25%, while the core developers are more equally distributed. The plots for the results with degree centrality and hierarchy are shown in the Figures A.11 and A.12.

Figure 4.9: Violin plots for the code-acceptance rates of core and peripheral developers in the HTTPD project. The classification metric is hierarchy.

The results for HTTPD show statistical significance for all three classifications. This means that patch acceptance rates of core developers are overall greater than the ones of peripheral developers. This supports our hypothesis. We show the violin plot for the classification with the hierarchy metric in Figure 4.9. The results for the degree and the eigenvector centrality are shown in the Figures A.13 and A.14.

Based on the results that we find for the three case studies, we can neither accept nor refuse the hypothesis. The only case study that shows statistically significant results for all three classifications is HTTPD, but the other two are inconclusive. In conclusion, we state that the hypothesis is **inconclusive**.

4.2.4 Hypothesis H4: Number of Patch Revisions

In Hypothesis H4, we state that core developers should need to submit less patch revisions until a patch is accepted than peripheral developers. The statistical results of the analysis are shown in Table 4.7.

	Degree		Eigenvector		Hierarchy	
Project	W	p-value	W	p-value	W	p-value
JAILHOUSE	42	0.97	18	0.95	19	0.82
BUSYBOX	2,658	0.92	2,461.5	0.78	189	0.78
HTTPD	49.5	0.41	38.5	0.11	16	0.31

Table 4.7: This table shows the statistical results of the analysis of Hypothesis H4. W and p mark the results for the Wilcoxon-Mann-Whitney tests, where p is the more important value since it describes the statistical significance.

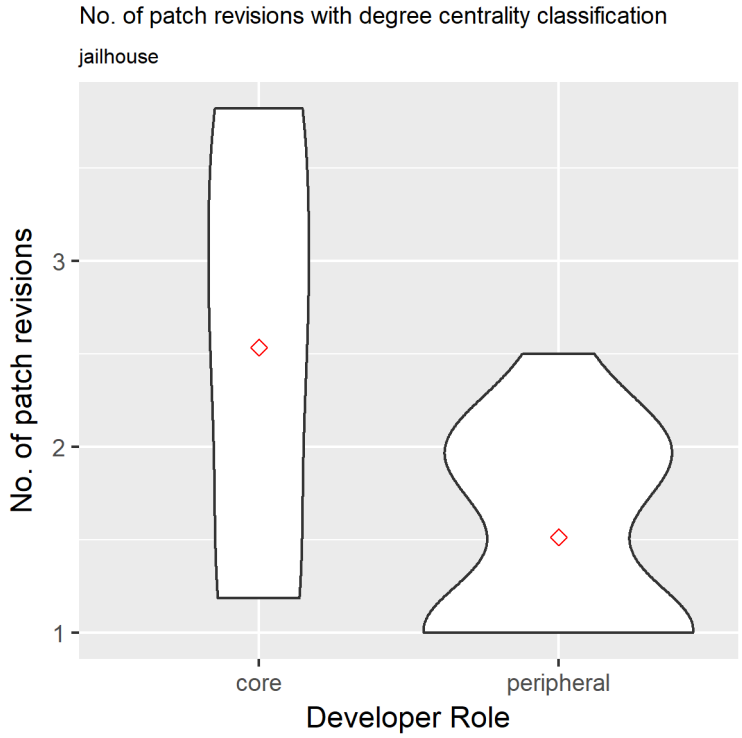


Figure 4.10: Violin plots for the number of patch revisions of core and peripheral developers in the JAILHOUSE project. The classification metric is degree centrality.

The results for the JAILHOUSE do not support our hypothesis. The results for the classification with degree centrality even indicate that there could be statistical

significance for the opposite hypothesis. This means that the number of patchsets could be higher for core developers than for peripheral developers in this case. The plot for this is shown in Figure 4.10. We can clearly find that the number of patch revisions is lower for peripheral developers than for core developers. The results for the classifications with eigenvector centrality and the hierarchy metric tend to this conclusion as well. The plots for these two classifications are shown in the Figures A.15 and A.16.

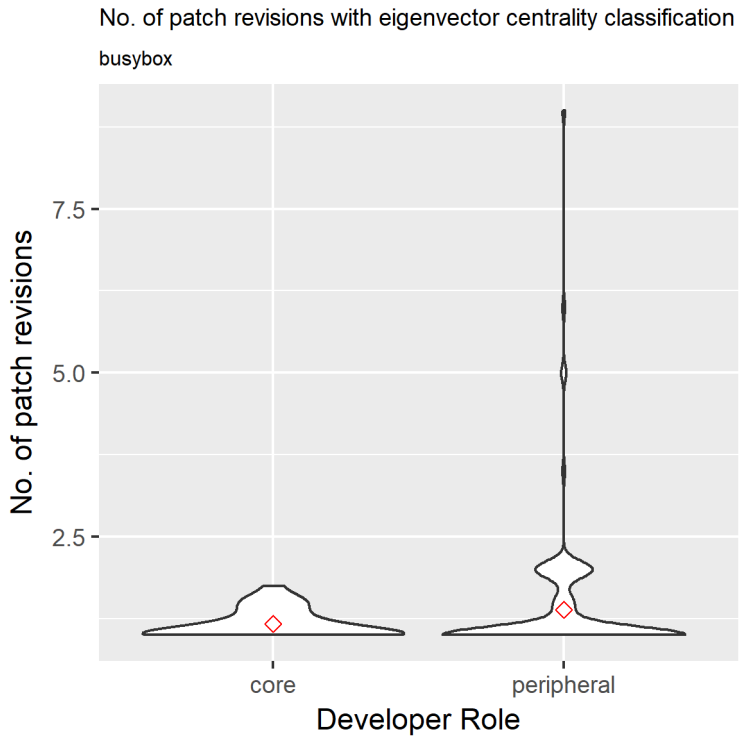


Figure 4.11: Violin plots for the number of patch revisions of core and peripheral developers in the BusyBox project. The classification metric is eigenvector centrality.

The next project, we take a look at, is BusyBox. The results for this case study do not support the hypothesis. Since all three classifications have a similar analysis outcome, we only present the plot for the classification with degree centrality in Figure 4.11. Here, we can see that the numbers of patch revisions are pretty similar for core and peripheral developers, but the few outliers for the peripheral developers have to send significantly more patchsets than the outliers among the core developers.

The last results we present are the ones for httpd. These results basically support the hypothesis but since they do not show statistical significance, we can not ultimately confirm the correctness of these indications. The results for the classifications with eigenvector centrality and the hierarchy metric both show that core developers only need to send one patch revision until the patch is accepted. Peripheral developers obviously need more revisions. We show these results in Figure 4.12. Since the plots for the classifications with eigenvector centrality and hierarchy very much look alike, we only present one of them. The plot for the classification with degree

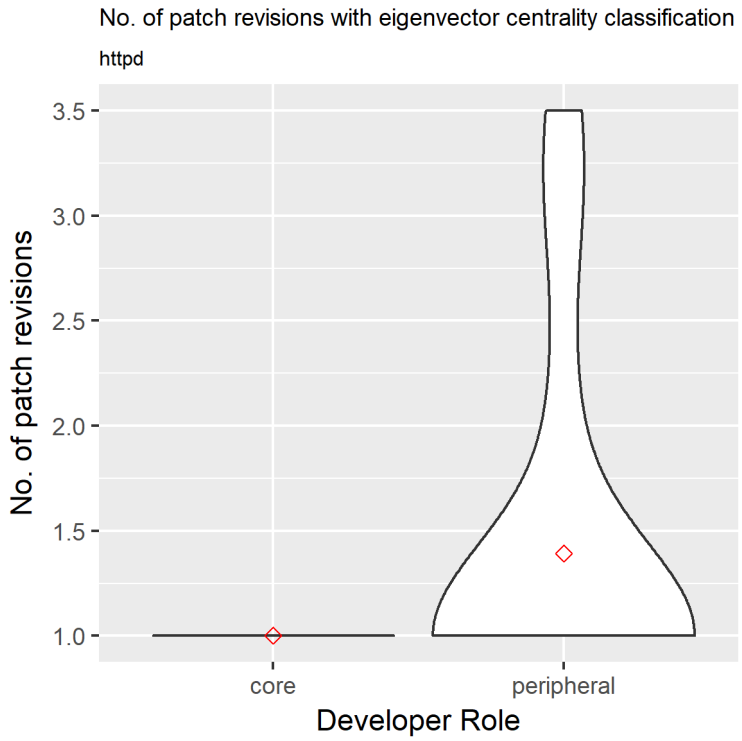


Figure 4.12: Violin plots for the number of patch revisions of core and peripheral developers in the HTTPD project. The classification metric is eigenvector centrality.

centrality shows that core developers need between 1 and 2 revisions and peripheral developers need between 1 and 3.5. These results are shown in Figure A.17.

In conclusion, we have to **reject** the hypothesis, since we do not have clear results. Some of our results even indicate that the opposite of our hypothesis could be the case.

4.3 Discussion

In this next section, we discuss the results we presented in Section 4.2 and set them in contrast to the results of the original paper by Bosu et al. [BC14].

4.3.1 Hypothesis H1: First-Feedback Interval

The first hypothesis, Bosu et al. [BC14] and we pose, is that core developers have a shorter first-feedback interval than peripheral developers (H1). The findings of the original study are able to confirm this hypothesis. We have to reject it.

In fact, the results for JAILHOUSE even indicate that the first-feedback interval is longer for core developers. We argue that this could be due to the fact that core developers work on more complex patches. This could lead to a longer time span until a reviewer has a first feedback on the sent patch. Our other two case studies show no statistically significant difference in the first-feedback interval for core and peripheral developers. This means that, no matter what role a developer fits in, all developers get a first feedback in the same time frame.

Since we find different results for our three case studies, we can not generalize them. The fact that we find differences in the results could show that the first-feedback interval depends more on the OSS project than the developer role.

4.3.2 Hypothesis H2: Review Interval

Hypothesis H2 of the original paper and this study is that core developers have a shorter review interval than peripheral developers. Bosu et al. [BC14] accept this hypothesis based on their analysis. We, on the other hand, have to reject the hypothesis.

One of the classifications of JAILHOUSE even shows results that could militate in favor of the opposite of the hypothesis. Again we argue, that this could be due to the fact, that core developers could be working on more complex patches than peripheral developers. This could lead to a longer review process. The results for the other case studies indicate that there is no significant difference between the review interval for core and peripheral developers. But the results also show that some outliers among the peripheral developers have to wait a significantly longer time for the completion of the review process than the core developers. This could indicate that the hypothesis might be true, at least, in some cases. The last point we want to address here is that the classification with the hierarchy metric for HTTPD shows significant results that support the hypothesis. Again, this leans toward the hypothesis being correct. Nevertheless, this is only one of 9 results and our results for HTTPD might not be meaningful for the whole project, which we elaborate further in Section 5.1.

4.3.3 Hypothesis H3: Code-Acceptance Rate

The third hypothesis Bosu et al. [BC14] and we pose is that core developers have a higher code-acceptance rate than peripheral developers. The original study accepts this hypothesis based on their results. With the outcome of our study, we can not accept the hypothesis. But since more than half of our results show significant results that tend toward the acceptance of the third hypothesis, we do not reject it either. The remaining results, except for one, show clear tendencies toward the fact that core developers have a higher code-acceptance rate than peripheral developers. For these reasons, we state that the results are overall inconclusive for the hypothesis.

As we can not definitely conclude whether core developers have a higher code-acceptance rate than peripheral developers, or vice versa, we give reasons for both cases. If the acceptance rate of core developers is higher, we can argue that this should be due to the fact that they are usually more involved with the project. This indicates that they have a deeper knowledge of the project and can therefore produce higher quality patches that can be accepted more easily. If we had to argue for the fact that peripheral developers have a higher code-acceptance rate, we could state that peripheral developers usually only work on small bug fixes or extensions. This could make it easier to get the patch accepted, as such small, patches are not so error-prone as large, complex patches.

4.3.4 Hypothesis H4: Number of Patch Revisions

Hypothesis H4 of the original study and this thesis is that core developers need to submit less patch revisions until a patch is accepted. Bosu et al. find that their results are overall inconclusive, but show tendencies toward the hypothesis. We, on the other hand, have to reject the hypothesis.

Our results for JAILHOUSE show that the numbers of patch revisions of core developers is even higher than the one for peripheral developers. But since the results do not show statistical significance, we can not confirm this either. Nevertheless, we hypothesize that the high number of patch revisions for core developers is, as before, based on the fact that core developers might be working on more complex parts of the system. This could lead to a more fault-prone development process and thus to more reworks being necessary. Most peripheral developers need more than one revision, too. But we hypothesize that this is due to other reasons than the ones for core developers. A possible explanation could be that peripheral developers are not so familiar with the development process and the coding guidelines of a project. This could lead to many formal mistakes in contributions which have to be corrected with another patch revision.

4.3.5 Influence of Developer Roles

The research question of this thesis, which we asked in Section 3.2, is how developer roles influence contributions to OSS projects. This is basically the same question Bosu et al. [BC14] ask in the original study. They answered it by finding that core developers have a shorter first-feedback interval, a shorter review interval and a higher code-acceptance rate than peripheral developers.

Based on our results we can only say that the influence seems to be non existent in our case studies, although our results for the code-acceptance rate indicate that core developers could get their contributions accepted at a higher rate. But since these results are not conclusive overall, we can not definitely confirm this.

Some of the results even indicate that the choice of the project has a higher influence on the contributions. This could be substantiated by the fact that we get different results for the different case studies in some cases.

4.3.6 Difference to the Original Study

This next part deals with the differences between our thesis and the original study [BC14]. The main differences here are the classification of developers and the choice of case studies.

While Bosu et al. use six different centrality metrics and combine them by using a K-means clustering algorithm, we use three network metrics separately. The problem here is that we partially get different results for the different classification methods we use, while the original study only has one result per hypothesis and case study as outcome. This could make it difficult to compare the results of the two studies, since the classification of the developers forms the base of the whole study.

Moreover, Bosu et al. choose eight OSS projects that use GERRIT as contribution tool, while we choose three projects that use mailing-lists as contribution tool. The

choice of different case studies in two studies with the same goal is basically a good idea. But in our case there could arise some issues with that. Gerrit offers a lot of features that a mailing-list can not offer. For example one could integrate a tool for automated testing of a contribution into the submission process. This could lead to developers having to change their contributions multiple times before even submitting them. These patch revisions would not be included in the data. This could also affect the review time since the time a developer needs for the first few changes are not included. On a mailing-list, on the other hand, no such tools can be included. The patch has to be submitted to find mistakes in it.

4.3.7 Influence of the Project Size

In this section we want to talk about the difference in size between our three case studies and what influence this could have on our results. For this reason we state that Jailhouse is the smallest of our two projects and can be considered a small OSS project, while BusyBox and httpd, as we can see in Table 3.2, are significantly larger.

When we look at the results for Jailhouse, we can see that the results for the first-feedback interval, the review interval, and the number of patch revisions show strong indications that peripheral developers have shorter time intervals and need less revisions. We argue that this could be due to the fact that Jailhouse makes more efforts of keeping the support of peripheral developers, due to its size. Since it is a rather small project, there are also fewer people that contribute to it and thus has to rely more on the support of peripheral developers. This could lead reviewers to giving more thorough support to peripheral developers rather than to core developers. This could definitely lead to a shorter first-feedback interval and possibly to the faster completion of the review process. Moreover, the higher amount of assistance for peripheral developers could also lead to them needing fewer revisions until a patch is in a acceptable state.

To conclude this, we can say that there are some strong indications that peripheral developers achieve faster results for their contributions with less revisions than core developers in small OSS projects. However we can not confirm this and leave the answering of this question to future studies.

5 Threats to Validity

In this chapter we present the threats to internal, construct, and external validity.

5.1 Internal Validity

One of the main threats to internal validity is our project selection. We choose three OSS projects that use a mailing-list as contribution tool. This means that we can not determine, whether our results also apply to projects with other contribution tools. The next threat that goes hand in hand with the project selection, is the selection of the particular mailing-list, we choose for every project. We do use the main development mailing-list for each project, but large OSS projects usually have more than one mailing-list overall. This leads to the fact that we can not guarantee that the mailing-list we analyze is the only one that is used for patch submissions. Therefore, we do not know if we analyze all patch submissions of a project.

The next big threat are the tools we select for our analysis. In the original study, Bosu et al. [BC14], use only OSS projects that use Gerrit as contribution tool. This makes is easier get all necessary data for all of the code submissions, since they can simply be mined from Gerrit. We, on the other hand, have to rely on the correctness of multiple tools and have to gather the information surrounding code submissions by analyzing them by hand. The first tool we rely on is Codeface. Here we have to rely on the correctness of the data extraction, since it forms the base for all further analyses. Moreover, we have to rely on the correctness of PaStA, the tool we use to detect patches on the mailing-list and map them to commits in the repository of the respective project.

One issue here is that PaStA is only able to map 5% of the submissions to the mailing-list, to commits in the httpd project. This could have multiple reasons. One of them could be that we use a mailing-list that does not contain a lot of successful patches, or patches generally. This could also explain the fact that only 839 of the over 50,000 mails within the mailing-list, are found to contain patches. Another reason might be that PaStA is simply not able to find the patches, due to a unknown or wrong format of them.

5.2 Construct Validity

The threats to construct validity mainly center on the network metrics we use to determine the role of a developer. In contrast to Bosu et al. [BC14], we use only three of such network metrics and we look at them separated and do not combine them, like they do in the original study. Since the classification results differ a lot, in some cases, we can not validate what metric we use for classification is the most meaningful for each project.

Furthermore, just like the original study, we assume that the outcomes of code-review requests are dependent on the role a developer exhibits. However, there may be a lot of other parameters that can influence the outcomes of such review requests, such as the complexity of the patch or whether a developer is working on the OSS project voluntarily or on behalf of a company.

5.3 External Validity

Since we only analyze OSS projects that use a mailing-list as contribution tool, we can not determine to what extent our results can be generalized for other OSS projects. This gets supported, by the fact that we find different results than the original study [BC14]. Moreover, since there is a large variety among OSS projects regarding all sorts of aspects, it is generally difficult to apply results that have been found for a few projects, to all of the other projects.

6 Conclusion

In this last chapter of our thesis, we want to summarize what we did and what we found. Moreover, we want to give prospects of future work.

6.1 Summary

In this thesis we investigated the influence of developer roles on contributions to OSS projects. We did this as a replication study, to confirm or refuse the results of the original study by Bosu et al. [BC14]. The main parameters for our investigation were the time span a developer has to wait for a first feedback on the contribution and the amount of time, the developer has to wait until the review process of the contribution is finished. Furthermore, we looked at the code-acceptance rate of developers and how many patch revisions they have to submit, until the patch is accepted.

To analyze the topic of our thesis, we extracted data from the version control systems and the mailing-lists of our three case studies. These case studies are: JAILHOUSE, BUSYBOX and HTTPD. Based on this extracted data we constructed developer networks with mail interactions as relation between the developers. We used these networks to classify the developers of each case study into core and peripheral developers, by applying three different network metrics. Subsequently, we mapped the patches within the mailing-lists of our projects to their respective commits in the version control systems. Based on these data we performed the analysis of the four parameters, we presented in the first paragraph.

Unlike the original study, we were not able to confirm, at least, most of our hypotheses. While they find that core developers have a shorter first-feedback interval than peripheral developers, we have to reject this assumption. In fact, we found that for JAILHOUSE, there are strong indications that the exact opposite of the assumption is true. The next parameter, regarding OSS contributions, we investigated is the review interval. We found that there is no significant difference in the review intervals of core and peripheral developers. Again this disagrees with the original study. As for code acceptance rate, we found our results to be overall inconclusive. But since

more than half of our results suggest that core developers have a higher acceptance rate than peripheral developers, we do not refuse this hypothesis, either. The original study accepts this hypothesis. For the last parameter we did not find a notable difference in the numbers of patch revisions of core and peripheral developers, so we reject the assumption that core developers need less patch revisions than peripheral ones. Bosu et al. find their results, regarding this hypothesis, to be inconclusive overall.

In conclusion, we could not confirm the findings of the original study. Our motivation for this study was to find out in what way developers can influence the outcomes of their contributions to OSS projects, regarding their role within the project. Since we did not find significant results for this, we can only conclude that the developer role has little to no influence on contributions to the three projects, we used in our thesis.

6.2 Future Work

To improve and extend the study in the future, it would be valuable to include more network metrics for the classification of developers. It would also be good to find some kind of combination metric, to not having to work with multiple classifications, but one overall classification. Moreover, we should analyze more OSS projects, with maybe even different contribution tools. This would help us determine the validity of our study and show whether or not it is generally applicable to OSS projects.

Another good idea would be to include all mailing-lists of a project into the analysis, not only the main development list. This could lead to a more complete detection of patches, and therefor more solid results.

The last thing we need to do in the future is to determine whether the few results that indicate the opposite of our hypotheses, show statistical significance for this. For that reason we have to repeat the analysis but switch the alternative hypotheses of the Wilcoxon Mann-Whitney tests from greater to less and vice versa.

A Appendix

In the Appendix we provide all violin plots that we do not present in the main thesis.

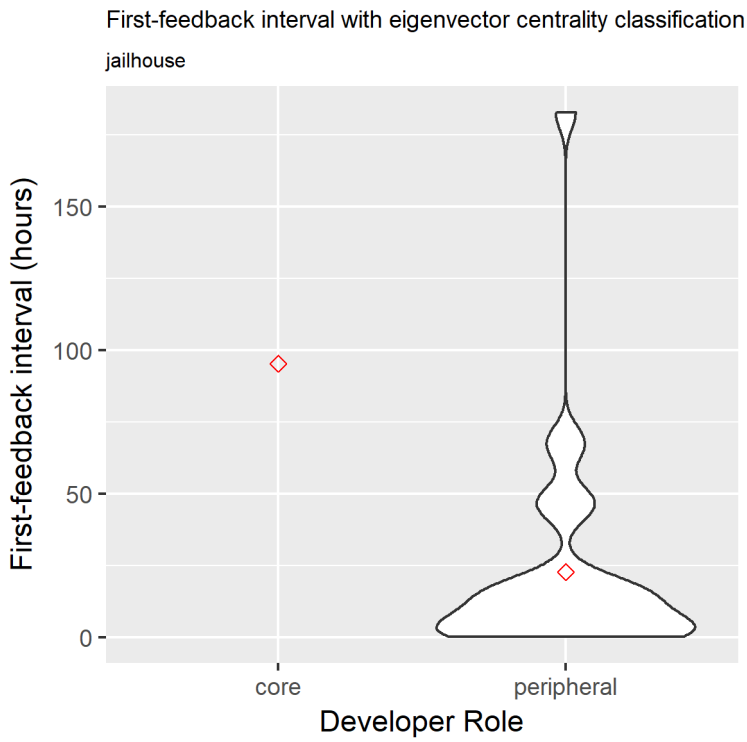


Figure A.1: Violin plots for the first-feedback intervals of core and peripheral developers in the JAILHOUSE project. The classification metric is eigenvector centrality.

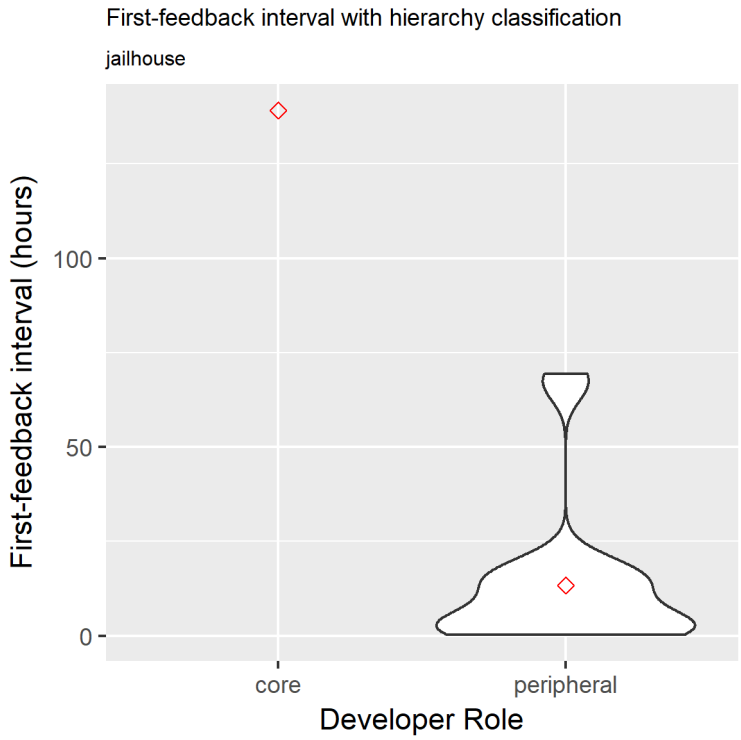


Figure A.2: Violin plots for the first-feedback intervals of core and peripheral developers in the JAILHOUSE project. The classification metric is hierarchy.

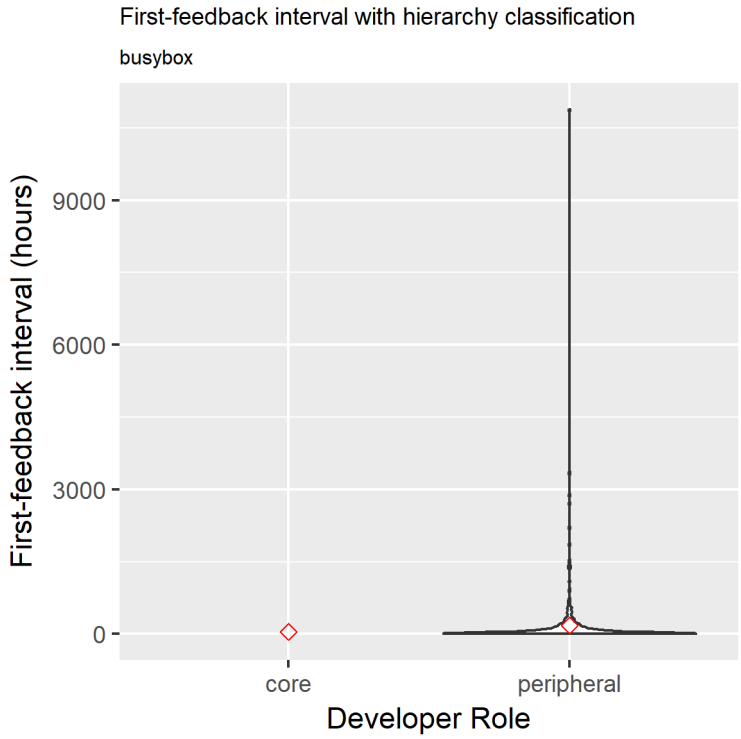


Figure A.3: Violin plots for the first-feedback intervals of core and peripheral developers in the BUSYBOX project. The classification metric is hierarchy.

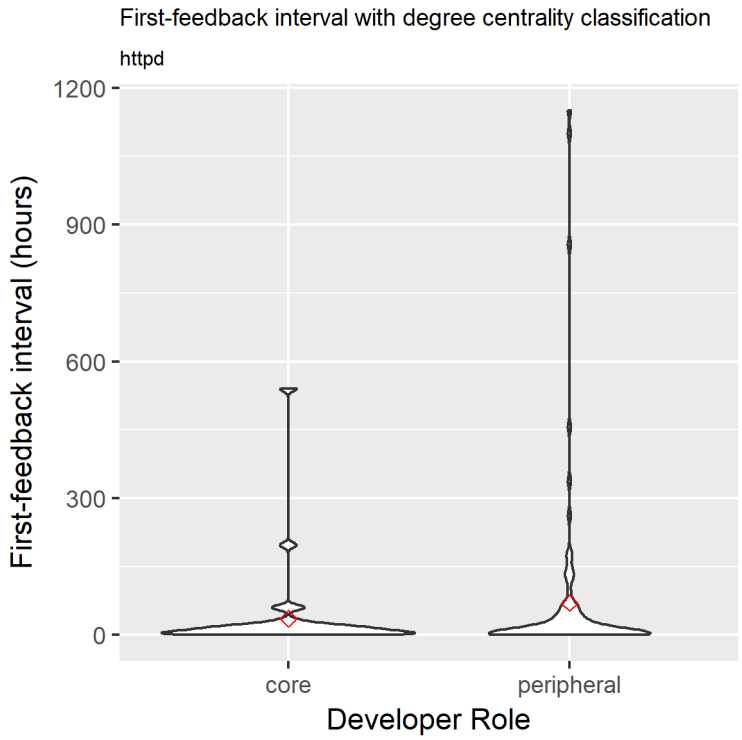


Figure A.4: Violin plots for the first-feedback intervals of core and peripheral developers in the HTTPD project. The classification metric is degree centrality.

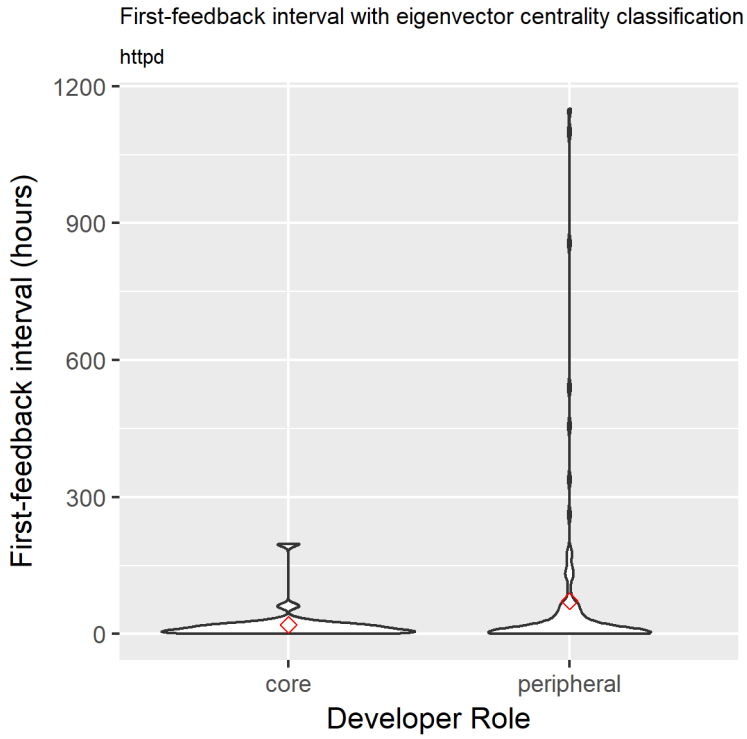


Figure A.5: Violin plots for the first-feedback intervals of core and peripheral developers in the HTTPD project. The classification metric is eigenvector centrality.

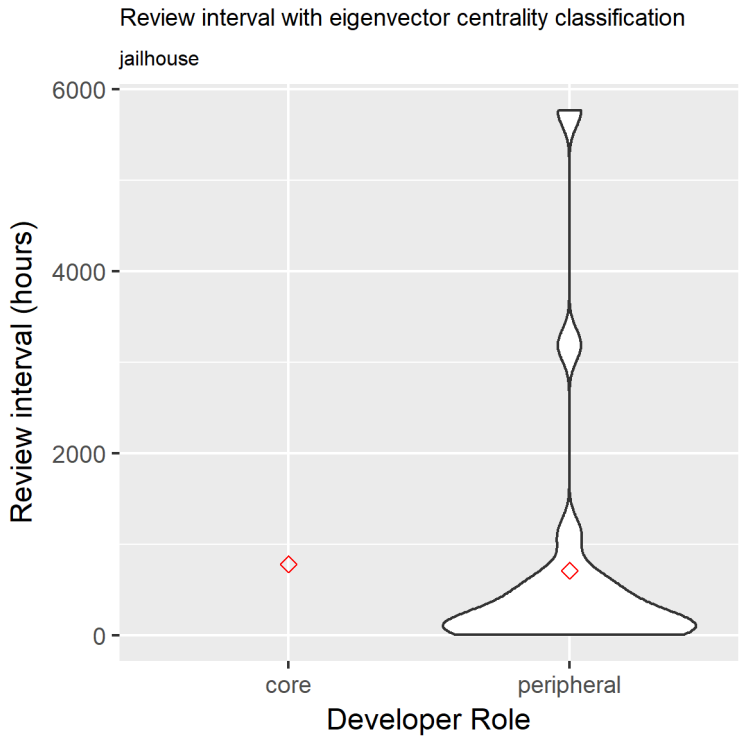


Figure A.6: Violin plots for the review intervals of core and peripheral developers in the JAILHOUSE project. The classification metric is eigenvector centrality.

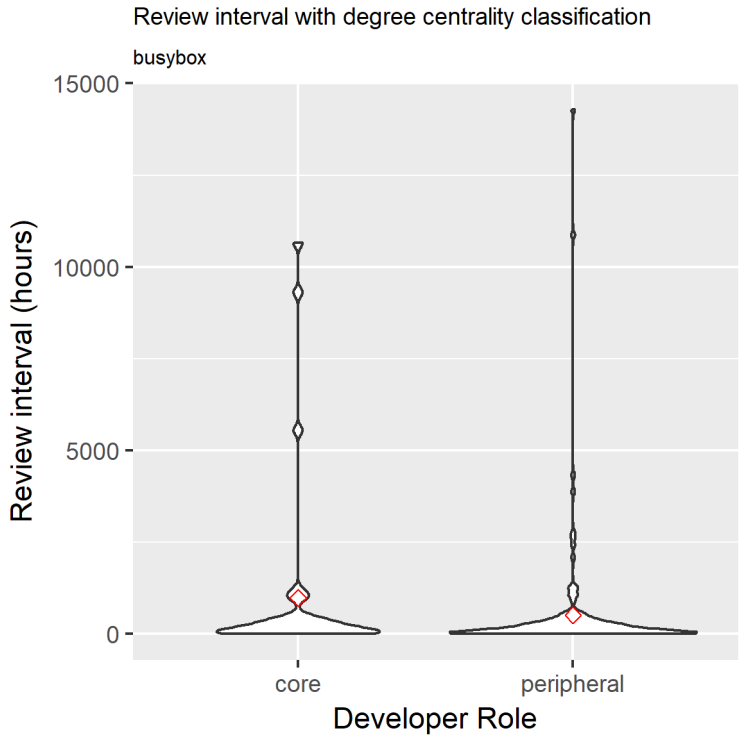


Figure A.7: Violin plots for the review intervals of core and peripheral developers in the BUSYBOX project. The classification metric is degree centrality.

Figure A.8: Violin plots for the review intervals of core and peripheral developers in the HTTPD project. The classification metric is degree centrality.

Figure A.9: Violin plots for the code-acceptance rates of core and peripheral developers in the JAILHOUSE project. The classification metric is eigenvector centrality.

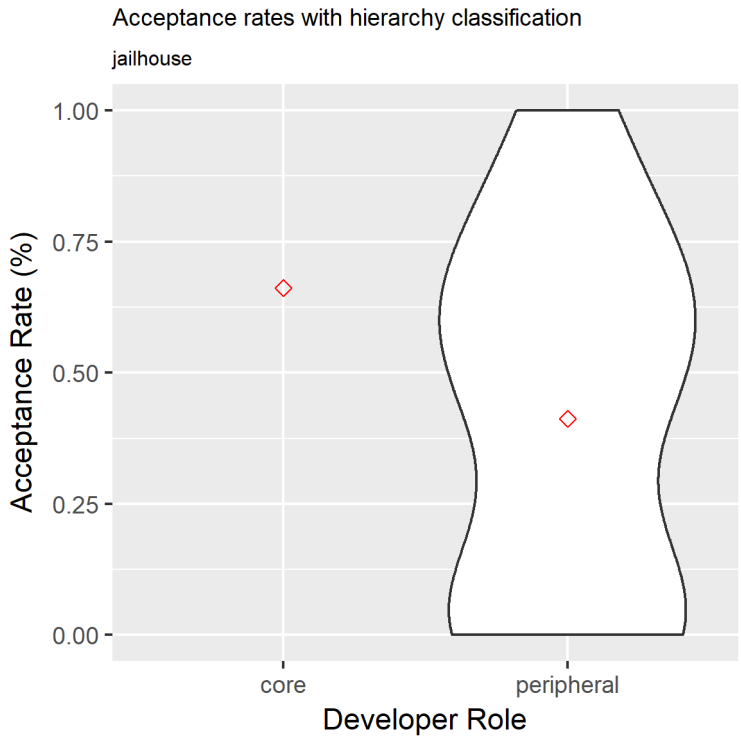


Figure A.10: Violin plots for the code-acceptance rates of core and peripheral developers in the JAILHOUSE project. The classification metric is hierarchy.

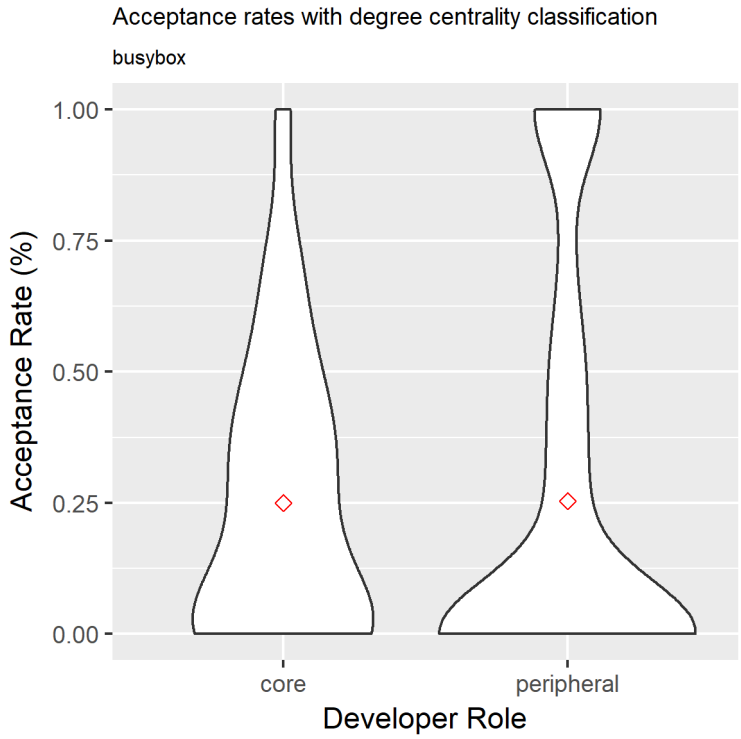


Figure A.11: Violin plots for the code-acceptance rates of core and peripheral developers in the BUSYBOX project. The classification metric is degree centrality.

Figure A.12: Violin plots for the code-acceptance rates of core and peripheral developers in the BusyBox project. The classification metric is hierarchy.

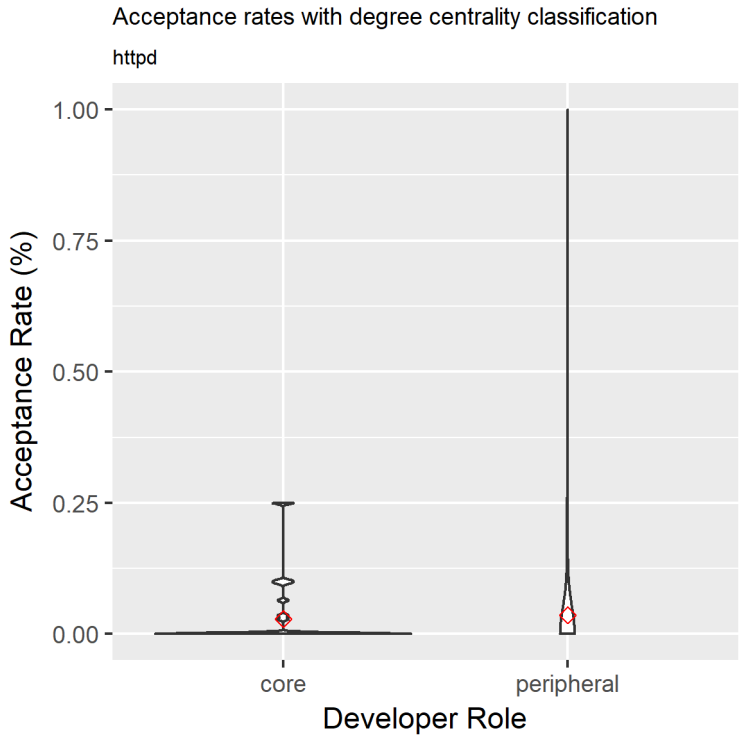


Figure A.13: Violin plots for the code-acceptance rates of core and peripheral developers in the httpd project. The classification metric is degree centrality.

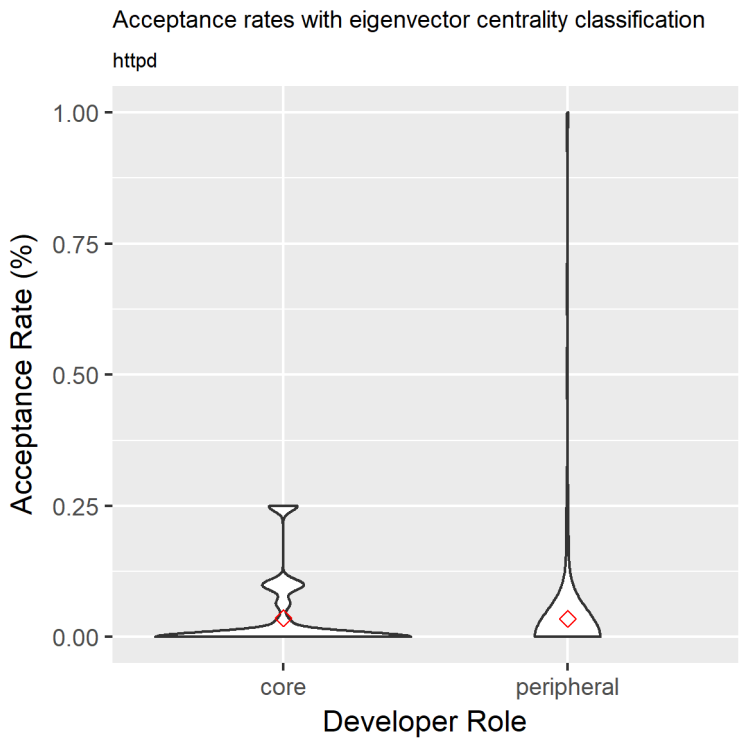


Figure A.14: Violin plots for the code-acceptance rates of core and peripheral developers in the HTTPD project. The classification metric is eigenvector centrality.

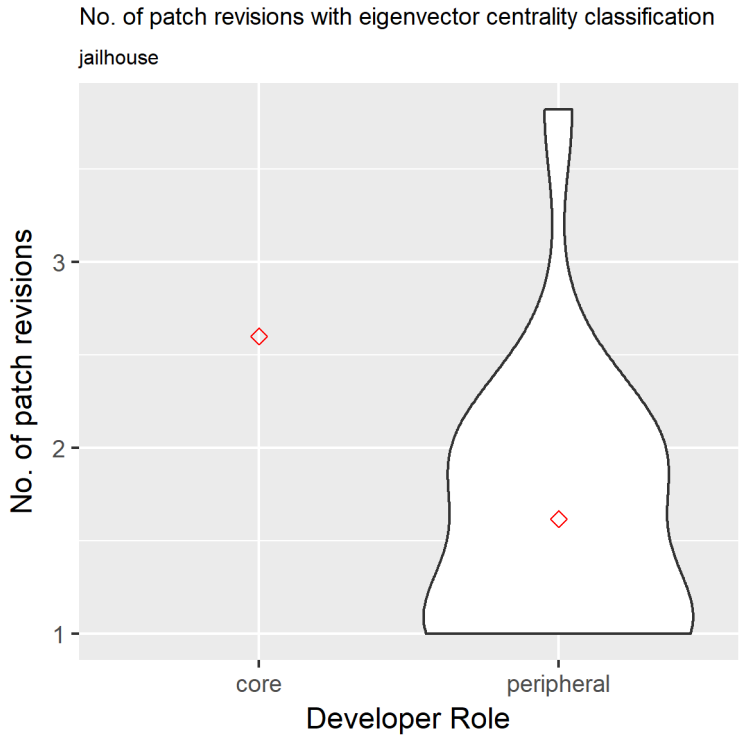


Figure A.15: Violin plots for the number of patch revisions of core and peripheral developers in the JAILHOUSE project. The classification metric is eigenvector centrality.

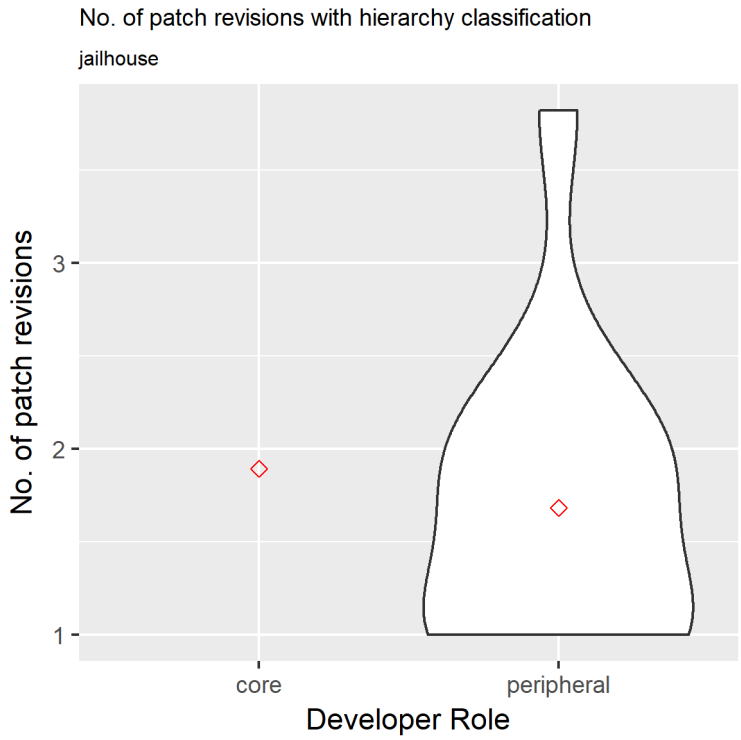


Figure A.16: Violin plots for the number of patch revisions of core and peripheral developers in the JAILHOUSE project. The classification metric is hierarchy.

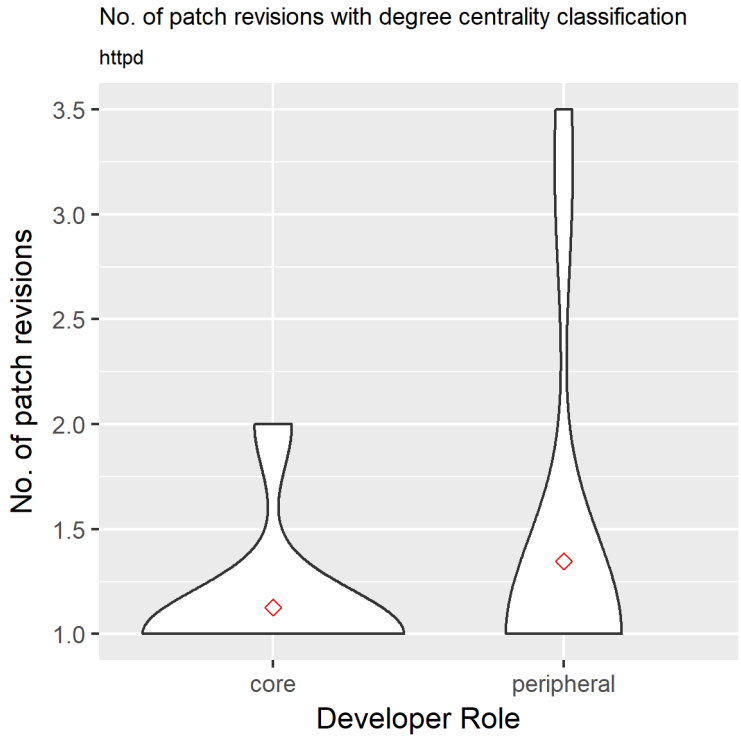


Figure A.17: Violin plots for the number of patch revisions of core and peripheral developers in the HTTPD project. The classification metric is degree centrality.

Bibliography

[AJ07] Jai Asundi and Rajiv Jayant. Patch review processes in open source software development communities: A comparative case study. In *Proceedings of the Annual Hawaii International Conference on System Sciences*, HICSS, page 166c. IEEE Computer Society, 2007. (cited on Page 3 and 4)

[BC14] Amiangshu Bosu and Jeffrey C. Carver. Impact of developer reputation on code review outcomes in oss projects: An empirical investigation. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM, pages 33:1–33:10. ACM, 2014. (cited on Page iii, 1, 8, 11, 12, 13, 14, 15, 16, 19, 35, 36, 37, 39, 40, and 41)

[BE05] Ulrik Brandes and Thomas Erlebach. *Network Analysis: Methodological Foundations*. Springer, 2005. (cited on Page 5, 6, and 7)

[BLM+06] S. Boccaletti, V. Latora, Y. Moreno, M. Chavez, and D-U. Hwang. Complex networks: Structure and dynamics. *Physics reports*, 424(4):175–308, 2006. (cited on Page 7)

[CWLH06] Kevin Crowston, Kangning Wei, Qing Li, and James Howison. Core and periphery in free/libre and open source software team communications. In *Proceedings of the Annual Hawaii International Conference on System Sciences*, HICSS, pages 118–127. IEEE Computer Society, 2006. (cited on Page 4, 5, and 8)

[JAG13] Yujuan Jiang, Bram Adams, and Daniel M. German. Will my patch make it? and how fast?: Case study on the linux kernel. In *Proceedings of the Working Conference on Mining Software Repositories*, MSR, pages 101–110. IEEE, 2013. (cited on Page 8)

[JAHM17] Mitchell Joblin, Sven Apel, Claus Hunsen, and Wolfgang Mauerer. Classifying developers into core and peripheral: An empirical study on count and network metrics. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 164–174. IEEE, 2017. (cited on Page 6, 7, 9, 19, and 24)

[JAM17] Mitchell Joblin, Sven Apel, and Wolfgang Mauerer. Evolutionary trends of developer coordination: A network approach. *Empirical Software Engineering*, 22(4):2050–2094, 2017. (cited on Page 7)

[JMA+15] Mitchell Joblin, Wolfgang Mauerer, Sven Apel, Janet Siegmund, and Dirk Riehle. From developer networks to verified communities: A fine-grained approach. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 563–573. IEEE Computer Society, 2015. (cited on Page 17)

[Job17] Mitchell Joblin. *Structural and Evolutionary Analysis of Developer Networks*. PhD thesis, Universität Passau, Germany, 2017. (cited on Page 5, 6, 7, and 8)

[LK77] J. Richard Landis and Gary G. Koch. The measurement of observer agreement for categorical data. *Biometrics*, 33(1):159–174, 1977. (cited on Page 19)

[LLFH09] J. M. Gonzalez-Barahona L. López-Fernández, G. Robles and I. Herraiz. Applying social network analysis techniques to community-driven libre software projects. *Integrated Approaches in Information Technology and Web Engineering: Advancing Organizational Knowledge Sharing*, 1:28–50, 2009. (cited on Page 5)

[LW05] K.R. Lakhani and R.G. Wolf. Why hackers do what they do: Understanding motivation and effort in free/open source software projects. *Perspectives on Free and Open Source Software*, 1:3–22, 2005. (cited on Page 1 and 11)

[Mac67] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of the Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. University of California Press, 1967. (cited on Page 13)

[MBR13] Murtuza Mukadam, Christian Bird, and Peter C. Rigby. Gerrit software code review data from android. In *Proceedings of the Working Conference on Mining Software Repositories*, MSR, pages 45–48. IEEE, 2013. (cited on Page 12)

[MFH02] Audris Mockus, Roy T. Fielding, and James D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002. (cited on Page 4)

[NYN+02] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. In *Proceedings of the International Workshop on Principles of Software Evolution*, IWPSE, pages 76–85. ACM, 2002. (cited on Page 4)

[RB03] Erzsébet Ravasz and Albert-László Barabási. Hierarchical organization in complex networks. *Physical Review E*, 67(2), 2003. (cited on Page 7)

[RG06] P. C. Rigby and D. M. German. A preliminary examination of code review processes in open source projects. Technical Report DCS-305-IR, University of Victoria, 2006. (cited on Page 9)

[RGS08] Peter C. Rigby, Daniel M. German, and Margaret-Anne Storey. Open source software peer review practices: A case study of the apache server. In *Proceedings of the International Conference on Software Engineering*, ICSE, pages 541–550. ACM, 2008. (cited on Page 12)

[RLM16] Ralf Ramsauer, Daniel Lohmann, and Wolfgang Mauerer. Observing custom software modifications: A quantitative approach of tracking the evolution of patch stacks. In *Proceedings of the International Symposium on Open Collaboration*, OpenSym, pages 4:1–4:4. ACM, 2016. (cited on Page 20)

[WND08] Peter Weißgerber, Daniel Neu, and Stephan Diehl. Small patches get in! In *Proceedings of the International Working Conference on Mining Software Repositories*, MSR, pages 67–76. ACM, 2008. (cited on Page 8)

[ZZM16] Jiaxin Zhu, Minghui Zhou, and Audris Mockus. Effectiveness of code contribution: From patch-based to pull-request-based tools. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE, pages 871–882. ACM, 2016. (cited on Page 3)

Eidesstattliche Erklärung:

Hiermit versichere ich an Eides statt, dass ich diese Bachelorarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Christian Hechtl

Passau, den 07. April 2018