Universidade Federal de Campina Grande
Centro de Engenharia Elétrica e Informática
Coordenação de Pós-Graduação em Ciência da Computação

Automatic Decomposition of Code Review Changesets in Open Source Software Projects

Victor da Cunha Luna Freire

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Ciência da Computação
Linha de Pesquisa: Engenharia de Software

Jorge Cesar Abrantes de Figueiredo e João Arthur Brunet Monteiro
(Orientadores)

Campina Grande, Paraíba, Brasil

Resumo

Revisão de código é uma atividade amplamente utilizada para garantia da qualidade de código-fonte. Como uma típica atividade de revisão, revisão de código depende fortemente da habilidade dos desenvolvedores de entenderem o código sendo revisado. Como consequência, a presença de conjuntos de mudanças grandes contendo várias modificações independentes (e.g. consertos de bugs, refatoramentos, adições de funcionalidades, etc) pode afetar negativamente a eficácia da revisão de código. Para lidar com esse problema, Barnett et al. desenvolveram ClusterChanges — uma técnica de análise estática leve para decompor conjuntos de mudanças em partições diferentes que podem ser revisadas independentemente. Eles descobriram que ClusterChanges pode, de fato, criar partições diferentes para modificações independentes dentro do mesmo conjunto de mudança e que desenvolvedores concordam com sua decomposição. Porém, a implementação dos autores de ClusterChanges não está disponível publicamente e eles restringiram sua análise da técnica a projetos de software que são: i) de código fechado, (ii) escritos em C# e iii) desenvolvidos por uma única organização. Portanto, ainda há evidência empírica limitada da aplicabilidade de ClusterChanges para contextos diferentes. Para resolver essa situação, nós criamos JClusterChanges, uma implementação livre e de código aberto (FOSS) de ClusterChanges para software Java, e replicamos o estudo original de Barnett et al. usando uma amostra de 1000 pull requests oriundos de projetos de código aberto escritos em Java e armazenados do GitHub. Ao fazer isso, nós descobrimos que conjuntos de mudanças de código aberto também podem ser grandes e conter várias modificações independentes semelhantemente aos changesets do estudo original. Assim, nossa pesquisa confirma que o problema é tanto real quanto relevante para outros ambientes além do da Microsoft, aumenta o corpo de conhecimento sobre esse assunto e prove uma implementação FOSS de ClusterChanges que mostra que é possível implementar a técnica para outros contextos e que pode ser usada para auxiliar pesquisa futura nesse assunto.

Abstract

Code review is a widely used activity for source code quality assurance. As a typical review activity, code review is strongly dependent on developers' ability to understand the code under revision. As a consequence, the presence of large changesets containing several independent modifications (e. g. bug fixes, refactoring, features additions etc) can negatively affect the efficacy of code review. To cope with this problem, Barnett et al. developed ClusterChanges — a lightweight static analysis technique for decomposing changesets in different partitions that can be reviewed independently. They have found that ClusterChanges can indeed create different partitions for independent changes within the same changeset and that developers agree with their decomposition. However, the authors' implementation of ClusterChanges is not publicly available and they restricted their analysis of the technique to software projects that are: i) closed source, ii) written in C# and iii) developed by a single organization. Therefore, there is still limited empirical evidence of ClusterChanges' applicability to different contexts. To address this situation, we created JClusterChanges, a free and open source implementation (FOSS) of ClusterChanges for Java software, and replicated the original Barnett et. al. study on a sample of 1000 pull requests from Java open source projects hosted on GitHub. By doing so, we found that open source changesets can also be large and contain several independent changes similarly to the changesets of the original study. Thus, our research confirms that the problem is both real and relevant to other environments besides Microsoft's, increases the body of knowledge about this subject and provides a FOSS implementation of ClusterChanges that shows that it is possible to implement the technique for other contexts and can be used to help future research on this subject.

Agradecimentos

Este trabalho não seria possível sem a ajuda de inúmeras pessoas. Gostaria de agradecer:

- Aos meus pais e meu irmão,
- Aos demais parentes,
- Aos meus orientadores, Jorge Abrantes e João Arthur,
- A todos os professores que contribuiram com meu mestrado, especialmente: Dalton Serey, Andrey Brito, Joseana Fechine, Tiago Massoni, Franklin Ramalho, Jacques Sauvé,
- Aos amigos: Diego Pedro, Andrey, Taciano, Laércio, Gabriel, Catharine, Arthur, Alysson, Aislan, Carlos Artur, Amilton, Saulo, Jeanderson, Katyusco, Caio, ...
- A todos os membros da banca avaliadora,
- Aos funcionários da COPIN e do SPLAB, especialmente, Lilian,
- A CAPES pelo auxílio financeiro,
- A todos que me ajudaram nessa trajetória e que ainda não foram citados.

Obrigado!

Contents

List of Figures

List of Tables

List of Source Code

Chapter 1

Introduction

1.1 Problem

One of the main goals of Software Engineering (SE) is to create high-quality software. In order to achieve it, software engineers need to perform quality assurance activities during software development [33].

Code review is one of the most important activities for improving software quality. In 1976, Fagan published the first code review process along with evidence that it had increased developer productivity and software quality at IBM [15]. Fagan's code review process and similar ones are known as inspections and, since their inception, researchers have been providing evidence of their efficacy. However, despite all this evidence, practitioners tend to not use inspections because of how costly and time-consuming they are [11][24][35].

Because of the drawbacks of formal inspections, developers have been increasingly applying lightweight code review processes known as "modern code review" (MCR) [1]. MCR usually consists of reviewing changesets in a distributed and asynchronous manner with the assistance of specialized tools [17][19][32]. A changeset is a set of code changes that a developer groups together and expects others to review. Even though these code review processes require less time than Fagan's inspections, they still positively affect software quality [1][4][5].

Modern code review is not trivial to perform. In particular, understanding the code under review is the main challenge faced by developers [1]. The more independent modifications contained in a changeset, the harder it is to understand what has been changed and, conse-

quently, review it [36]. For example, consider a changeset that contains not only modifications to refactor a module but also other modifications to add a new feature (Figure 1.1). This type of changeset has been called tangled changes [25] and composite changes [36] in previous research. In this dissertation, we[1] refer to them as composite changesets.

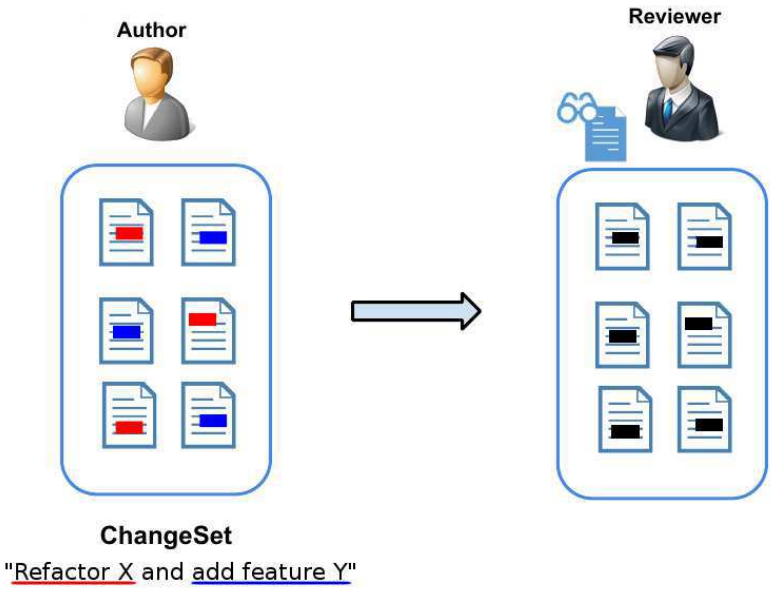

Figure 1.1: A changeset containing multiple independent modifications is difficult for another developer to review. Figure adapted from Barnett et al.'s study [2].

Composite changesets are not rare. Herzig and Zeller manually analyzed 7,000 changesets extracted from bug reports of 5 Java Open Source Software (OSS) projects and they estimated that up to 16% of these changesets are composite [25]. Similarly, Tao et al. manually analyzed 453 changesets from 4 different Java OSS and found that 17% of them were composite changesets [37].

One might wonder that developers would benefit from the existence of a tool able to separate unrelated changes. In fact, previous research indicates that developers would like an automatic partitioning tool not only to help them understand the composite changesets under review [2][14][36] but also to improve the accuracy of software repository mining programs [25].

[1] Although there is only one author, *we* is used instead of *I* to keep the style similar to conference and journal papers. Also, the first-person is used throughout this text to denote actions or decisions taken by the author of this work.

With this problem in mind, Barnett et al. [2] developed a static analysis technique for automatically partitioning changesets, which they named ClusterChanges (Figure 1.2).

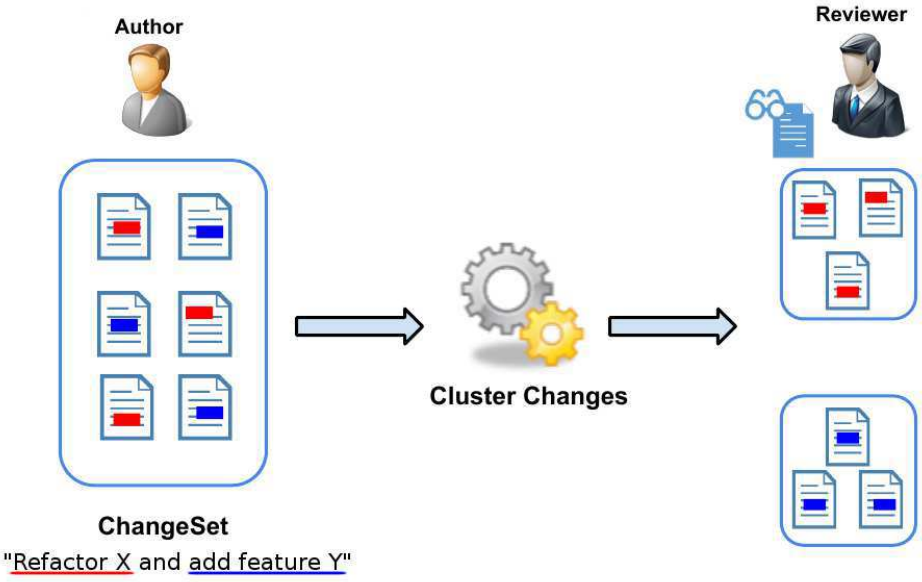

Figure 1.2: ClusterChanges can automatically partition a changeset to improve understanding. Figure adapted from Barnett et al.'s study [2].

Barnett et al. evaluated their technique and obtained promising results. First, they quantitatively analyzed it using 1000 changesets from Microsoft software projects written in C# that were submitted for code review. As a remarkable result, they found that, although ClusterChanges failed to detect some relationships between changes, it never marked two unrelated changes as being related, i.e. there were no false-positives. Secondly, they interviewed 20 Microsoft software developers. Out of the 20 developers who participated in the study, 16 agreed with the decomposition generated by ClusterChanges and all 20 developers believe that the tool could be useful for understanding changesets being reviewed [2].

Notwithstanding the excellent results, Barnett et al. [2] pointed out that their study suffers from the following threats to validity: (1) the changeset sample only contains C# code and (2) the changesets are from a single organization. In addition, we also believe that a noteworthy threat to external validity is that the analysis was restricted to closed source code, since OSS development practices tend to be significantly different [30][34].

1.2 Solution

Considering the aforementioned threats to validity, we replicated the quantitative substudy of the original study from Barnett et al. [2] using changesets from OSS projects written in Java by several organizations. More specifically, we (1) created JClusterChanges — a free and open source (FOSS) implementation of ClusterChanges for Java code changesets and (2) applied JClusterChanges to a sample of 1000 pull requests from the 10 most popular Java OSS projects hosted on GitHub.

Our replication consisted of following the original study as close as possible except that we deliberately changed the context to better understand how widely applicable the results were, i.e., to address the threats to external validity from the original study. According to Gómez et al. classification of replications [23], our study is a changed-populations/-experimenters replication.

Replication is an important mechanism for assuring the reliability of scientific knowledge, due to the fact that designing and executing experiments is a complex and error-prone activity. By means of this work, we increase the body of knowledge in Software Engineering and provide stronger evidence for previous work.

1.3 Main Contributions

We performed a quantitatively analysis on a sample of pull requests and we obtained similar results to the original study. Although small and independent changesets are demanded in OSS projects [34], we observed that OSS changesets are often composite like the closed source changesets from Microsoft analyzed in the original study. Therefore, we found evidence that ClusterChanges is also valuable for OSS developers.

The main contributions of this work are:

- Evidence that the problem also exists in OSS projects, i.e. composite changesets are common in OSS projects.
- Evidence that ClusterChanges is effective in other contexts. Our results suggest that ClusterChanges can assist developers working in any software organization using any modern code review process.

- JClusterChanges: a free and open source implementation of the ClusterChanges technique that can partition changesets written in the Java programming language. The tool is licensed under the GNU Affero General Public License version 3 [16] and is publicly available at: `https://sites.google.com/site/jclusterchanges/`
- The results of this study will be submitted as a paper to a conference.

1.4 Outline

This thesis is structured as follows: Section 2 provides the necessary background for understanding this work, Section 3 provides a formal description of the ClusterChanges technique, Section 4 describes our implementation of ClusterChanges, namely JClusterChanges, Section 5 presents the replication and its results, Section 6 gives an overview of related work on partitioning changesets and Section 7 concludes this work by outlining the main results and possible venues for future work.

Chapter 2

Background

This chapter contains the necessary background information for understanding this work. First, we provide an overview of code review, since this work is about a technique for improving its efficacy. Then, we describe pull-based software development to better understand the context of the replication. Finally, we explain the concepts and terminology of replication, which are necessary to perceive how our work increases the body of knowledge in Software Engineering.

2.1 Modern Code Review

Intuitively, we expect a developer to fix an error more easily if it is in code that he has been working on recently rather than code he wrote months ago. As a matter of fact, researchers have observed that the relative cost of fixing errors increases with each step of the software process (Figure 2.1). Thus, identifying errors as early as possible reduces the total cost of software development [15][33].

With this in mind and in order to improve software quality, Michael Fagan devised a formal review process where a group of developers carefully analyzes a software artifact over the course of several meetings in order to find errors. Although this process can be applied not only to code but also to other artifacts, e.g. test plan, architecture specification and requirements specification [15], we are interested here solely in its application as a code review process.

Fagan's inspection, as his code review process is commonly known, is ideally conducted

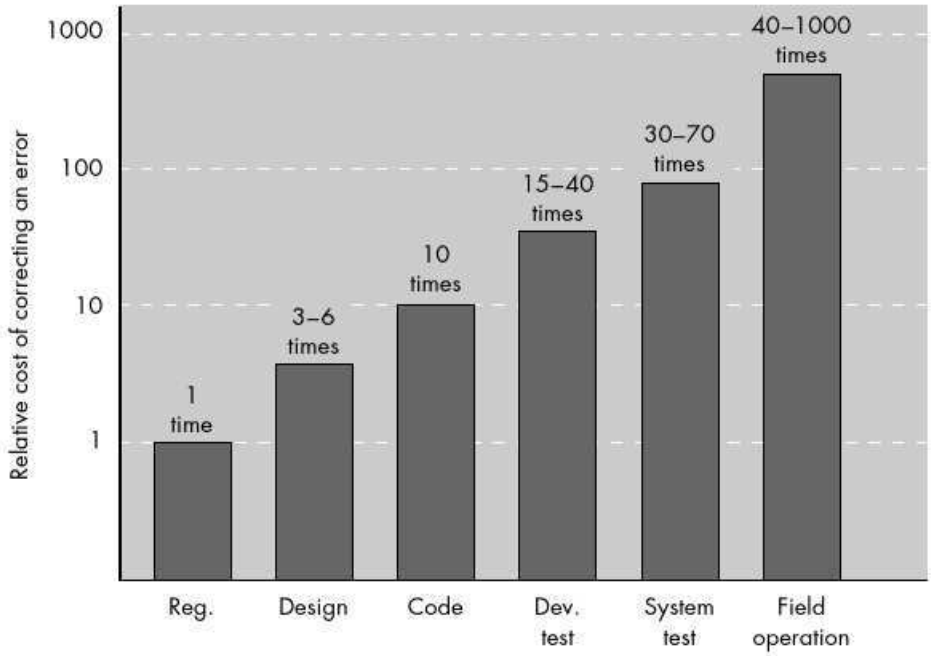

Figure 2.1: Relative cost of correcting an error increases. Figure extracted from Pressman's book on Software Engineering [33].

by a team of about four people where each one performs a different role. In practice, however, a person may fulfill multiple roles. The team must have a (1) moderator, someone who leads and guides the team, ensuring that the review process is done correctly, a (2) designer, the developer who designed the program, a (3) coder, the developer who implemented the program and a (4) tester, the developer in charge of creating the tests for the code under review [15].

After the team is established, they will perform the review process that consists of five steps (Figure 2.2) [15]:

1. Overview: the designer briefly explains to the team what piece of code will be reviewed and distributes all the relevant material such as the code itself

2. Preparation: each team member individually studies the material to review before the actual inspection meeting

3. Inspection: the inspection meeting itself where each line of code is analyzed at least once by the group with the goal of finding defects.

4. Rework: the designer and/or coder fix the defects found during the inspection meeting

5. Follow-up: the inspection team meets again and analyzes each of the fixes generated in the previous step to evaluate their quality. If a defect was not fixed or a new defect has been introduced, the process goes back to the rework step.

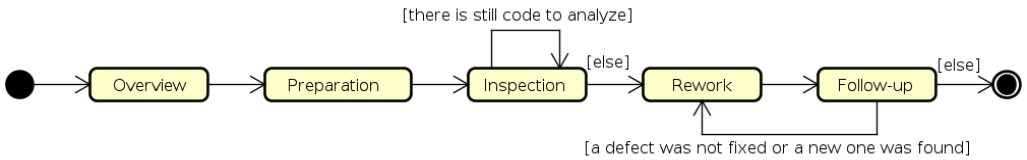

Figure 2.2: Workflow of Fagan's inspection.

Several research studies have found that Fagan's inspection and similar formal processes, which are collectively called software inspections, are effective. Most of these were case studies where software inspections were applied at a company. Researchers observed that software inspections tend to lead to cost savings and that they have high return on investment rates, usually of at least 30 [28].

Nevertheless, adoption of software inspections has not been as widespread as expected [35]. In general, companies are reluctant to adopt inspections because they perceive them as costly and time-consuming [11][24]. They also cite lack of training as a cause [11][24], which is not surprising given the complexity of software inspections.

Because of these issues with software inspections, organizations ended up devising simpler review processes known as MCR. MCR processes differ from software inspections by being lightweight, informal and tool-based [1]. They are being used extensively by not only OSS projects [34] but also big companies such as Google [20] and Microsoft [29].

The workflow of MCR revolves around a changeset (Figure 2.3). In order to accomplish development tasks such as adding a feature or fixing a bug, a developer performs a number of code changes, which are collectively called a changeset. Using a MCR tool, the developer shares this changeset with others who will act as reviewers. The reviewers assigned for a changeset are usually members of the development team and who are either chosen manually by the author of the changeset or automatically picked by the MCR tool.

The reviewers act as a quality gateway for the main codebase. Once a developer is assigned as a reviewer for a changeset, he will have to analyze it and make a decision:

- Vote for acceptance. If the reviewer does not find any defects and considers the changeset to meet the project and organization standards, he will vote for it to be accepted. The final decision is negotiated between all the reviewers assigned for that changeset. Accepted changesets are usually merged with the main codebase.

- Ask the author for clarifications and modifications. If the reviewer identifies a problem in the changeset or wants the author to explain a design decision for instance, he will create a comment associated with the relevant lines of code, where he will ask the author to do something. Once the author complies with all the requests, the reviewer will either vote for acceptance or rejection as explained in options 1 and 3.

- Vote for rejection. This usually happens If no amount of rework by the author will fix the changeset, as in the case of an unwanted feature for example.

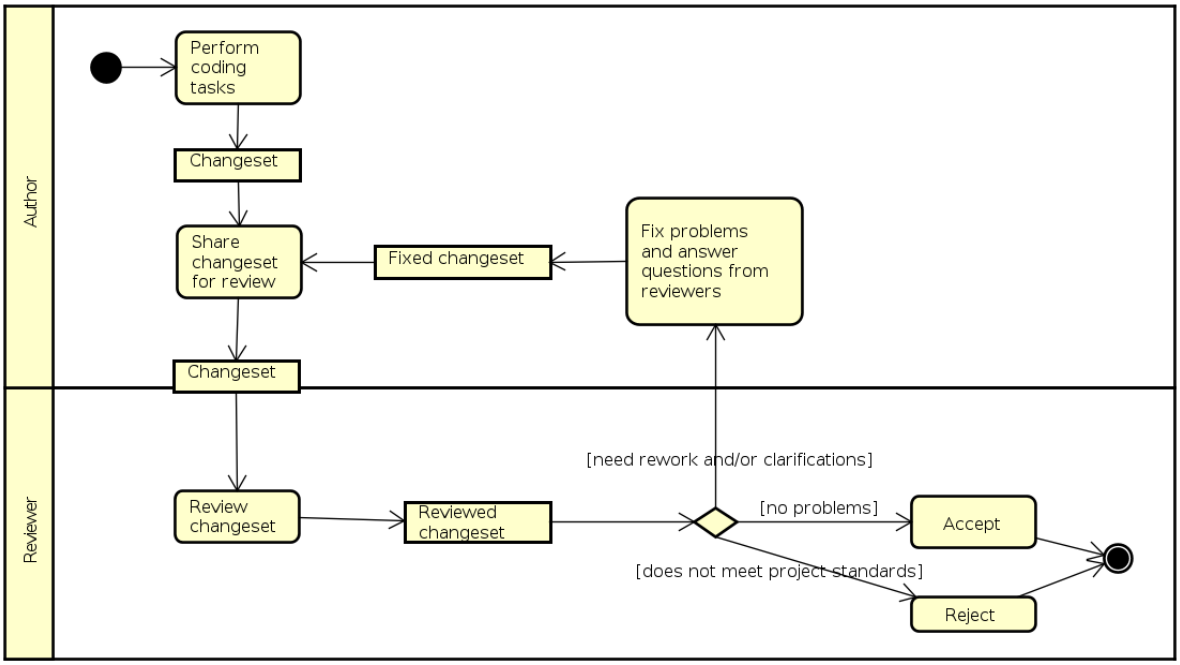

Figure 2.3: Workflow of modern code review.

While each organization and tool implements the process of MCR differently, the core workflow is essentially the same as described above.

Bachelli and Bird found that "code review is understanding". Developers consistently cited understanding as the main challenge of MCR and claimed that understanding is crucial

for finding defects during review. Understanding can be split into two categories: context understanding and change understanding. Context understanding is about understanding the code to be reviewed and change understanding consists of identifying what has been changed and why [1].

2.2 Pull-based software development

Pull-based software development is a distributed software development process that is popular among OSS projects, because it allows a team of core developers and outside contributors from different locations and timezones to collaborate effortlessly [22].

The fundamental concept of pull-based software development is a pull request. A developer who wants to contribute to the software project starts by forking the main project repository. Then, he performs code changes in his local repository. When the developer is satisfied with his work, he creates a request in a specialized tool where he describes what he has done and asks that his changeset be merged into the main project repository. This request is called a pull request because the developer is asking the project maintainers to pull his changes into the main repository.

MCR is a fundamental activity within pull-based software development. Ideally, all code intended to be merged with the main repository must be submitted as pull requests. In addition, before a pull request is merged, it must be positively reviewed by one or more maintainers – a core developer in charge of handling pull requests.

GitHub is the largest code host in the world with over 38 million projects and a large amount of those are OSS projects [18]. In 2013, over 100,000 OSS repositories used pull-based software development [22]. Therefore, there is a vast amount of data for researchers to investigate.

GitHub provides its own integrated pull request system (Figure 2.4). The system lets maintainers, core developers and outside contributors interact with the author of the pull request via a discussion thread and code comments [19]. When the maintainers are satisfied with the pull request, they can easily accept and merge it into the main repository. Alternatively, they can reject the pull request.

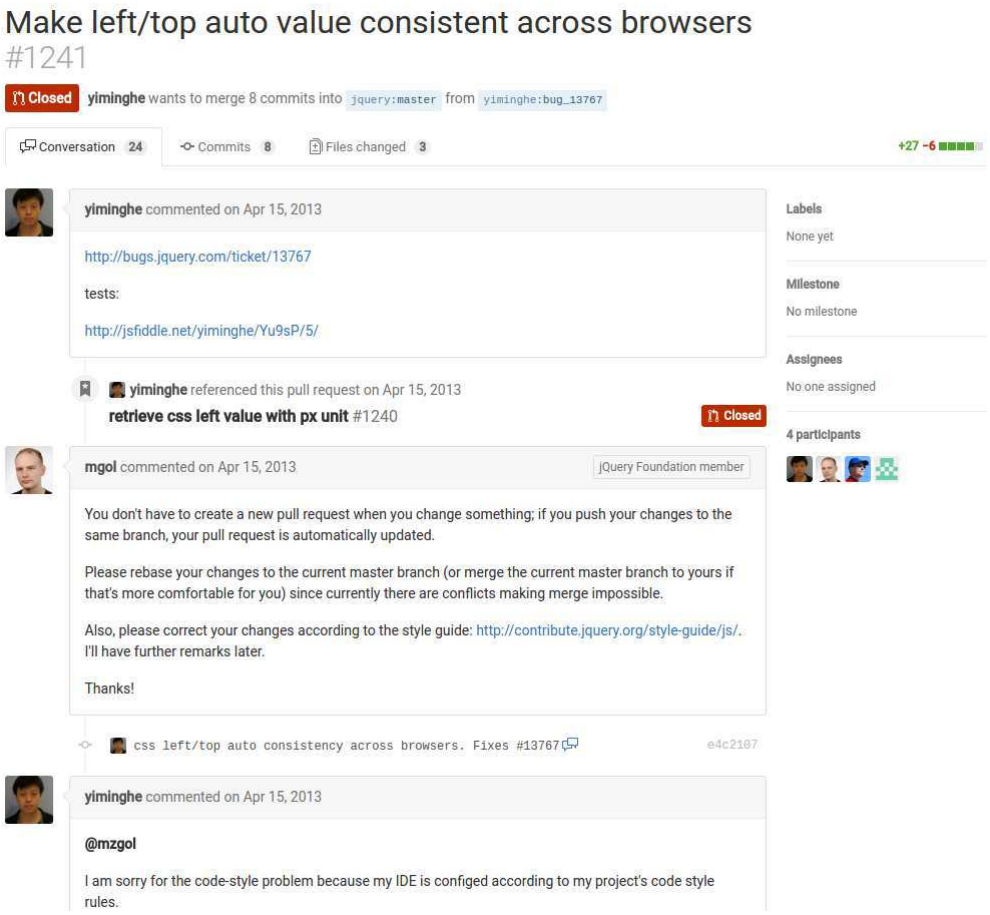

Figure 2.4: An example of a pull request from GitHub. The pull request's author discusses the changeset that he would like to be merged with a maintainer.

2.3 Replication

Replication is a fundamental activity in science because designing and executing experiments is complex and error-prone. In particular for Software Engineering, experimentation is even harder because the high variability in human skill and performance can drastically affect the results of a study [3][8]. In addition, we cannot generalize the results of an isolated study to other contexts because of the many sources of variation between different contexts [3][8].

More replications need to be done in SE. Of the thousands of SE papers published between 1994 and 2010, only 96 reported replications. In addition, the growth rate of replications in SE has been smaller than the growth rate of empirical studies during the same period [12].

Replication papers are inconsistent in their description of the original studies. They often do not provide enough information about the original study to allow their results to be compared [12][9]. For this reason, Carver proposed a set of guidelines for publishing replications [9]. Subsequently, Carver et al. [10] found evidence that other researchers deemed the guidelines useful. We followed their guidelines when writing this work and found them helpful as well.

2.3.1 Gómez et al. classification of SE replications

Gómez et al. [23] observed that there is no consensus among SE researchers on what a replication is and that previously proposed classifications have a number of shortcomings. Thus, they created a classification for replications in SE with the goal of addressing those shortcomings. More specifically, their classification aims to "(1) clearly define the bounds of a replication and the admissible changes; (2) exhaustively state all possible changes to the original experiment; (3) use self-explanatory terms; and (4) associate each replication type with its verification purpose(s)" [23].

In Gómez's classification [23], experiments have four dimensions that can be changed in a replication:

- Operational: consists of the operational definitions of the cause and effect constructs. For example, in this work, the cause construct is the ClusterChanges technique and its operationalization is JClusterChanges and how it is applied in the experiment.

- Protocol: the materials and instruments used. This dimension includes the experimental design, measuring instruments and data analysis techniques.

- Population: the subjects and/or objects used in the experiment. In this work, for instance, the objects are the pull requests to which JClusterChanges is applied.

- Experimenters: roles performed in the experiment. If the same experimenters participate in the replication, but they perform different roles, the replication is considered to have changed this dimension. Examples of roles are the data analyst and the experimental designer.

Each of the aforementioned dimensions when changed allows the replication to account for different threats to validity of the original study (Table 2.1) [23].

Dimension changed	Threats to validity addressed
None	Conclusion validity
Operational	Construct validity
Protocol	Internal validity
Population	External validity
Experimenters	Results are independent of experimenters
All	Validity of hypotheses

Table 2.1: Threats to validity addressed when a dimension is changed [23].

Since all dimensions can be changed in a replication, the question arises as to what the minimum requirements for a study to still be considered a replication are. Gómez et al. proposes that, for a study to be considered a replication, it must execute an experiment and keep some of the hypotheses of the original study, more specifically, "at least two treatments and one response variable need to be shared" [23].

Finally, a replication is named according to the dimensions that were changed except when no dimensions were changed or all of them were changed. In those cases, they are called repetition and reproduction, respectively. For example, a replication where the protocol and population were altered is called a changed-protocol/-population replication.

Chapter 3

ClusterChanges

This chapter contains a summary of the original study by Barnett et al. [2]. First, we provide a formal description of the ClusterChanges technique for automatically partitioning a changeset. Then, we describe how the researchers evaluated ClusterChanges and summarize their findings.

3.1 Formal Description

3.1.1 The Problem

Given a changeset containing various diff-regions that are possibly the result of different tasks, we would like to find a partitioning of those diff-regions such that: (1) each subset contains only diff-regions which are related to one another and (2) if a diff-region A is related to a diff-region B, then both A and B are in the same subset. In other words, we want to find a partitioning where each subset can be reviewed independently from the other subset. Note that the input does not contain the whole software source code, but only the files that have been modified.

In the next sections, we describe each aspect of ClusterChanges in both natural language and mathematical notation.

3.1.2 Definitions

Changeset

A changeset is a set of pair of files where each pair contains the original version of the file (before-file) and the modified version of the file (after-file).

Diff-region

A diff-region is a contiguous set of source code lines that have been added or changed. Since only the after-files of a changeset are analyzed, lines that were deleted are not considered. Diff-regions that cross type and method boundaries are split, i.e. no diff-region spans more than one type or method.

- F: set of diff-regions in the changeset

Span

A span is a sequence of characters in the source code.

- $span : (D \cup U \cup F) \rightarrow \{x_n\}_{n \in \mathbb{N}}$: a sequence of characters in the source code. Each character in the source code is uniquely identified by a natural number equal to its position in the document. The first character in the document is represented by 0, the second by 1 and so forth.

Definition

A definition is a group of statements that introduces an entity into a program and provides an identifier which can be used to refer to this entity. Such entities can be methods, fields and types for instance. In the Java Language Specification, this is known as a declaration [21].

- D: set of definitions in the changeset
- $defs(f) = \{d | d \in D \wedge span(f) \cap span(d) \neq \emptyset\} \quad f \in F$: set of definitions inside a diff-region

Use

A use is a reference to a definition by its identifier.

- U: set of uses in the changeset
- $Def : U \rightarrow D$: function that returns the definition associated with a use. If the associated definition of a certain use is not present in the changeset, the function is undefined for that use.
- $uses(f) = \{u | u \in U \wedge span(f) \cap span(u) \neq \emptyset\} \quad f \in F$: set of uses inside a diff-region

3.1.3 Relationships between Diff-regions

In this section, we describe how ClusterChanges infers if two diff-regions are related. Briefly, the gist of ClusterChanges is that the relationships between diff-regions can be determined by analyzing if any of their definitions and uses matches.

Def-use

If a diff-region f_1 has a definition d and a diff-region f_2 has a use u whose associated definition is d, then we say that there is a def-use relationship between diff-regions f_1 and f_2 (Figure 3.1).

- $defUsesInDiffs = \{(d, u) \quad | \quad d \in D \wedge u \in U \wedge \quad \exists f_1, f_2 \in F(f1 \neq f2 \wedge d \in defs(f_1) \wedge u \in uses(f_2) \wedge Def(u) = d)\}$: set of def-use relationships between the diff-regions in the changeset. If (d, u) is in the set, then d is inside a diff-region f_1, u is inside another diff-region f_2 and the associated definition of u is d.

Use-use

Given a definition d that is present in the changeset but is not inside any diff-region, if a diff-region f_1 has a use u_1 whose associated definition is d and a diff-region f_2 has a use u_2 whose associated definition is also d, then we say that there is a use-use relationship

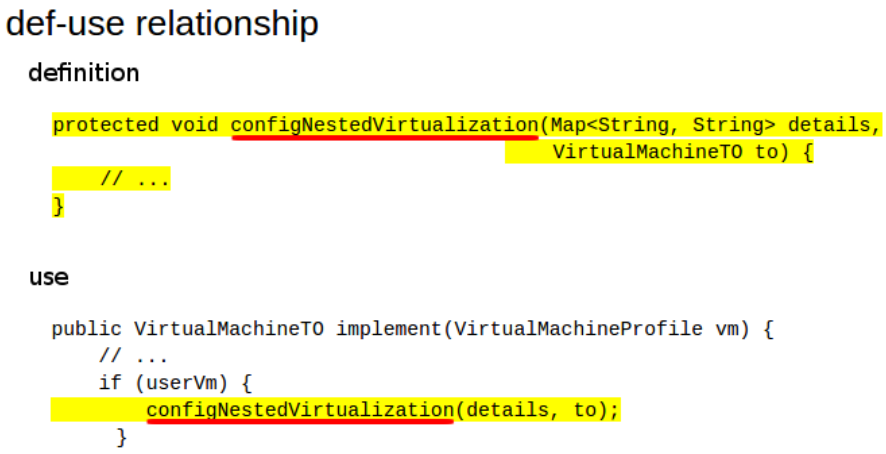

Figure 3.1: An example of a def-use relationship between the two highlighted diff-regions. The first diff-region contains the definition of the method *configNestedVirtualization* and the second diff-region has a use of that method.

between diff-regions f_1 and f_2. The definition d must be present in the changeset because the algorithm does not have access to the whole source code and it is not trivial to determine if two uses are referring to the same unknown definition. Furthermore, the definition d must not be inside any diff-region to avoid redundancy, because if it is inside one, then it will be part of a def-use relationship (Figure 3.2).

- $useUsesInDiffs = \{(u_1, u_2) \quad | \quad u_1 \in U \land u_2 \in U \land \quad \exists f_1, f_2 \in F(f1 \neq f2 \land u \in uses(f_1) \land u \in uses(f_2) \land Def(u_1) = Def(u_2)) \land \quad \forall f \in F(Def(u_1) \notin defs(f))\}$: set of use-use relationships between the diff-regions in the changeset. If (u_1, u_2) is in the set, then u_1 is inside a diff-region f_1, u_2 is inside another diff-region f_2, u_1 and u_2 have the same associated definition and this definition is present in the changeset but not inside any diff-region.

Same enclosing method

Besides relationships based on definitions and uses, ClusterChanges also considers diff-regions to be related if they are in the same method. This is because Barnett et al. not only observed that changes belonging to the same method are usually related [2] but also found that reviewers review changes to the same method at the same time [1].

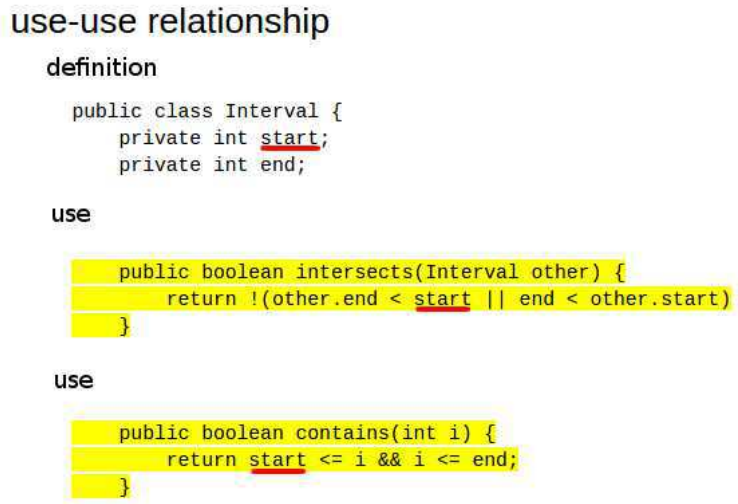

Figure 3.2: An example of a use-use relationship between the two highlighted diff-regions. Both diff-regions contain uses of the variable *start* whose definition is included in the changeset.

- $SameEnclosingMethod(f_1, f_2)$: this relation is true if the diff-regions f_1 and f_2 are inside the same method.

3.1.4 Partitioning

After identifying the relationships between the diff-regions, the changeset can be decomposed using graph theory. Let $G = (F, RelatedDiffs)$ be a directed graph with vertices F and edges $RelatedDiffs$, i.e. the vertices are the diff-regions in the changeset and the edges are the pairs in the relation $RelatedDiffs$. Thus, each partition of the changeset will be a component of the graph G, i.e. each partition will be a set of diff-regions that are related to one another but that are not related to any other diff-region outside the set.

The components of a graph can be found with the well-known union-find algorithm. Briefly, we start the algorithm with each vertex of the graph in a set of its own, then for each edge of the graph we merge the sets which contain the vertices connected by that edge.

We classify the resulting partitions as either trivial or non-trivial. Trivial partitions are those that contain only diff-regions that belong to the same method or that contain a single diff-region. Non-trivial partitions are the ones which are not trivial partitions, i.e. they have multiple diff-regions that are not all enclosed by the same method.

3.2 Example

In this section, we manually apply ClusterChanges to a small toy example to better understand the formal description presented in section 3.1.

3.2.1 Changeset

The changeset for this example consists of the modifications performed by a developer to accomplish two development tasks:

1. Bug fix: use *lastName* instead of *firstName* in the corresponding methods of class *Person*.
2. Feature addition: store the age of a person.

Only one file was changed by the developer. It is shown below before and after the developer modified it. The modifications are highlighted in yellow.

Source Code 3.1: Before-file (Person.java before changes)

```
class Person {
    private String firstName;
    private String lastName;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return firstName;
    }

    public void setLastName(String firstName) {
        this.firstName = firstName;
```

```
    }
}
```

Source Code 3.2: After-file (Person.java after changes)

```
class Person {
    private String firstName;
    private String lastName;
    private int age;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

3.2.2 Partitioning with ClusterChanges

We now apply ClusterChanges to the aforementioned changeset according to the formalism presented in section 3.1.

Definitions D

The definitions present in the after-file are highlighted below in blue. Note that, for ease of presentation, these are not the span of each definition. The symbol $ indicates the label given to that definition.

Source Code 3.3: Definitions in Person.java

```
class Person {
    private String $d0 firstName;
    private String $d1 lastName;
    private int $d2 age;

    public String $d3 getFirstName() {
        return firstName;
    }

    public void $d4 setFirstName(String $d5 firstName) {
        this.firstName = firstName;
    }

    public String $d6 getLastName() {
        return lastName;
    }

    public void $d7 setLastName(String $d8 lastName) {
        this.lastName = lastName;
    }

    public int $d9 getAge() {
        return age;
    }
```

```
    public void $d10 setAge(int $d11 age) {
        this.age = age;
    }
}
```

Uses U

The uses present in the after-file are highlighted below in blue. The symbol $ indicates the label given to that use.

Source Code 3.4: Uses in Person.java

```
class Person {
    private String firstName;
    private String lastName;
    private int age;

    public String getFirstName() {
        return $u0 firstName;
    }

    public void setFirstName(String firstName) {
        $u1 this.firstName = $u2 firstName;
    }

    public String getLastName() {
        return $u3 lastName;
    }

    public void setLastName(String lastName) {
        $u4 this.lastName = $u5 lastName;
    }

    public int getAge() {
        return $u6 age;
    }
```

```
    public void setAge(int age) {
        $u7 this.age = $u8 age;
    }
}
```

Diff-regions F

The diff-regions contained in the after-file are highlighted below in yellow. The symbol $ indicates the label given to that diff-region.

Source Code 3.5: Diff-regions in Person.java

```
class Person {
    private String firstName;
    private String lastName;
$f0     private int age;

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
$f1         return lastName;
    }

$f2     public void setLastName(String lastName) {
        this.lastName = lastName;
    }

$f3     public int getAge() {
        return age;
    }
```

```
$f4     public void setAge(int age) {
          this.age = age;
      }
}
```

Associated Definitions Def(u)

The definition associated with each use is shown below.

$$Def(u0) = d0$$
$$Def(u1) = d0$$
$$Def(u2) = d5$$
$$Def(u3) = d1$$
$$Def(u4) = d1$$
$$Def(u5) = d8$$
$$Def(u6) = d2$$
$$Def(u7) = d2$$
$$Def(u8) = d11$$

Spans

For ease of presentation, we represent the return of $span$ by the corresponding text in most cases. The actual formal definition of $span$ returns an interval of characters.

$span(d0) =$ "private String firstName;" *or* $\{19, ..., 42\}$
$span(d1) =$ "private String lastName;"
$span(d2) =$ "private int age;"
$span(d3) =$ "public String getFirstName() {...}"
$span(d4) =$ "public void setFirstName(String firstName) {...}"
$span(d5) =$ "String firstName"
$span(d6) =$ "public String getLastName() {...}"
$span(d7) =$ "public void setLastName(String lastName) {...}"

$span(d8) =$ "String lastName"

$span(d9) =$ "public String getAge() {...}"

$span(d10) =$ "public void setAge(int age) {...}"

$span(d11) =$ "int age"

$span(u0) =$ "firstName" *or* $\{154, ..., 162\}$

$span(u1) =$ "this.firstName"

$span(u2) =$ "firstName"

$span(u3) =$ "lastName"

$span(u4) =$ "this.lastName"

$span(u5) =$ "lastName"

$span(u6) =$ "age"

$span(u7) =$ "this.age"

$span(u8) =$ "age"

Definitions inside a diff-region defs(f)

$defs(f0) = \{d2\}$

$defs(f1) = \{d6\}$

$defs(f2) = \{d7, d8\}$

$defs(f3) = \{d9\}$

$defs(f4) = \{d10, d11\}$

Uses inside a diff-region uses(f)

$uses(f0) = \{\}$

$uses(f1) = \{u3\}$

$uses(f2) = \{u4, u5\}$

$uses(f3) = \{u6\}$

$uses(f4) = \{u7, u8\}$

defUsesInDiffs

$defUsesInDiffs = \{(d2, u6), (d2, u7)\}$

useUsesInDiffs

$$useUsesInDiffs = \{(u3, u4)\}$$

SameEnclosingMethod

In this example, there are not any pair of distinct diff-regions that are inside the same method.

RelatedDiffs

$$RelatedDiffs = \{(f0, f3), (f0, f4), (f1, f2)\}$$

Partioning

Using an algorithm (such as the union-find algorithm) to find the components of the graph $G = (F, RelatedDiffs)$, the partition P is obtained:

$$P = \{\{f0, f3, f4\}, \{f1, f2\}\}$$

We can see that the algorithm correctly partitioned the diff-regions. $\{f0, f3, f4\}$ is the partition that matches task 2 and $\{f1, f2\}$ matches task 1.

3.3 Evaluation of Barnett et al.

To evaluate the efficacy of ClusterChanges, Barnett et al. first performed a quantitative evaluation followed by a qualitative one. Subsections 3.3.1 and 3.3.2 explain the design of their evaluations and summarize their findings [2].

3.3.1 Quantitative Evaluation

Study Design

The authors of the original study selected at random a sample of 1000 changesets from the Microsoft Office project that were submitted for review and then ran their implementation of ClusterChanges for C# on them.

They described their dataset using boxplots (Figure 3.3) of the following size metrics:

- Files changed (number of files which were added, modified or deleted)
- Methods changed (number of methods affected by the changes)
- Diff-regions

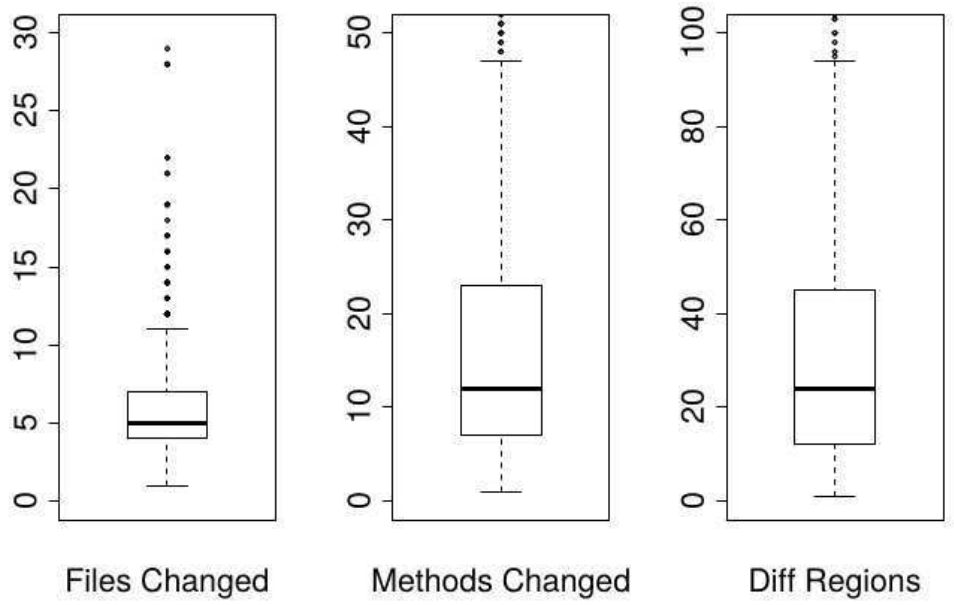

Figure 3.3: Boxplots of change sizes from the original study. Figure extracted from Barnett et al.'s study [2].

Findings

Barnett et al. summarized their results using two histograms: one for the number of non-trivial partitions (Figure 3.4) and another for the number of trivial partitions (Figure 3.5).

Assuming that ClusterChanges correctly identifies non-trivial partitions (no false-positives), the results show that as much as 42% of the changesets can be decomposed. Furthermore, the average proportion of changed methods that are contained in non-trivial partitions is 66%, which suggests that the partitioning generated by ClusterChanges, although imperfect, covers a significant portion of the changeset.

The ideal evaluation of the partitionings suggested by ClusterChanges would be comparing each of them to the mental model of the developer who created that changeset. However,

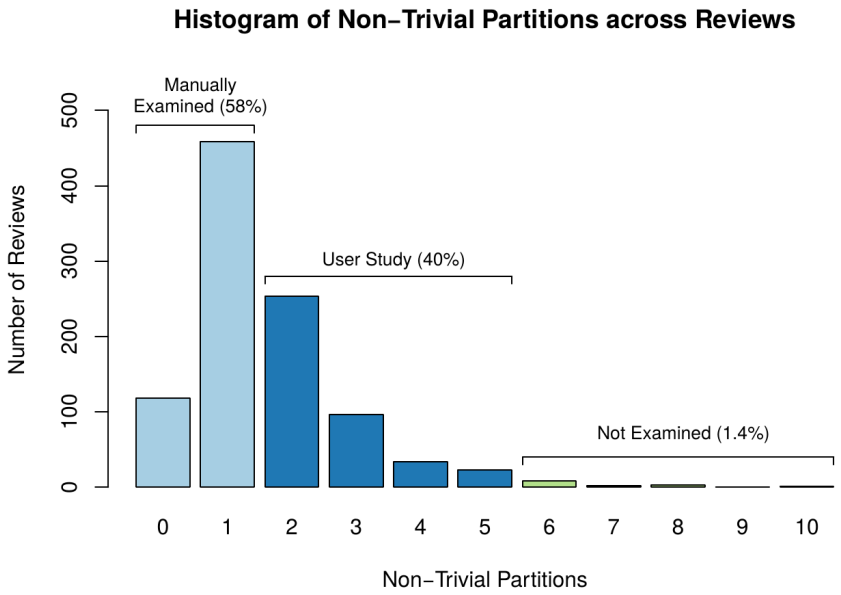

Figure 3.4: Distribution of non-trivial partitions from sample of 1000 changesets submitted for review at Microsoft. Figure extracted from Barnett et al.'s study [2].

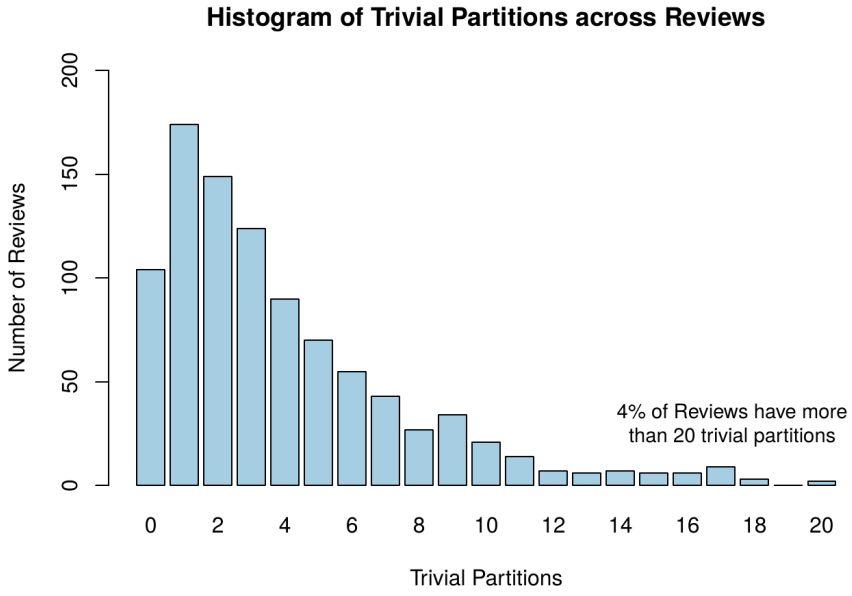

Figure 3.5: Distribution of trivial partitions from sample of 1000 changesets submitted for review at Microsoft. Figure extracted from Barnett et al.'s study [2].

given that such a study would be too costly, Barnett et al. divided the changesets into four groups based on their number of partitions and performed different analyses for each group.

- ≤ 1 **non-trivial partition:** Barnett et al. manually analyzed a random sample of 50 of these changesets to determine if the partitioning generated by ClusterChanges was not obviously wrong by looking at the commit messages and the code changes. 6 of those changesets had commit messages that suggested more than one task and they were all reasonably partitioned. Moreover, the authors found no false-positives.

- $[2, 5]$ **non-trivial partitions:** These were subjected to a qualitative evaluation where the authors interviewed the developers who made them to determine if the partitioning was correct. This study and its results are detailed in the next subsection.

- ≥ 6 **non-trivial partitions:** The authors of the original study did not analyze this group because only 1.4% of the changesets fall into it and it would take too much time to interview developers about these changesets.

- ≥ 10 **trivial partitions:** The authors analyzed 15 out of 199 changesets that had at least 10 trivial partitions and concluded that missing relationships are what caused so many trivial partitions.

3.3.2 Qualitative Evaluation

Study Design

As a first step in designing their qualitative study, Barnett et al. selected three research questions to guide the study:

RQ1 Do developers agree with our decomposition of their changes?

RQ2 What role do trivial partitions play?

RQ3 Can organizing a changeset using our decomposition help reviewers?

To answer these research questions, the authors used a firehouse research study design where they conducted 20 semi-structured interviews with Microsoft developers from 13 different projects. In this type of study, the authors wait for the expected event to happen and

then interview the developer as soon as possible so that his memory of the event is still fresh. The expected event here was the submission of a changeset for code review that contained between 2 and 5 non-trivial partitions and the interviews happened in at most three days after the changeset submission.

Findings

The authors answered each of the research questions using the study results:

- **RQ1:** 16 out of the 20 interviewees stated that the non-trivial partitions were correct and complete. Of the 4 who did not agree with the non-trivial partitions, 2 claimed that there should be only one non-trivial partition and the other 2 did agree with the non-trivial partitions, but explained that some of the trivial partitions should be grouped into another non-trivial partition.

- **RQ2:** To answer this research question, the authors focused on the participants who considered the trivial partitions to be correct. The authors hypothesized that non-trivial partitions were more important than trivial ones, however the participants reported mixed opinions on this hypothesis. In addition, the authors were also interested in knowing if the trivial partitions were easier to understand than non-trivial ones and indeed as many as 16 participants out of 20 found trivial partitions easier to understand, while 3 were not sure about it and 1 thought that both were equally challenging to understand.

- **RQ3:** All participants agreed with the rationale behind ClusterChanges, which is to help developers understand changes. As for the technique itself, 18 participants would like to use the tool in their next changesets, while 2 participants expressed doubts about the usefulness of the tool.

3.3.3 Threats to Validity

Although the study obtained promising results, it suffers from a few threats to external validity. These threats mean that the results of the study may not be generalizable to other contexts.

- the changeset sample only contains C# code. It is not clear whether the technique works for other programming languages.

- changesets are from a single organization, namely Microsoft. Maybe the technique is not suitable for other organizations.

- the changesets consist of closed source code. ClusterChanges may not be useful for OSS project, because researchers have shown that OSS development practices are often significantly different from their closed source counterparts [34][30].

Chapter 4

JClusterChanges

In this chapter, we present JClusterChanges — our implementation of the ClusterChanges technique (Section 3) for Java software. We had to create JClusterChanges to carry out the replication because the original implementation of ClusterChanges for C# software is not publicly available. We also created a Web GUI for JClusterChanges in order to improve its usability.

JClusterChanges is free and open source software. It is licensed under the GNU Affero General Public License version 3 [16] and is publicly available at: `https://sites.google.com/site/jclusterchanges/`

4.1 Implementation

We decided to implement ClusterChanges for Java instead of C# for several reasons (in order of importance):

- Java is a more popular programming language for OSS projects (Figure 4.1)
- There are multiple OSS Java code parsers available
- In case Barnett et al. publicly release their implementation of ClusterChanges, our implementation is more valuable to the community if it targets a different language
- We had previous experience with the Java programming language

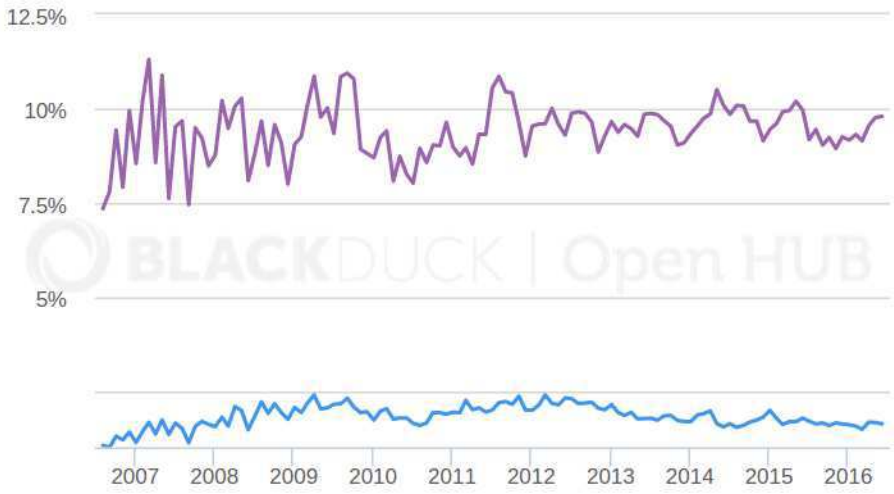

Figure 4.1: Proportion of monthly commits in OSS projects which contained code of a certain programming language over the years. From 2007 to 2016, between 7% and 12% of the commits of each month had Java code, while less than 2.5% of the monthly commits had C# code. In other words, there were roughly three times more commits containing Java code than commits containing C# code between the years 2007 and 2016. Figure generated by Open Hub [7].

A parser is needed in order to identify def-use, use-use and same enclosing methods relationships between diff-regions. We could not apply the Roslyn compiler as in the original study because it only works with C# code. Thus, we used the Eclipse Compiler for Java (ECJ) [38] for parsing Java code. We chose ECJ because it's a mature, open and incremental compiler whose main goal is analyzing partial programs that may not be fully compilable. For instance, the Eclipse IDE uses ECJ for performing real-time static analysis while the developers are editing the source code.

To verify and validate JClusterChanges, we applied automated test cases and analyzed the results it generated for changesets from real software projects.

JClusterChanges provides a command-line interface that takes as input a changeset represented by a folder with after-files and patch files. The after-files are the files after being modified by the changeset author and the patch files contain the diff-regions, i.e. information about what was modified in each file. If desired, the before-files can be generated from those two by using a diff tool.

The output of JClusterChanges is a set of files in the comma-separated values (CSV) file format. These files contain not only the partitioning of the changeset but also all definitions, uses, diff-regions, relationships between diff-regions that were identified. Figure 4.2 summarizes how JClusterChanges works.

4.2 Web Front End

We also developed a Web GUI for visualizing the results of JClusterChanges (Figure 4.3). It improves the usability of JClusterChanges and allows users to try the tool without having to install it on their computers.

Using the GUI, users can choose a pull request from GitHub and visualize the partitioning generated by JClusterChanges. First, the user types the link to the pull request he wishes to partition. Then, the resulting partitions are displayed as a tree structure in the left hand side of the screen. When the user selects a diff-region in the tree, it is displayed in the editor to its right. This way, the user can review each partition independently (Figure 4.3).

Regarding its implementation, the Web GUI follows a client-server architecture. The server downloads the pull request specified by the user from GitHub, parses it with JClus-

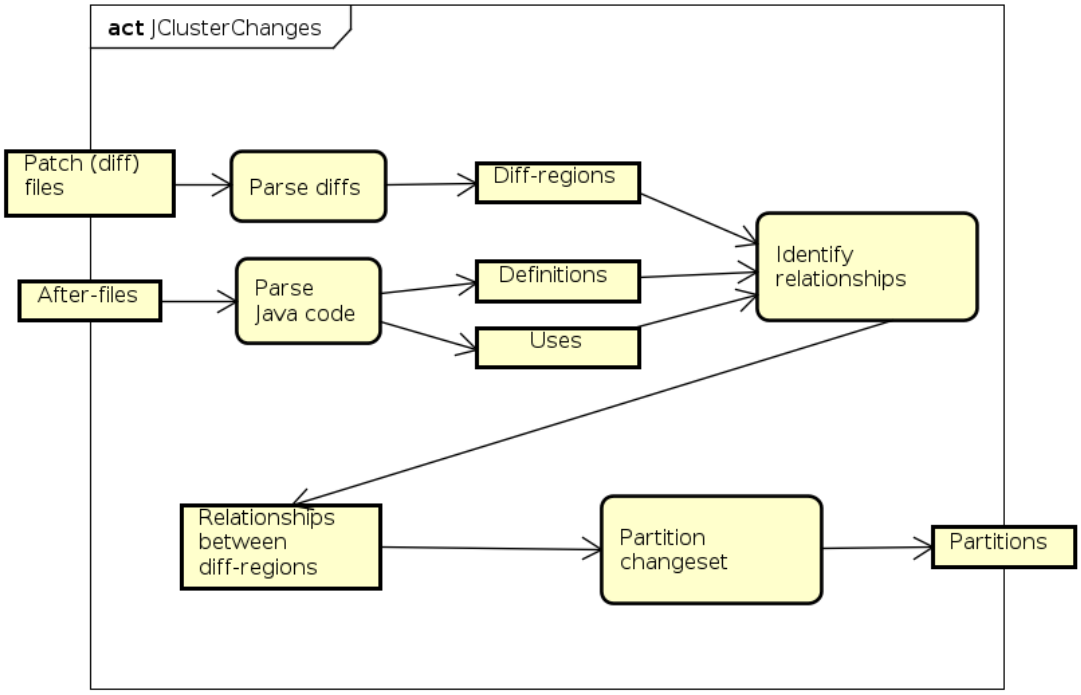

Figure 4.2: A UML 2 activity diagram which provides a high-level view of how JClusterChanges partitions a changeset.

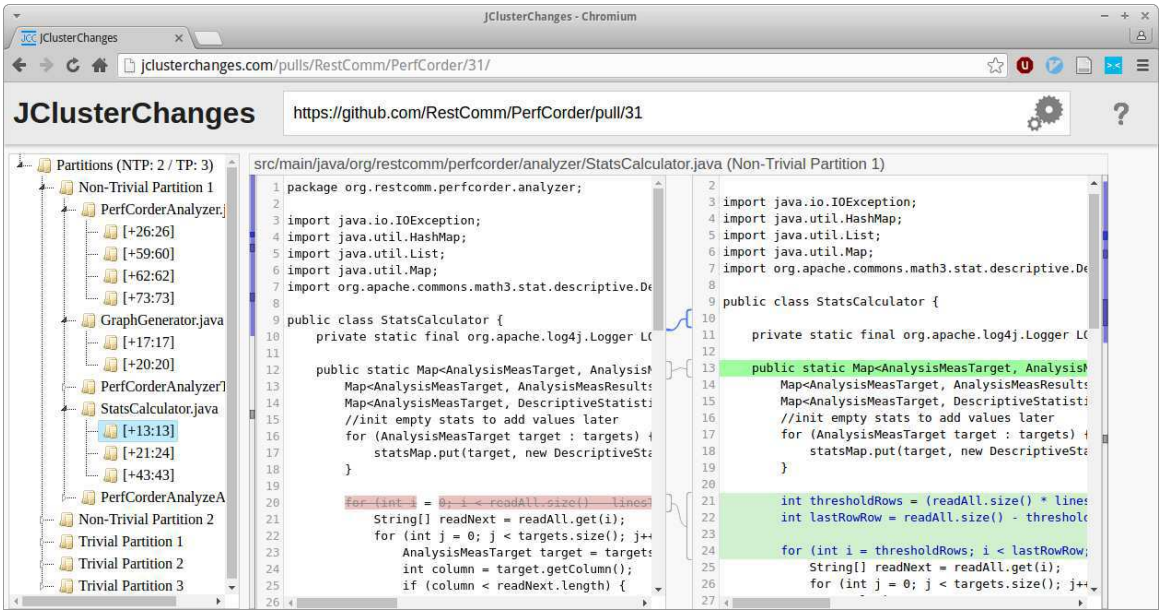

Figure 4.3: Web GUI for JClusterChanges

terChanges and sends the results to a JavaScript client app which lets the user interactively examine the results.

We decided to integrate the Web GUI with GitHub because we hope to use it in the future to replicate the qualitative study by interviewing OSS developers from GitHub.

Chapter 5

Replication

In this chapter, we describe our replication of the quantitative substudy of the original study from Barnett et al. [2]. First, we explain the study design and the rationale behind it. Then, we explain how data was collected. Next, we present the results and discuss them. Finally, we list the main threats to validity.

5.1 Study Design

The goal of this study is to address the threats to external validity from the original study which were enumerated in Section 3.3.3. To accomplish this, we perform a replication where we change the population dimension as recommended by Gómez et. al classification of SE replications (Table 2.1). Hence, this is a changed-population/changed-experimenters replication.

This study can be divided into three phases. First, we selected at random a total of 1000 pull requests from the 10 most popular Java OSS projects hosted at GitHub. After that, we used JClusterChanges to automatically partition them. Finally, we quantitatively analyzed the dataset similarly to the original study.

The reasons for choosing software written in the Java programming language are detailed in Section 4.1.

As for OSS projects, there are two main reasons for choosing them as context. First, it is a different context as desired, because researchers have shown OSS development practices to be significantly different from their closed source counterparts [34][30]. Second,

OSS projects have a wealth of data easily accessible on the Internet, which facilitates data collection and replicability of empirical studies.

Our choice of GitHub stems from the fact that not only it is the largest code host in the world as explained in Section 2.2 but it also provides a REST API that makes it easy to mine data from it.

Out of all the Java OSS projects in GitHub, we mined the most popular ones because we hypothesized that these would have large numbers of diverse pull requests to analyze.

We chose to analyze pull requests because we believe they closely mirror the changesets used in the original study, that is, both are sets of pairs of changed files (before-file and after-file) that are submitted for review and if approved will be merged into the target source repository.

5.2 Data Collection

In this section, we explain how we collected the sample of 1000 pull requests from GitHub.

5.2.1 Software Project Sample Selection

We chose 10 software projects from GitHub in the following way:

1. In the GitHub web page, we asked for the list of open source Java projects in descending order of stars (search string: stars:>1 language:java). We hypothesized that the projects with the most stars are the most popular and would have the most pull requests.

2. Then we manually analyzed each software project in the list (until we had 10 projects) and selected it for the study if:

    - it used GitHub's pull request system (most projects do not use pull requests [26]);
    - it was not a mirror of a repository maintained somewhere else. When this is case, the GitHub's pull request system is not being used by the project;
    - it had at least 300 pull requests which contained Java source code (in 2014, 95% of GitHub projects had at most 25 pull requests [26]);

- it was targeted at the JVM (i.e. Android exclusive projects were not considered). We wanted to avoid compatibility issues.

5.2.2 Pull Request Sampling

For each software project, we:

1. Sampled 300 pull requests at random that matched the following criteria:
   - Had at least one Java source code after-file, since we are interested in analyzing Java code
2. Ran JClusterChanges on these 300 pull requests;
3. Of these 300 pull requests, sampled 100 pull requests at random that matched the following criteria:
   - Was analyzed by JClusterChanges without errors or warnings (we discuss this further at the Threats to Validity section)

5.3 Results

5.3.1 Pull Request Sizes

Similarly to the original study, we use boxplots of three metrics to describe the sizes of the changesets in the dataset (Fig. 5.1). The boxplots indicate that pull requests tend to be small. Nevertheless, the presence of numerous outliers show that big pull requests occur frequently. It is likely that these outliers contain independent modifications and, thus, it is likely that the problem is real, i.e. that composite changesets are not rare.

Since the original study data is not publicly available, we could only compare the results visually. Figure 5.2 shows the boxplots of the original study and the boxplots of this study side-by-side in order to allow them to be more easily compared.

The boxplots show that the changesets of this study are generally smaller than the ones from the original study.

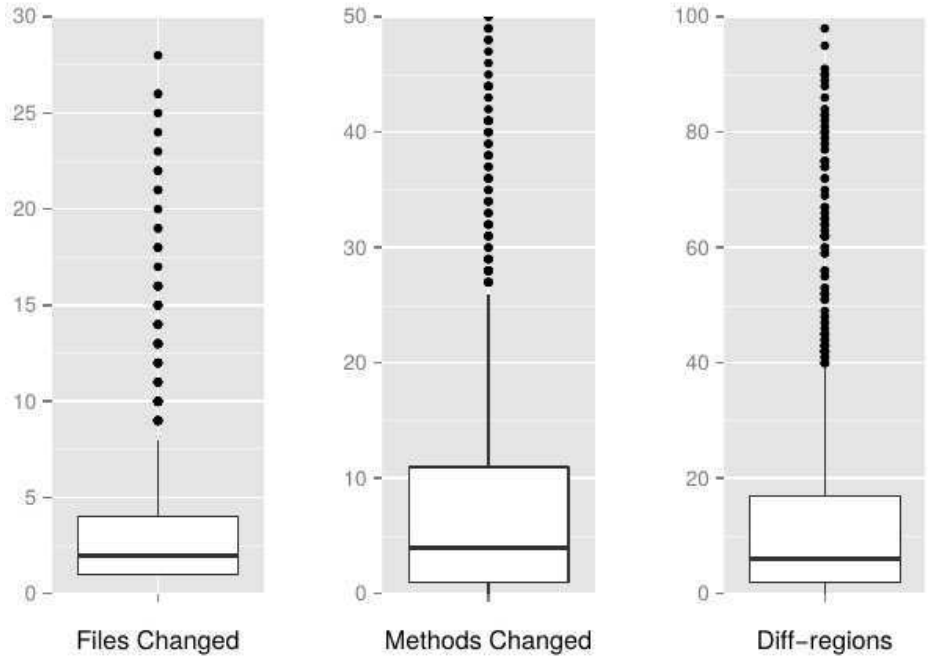

Figure 5.1: Boxplots of change sizes from this study.

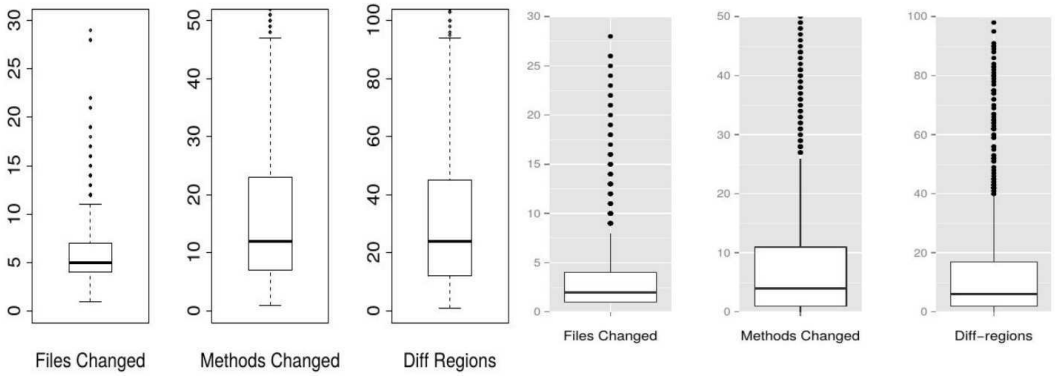

Figure 5.2: Comparison between the boxplots of change sizes from the original study (left-hand side) and the ones from this study (right-hand side).

5.3.2 Partitions

According to the histogram of non-trivial partitions (Fig. 5.3), the three most common cases in descending order are pull requests with: 1 non-trivial partition (41.5%), 0 non-trivial partitions (36.9%) and 2 non-trivial partitions (12.2%). Moreover, 90.6% of the pull requests have at most 2 non-trivial partitions and 95.3% of the pull requests have at most 3 non-trivial partitions.

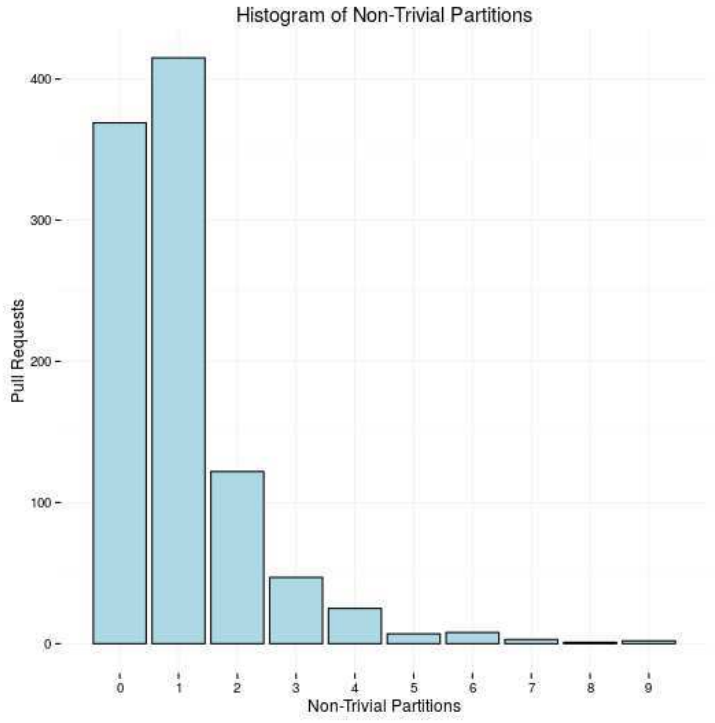

Figure 5.3: Distribution of non-trivial partitions from sample of 1000 pull requests.

37% of the changesets did not have non-trivial partitions. This happens when the modifications are too small or consist of relationships not detected by JClusterChanges, thus the diff-regions are spread out in several trivial partitions. Since the median of diff-regions was small, the most frequent case was a changeset with few modifications.

Again, we could only compare the results visually because the original study data is not publicly avaiable. Figure 5.4 shows the histograms of non-trivial partitions of the original study and of this study side-by-side.

The histogram of non-trivial partitions obtained in this study is visually similar to the one from the original study (Fig. 5.4). More specifically, the tail of both distributions is similar

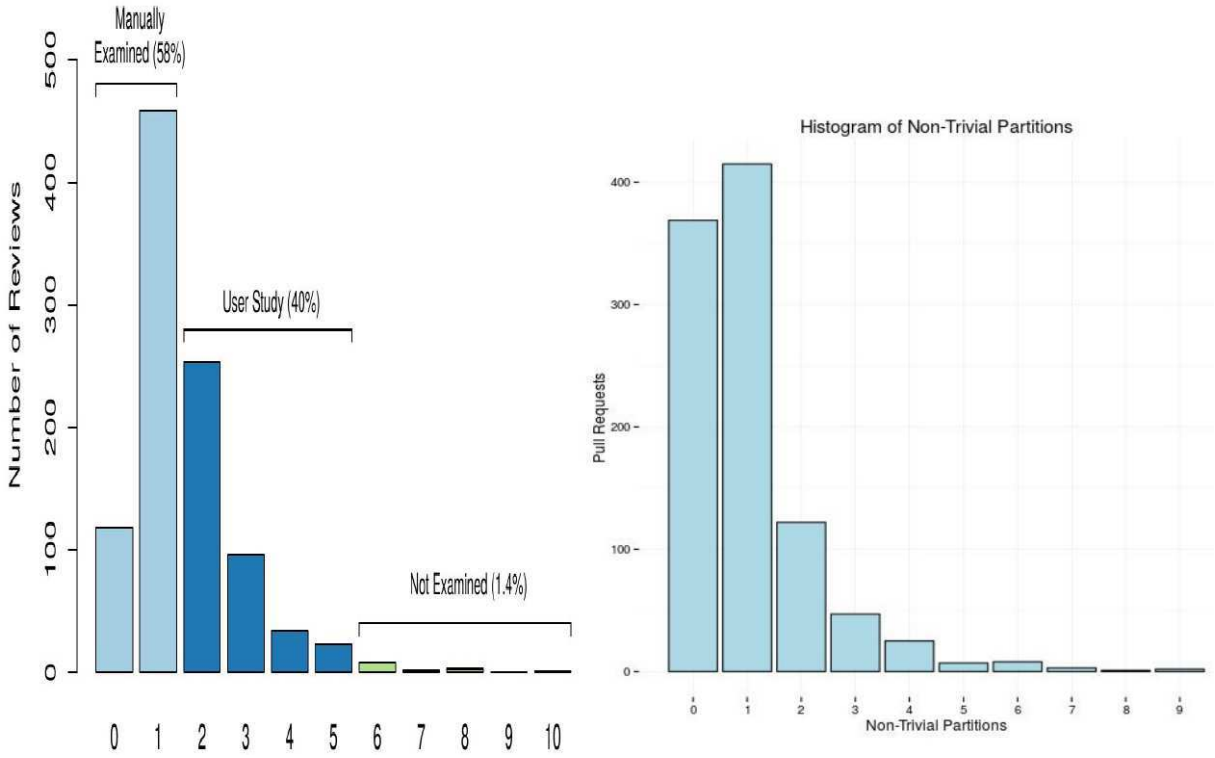

Figure 5.4: Comparison between the histogram of non-trivial partitions of the original study (left-hand side) and the one from this study (right-hand side).

and the most common cases are the same. But, the changesets from this study tend to have fewer non-trivial partitions.

As for the histogram of trivial partitions (Fig. 5.5), the three most common cases in descending order are pull requests with: 1 trivial partition (30.5%), 0 trivial partitions (25.9%) and 2 trivial partitions (12.6%). Unlike the distribution of non-trivial partitions, there is a long tail and 31% of the pull requests have more than 2 trivial partitions

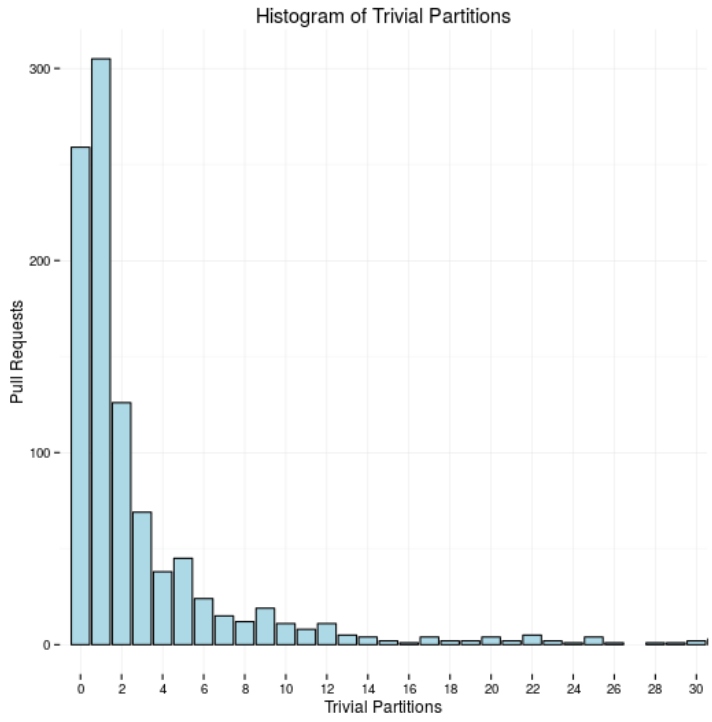

Figure 5.5: Distribution of trivial partitions from sample of 1000 pull requests.

Figure 5.6 shows the histograms of trivial partitions of the original study and of this study side-by-side.

A visual comparison between the histogram of trivial partitions from this study and the one from the original study (Fig. 5.6) shows a certain similarity between them. However, the changesets from this study tend to have fewer trivial partitions. Furthermore, while 4% of the changesets from the original study have more than 20 trivial partitions, 3% of the changesets from this study have more than 20 trivial partitions.

We followed the original study grouping strategy and summarize the percentage of pull requests in each of the four groups in Table 5.1. There is a marked difference between the

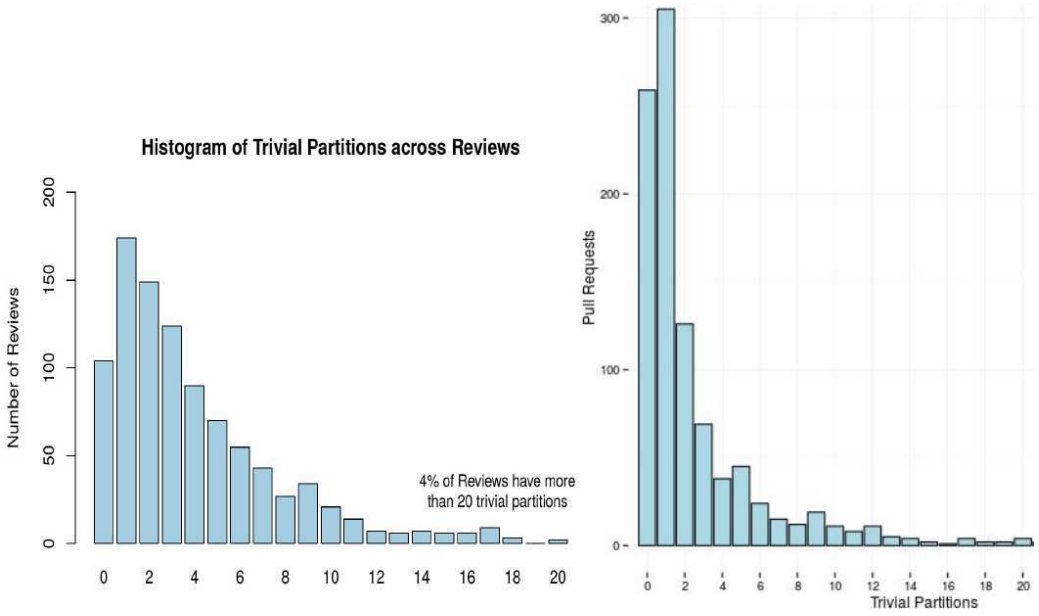

Figure 5.6: Comparison between the histogram of trivial partitions of the original study (left-hand side) and the one from this study (right-hand side).

Table 5.1: Groupwise comparison between this study dataset and the original study one.

Group	This Study	Original Study
≤ 1 **non-trivial partition**	78.4%	58%
$[2, 5]$ **non-trivial partitions**	20.1%	40%
≥ 6 **non-trivial partitions**	1.5%	1.4%
≥ 10 **trivial partitions**	7.7%	11.9%

proportions of pull requests with at most 1 non-trivial partition and the proportions of those with between 2 and 5 non-trivial partitions.

5.3.3 Pull Requests with ≤ 1 Non-Trivial Partitions

We manually investigated a random sample of 50 pull requests that have at most one non-trivial partition with the goal of determining if ClusterChanges grouped unrelated diff-regions in the same non-trivial partition. As in the original study, we manually analyzed these pull requests following a process similar to Herzig and Zeller [37].

Only two pull requests clearly contained independent changes and JClusterChanges separated those changes into different partitions. One of the pull requests consisted of small changes to two methods and JClusterChanges correctly split it into 2 trivial partitions and the other pull request consisted of a refactoring and a bug fix and was split into 1 non-trivial partition and 1 trivial partition as expected. Finally, JClusterChanges never grouped unrelated diff-regions in the same partition in these 50 pull requests.

5.3.4 Pull Requests with > 10 Trivial Partitions

In accordance with the original study hypothesis that "a single developer's change cannot consist of so many independent (sub) changes" [2], we manually investigated 15 of the 77 pull requests with more than 10 trivial partitions in order to determine what relationships ClusterChanges does not detect and, as a result, gain insight on how it can be improved.

We obtained results that are similar to the ones from the original study. The three most common relationships that JClusterChanges could not detect were:

- Refactoring patterns such as code style and formatting changes, e.g., renaming the arguments of several methods, marking methods of a class as final and breaking lines longer than 80 characters;
- Changes dependent on code not available in the pull request;
- Code called indirectly. For instance, methods meant to be called by a test framework, overridden methods in a subclass that are only called in the code by the base class and methods of a REST API which are called automatically by the Web library being used.

Figure 5.7 shows an example of a refactoring pattern that JClusterChanges could not detect. In the example, the developer changed the name of several local variables from *t* to *e*. Although these changes are clearly related for another developer reviewing the code, JClusterChanges mark them as related because there are not def-use and use-use relationships between them.

Figure 5.8 presents an example of related changes not identified by JClusterChanges because the code is called indirectly. Since the methods are called by a Web framework during

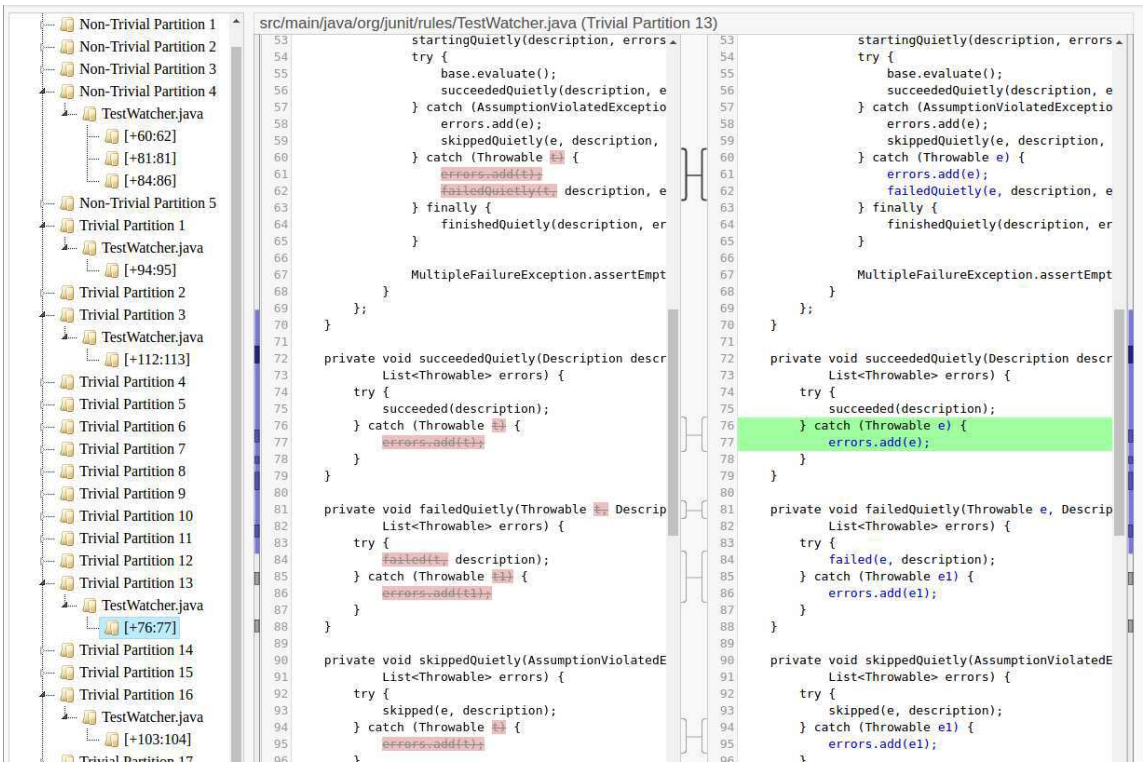

Figure 5.7: An example of related changes that JClusterChanges fails to group together into a single partition. They are related because they follow the same refactoring pattern, where several different local variables with the same name are renamed in a similar way.

runtime, JClusterChanges is unable to identify def-use and use-use relationships between them. Thus, the methods end up in separate trivial partitions.

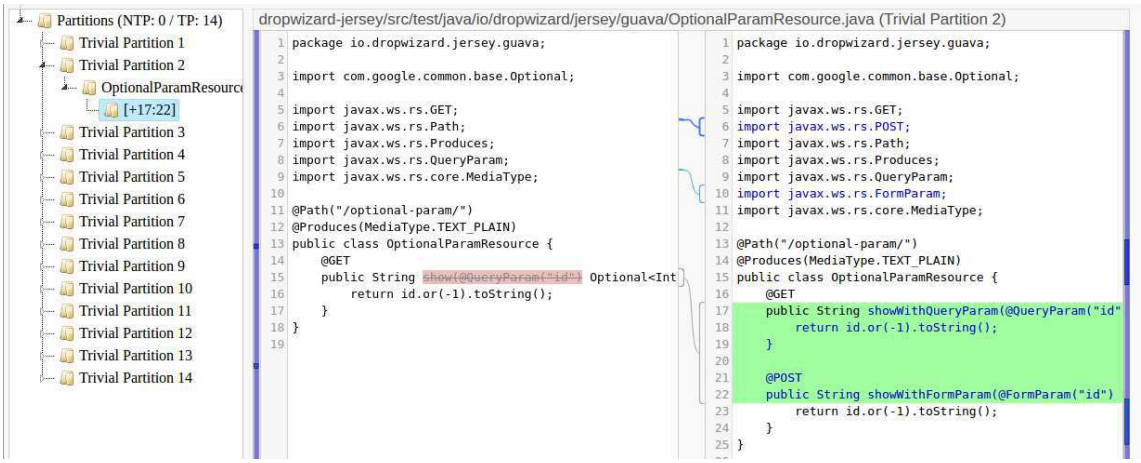

Figure 5.8: Another example of related changes that JClusterChanges fails to identify. The methods are not marked as related because they are called indirectly.

5.4 Discussion

The similarity between the histograms of non-trivial partitions and trivial partitions of this study and the ones from the original study suggests that ClusterChanges is as effective in the context of OSS projects as in the context of closed source software projects.

The boxplots of size metrics (Fig. 5.2) indicate that that this dataset changesets are smaller than the ones analyzed in the original study. This suggests that pull requests tend to be more cohesive and less complex than the changes sent for review at companies such as Microsoft. We hypothesize that this is because pull requests are often created by outside developers who are not familiar with the code.

Given the smaller size of this dataset changes compared to the changes of the original study dataset, we expected to see fewer non-trivial partitions in this dataset and this prediction proved to be correct since there were about half as many changesets as the original study with more than 1 non-trivial partition. Nevertheless, the number of changesets with multiple partitions is significant and is evidence that large changesets with independent changes are also commonplace in OSS projects.

22% of the pull requests have more than 1 non-trivial partition (Fig. 5.3). Considering that ClusterChanges never identified a false-positive, this means that at most 22% of the changesets are composite. Furthermore, this proportion is close to the estimates of 16% and 17% provided by past studies as to how many changesets in OSS projects are composite. This suggests that the partitioning is correct.

Our manual analysis of 50 pull requests provided further evidence that the def-use relationship is a fundamental one as claimed in the original study since we did not found any false positives. A false-positive would be an instance where JClusterChanges had put two unrelated diff-regions in the same partition and this did not happen in any of the pull requests analyzed. Such absence of false positives means that the tool provides an upper bound on the number of independent partitions within a changeset. Furthermore, we believe that this is of vital importance for user acceptance of the tool, because false positives have been a significant obstacle in the adoption of static analysis tools for finding bugs [6].

Although there were no false positives, we did observe several types of relationships between diff-regions that ClusterChanges did not detect. Diff-regions that were related by logical patterns but did not have code dependencies between them were a common type of false negative. A few real examples were: adding the prefix "Abstract" to the name of all abstract classes, changing the access modifier of several fields from private to protected and adding final modifiers to fields. Tao et al. had some success using pattern matching techniques for detecting such relationships [37], so their approach could potentially be used to improve the efficacy of ClusterChanges with regards to these relationships.

We also frequently saw missed use-use relationships because these depended on definitions not present in the changeset, e.g. calls to a Logger API not present in the changeset and test cases for code that was not changed. One way to address this, at least in the context of GitHub pull requests, would be to modify ClusterChanges to consider the whole code base instead of just the files changed. But, future research is needed to determine whether this can be done efficiently since there would be a lot more source code for the tool to analyze.

A type of relationship that is also often missed and that would be harder to detect than the two aforementioned ones is when diff-regions are related by indirect code calls, e.g. code is dynamically related via callbacks or a dependency injection framework.

5.5 Threats to Validity

At the time of writing and during the quantitative evaluation, some changesets are not being fully analyzed by our tool. This seems to be caused by limitations in ECJ when parsing certain features of Java, e.g., we have observed that ECJ is not detecting the use of a field when this use is inside a lambda function. As this may result in missing relationships and seem to be an implementation issue unrelated to ClusterChanges technique itself, we have excluded such changesets from the dataset for now.

We have not finished our replication of the qualitative study due to time constraints. Even though our results provide evidence that OSS developers face a similar situation to Microsoft developers and that ClusterChanges behave similarly in an OSS context, it is still possible that OSS developers do not agree with the partitionings or that they would not like to use the tool, since OSS development practices tend to be significantly different [30][34].

Chapter 6

Related Work

We found five other studies that present techniques for automatic decomposing changesets. In this chapter, we summarize these techniques.

Herzig and Zeller [25] presented a technique that combines five heuristics to identify related changesets: FileDistance, PackageDistance, CallGraph, ChangeCouplings and DataDependency. The heuristics of CallGraph and DataDependency are similar to the def-use relationship because they detect static dependencies between regions of change. Although their algorithm had decent performance, it was evaluated using artificially created composite changesets.

Kirinuki et al. [27] devised a technique based on pattern matching that is able to warn users if they are about to commit a potentially composite change by analyzing the history of the software repository. They presented an early evaluation of their technique using a single OSS project that shows the feasibility of their approach, but, unlike ClusterChanges, there was a considerable number of false positives.

Nguyen et al. [31] observed that changesets containing a mix of bug fixing code changes and unrelated non-fixing code changes reduce the efficacy of MSR techniques. So, they created a tool that can filter files containing non-fixing code changes in such a changeset by decomposing it into independent clusters and afterwards using natural language processing algorithms. Although their decomposition algorithm uses a dependency-based analysis similar to ClusterChanges, it is specialized for detecting bug fixing changes and it works on a coarser granularity level: it groups files together instead of diff-regions. Moreover, this study and the two aforementioned ones differ from ClusterChanges by focusing on the impact of

composite changes on mining software repositories (MSR).

Dias et al. [13] created a technique based on machine learning for decomposing changes before developers commit them with the goal of avoiding the creation of composite commits. Their technique relies on processing fine-grained changes which are recorded by the IDE while the developers work on the source code, e.g. add method and delete class. Differently from ClusterChanges, the technique needs to be calibrated for each developer to maximize its efficacy.

Tao and Kim [37] also published a heuristic-based technique for automatically partitioning changesets. Instead of def-use relationships, their technique uses three other heuristics: one that detects and groups formatting changes in a single partition, another which uses the concept of program slicing for identifying static dependencies and a third which detects related changes using pattern matching. Improvements over ClusterChanges include the fact that their technique considers code deletions and that they performed a controlled experiment to evaluate it. As a downside, the evaluation of the technique is based on the unverified assumption that a changeset has multiple acceptable partitions. Hence, the authors of the analyzed changesets were not contacted in order to obtain the correct partitioning, the ground truth.

Chapter 7

Conclusion

In this work, we studied whether an existing technique for automatically decomposing changesets containing independent modifications could be applied to OSS projects. To accomplish this, we created our implementation of this technique and replicated an evaluation study of the technique.

Large changesets containing independent modifications negatively affect the efficacy of modern code review. Because of this, Barnett et al. developed and evaluated an algorithm called ClusterChanges for automatically decomposing such changesets into partitions that can be reviewed independently. Although their study of ClusterChanges had promising results, their analysis was restricted to proprietary software projects from a single organization and their implementation of the technique was not publicly released.

To address those limitations, we created JClusterChanges, a FOSS implementation of ClusterChanges for Java software projects, and used it to evaluate the technique in the context of Java OSS projects from different organizations.

We obtained results similar to the ones in the original study. Hence, we have provided evidence that ClusterChanges is generalizable to other contexts and further evidence that the problem does exist. In addition, JClusterChanges shows that it is possible to implement the technique for other programming languages and it can be used to help future research on this subject, e.g. by using it as a baseline to evaluate other techniques and by extending its implementation to evaluate changes to the technique.

As with any scientific study, our paper suffers from threats to validity. Some changesets are not being fully parsed by JClusterChanges and they were thus excluded from the study.

However, this may still have affected the results, e.g. it is possible that those changesets had multiple non-trivial partitions. Another noteworthy limitation is that we did not replicate the qualitative study. As a result, it is possible that OSS developers do not agree with the partitionings created by JClusterChanges.

In keeping with the spirit of OSS, to aid future research on the topic of automatic decomposition of changesets and to encourage replication of this work, we provide the material and results of this research at: https://sites.google.com/site/jclusterchanges/

7.1 Contributions

The main contributions of this work are:

- Evidence that the problem also exists in OSS projects, i.e. composite changesets are common in OSS projects.
- Evidence that ClusterChanges is effective in other contexts. Our results suggest that ClusterChanges can assist developers working in any software organization using any modern code review process.
- JClusterChanges: a free and open source implementation of the ClusterChanges technique that can partition changesets written in the Java programming language.
- The results of this study will be submitted as a paper to a conference.

7.2 Future Work

As for future work, there are numerous possibilities. Presently, we are working on a replication of the qualitative portion of the original ClusterChanges study. This will address one of the limitations of this study, namely whether OSS developers would find JClusterChanges useful.

Another possibility would be to try to improve JClusterChanges according to the discussion on missing relationships from both this paper and the original study. Both studies combined provide a considerable amount of information on the relationships not detected by ClusterChanges.

Comparing the efficacy of ClusterChanges with other published techniques for decomposing changes could also yield valuable insight. It may be possible to substantially increase the effectiveness of JClusterChanges by combining ClusterChanges with another existing decomposition technique.

Bibliography

[1] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 712–721. IEEE Press, 2013.

[2] Mike Barnett, Christian Bird, Joao Brunet, and Shuvendu K. Lahiri. Helping Developers Help Themselves: Automatic Decomposition of Code Review Changesets. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 2015.

[3] Victor R. Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *Software Engineering, IEEE Transactions on*, 25(4):456–473, 1999.

[4] Gabriele Bavota and Barbara Russo. Four eyes are better than two: On the impact of code reviews on software quality. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 81–90. IEEE, 2015.

[5] Moritz Beller, Alberto Bacchelli, Andy Zaidman, and Elmar Juergens. Modern code reviews in open-source projects: which problems do they fix? In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 202–211. ACM, 2014.

[6] Al Bessey, Dawson Engler, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, and Scott McPeak. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, February 2010.

[7] Black Duck Software, Inc. Open Hub - comparison between C# and Java with regard to monthly commits. `https://www.openhub.net/languages/`

compare?language_name%5B%5D=csharp&language_name%5B%5D=java&language_name%5B%5D=-1&language_name%5B%5D=-1&measure=commits. [Online; accessed 01-Aug-2016].

[8] A. Brooks, M. Roper, M. Wood, J. Daly, and J. Miller. Replication's Role in Software Engineering. In Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 365–379. Springer London, 2008.

[9] Jeffrey C. Carver. Towards reporting guidelines for experimental replications: A proposal. In *1st International Workshop on Replication in Empirical Software Engineering*. Citeseer, 2010.

[10] Jeffrey C. Carver, Natalia Juristo, Maria Teresa Baldassarre, and Sira Vegas. Replications of software engineering experiments. *Empirical Software Engineering*, 19(2):267–276, April 2014.

[11] M. Ciolkowski, O. Laitenberger, and S. Biffl. Software reviews: The state of the practice. *IEEE Software*, 20(6):46–51, November 2003.

[12] Fabio Q. B. da Silva, Marcos Suassuna, A. César C. França, Alicia M. Grubb, Tatiana B. Gouveia, Cleviton V. F. Monteiro, and Igor Ebrahim dos Santos. Replication of empirical studies in software engineering research: a systematic mapping study. *Empirical Software Engineering*, September 2012.

[13] Martín Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 341–350. IEEE, 2015.

[14] Martín Dias, Stéphane Ducasse, Damien Cassou, and Verónica Uquillas-Gómez. Do Tools Support Code Integration? A Survey. *The Journal of Object Technology*, 16(2):2:1, 2016.

[15] M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[16] Free Software Foundation. Gnu Affero General Public License. `https://www.gnu.org/licenses/agpl-3.0.en.html`. [Online; accessed 01-Aug-2016].

[17] Gerrit team. Gerrit. `https://www.gerritcodereview.com/`. [Online; accessed 21-Jul-2016].

[18] GitHub. About github. `https://github.com/about`. [Online; accessed 01-Aug-2016].

[19] GitHub. Github's features. `https://github.com/features`. [Online; accessed 21-Jul-2016].

[20] Google. Google web toolkit (gwt) uses gerrit. `http://googlewebtoolkit.blogspot.com/2013/07/gwt-news.html`. [Online; accessed 21-Jul-2016].

[21] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java®Language Specification: Java SE 7 Edition*, 2011.

[22] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[23] Omar S. Gómez, Natalia Juristo, and Sira Vegas. Understanding replication of experiments in software engineering: A classification. *Information and Software Technology*, 56(8):1033–1048, August 2014.

[24] L. Harjumaa, I. Tervonen, and A. Huttunen. Peer Reviews in Real Life - Motivators and Demotivators. pages 29–36. IEEE, 2005.

[25] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 121–130. IEEE, 2013.

[26] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101. ACM, 2014.

[27] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Hey! Are You Committing Tangled Changes? In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 262–265, New York, NY, USA, 2014. ACM.

[28] Sami Kollanus and Jussi Koskinen. Survey of software inspection research. *The Open Software Engineering Journal*, 3(1):15–34, 2009.

[29] Microsoft. Codeflow. `http://news.microsoft.com/2012/01/05/a-bar-an-idea-and-a-garage-the-story-of-codeflow/`. [Online; accessed 21-Jul-2016].

[30] 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 (TOSEM)*, 11(3):309–346, 2002.

[31] Hoan Anh Nguyen, Anh Tuan Nguyen, and Tuan N. Nguyen. Filtering noise in mixed-purpose fixing commits to improve defect prediction and localization. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 138–147. IEEE, 2013.

[32] Phacility, Inc. Phabricator. `https://www.phacility.com/phabricator/`. [Online; accessed 21-Jul-2016].

[33] Roger S. Pressman. *Software engineering: a practitioner's approach*. McGraw-Hill series in computer science. McGraw Hill, Boston, Mass, 5th ed edition, 2000.

[34] Peter C. Rigby, Daniel M. German, Laura Cowen, and Margaret-Anne Storey. Peer Review on Open-Source Software Projects: Parameters, Statistical Models, and Theory. *ACM Transactions on Software Engineering and Methodology*, 23(4):1–33, September 2014.

[35] F. Shull and C. Seaman. Inspecting the History of Inspections: An Example of Evidence-Based Technology Diffusion. *IEEE Software*, 25(1):88–90, January 2008.

[36] Yida Tao, Yingnong Dang, Tao Xie, Dongmei Zhang, and Sunghun Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 51. ACM, 2012.

[37] Yida Tao and Sunghun Kim. Partitioning Composite Code Changes to Facilitate Code Review. In *Proceedings of the 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 180–190. IEEE, May 2015.

[38] The Eclipse Foundation. Eclipse Java development tools (JDT). `https://www.eclipse.org/jdt/`. [Online; accessed 26-November-2015].