Automatic Ordering of Code Changes for Review

Version of June 26, 2018

Enrico Fregnan

Automatic Ordering of Code Changes for Review

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Enrico Fregnan
born in Gavardo, Italy

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
`www.ewi.tudelft.nl`

Automatic Ordering of Code Changes for Review

Author: Enrico Fregnan
Student id: 4623932
Email: eafregnan@gmail.com

Abstract

Code review has been proved to be an extremely important practice to ensure software quality. In recent years, the trend has moved towards modern code review, a lightweight and less strict paradigm. Despite its many advantages, this approach still has room for further improvement, especially in the area of cognitive support for reviewers.

Previous research stated how ordering code changes based on their relations may constitute an effective way to support reviewers. Based on this premise, this work focuses on studying how this ordering theory may be applied in practice. As result, a tool that automatically orders the modifications in a commit has been created.

Moreover, the tool has been tested and an initial investigation of the perceived usefulness of its relations has been conducted. Finally, it has been investigated if the ordering produced by the tool is identified as useful by the developers and which factors may influence this choice.

Thesis Committee:

Chair:	Dr. A. Zaidman, Faculty EEMCS, TU Delft
University supervisor:	Dr. A. Bacchelli, Department of Informatics, University of Zurich
External supervisor:	M.Eng. T. Baum, Fachgebiet Software Engineering, Leibniz Universität Hannover
Committee Member:	Dr. A. Bozzon, Faculty EEMCS, TU Delft
Committee Member:	Dr. G. Gousios, Faculty EEMCS, TU Delft

Preface

My Master's studies have been an amazing and enriching experience. During these two years, I had the chance to grow not only as a student, but also as a person. As a conclusion of this incredible journey, I am happy to present my Thesis project, conducted in fulfillment of the Master of Science in Computer Engineering.

I would like to thank my supervisor Alberto Bacchelli for his guidance during this project. Having the possibility to work in his group sparked my interest in doing research. Therefore, I am thrilled to have received the opportunity to join his amazing team. Furthermore, I want to thank my external supervisor Tobias Baum for his constant help. His deep knowledge of the subject and his precious feedback helped me greatly in conducting this research work. I want to thank Dr. Andy Zaidman, Dr. Alessandro Bozzon and Dr. Georgios Gousios for having accepted to be part of my thesis committee.

A huge thanks to my family and friends. To my parents for having given me the opportunity to conduct my studies abroad and their presence. To my sister Irene and all my friends for their incredible support during these years. Among them, a special thanks goes to Alessandra, Davide, Anna, Alessio and Lidia, who have always been there for me despite the many kilometers between us.

A special mention goes to Lorenzo for his feedback during the whole project and his patience in reading the drafts of this work. Finally, I would like to thank all the people who participated in the interviews and the online survey. Their help has been fundamental for the completion of this work.

Enrico Fregnan
Delft, the Netherlands
June 26, 2018

Contents

List of Figures

Chapter 1

Introduction

The main focus of this work is to improve the support to the developers during code review. To this aim, we implemented and tested a tool that relies on an ordering theory among code changes. Moreover, we conducted an initial assessment of its usefulness.

This chapter introduces the problem investigated, the research method followed (thesis statement, research questions and research steps) and the thesis outline.

1.1 Problem Description

Code review has been proven fundamental to assess the quality of code [1] [8]. In recent years, the trend has moved from the traditional code Inspection [19] [20], a strict and rigidly regulated procedure, towards modern code review. These new techniques present less strict paradigms, which can be easily integrated with the needs of a specific development team, and currently possess a vast popularity [40] [7]. However, the actual approaches still allow further improvements, especially in terms of cognitive support for the reviewers.

As assessed by LaToza et al., understanding the rationale behind a code change is a major problem for code review [33]. During a survey conducted among a group of developers, 66% of the interviewees reported it as a serious issue, while 56% of them reported issues in understanding code that somebody else wrote. Moreover, a better comprehension of the elements under review may increase the effectiveness of the review process [5] [16]. These findings have also been supported by Bacchelli and Bird [3], based on their interviews with Microsoft developers, and by Tao et al. [46], who conducted an online survey involving 180 software engineers in the same company.

To increase the cognitive support for the developers during code review, Baum et al. proposed to present the code changes to be reviewed in a meaningful way, based on the relations among different code entities [8]. In fact, actual tools such as GitHub present the list of changes ordered alphabetically at file-level, and by line number inside the same file. However, this order is often considered sub-optimal by software professionals, especially when dealing with large code changes [5]. The importance of ordering code changes as a mechanism to improve review has also been mentioned in the research conducted by Dunsmore et al. [17]: the interviewed developers identified code ordering as a means to improve code inspection. Following a theory-generating methodology [44], Baum et al. derived a set of principles and a formal theory to order code changes combing data from multiple sources (e.g., log data from tool-based review sessions and interviews with developers) [8].

The aim of our work is to implement this ordering theory in a tool and produce an initial assessment of the usefulness that this approach may have in practice.

1.2 Thesis Statement

In this work, we aim to investigate the following hypothesis:

A tool that automatically orders code changes in a commit, based on one or more relations that they share with each other, leads to results useful to the developers to perform code review.

To validate this claim, we developed a tool that automatically orders code changes based on the relations between them. We investigated the developers' perception of the usefulness of the relations implemented and tested the correctness of the order produced by the tool. Moreover, we evaluated the usefulness of this order compared to other possible orders of code changes to perform code review of a selected commit.

1.3 Research Method

In this section, the methodology of our investigation is described. We present the research questions and the steps followed.

1.3.1 Research Questions

The main goal of this work is to develop a tool to automatically order code changes in a commit, according to the formal theory proposed by Baum et al. [8], and perform an initial assessment of its usefulness. To investigate our thesis, we developed 3 research questions.

First of all, we need to evaluate how this ordering theory may be implemented in practice. Therefore, our first research question is:

RQ1 *Is it possible to implement the ordering formal theory in a tool?*

This first exploratory question aims to investigate if the ordering theory may be effectively integrated in a tool. To answer it, we need to identify the ideal instruments to analyse code changes and extract relations among them.

Once the tool has been built, we need to evaluate the correctness of the order produced. For this reason, we formulate our second research question as:

RQ2 *How can the tool correctness be evaluated and with which results?*

With this question the goal is to check the operations of the tool to verify their correct implementation, control the constructs covered and, possibly, identify further relations that need to be included. To answer it, two test sets have been manually created: the former has been produced by the tool developer, while the latter has been created by external developers. Each of these sets is formed by commits manually ordered according to the ordering theory, using one or more relations supported by the tool.

Finally, we aim to understand the impact that the ordering produced by our tool can have in performing code review tasks. For this reason, the final research question is:

RQ3 *How useful is the tool ordering to perform code review?*

To answer this question, we developed an online survey where a group of developers was asked to select the most useful ordering of code changes to review a given commit. Our aim is to investigate if the order created by the tool is chosen as preferable over other orders. Moreover, factors that may influence this choice are also investigated: e.g., size of the commit or the relation used.

1.3.2 Research Steps

To answer our research questions, we followed these research steps:

1. **Implement a tool to automatically order the changes in a commit**: To implement the tool referred in RQ1, we decided to work using Java. The choice of this programming language was motivated by two main factors: the possibility to easily integrate code already developed by Baum et al. [8] in the context of their research and the availability of libraries and tools to extract and analyse code changes. In particular, we used JGit[1] to extract the modifications from a given commit and JavaParser and Java Symbol Solver[2] to solve the dependencies among different code portions.

2. **Create a test set**: To answer RQ2, a set of Java-based Open-Source projects has been selected. For the internally produced test set, 10 different projects were used. From each of them, 10 different commits were manually ordered using each of the relations supported by the tool and compared against the tool output. The externally produced test set was created by a group of interviewed Java developers. Each of them was asked to produce three different orders using a project on which they worked or an Open-Source project given as backup option, if they could not choose a project of their own.

   At the same time, we investigated which of the relations implemented in the tool were perceived as useful by the developers and which, on the contrary, were not considered to lead to orders significant for review.

[1]JGit library: https://www.eclipse.org/jgit

[2]JavaParser: http://javaparser.org; Recently JavaParser directly includes Java Symbol Solver.

3. **Evaluate the tool usefulness**: Finally, an investigation of the tool usefulness has been conducted through an online survey (RQ3). Every participant was asked to answer eight questions: in each of them four different orders were presented and the respondent had to select the one that he or she would have used to review the given commit. Among them only one was the optimal order produced by the tool. Furthermore, commits of different sizes and different relations (or combinations of them) were used.

1.4 Thesis Outline

This thesis work is organized as follows: Chapter 2 contains a presentation of the Code Review and Cognitive Support approaches developed in the software engineering community, together with a brief explanation of the formal theory developed by Baum et al. [8]. Chapter 3 contains a presentation of the tool and the choices made during its development. Furthermore, it shows how the internally produced test set has been created. Chapter 4 presents two examples of how the tool works in practice. Chapter 5 includes the explanation of the survey conducted to create the external test set and to evaluate the usefulness of the tool. Finally, Chapter 6 reports the contributions of our work, together with a discussion of the results and ideas for future work.

Chapter 2

Background and Related Work

Code review is a well-explored domain in software engineering. This chapter discusses previous methodologies developed to perform it, together with a brief explanation of the formal ordering theory on which the tool is based.

The chapter is divided into four sections: "*Code Review*", "*Modern code review*", "*A theory to order code changes*" and "*Other cognitive support approaches*". In the first section, the classical methods to perform code review are presented. In the second one, the new paradigm of modern code review is explained, together with the reasons behind its introduction. Section 3 contains a description of the main terminology and the theory used in this work. Finally, section 4 presents an overview of other techniques developed to increase the cognitive support to developers during code review.

2.1 Code Review

Code review is a fundamental and well-assessed method to ensure software quality [7]. It has been formalised by Baum et al. [6] as "*a software quality assurance activity*" with the following properties:

- The main checking is done by one or several humans;
- At least one of these humans is not the code's author;
- The checking is performed mainly by viewing and reading source code;
- It is performed after implementation or as interruption of implementation.

Finally, the people who perform this task are defined as *reviewers*.

Three major types of code review processes emerge from the literature [28] [42]: *Code Inspection* [19] [20], *Technical Review* [22] [47] and *Structured Walkthrough* [49].

Code Inspection has been formalized by Fagan [19] [20]. This highly structured approach is conducted in meetings and based on line-by-line reviews. Its usefulness has been well-assessed through the years. In particular, this approach revealed itself useful to find bugs and defects in the code [43]. Furthermore, it also has benefits on software quality, predictability and information on development operations [28]. In Fagan's Inspection, a moderator is in charge of the creation of a team (recommended team size of four people) and of checking that the object of the review meets a number of entrance criteria [15]. Then, the process can be organised in six phases: planning, overview, preparation, inspection, rework and follow-up. In the first step a meeting is planned, then in the second phase an introductory meeting takes place: roles (author, reader or tester) are assigned and documents produced and distributed. In the preparation phase, the reviewer is prepared to perform the task that takes place in the subsequent inspection phase. Finally, in the last two steps of the process the errors are corrected (rework) and a check on the quality of the corrections is done, together with an assessment on the need of further inspection (follow-up phase).

Technical Review also presents among its benefits the detection and correction of errors, together with a reduction in maintenance costs [28]. Technical review is conducted by a group of reviewers not part of the producing unit of the work under analysis and their task is to address potential technical issues in the project [47]. Reports of these reviews serve to guarantee that the code under analysis respects the specifications for which it was designed. Some of the documents generated after a review are meant to report its outcomes to the management, others contain lists of the issues found and that need to be resolved. Like Fagan's Inspection, this approach is based on meetings too.

Structured Walkthrough's purpose is to analyse the work under different perspectives. Its benefits are error detection and standards establishment [49]. A walkthrough is a peer group review of a product (in this case, a software) based on formal or semi-formal meetings, preceded and followed by preparations and revision activities. Walkthroughs analyse the product with the main goal to validate its general approach; this may lead to focus less on finding specific errors [47].

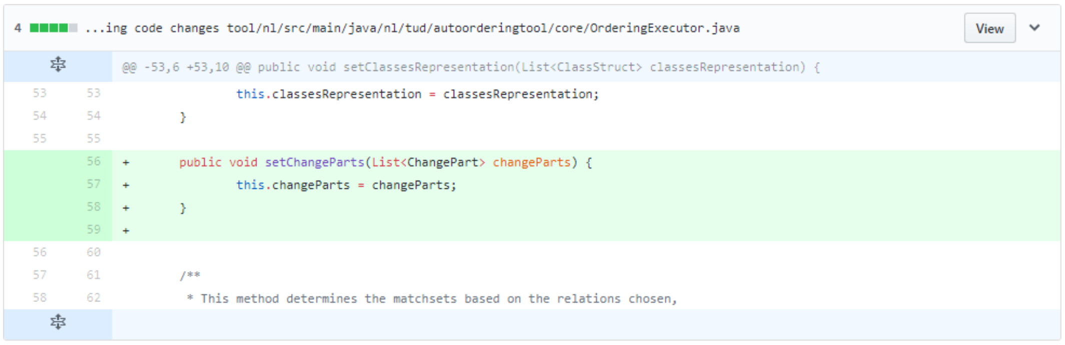
Fagan's Inspection has been the object of further investigation to identify possible alternatives. Many different possibilities have been developed, such as Two-Person Reviews [11], N-Fold Inspection [35], Phased Inspections [30] and Verification-Based Inspection [18]. However, all these solutions require face-to-face review meetings.

These meetings are costly, but they do not increase significantly the defect detection effectiveness [26]. An initial attempt to remove them has been done by Parnas and Weiss [36]: their approach, called Active Design Review, tries to minimize the impact of meetings in the review process. Nonetheless, meetings still constitute an important part in their method.

2.2 Modern code review

In recent years, the trend in this field has moved from classical code review techniques towards the application of modern code review (also known as "*change-based review*" [5], "*differential code review*" [9] or "*patch review*" [6]). This new approach to code review is characterized by less strict practices that allow modifications in the process to fit the needs of a specific development team [39]. In fact, its main characteristics are being informal and tool-based [3].

Peer review is often used in Open-Source projects development. In fact, it represents the



```
4 ...ing code changes tool/nl/src/main/java/nl/tud/autoorderingtool/core/OrderingExecutor.java        View

@@ -53,6 +53,10 @@ public void setClassesRepresentation(List<ClassStruct> classesRepresentation) {
                this.classesRepresentation = classesRepresentation;
        }

+       public void setChangeParts(List<ChangePart> changeParts) {
+               this.changeParts = changeParts;
+       }
+

        /**
         * This method determines the matchsets based on the relations chosen,
```

Figure 2.1: A change part as shown in GitHub

way in which OSS (Open Source Software) developers can set rules about what constitutes a good contribution valid for the whole community [40]. The process starts when a developer submits a patch (e.g., a commit). Then, the reviewers read through the modifications and comment them. Based on this feedback, the person who introduced the changes can improve the code. Once it reaches the community standard, the modifications are committed to the main code base of the project [2] [40]. Two main review styles can be applied: Review-Then-Commit (RTC) or Commit-Then-Review (CTR). The former is the style expressed in the steps mentioned above. The latter allows trusted developers to commit changes before they are reviewed.

Furthermore, a vast amount of tools exists to support change-based code review (both general-cases and specialized tools) [5]. Companies sometimes develop their own internal tools: e.g., Microsoft uses CodeFlow, a tool which operates in a fashion similar to RTC [40]. Other examples can be found in the Open-Source tool Gerrit[1] or Facebook's Phabricator [21].

Rigby and Bird analyzed different code review processes on a set of projects (both Open-Source and private) to identify possible common features [40]. They confirmed that the general trend is evolving towards a lightweight and flexible process. This shows a marked contrast with classical software inspection: a procedure with strict steps and rigid roles of the people involved. Furthermore, their study revealed that review tends to happen early in the development process, quickly and frequently and that the whole process has changed from being a bug hunt to a group problem solving activity. Moreover, it also involves knowledge transfer and team awareness [3]. Finally, the use of code review tools, compared to other means such as emails, provides advantages in terms of traceability. The success of modern code review techniques can be derived from their wide adoption.

2.3 A theory to order code changes

In this work, we relied on definitions extracted from previous work on ordering code changes for review [8]. For this reason, they are briefly explained in the following section.

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

Table 2.1: ORDERING PRINCIPLES

Principle	**Description**
Principle 1	Group related change parts as closely as possible.
Principle 2	Provide information before it is needed.
Principle 3	In case of conflicts between principles 1 and 2, prefer principle 1 (grouping).
Principle 4	Closely related change parts from chunks treated as elementary for further grouping and ordering.
Principle 5	The closest distance between two change parts is "*visible on the screen at the same time*".
Principle 6	To satisfy the other principles, use rules that the reviewer can understand. Support this by making the grouping explicit to the reviewer.

- **Change part**: a change part is a portion of code that has been changed in the new version of the file committed compared to the old version. A change part can be constituted by the addition, removal or modification of the code in the file. Fig 2.1 shows an example of an added change part as displayed by GitHub. In this case, the change part is the portion of code contained in the green area. It has a beginning line number (56) and an ending line number (59).
- **Relation**: a relation exists between two or more change parts that share a link or a property with each other. Recalling the definition given by Baum et al.: "*A relation consists of a type (e.g. call flow, inheritance, similarity) and an ID that allows to distinguish several relations of the same type (e.g. the name of the called method)*" [8].
- **Match set**: a group of change parts connected by a relation.
- **Tour**: an ordering among the change parts.
- **Review efficiency**: the number of defects retrieved for every hour spent reviewing [8].
- **Review effectiveness**: the ratio between the defects found over all the defects in the code change [8] [10].

Furthermore, Baum et al. formalized a set of principles that leads an order to have a higher review effectiveness and efficiency compared to other orders. These principles are reported in Table 2.1.

Based on these principles, a formal theory to order code changes has been created for code review. Its main goal is to define a partial order ($\geq_T \subseteq Tour \times Tour$) among tours such as the first one is better then the second in terms of review utility. Two important propositions must hold:

$$\begin{aligned}\forall t_1, t_2 : t_1 \geq_T t_2 \Rightarrow (&reviewEfficiency(t_1) \geq reviewEfficiency(t_2) \wedge \\ &reviewEffectiveness(t_1) \geq reviewEffectiveness(t_2))\end{aligned} \tag{2.1}$$

$$\exists t_1, t_2 : t_1 \geq_T t_2 \wedge \neg(t_2 \geq_T t_1) \Rightarrow reviewEfficiency(t_1) > reviewEfficiency(t_2) \tag{2.2}$$

Proposition 2.1 states that if a tour is better than another in the partial order, it can not be worse in terms of review efficiency and effectiveness. Furthermore, proposition 2.2 adds strength to this concept stating that if there is a tour that is better than another in the order ($\geq_T$), it is also better in terms of review efficiency.

$\geq_T$ has a parametric definition based on a set P of grouping patterns and the part graph g. A grouping pattern captures the ordering preferences of a reviewer. It is formed by a matching rule (relation) to identify a subset of related change parts and a rating function (*rate*) to evaluate the permutation of the matched change parts. The definition of $\geq_T$ is also based on the 'match set' (MS) helper construct. It is formed by all the occurrences of a grouping pattern in a tour. The definition of $\geq_T$ is given by the following relation:

$$\begin{aligned} t_1 \geq_T t_2 \Leftrightarrow & mS(t_1, g) \geq_T mS(t_2, g) \\ \Leftrightarrow & mS(t_1, g) \supset mS(t_2, g) \vee \\ & (mS(t_1, g) = mS(t_2, g) \wedge \\ & \forall m \in mS(t_1, g) : rate(m, t_1) \geq rate(m, t_2)) \end{aligned} \tag{2.3}$$

Proposition 2.3 states that a tour is better than another when it has more matches or the same matches with higher ratings (its MS is better). *mS* is a recursive function, $mS : Tour \times PartGraph \rightarrow MS$, defined as:

$$mS(t, g) := \bigcup_{p \in P} \Big(pM(p, t, g) \cup \bigcup_{m \in pM(p, t, g)} mS(shrink(t, g, m.v)) \Big) \tag{2.4}$$

In definition 2.4, *pM* stands for patternMatches: the matches found for a pattern in a tour. *shrink* is a function that takes as parameters a tour, its graph and a list of change parts belonging to a match set and replaces all these change parts with a composite part, creating a new tour.

2.4 Other cognitive support approaches

Apart from the ordering theory discussed in the previous section, other techniques to improve the cognitive support for the developers have been proposed by the software engineering research community during the years.

A first approach to the problem relies on clustering related code changes. This approach divides in groups the code changes in a commit, based on the relations that they share with each other. To this aim, Barnett et al. proposed CLUSTERCHANGES [4], a tool that clusters similar code regions based on the *def-use* relation: any use of a software entity (types, fields methods and local variables) is mapped to its correct definition. A different approach is the one implemented in EpiceaUntangler [14], a tool that exploits machine learning techniques to identify if two modifications belong to the same cluster. The set of features used in this approach includes the analysis of code structure, its content, the distance between two changes and the variables accessed. A similar work has been conducted by Kreutzer et al.: they introduced *C3* (Clustering of Code Changes), a fully automated approach to cluster similar code changes [31].

Clustering of similar code regions has been done also with semantic analysis. Maletic and Marcus [34] proposed a method to cluster software entities based on LSI (Latent Semantic Indexing): a machine learning model developed to analyze relations among words and documents [13]. Their approach was further refined by Kuhn et al., who developed a tool called Hapax to cluster software entities based on their semantic similarity [32]. Hapax works at different levels: systems, classes and methods.

A different way to increase the cognitive support for the reviewers is to untangle code changes. In fact, a change may involve many different portions of code and bundle unrelated modifications together: this type of change is often referred to as "*chunky change*" or "*code bomb*" [45]. It constitutes an obstacle in the code review process [29]. For this reason, different approaches have been proposed to solve this issue.

A technique to untangle code changes in a commit has been investigated by Kirinuki et al. [29]. It relies on change patterns to suggest to the developers code changes that may be tangled, so they can consider to divide them in different commits. A similar work has been conducted by Herzig and Zeller [24]. They proposed a heuristic-based algorithm to untangle code changes. For every pair of change operations, the algorithm decides if they belong to the same partition (are related) or to different ones (are not related). To take this decision, it relies on a set of metrics that capture the dependencies among them: e.g., file distance or change coupling [51]. The solution presented by Tao and Kim also relies on a heuristic to group similar code changes [45]. In particular, two code changes are identified as related if they are formatting-only changes, they have static dependencies (computed using program-slicing) or they have similar patterns. Herzig et al. developed a heuristic-based untangling algorithm that creates partition sets of modifications related to each other from a bigger change set [25]. It is fully automatic and based on static code analysis.

A different approach to the problem has been presented by Kawrykow and Robillard, who focused on identifying non-essential changes (e.g., trivial types updates and local variables renames) in a commit [27]. To this aim, they developed a tool called DIFFCAT. It works comparing the two versions of the AST (Abstract Syntax Tree) related to a change: the one from the file before the modification and the one after the modification. From their comparison, DIFFCAT is able to detect non-essential changes.

Finally, Zhang et al. presented CRITICS, an approach to inspect changes for review based on the data and flow context [50]. This tool gives to the user the possibility to visualize changes with a similar context. Its aim is to help developers to identify missing or inconsistent updates.

Chapter 3

An ordering tool

The main goal of this work is the development of a tool, called **CodeChangeOrderer**, to automatically order the modifications contained in a commit. During its construction, different challenges had to be faced: e.g., how to create groups of related change parts or the selection of the relations supported by the tool. Furthermore, the options made available to the user had to be analysed to understand if they could lead to interesting results.

3.1 CodeChangeOrderer structure and implementation choices

The tool, written in Java, works using a local clone of the GitHub project repository under analysis and the unique code associated to the selected commit. The Java library JGit is used to analyse the commit and extract the code diff and the related modifications.

To match the different change parts, CodeChangeOrderer needs to analyse the entities contained in the code. To perform this task, it relies on a parser and a symbol solver. This choice required to restrict the scope of our tool to a particular programming language, therefore our tool works only on Java files. This choice was motivated by the popularity of Java as a programming language and the vast availability of Open-Source projects based on this language, against which the tool output can be tested. To analyse the code and resolve the relations among the different elements, we relied on a language parser. In particular, we selected JavaParser and Java Symbol Solver. The reasons behind this choice were: the open-source license under which JavaParser is released, its simple architecture and the active community that works on it. We argue that this last criterion is important because it may allow the tool to be further expanded in the future with new relations. To solve the dependencies in the code, Java Symbol Solver requires either the path of all the project repositories or the JAR files associated to them. The current tool implementation relies on the second option. This choice was motivated by the significant time overhead necessary to compute all the project directories and resolve the dependencies in them compared to the solution that relies on the JAR files. Furthermore, the use of the JAR files allows the tool to solve dependencies with files belonging to external libraries: these can not be resolved using the project local directories since external files are not contained in them.

We decided to limit the scope of our analysis and the order produced to change parts level, without extending it to the single lines of code changed. In fact, we argue that such a fine granularity order is counter-productive to help the developers during code review: reordering lines of code inside the same change part only makes the code difficult to understand. Consequently, it may decrease the cognitive support for the reviewers.

3.1.1 Relations

To select the relations to implement in the tool, the set proposed by Baum et al. has been analysed [8]. In their research, the authors reported a group of relations emerged from their interviews with the developers. The idea of these relations is to specify the concept of *relatedness*, which was identified as fundamental principle on which the change parts must be ordered. The results of their investigation have been reported in a table (presented in the Appendix as Table A.1), indicating the relation name, if it is directed and a brief description.

Analysing these relations, the aim was to step from the general formulation with which they are presented to a more specific definition. In fact, only restricting their scope in terms of code constructs (and specifically Java constructs), it was possible to implement them in the tool. From the list presented in Table A.1, the “*Similarity*” relation was excluded from our investigation due to its excessively broad spectrum: in fact, the concept of similarity was not explicitly defined by the developers and, therefore, open to multiple interpretations. For the same reason, we also excluded the *Common Identifier* relation. The *Logical dependency* relation may constitute a significant contribution to help the code review process. However, since it matches code changes that are likely to be modified together, it needs data from the Software Configuration Management (SCM) or from a tool that tracks the developers' activities in the IDE. Therefore, it was outside the scope of this research, since our tool has been designed to be independent from developing environments or other code review tools. The same reasoning also applies to the last relation reported in the table, the *Development flow* relation, which aims to follow the development flow of the programmer.

Starting from these general relations, a new set of relations has been created to be implemented in the tool. The main idea behind it was to split some of the general relations (e.g. *Data flow* or *Call flow*) indicated in the research conducted by Baum et al. [8] to cover the different constructs available in Java. The resulting set of relations is shown in Table 3.1.

Table 3.1: TYPES OF RELATIONS AMONG CHANGE PARTS

Name	Description
Same file	Change parts belonging to the same file are considered as related.
Same format	Change parts belonging to files having the same format are considered as related.
Inheritance	Change parts of a class are considered related to the ones belonging to its parent class.
New object	A change part that instantiates a new object is considered as related to the change parts belonging to the class of the created object.
Method call	A change part that calls a method is considered as related to the change parts belonging to the method's definition.

Declare-use	A change that contains a local variable use is considered as related to the change part containing this variable declaration.
Field access	A change part that accesses a class field is considered as related to the change part containing the field declaration.
Parameter-use	A change part containing a parameter use is considered as related to the one containing the parameter declaration.

The relations *File Order* and *File Type* from Table A.1 have been reported as *Same file* and *Same format* without any modification. In fact, these two relations constitute a special case in our set of relations since they do not rely on the analysis of Java code. Therefore, JavaParser and Java Symbol Solver functions are not applied to produce them. For this reason, they can be applied to all kinds of files in a commit.

The *Class Hierarchy* relation has also been implemented in the tool (called *Inheritance* relation), relying on JavaParser to solve the parent-child connection among the classes containing the change parts. However, it is important to underline a difference in our definition compared to the one proposed by Baum et al. [8]. On the one hand, their definition states that two modified classes are linked only if a method overrides a method defined in the parent class. On the other hand, our definition states that two classes (a parent and a child class) are linked by the inheritance relation if both of them have been modified in the considered commit: in other words, if both of them contain at least a change part. Therefore, our relation has a more general scope that the one presented in Table A.1.

The *Data flow* and *Call flow* relations have been divided in two relations that reflect Java language constructs: method call and new object. The definition of the *Method call* relation reflects the one indicated by Baum et al. for the *Call flow* relation. The *New object* relation reflects instead the instruction to create new objects, available in Java and that we identified as worthy to be included due to its fundamental importance in the language. In fact, we argue that an instruction that allows the interaction between different objects should be regarded as important in an object-oriented paradigm.

The general *Declare-use* relation, as defined in Table A.1, has also been covered in the tool. However, it has been divided in three relations, *Declare-use*, *Parameter-use* and *Field access*. The first covers the case of variable declaration and use, but not the cases in which the variable is a method's parameter or a class field, which are instead covered by the *Parameter-use* and *Field access* relations, respectively. The reasons behind this choice were two: the way in which JavaParser treats these entities and giving more freedom to the user. Regarding the former reason, JavaParser resolves a parameter, a local variable and a field declaration (in the meaning used in our relation) as different objects. This encouraged us to consider the creation of three separate relations. The latter reason, offering more choices to the user, was the main reason behind our implementation decision. In fact, we argue that leaving more freedom to the user in selecting different relations constitutes a significant advantage to increase the cognitive support offered by the tool. He or she can choose to use the relations separately or to combine them. Selecting all three of these relations at the same time leads back to the original *Declare-use* definition from Table A.1.

The tool works with one relation at a time or with multiple relations together. It constructs the related match sets for all the relations selected and produces an ordering that respects

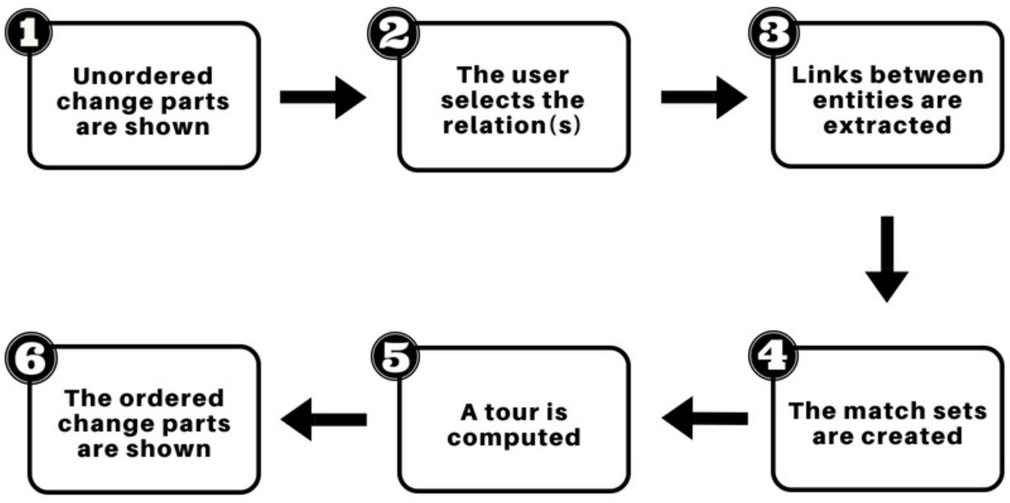

1
Unordered change parts are shown
2
The user selects the relation(s)
3
Links between entities are extracted
4
The match sets are created
5
A tour is computed
6
The ordered change parts are shown

Figure 3.1: Steps followed by CodeChangeOrderer

all the constraints. In case of conflict between two or more relations (or match sets of the same relation), the tool produces a tour with the principle of grouping as close as possible related change parts (Principle 1 in Table 2.1). However, it may not be possible to order them immediately next to each other, as it happens in a scenario without conflicts.

3.2 Tool functionality

In this section, CodeChangeOrderer operating principles are explained. First, the steps followed in the basic tool functioning are shown. Then, further options implemented in it are explained. Moreover, the graphical user interface developed for the tool is briefly presented. Finally, CodeChangeOrderer limits are reported together with the motivations behind them.

3.2.1 Steps of the tool's algorithm

The first time the user launches the tool, CodeChangeOrderer asks for the project path and the identifier of the commit to review. It saves this information to show immediately the last project and commit used when the user opens the tool a second time. Once the GUI is displayed, the user can decide to change the commit or the project under analysis: in this case, the list of change parts shown is updated accordingly.
To produce an optimal ordering of a set of change parts, CodeChangeOrderer operates according to the following steps (an overview of them is presented in Figure 3.1):

1. Using the JGit library, the diff of the commit under analysis is extracted and the change parts contained in it are shown. These are displayed ordered in the default

fashion of GitHub: the different files are ordered based on the alphabetical order, while inside the same file change parts are ordered based on the line number.

2. The user can select the relation (or group of relations) that he or she wants to apply to produce the order. Once he or she is satisfied with the relations choice, he/she can confirm the selection. The list of the relations currently supported by CodeChange-Orderer has been presented in section 3.1.1. Albeit all relations may be selected, the tool does not allow the user to choose the same relation more than once: even if this option was enabled, it would not lead to an order different than the one produced using the relation only once.

3. Based on the relation(s) selected, the links between software entities are extracted using JavaParser and Java Symbol Solver. To avoid an unnecessary time overhead, only the code elements needed to create the match sets for the chosen relation(s) are extracted. This saves time avoiding an overuse of the symbol solver. It is important to notice that, if the user selects only the *Same file* relation or the *Same format* relation, this step is skipped since no Java instructions need to be analyzed to group together change parts based on these criteria.

4. The match sets are created: the tool analyses the elements extracted in the previous step to identify change parts that are related to each other and it clusters them together in the same group. If more than one relation has been selected, the match sets for all the relations are constructed. Therefore, the same change part will appear in more than one match set.

5. Based on the match sets created, a tour is computed. To do so, all the change parts contained in the commit are ordered to respect the relation constraints. When no further constraints apply, the order of the different change parts is determined by the line number with which they begin.

6. Finally, the ordered modifications are shown to the user, who can now use the produced order to perform code review of the selected commit. Then, he or she is free to select a different set of relations to produce a new order to keep reviewing the same commit, change it or close the tool.

These steps give an overall description of the procedure followed by CodeChangeOrderer. However, the algorithm used to extract and order the change parts has been described only superficially. For this reason, the last four steps (from 3 to 6) are now explained in more detail.
The algorithm starts from step 3, where the Java files contained in the commit are analysed and their properties (e.g., methods declared, methods called, variables declared etc.) are saved in data structures: each of them corresponds to a class. Only code elements contained in the portion of code modified in the commit are considered.
To create the match sets (step 4), each change part is associated to the above-mentioned data structure related to its Java file. The creation of these links allows the tool to compute the relations among different classes and, after that, trace back the correct change parts to

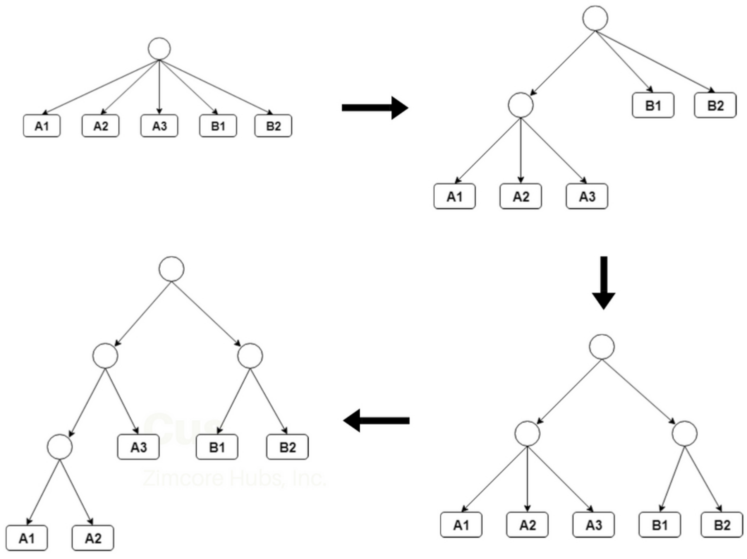


Figure 3.2: Example of the construction of the change parts tree by the algorithm based on three match sets {A1, A2, A3}, {B1, B2} and {A1, A2}.

create a match set. If a change part belongs to a file of a format different from Java, no link is created. In this case, this change part is grouped alone in a match set. When the *Same file* or *Same format* relations are chosen, CodeChangeOrderer does not need to link each change part to its correspondent data structure since the information contained in the change parts are enough to construct the match sets.
In step 5, the algorithm[1] relies on a tree structure: a data structure formed by nodes and edges without any cycle. A node without children is called leaf, while a node without parent is called root. Finally, a node that possesses both a parent and descendants is an intermediate node. The algorithm starts by creating a tree having only the root node and all the change parts as leaves. Then, it iterates over the match sets: for each of them, the algorithm finds the related change parts and adds in the tree an intermediate node having these change parts as leaves. To do so, the algorithm starts from the root and iterates over all the children of each node. Figure 3.2 shows this procedure considering an example with five change parts belonging to two files *A* and *B*. The *Same file* and *Declare-use* relation, assuming A1 and A2 as related, are used. Therefore, the tree is constructed using the match set {A1, A2, A3}, {B1, B2} and {A1, A2}.
Each intermediate node possesses a flag that indicates if its children can be reordered or not. This flag is set when the node is constructed, depending on the position of the leaves belonging to the match set under analysis. A node is non-reorderable when the children in the middle are linked to both the ones at top and at bottom: e.g, considering the two

[1]The algorithm used in this step is based on the one implemented in CoRT: https://github.com/tobiasbaum/reviewtool

match sets {A, B} and {B, C}, the algorithm creates a non-reorderable node having these three change parts as leaves. In fact, after the first match set has been considered, the tree has an intermediate node, with leaves A and B, and a leaf C connected directly to the root. This intermediate node is reorderable since both leaves A and B can be moved to comply with further match sets, if necessary. Considering the second match set {B, C} leads to the creation of a non-reorderable node: its leaves can not be moved anymore to not violate the match sets constraints.
CodeChangeOrderer keeps track of the match sets that have been satisfied and the ones that have not yet been satisfied at the end of this step. Then, the tool checks if it is possible to satisfy them by folding a group of change parts. Once this step is completed, the algorithm resolves the requests to position a certain change part before or after others in the same match set (used in the positioning and center options). To complete this operation, the algorithm iterates over the positioning requests and for each of them it finds the subtree containing the change parts belonging to the related match set. Then, it moves the target change part(s) in the selected position, creates a new subtree and substitutes it with the one found before.
Now CodeChangeOrderer returns the resulting tour, converting the tree constructed before in an ordered list. It traverses the whole tree from the root to the leaves. Once they are reached, they are added in a sublist. Then, the algorithm moves backwards, considering the sublist belonging to all the children of a node and merging them in a bigger list. If the node is reorderable, the position of a sublist in this ordered list is determined by the line number of its first element: the criterion of positioning first the change part with lowest line number is used to solve situations when no other constraints apply. On the contrary, if the node is non-reorderable, its sublists are merged respecting the order in the tree. This procedure is applied iteratively until the root is reached again. Finally, each change part in the ordered list produced by the algorithm is matched with the related modification. The resulting ordered list of modifications is shown to the user.

3.2.2 Further options

Alongside the basic operating cycle of the tool, three options have been implemented to allow the user to have more control on the produced order: the positioning, the "*Add unmodified code*" and the center options.

Positioning option: this option allows the user to select the position of a specific change part (or group of change parts) in a relation. This option exploits the concept of direction of a relation, allowing the user to decide in which way direct it. Therefore, it has been implemented only for directed relations: all relations presented in table 3.1, excluding *Same file* and *Same format*. Considering as example the *Method call* relation, the user can choose to position the change part containing the method call before or after the change parts belonging to the method declaration. Moreover, it is also possible to not specify a preference for this option. If that is the case, the tool decides which change part(s) to display first based on the line number.

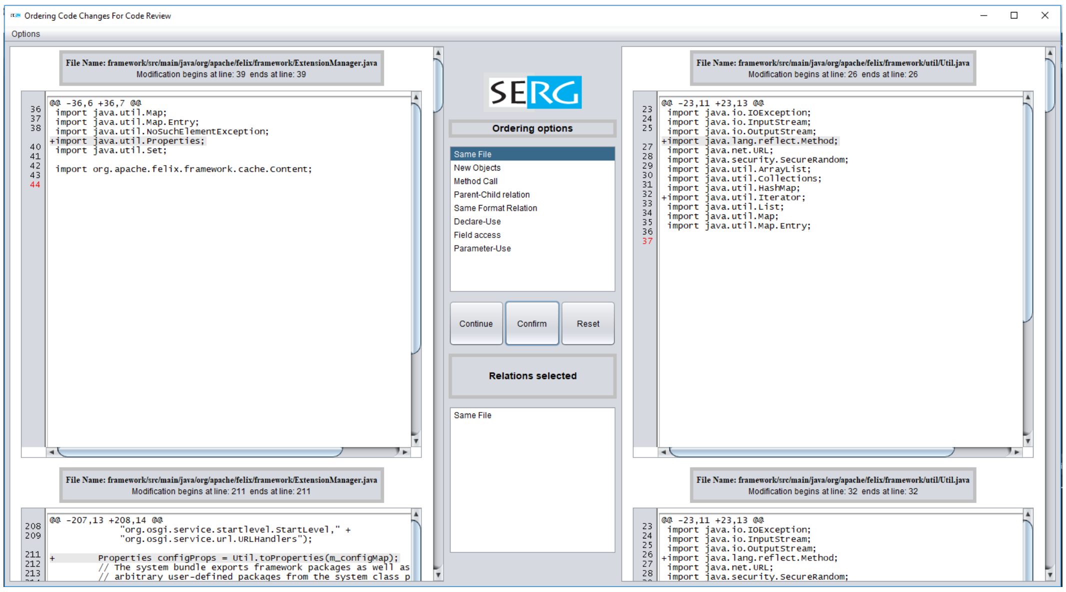
Add unmodified code option: this option allows the user to include files not in the commit

under review to the set of modifications to order. This option is available only for the *New object*, *Method call* and *Inheritance* relations, since the other relations either work at a level of granularity too fine (e.g., *Declare-use*) or do not involve files outside the commit (e.g., *Same file*). If this option is selected, JavaParser and Java Symbol Solver are used to analyse the content of the change parts (as in the normal procedure of the tool explained in Section 3.2). However, in constructing the match set, the tool will consider also references to classes not included in the commit. Subsequently, it will search for these unmodified related files in the cloned local directories of the project. If the files are found, they are added to the set of change parts to order: a file added in this way is treated as a single change part. Furthermore, line number zero is associated to it: in this way, if no other positioning options are specified, priority will be given to this file and they will be positioned as close as possible to the beginning of the ordering, depending on the constraints enforced by the relations. Attention has been given to the problem of identifying the correct class if the project contains classes with the same name.
If this option is selected for the *Method call* relation, then an extra step is done to identify the related unmodified method. To this aim, JavaParser is applied to find the right method declaration in the retrieved class. As result, the tool does not show the whole class, but only the implementation of the related method.
The searching operation among the project directories may introduce a significant time overhead. This constitutes a drawback of this option. Moreover, the tool is unable to retrieve files belonging to external libraries since they are not included in the project repositories.
We argue that this option may give a deeper understanding of the review context to a developer. As motivating example, consider the case in which a class presents a modification in a method inherited from its parent class, but that is not included in the commit. Having the possibility to immediately show to the user also the parent class code can save him/her the time necessary to manually retrieve it from the project codebase. A further limit of this option is that the number of new files added may become easily very large. In this case, the gain of this procedure is undermined, since the order produced may become excessively complex.

Center option: when the same change part is linked to more than one other change part by a relation, the tool offers three different options to decide its position. This option can not be applied to undirected relations (e.g., *Same file* and *Same format*).

- *Center in the middle*: with this option the change part connected to all the others is positioned in the middle of all the related change parts. In this way, it is positioned as close as possible to all the related change parts in the order. However, while being the option that satisfies the principle of grouping related change parts close to each other (Principle 1 in Table 2.1), we argue that it can be sub-optimal to give knowledge support to the reviewer. For this reason, the other two options were implemented.

- *Center first*: this option positions a change part (or more than one) before the ones to which it is related. Albeit it is less adherent to the "*position related change parts close together*" principle (refer to Table 2.1), we argue that this may lead to a gain in terms of clarity and usefulness for the review.



File Name: framework/src/main/java/org/apache/felix/framework/ExtensionManager.java
Modification begins at line: 39 ends at line: 39

```
@@ -36,6 +36,7 @@
 import java.util.Map;
 import java.util.Map.Entry;
 import java.util.NoSuchElementException;
+import java.util.Properties;
 import java.util.Set;

 import org.apache.felix.framework.cache.Content;
```

File Name: framework/src/main/java/org/apache/felix/framework/ExtensionManager.java
Modification begins at line: 211 ends at line: 211

```
@@ -207,13 +208,14 @@
            "org.osgi.service.startlevel.StartLevel," +
            "org.osgi.service.url.URLHandlers");

+        Properties configProps = Util.toProperties(m_configMap);
         // The system bundle exports framework packages as well as
         // arbitrary user-defined packages from the system class p
```

File Name: framework/src/main/java/org/apache/felix/framework/util/Util.java
Modification begins at line: 26 ends at line: 26

```
@@ -23,11 +23,13 @@
 import java.io.IOException;
 import java.io.InputStream;
 import java.io.OutputStream;
+import java.lang.reflect.Method;
 import java.net.URL;
 import java.security.SecureRandom;
 import java.util.ArrayList;
 import java.util.Collections;
 import java.util.HashMap;
+import java.util.Iterator;
 import java.util.List;
 import java.util.Map;
 import java.util.Map.Entry;
```

File Name: framework/src/main/java/org/apache/felix/framework/util/Util.java
Modification begins at line: 32 ends at line: 32

```
@@ -23,11 +23,13 @@
 import java.io.IOException;
 import java.io.InputStream;
 import java.io.OutputStream;
+import java.lang.reflect.Method;
 import java.net.URL;
 import java.security.SecureRandom;
```

Figure 3.3: The Graphical User Interface developed for the tool. A commit taken from the Felix project was used.

- *Center second*: in this option, the related change part is positioned after one of the change parts to which it is related, but before all the other connected change parts. The rationale behind this choice is to give to the user an example of method or variable use before showing the declaration and the other uses.

An example of how this option is applied in practice is presented in Section 4.

3.2.3 Graphical Interface

Since the aim of the tool is to produce an order that may be used by the developers, importance has been given to the construction of a suitable Graphical Interface.
As shown in Figure 3.3, it is composed of three main parts. On the left, trough a scroll box, the modifications presented in the commit are shown. Each modification is formed by a text box stating the file name and the starting and ending line number and by the related code. The order with which they are presented is the same used by GitHub: files in alphabetical order and code changes in the same file ordered by line number. The portion of code modified is highlighted in gray.
In the central part, the different relations that the user can select to produce an order are shown. Underneath them, three buttons are present: "*Continue*", "*Confirm*" and "*Reset*". The first option allows the user to confirm the choice of a relation and select more relations to use for the order: once the button is clicked, the list of relations shown to the user is updated removing the already selected relation, since no relation can be used more than once. Furthermore, the tool starts to compute an ordering using the relation selected: if more relations are chosen, the order is updated to include them. The second option allows the

user to confirm his/her choice of relations, so the tool can start to compute the appropriate ordering of change parts. Finally, the "*Reset*" button allows the user to reset the list of relations to display again all the relations available in the tool. Below these buttons, the list of relations selected to produce the order is shown.
On the right side, the change parts ordered according to the user's choices are displayed. These are presented in the same way as the unordered modifications, but their order is rearranged to comply with the constraints imposed by the relation(s) chosen by the user.
The "*options*" bar in the top-left corner allows the user to enable or disable the "*Add unmodified files*" option, change the commit or the project under analysis and show a summary of the order of the change parts and the match sets constructed. These two last options were meant as aid to verify the correct functioning of the tool. Furthermore, they may give to the user a better insight in understanding the change parts that are connected to each other. If the user decides to change the project under analysis, he or she is also asked to insert the identifier of the commit, belonging to the new project, that he/she wants to review.

3.2.4 Tool limitations

Due to our implementation choices, the tool has the limits presented in this section.

- **Usage of JAR files**: the use of JAR files to solve the dependencies among different classes and external libraries may introduce imprecisions. In fact, to obtain complete precision in solving the links between software entities, the release date of the JAR files needs to be close to the one of the commit under analysis. Increasing the difference between them may introduce imprecisions in the creation of the match sets, consequently making the order produced by the tool sub-optimal. In fact, if a dependency among two different change parts is not solved correctly, the tool is not able to identify them as related. Therefore, they are located in two different match sets instead of the same one, consequently leading to an order that does not reflect the dependency between them.

- **Removed change parts**: the tool does not analyse the code inside removed change parts. Therefore, relations existing among removed portions of code can not be solved. The reason behind this decision is that analyzing both removed and added change parts may highlight dependencies that have been removed. We argue that this may lead to an ordering not useful for review. Furthermore, in case of a modified change part, both the old and the new code are present together in the diff. This may lead to solve a relation that has been changed in the new version of the file.

3.3 Internal test set

To verify the correct implementation of the relations in the tool, a test set was created. To construct it, a group of 10 different projects has been selected and, subsequently, 10 commits from each project. To guarantee the meaningfulness of our approach, these projects have been chosen based on the following criteria:

Table 3.3: SELECTED PROJECTS

Project	**# of commits**[2]	**# of classes**[3]
Spring-Framework	16,597	6830
Lucene-solr	30,098	7629
Titan	4,434	904
Camel	32,390	18458
Wicket	20,252	3255
Hbase	15,341	3627
Felix	14,658	4561
CXF	14,048	7163
Aries	5,429	2386
org.ops4j.pax.web	2,964	732

- *Open-Source*: the selected projects must be publicly available on GitHub. This limitation has been imposed for two reasons: the verifiability of the study and the implementation of the tool. The former because other researchers must be able to reproduce easily the same conditions that we used in our work, being able to clone the project repository and identify the considered commits through their unique identifier. The latter reason has its rationale in the fact that the tool needs to work on a local clone of the project repository to extract the code diff. Furthermore, also the "*Add unmodified code*" option needs access to the project directories to search for the related files. Moreover, projects developed in a company may have strict disclosure rules and, therefore, their source code could not be available.

- *Java-based*: currently the tool only supports the complete analysis of Java code. For this reason, selecting projects based on other programming languages would make it impossible to check the correctness of the majority of our relations: only the *Same file* and *Same format* relations can be applied to all kind of files since they do not rely on a parser.

The projects selected are shown in Table 3.3. To increase the variability of the test set, projects of different sizes have been selected. Furthermore, not all projects are still in development. The aim is to cover as many situations as possible. For this reason, we argue that these differences, together with the different policies and coding rules of the projects, may lead to better coverage of code instructions and constructs available in Java.
Moreover, to select the commits from each project the following criteria were applied:

- *JAR files availability*: to solve the dependencies among different code entities the tool relies on JavaParser and Java Symbol Solver, which requires the access to the JAR

[2]Retrieved on May 27 2018.
[3]Retrieved on June 11 2018.

Table 3.4: SPRING-FRAMEWORK TEST SET

Commit[5]	Files	Change Parts	Orders	Same file
e1fa65a	2	4	Manually computed	{AJVA2, AJVA52}, {HJVA2, HJVA54}
			Tool	{AJVA2, AJVA52}, {HJVA2, HJVA54}
b474916	2	5	Manually computed	{J2T2, J2T176}, {J2TT2, J2TT171, J2TT181}
			Tool	{J2T2, J2T176}, {J2TT2, J2TT171, J2TT181}
4f28c28	2	5	Manually Computed	{RT2, RT304, RT307}, {DUTH2, DUTH36},
			Tool	{RT2, RT304, RT307}, {DUTH2, DUTH36}
fdde9de	1	1	Manually Computed	{AABPP675}
			Tool	{AABPP675}
30f6e44	2	3	Manually Computed	{RTT1340, RTT1648}, {RT1471}
			Tool	{RTT1340, RTT1648}, {RT1471}
10caaef	1	2	Manually Computed	{SP188, SP191}
			Tool	{SP188, SP191}
c7f60d1	4	41	Manually Computed	{AJF2, AJF28, AJF46, AJF81} (...) [6]
			Tool	{AJF2, AJF28, AJF46, AJF81} (...)
4dc9645	2	15	Manually Computed	{AHLR2, AHLR27, AHLR121, AHLR125, (...)
			Tool	{AHLR2, AHLR27, AHLR121, AHLR125, (...)
196f3f8	2	8	Manually Computed	{HWHA2, HWHA185, HWHA194, HWHA207}, (...)
			Tool	{HWHA2, HWHA185, HWHA194, HWHA207}, (...)
0c28928	6	38	Manually Computed	{AREM2, AREM76, AREM79, AREM85, (...)
			Tool	{AREM2, AREM76, AREM79, AREM85, (...)

files of the project. Therefore, only commits involving modules for which the JAR files are publicly available can be selected[4].

- *Limit on the number of commits from the same author*: to increase the variability of the cases covered in our test set, we limited the number of commits authored by the same person to a maximum of four. In fact, we argue that the same person is more prone to use the same coding style. One of our main goals is to identify constructs that have not been covered by the tool implementation.

To test the ordering and the match sets created by the tool, a manual order was first computed based on the definition of the applied relation. In a second step, this order was compared with the one automatically produced by the tool. To have a better insight in the tool procedures and test also their correctness, during the ordering the match sets produced by the tool have also been indicated. In this way, it is possible to check that the tool has correctly identified the relations among different change parts. This procedure also helped us to identify new relations that have not been covered by the tool before, but that may have been worth investigating. In fact, manually analyzing code to construct the test cases allowed us to retrieve new links between change parts. An example may be found in the *Field access* relation. This relation was excluded by our tool at first, but added in a later time due to its frequent use in the projects under analysis.
This testing procedure has been applied to each relation implemented in the tool and for the *unmodified code option*, which is supported only by the *New object*, *Method call* and *Inheritance* relations.

[4]The JAR files were retrieved online using the following website: www.mvnrepository.com

Table 3.4 shows the test set produced by applying the *Same file* relation for the 10 commits selected from the Spring-framework[7] project. In the last columns, for each commit the order produced manually and the one produced by the tool are shown. The brackets are used to delimit the different match sets constructed by the tool. For readability, each file has been indicated using only the initial letters of the words forming its name. Consequently, each change part has been reported using this abbreviation together with the number of the first line of the modification in the new version of the file. Moreover, if no extension is specified for the file, we imply it to be a Java file. Otherwise, its extension is specified after the file name. A similar approach has been applied to all the other relations supported by the tool and to the remaining projects. For readability, only this example has been reported here. However, the complete test set is available online[8].

3.3.1 Threats to validity

While the creation of a test set allowed us to verify the correctness of the relations and the ordering of the tool, this approach presents some limits.
First of all, the test cases were produced through a manual inspection of the code to identify the relations among the different change parts. This procedure may have introduced some bias in the produced output, especially in the case of large commits, fine granularity relations (e.g., *Declare-use*) or the "*Add unmodified code*" option. Particularly in this last case, the vast amount of links between software entities that needs to be considered may have introduced small imprecisions in the two orderings (both the manually and the automatically generated one).
JavaParser may introduce some imprecisions in the resolution of the relations among change parts. This may lead to the production of a sub-optimal order. To be able to solve correctly all the Java constructs (e.g., method calls) the tool requires an updated version of all the JAR files of the source directories and libraries used: modifications in the project repository done after the commit date may undermine the tool capability to correctly solve the dependencies. Therefore, a constantly updated version of the JAR files is necessary for the tool to operate correctly. Violating this condition may cause differences between the manually computed order and the one produced by the tool in our test set.

[5]Only the first 7 digits of the commit identifier are reported for clarity reasons
[6]For clarity, not all the change parts have been reported in the table
[7]Spring-framework: https://github.com/spring-projects/spring-framework
[8]https://github.com/EFregnan/Automatic-ordering-of-code-changes_related-material

Chapter 4

CodeChangeOrderer in practice

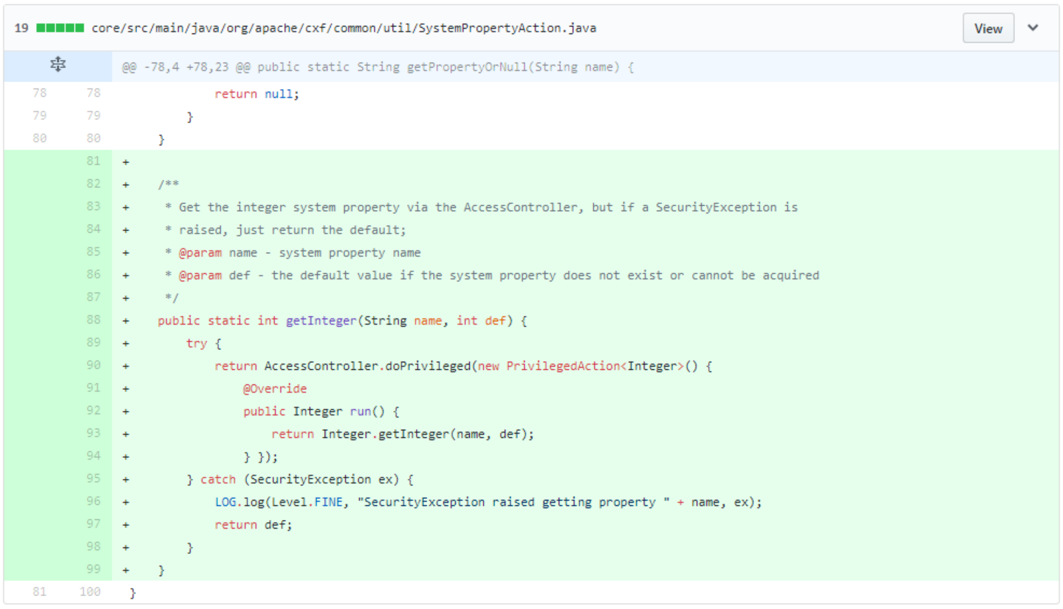
In this chapter, three examples of the orders produced by the tool are presented. Their aim is to clarify the concepts introduced in chapter 3. Three commits are considered for the following analysis: the first is a commit[1] from the CXF[2] project, the second[3] belongs to the Felix project[4], while the last one[5] is extracted from the Lucene-solr project[6]. All three of these commits are included in the internally produced test set for the tool.
The *Same file* relation and the *Method call* relation are applied to the first commit, together with their combination. The *Same file* and the *Declare-use* relations, and their combination, are applied to the second commit. Finally, the *New object* relation is used in the last commit to present the *positioning* and "*Add unmodified code*" options. For clarity reasons, only a subset of the relations available in the tool can be shown in this chapter. However, the orders generated applying the remaining relations to these commits may be found in the related material (internally produced test set).

In the next sections, the following formalism is used: each change part is identified by the file name and the initial line of the modification. This is the first line on which the modification begins in the new version of the file. For brevity, each file is indicated using only the initial letters of the words forming its name: e.g., SystemPropertyAction is shortened to SPA. Therefore, the change part belonging to this file (shown in Figure 4.1) is reported as SPA81.

4.1 First example

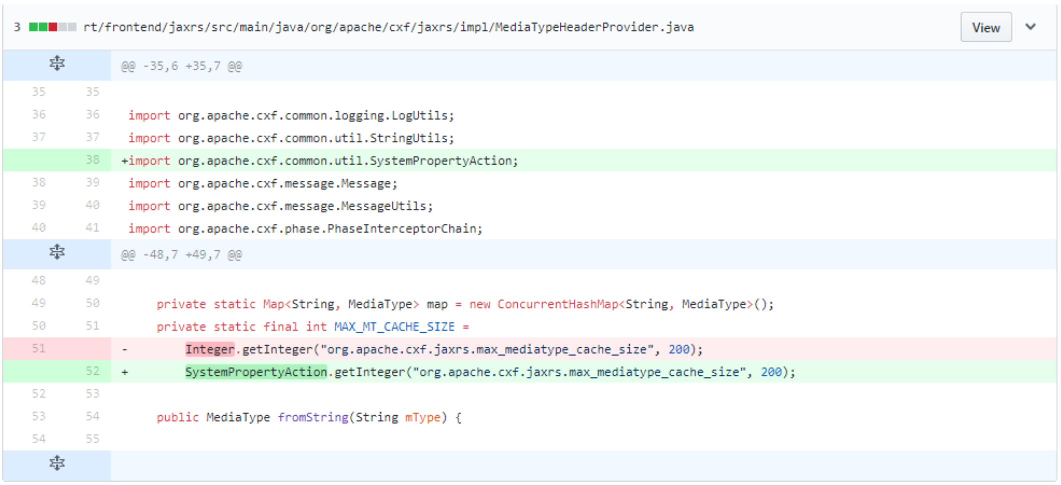
The CXF commit under analysis modifies four different Java files: SystemPropertyAction.java, MediaTypeHeaderProvider.java, URITemplate.java and ProviderCache.java

[1]Commit ID: 9130f84fafca18620888aecfe1d0d8a0cbce1fc2
[2]CXF Project: https://github.com/apache/cxf
[3]Commit ID: cb4b81cd08d9a5ecb62dd410a6fb8c163b0009de
[4]Felix Project: https://github.com/apache/felix
[5]Commit ID: 1f3d971a757edd694adbc492f2de08263921eb01
[6]Lucene-solr project: https://github.com/apache/lucene-solr



```
19 core/src/main/java/org/apache/cxf/common/util/SystemPropertyAction.java

@@ -78,4 +78,23 @@ public static String getPropertyOrNull(String name) {
78   78               return null;
79   79           }
80   80       }
     81  +
     82  +    /**
     83  +     * Get the integer system property via the AccessController, but if a SecurityException is
     84  +     * raised, just return the default;
     85  +     * @param name - system property name
     86  +     * @param def - the default value if the system property does not exist or cannot be acquired
     87  +     */
     88  +    public static int getInteger(String name, int def) {
     89  +        try {
     90  +            return AccessController.doPrivileged(new PrivilegedAction<Integer>() {
     91  +                @Override
     92  +                public Integer run() {
     93  +                    return Integer.getInteger(name, def);
     94  +                } });
     95  +        } catch (SecurityException ex) {
     96  +            LOG.log(Level.FINE, "SecurityException raised getting property " + name, ex);
     97  +            return def;
     98  +        }
     99  +    }
81  100   }
```

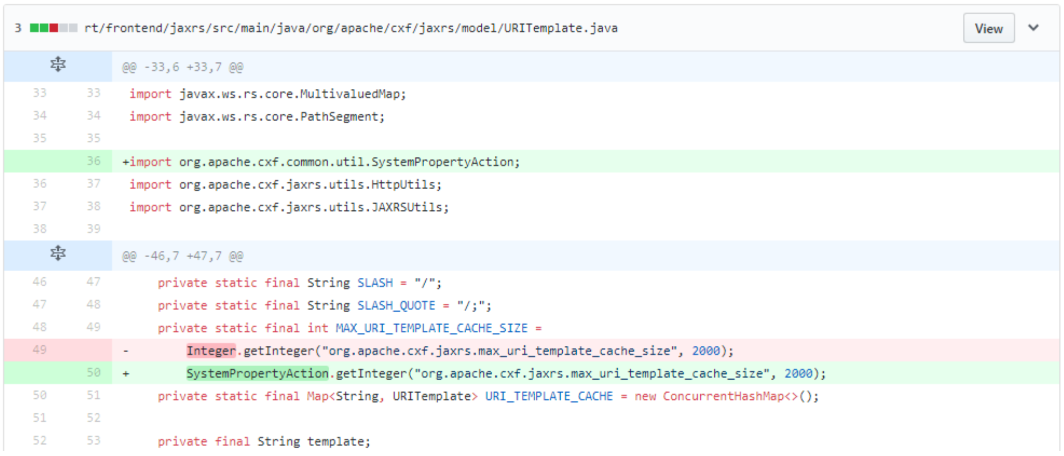
Figure 4.1: CXF Commit: SystemPropertyAction.java



```
3 rt/frontend/jaxrs/src/main/java/org/apache/cxf/jaxrs/impl/MediaTypeHeaderProvider.java

@@ -35,6 +35,7 @@
35   35
36   36   import org.apache.cxf.common.logging.LogUtils;
37   37   import org.apache.cxf.common.util.StringUtils;
     38  +import org.apache.cxf.common.util.SystemPropertyAction;
38   39   import org.apache.cxf.message.Message;
39   40   import org.apache.cxf.message.MessageUtils;
40   41   import org.apache.cxf.phase.PhaseInterceptorChain;
@@ -48,7 +49,7 @@
48   49
49   50       private static Map<String, MediaType> map = new ConcurrentHashMap<String, MediaType>();
50   51       private static final int MAX_MT_CACHE_SIZE =
51       -        Integer.getInteger("org.apache.cxf.jaxrs.max_mediatype_cache_size", 200);
     52  +        SystemPropertyAction.getInteger("org.apache.cxf.jaxrs.max_mediatype_cache_size", 200);
52   53
53   54       public MediaType fromString(String mType) {
54   55
```

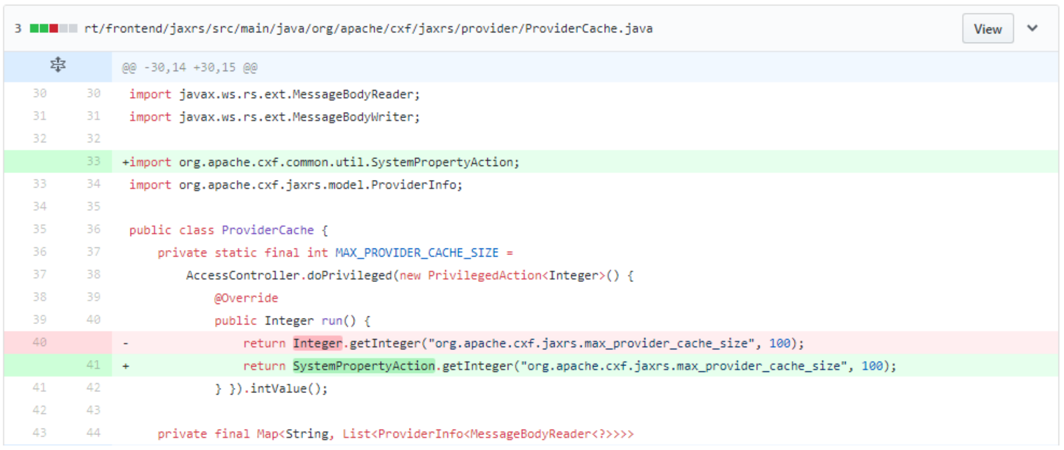
Figure 4.2: CXF Commit: MediaTypeHeaderProvider.java

The first class, shown in Figure 4.1, contains only one change part: SPA81. The second class, MediaTypeHeaderProvider (Figure 4.2), contains two change parts: MTHP38 and MTHP52. The third class URITemplate (Figure 4.3) contains two change parts: URIT36 and URIT50. Finally, the last class ProviderCache has two change parts too: PC33 and PC41.



```
3 rt/frontend/jaxrs/src/main/java/org/apache/cxf/jaxrs/model/URITemplate.java    View
        @@ -33,6 +33,7 @@
33  33   import javax.ws.rs.core.MultivaluedMap;
34  34   import javax.ws.rs.core.PathSegment;
35  35
    36  +import org.apache.cxf.common.util.SystemPropertyAction;
36  37   import org.apache.cxf.jaxrs.utils.HttpUtils;
37  38   import org.apache.cxf.jaxrs.utils.JAXRSUtils;
38  39
        @@ -46,7 +47,7 @@
46  47       private static final String SLASH = "/";
47  48       private static final String SLASH_QUOTE = "/;";
48  49       private static final int MAX_URI_TEMPLATE_CACHE_SIZE =
49      -         Integer.getInteger("org.apache.cxf.jaxrs.max_uri_template_cache_size", 2000);
    50  +         SystemPropertyAction.getInteger("org.apache.cxf.jaxrs.max_uri_template_cache_size", 2000);
50  51       private static final Map<String, URITemplate> URI_TEMPLATE_CACHE = new ConcurrentHashMap<>();
51  52
52  53       private final String template;
```

Figure 4.3: CXF Commit: URITemplate.java



```
3 rt/frontend/jaxrs/src/main/java/org/apache/cxf/jaxrs/provider/ProviderCache.java    View
        @@ -30,14 +30,15 @@
30  30   import javax.ws.rs.ext.MessageBodyReader;
31  31   import javax.ws.rs.ext.MessageBodyWriter;
32  32
    33  +import org.apache.cxf.common.util.SystemPropertyAction;
33  34   import org.apache.cxf.jaxrs.model.ProviderInfo;
34  35
35  36   public class ProviderCache {
36  37       private static final int MAX_PROVIDER_CACHE_SIZE =
37  38           AccessController.doPrivileged(new PrivilegedAction<Integer>() {
38  39               @Override
39  40               public Integer run() {
40      -                 return Integer.getInteger("org.apache.cxf.jaxrs.max_provider_cache_size", 100);
    41  +                 return SystemPropertyAction.getInteger("org.apache.cxf.jaxrs.max_provider_cache_size", 100);
41  42               } }).intValue();
42  43
43  44       private final Map<String, List<ProviderInfo<MessageBodyReader<?>>>>
```

Figure 4.4: CXF Commit: ProviderCache.java

4.1.1 Same file

First, the *Same file* relation is applied to the change parts in the commit. The rationale behind this relation is to position change parts belonging to the same file close to each other. Therefore, the two change parts belonging to MediaTypeHeaderProvider are grouped together. The same applies for the two change parts belonging to the URITemplate and ProviderCache, respectively. SystemPropertyAction contains only one change part, so it is grouped alone.
The following match sets are constructed:

{MTHP38, MTHP52}, {URIT36, URIT50}, {PC33, PC41} and {SPA81}.

If no relation applies or all the constraints are already satisfied, the tool orders the change parts based on the line number.

Therefore, the order produced by the tool is the following:

PC33, PC41, URIT36, URIT50, MTHP38, MTHP52, SPA81

4.1.2 Method call

Change parts may also be ordered according to the method call flow, where a change part containing a call of a method is positioned close to the change part(s) belonging to the related method declaration. The tool analyses the code contained in the change parts using JavaParser and Java Symbol Solver and creates the match sets. If a change part has no relation to the others, so it does not contain any method call to methods modified in the commit, it is grouped alone in a match set.
In the considered commit, the change parts MTHP52, URIT50 and PC41 all contain a call to the method *getInteger* defined in the change part SPA81 (Figure 4.1). Therefore, the tool produces the following match sets:

{MTHP52, SPA81}, {URIT50, SPA81}, {PC41, SPA81}, {MTHP38}, {URIT36} and {PC33}.

Change part SPA81 is positioned in multiple match sets since it is linked to different change parts. However, it will be shown only once in the final order.

The order produced by the tool is the following:

PC33, URIT36, MTHP38, PC41, SPA81, URIT50, MTHP52

Since multiple change parts are related to the same one (MHTP52, URIT50, PC41 are related to SPA81), the *centering option* may be applied. The order shown above uses the "*center in the middle*" option. Therefore, change part SPA81 is positioned between all the change parts to which it is related.
Applying the "*center first*" option, change part SPA81 must be positioned before all the change parts to which is related. Therefore, the order produced by the tool is the following:

PC33, URIT36, MTHP38, **SPA81**, PC41, URIT50, MTHP52

Finally, recurring to the "*center second*" option, change part SPA81 must be ordered after a related change part, but before the others. Therefore, one use of the method is shown before the method declaration.
The order produced by the tool is the following:

PC33, URIT36, MTHP38, PC41, **SPA81**, URIT50, MTHP52

4.1.3 Same file and Method call

The tool allows the user to select more than one relation. In this example, the relations applied before are combined to produce an order that takes in account both their constraints. In fact, the tool output must respect to the maximum extent possible the match sets created by the two relations.
In this case, the tool creates two groups of match sets, the one reflecting the same file relation and the one reflecting the method call relation. These are:

- **Same file**: {MTHP38, MTHP52}, {URIT36, URIT50}, {PC33, PC41} and {SPA81}
- **Method call**: {MTHP52, SPA81}, {URIT50, SPA81}, {PC41, SPA81}, {MTHP38}, {URIT36} and {PC33}

Based on these constraints, the order produced by the tool is the following:

PC33, PC41, SPA81, URIT50, URIT36, MTHP52, MTHP38

Where change part SPA81, containing the method declaration, is positioned close to the change parts (PC41, URIT50 and MTHP52) containing the method use. However, these change parts must also be close to the ones belonging to the same file.

4.2 Second example

The Felix commit under analysis contains two Java files: ExtensionManager.java and Util.java. The first class contains eight change parts: EM39, EM211, EM218, EM226, EM228 (Figure 4.5), EM238, EM246 and EM248 (Figure 4.6). The second class, Util.java (Figure 4.7), contains four change parts: U26, U32, U40 and U134.

4.2.1 Same file

The *Same file* relation is applied to the change parts in the commit: change parts belonging to the same file must be grouped in the same match set and positioned next to each other in the order produced by the tool.
The following match sets are constructed:

{EM39, EM211, EM218, EM226, EM228, EM238, EM246 and EM248} and {U26, U32, U40 and U134}.

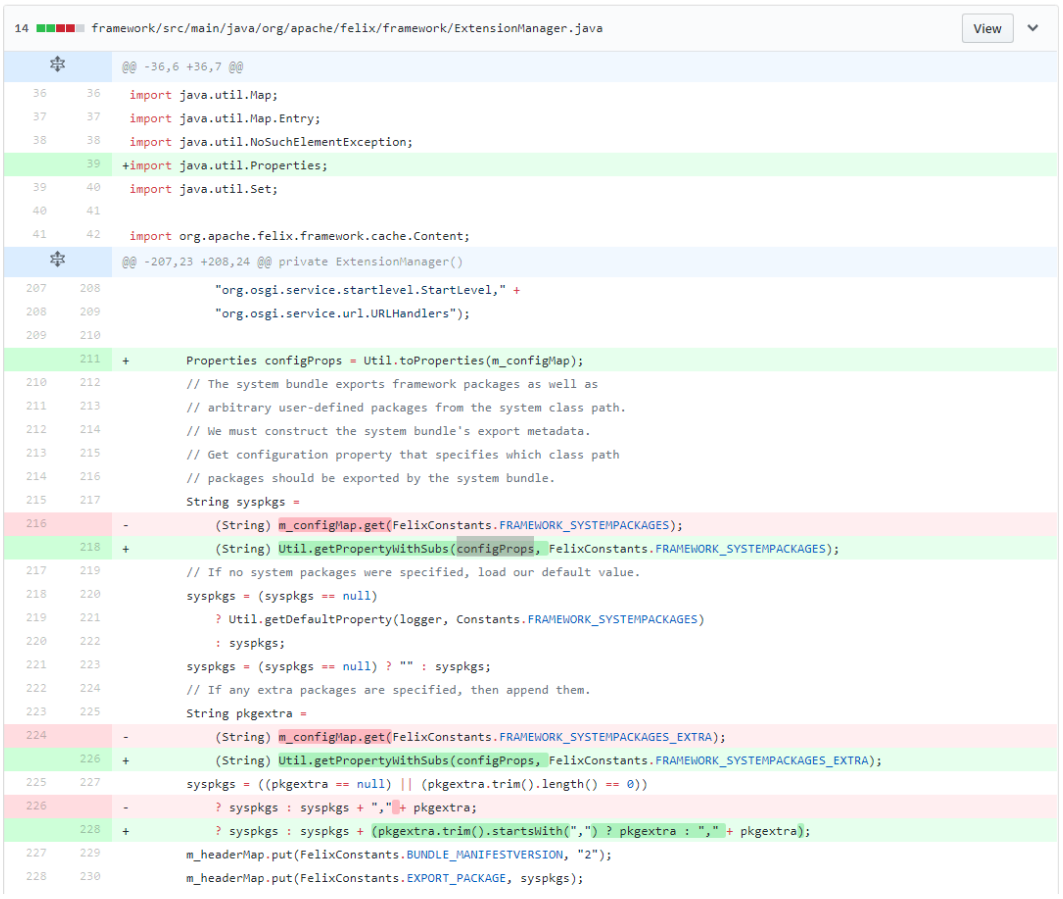
The order produced by the tool is the following:

U26, U32, U40, U134, EM39, EM211, EM218,
EM226, EM228, EM238, EM246, EM248

4.2.2 Declare-use relation

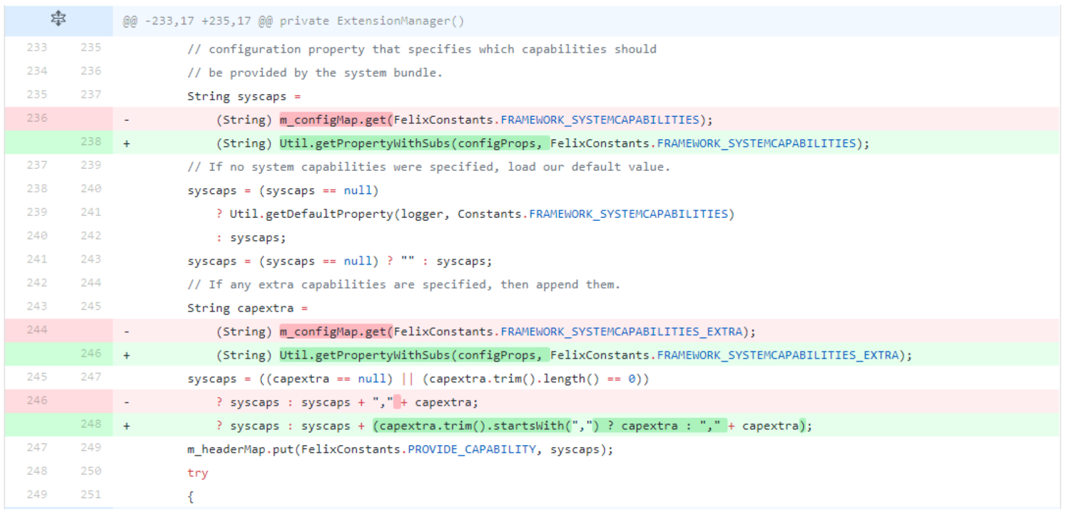
Applying the *Declare-use* relation, each variable use is positioned close to the related variable declaration. However, only variable uses and declarations contained in portions of code modified in the commit are considered. Furthermore, only added portions of code are analyzed to extract links among the change parts.
According to these limitations, the tool constructs the following match sets:

{EM211, EM218}, {EM211, EM226}, {EM211, EM238}, {EM211, EM246}, {EM39}, {EM228}, {EM248}, {U26}, {U32}, {U40} and {U134}.



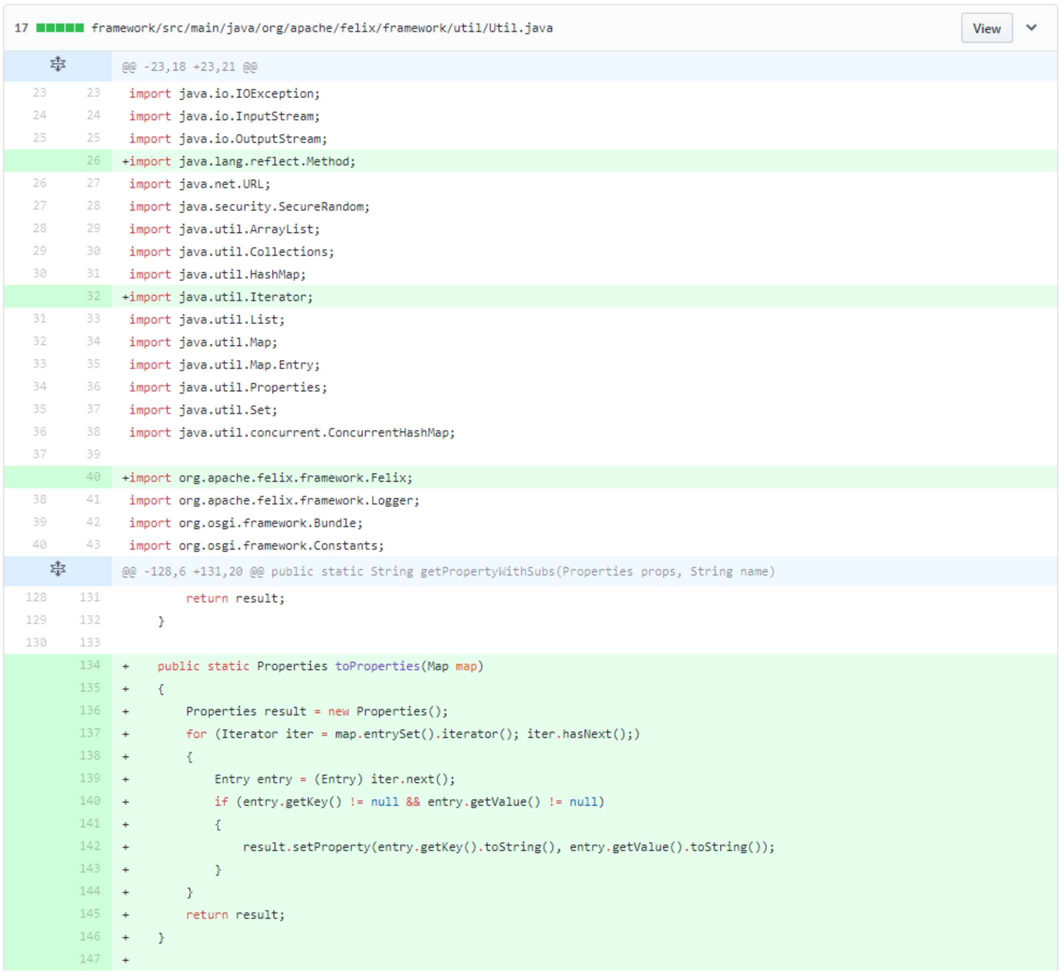
```
14 framework/src/main/java/org/apache/felix/framework/ExtensionManager.java          View
@@ -36,6 +36,7 @@
 import java.util.Map;
 import java.util.Map.Entry;
 import java.util.NoSuchElementException;
+import java.util.Properties;
 import java.util.Set;

 import org.apache.felix.framework.cache.Content;
@@ -207,23 +208,24 @@ private ExtensionManager()
             "org.osgi.service.startlevel.StartLevel," +
             "org.osgi.service.url.URLHandlers");

+        Properties configProps = Util.toProperties(m_configMap);
         // The system bundle exports framework packages as well as
         // arbitrary user-defined packages from the system class path.
         // We must construct the system bundle's export metadata.
         // Get configuration property that specifies which class path
         // packages should be exported by the system bundle.
         String syspkgs =
-            (String) m_configMap.get(FelixConstants.FRAMEWORK_SYSTEMPACKAGES);
+            (String) Util.getPropertyWithSubs(configProps, FelixConstants.FRAMEWORK_SYSTEMPACKAGES);
         // If no system packages were specified, load our default value.
         syspkgs = (syspkgs == null)
             ? Util.getDefaultProperty(logger, Constants.FRAMEWORK_SYSTEMPACKAGES)
             : syspkgs;
         syspkgs = (syspkgs == null) ? "" : syspkgs;
         // If any extra packages are specified, then append them.
         String pkgextra =
-            (String) m_configMap.get(FelixConstants.FRAMEWORK_SYSTEMPACKAGES_EXTRA);
+            (String) Util.getPropertyWithSubs(configProps, FelixConstants.FRAMEWORK_SYSTEMPACKAGES_EXTRA);
         syspkgs = ((pkgextra == null) || (pkgextra.trim().length() == 0))
-            ? syspkgs : syspkgs + "," + pkgextra;
+            ? syspkgs : syspkgs + (pkgextra.trim().startsWith(",") ? pkgextra : "," + pkgextra);
         m_headerMap.put(FelixConstants.BUNDLE_MANIFESTVERSION, "2");
         m_headerMap.put(FelixConstants.EXPORT_PACKAGE, syspkgs);
```

Figure 4.5: Felix Commit: ExtensionManager.java; first five change parts



```
@@ -233,17 +235,17 @@ private ExtensionManager()
         // configuration property that specifies which capabilities should
         // be provided by the system bundle.
         String syscaps =
-            (String) m_configMap.get(FelixConstants.FRAMEWORK_SYSTEMCAPABILITIES);
+            (String) Util.getPropertyWithSubs(configProps, FelixConstants.FRAMEWORK_SYSTEMCAPABILITIES);
         // If no system capabilities were specified, load our default value.
         syscaps = (syscaps == null)
             ? Util.getDefaultProperty(logger, Constants.FRAMEWORK_SYSTEMCAPABILITIES)
             : syscaps;
         syscaps = (syscaps == null) ? "" : syscaps;
         // If any extra capabilities are specified, then append them.
         String capextra =
-            (String) m_configMap.get(FelixConstants.FRAMEWORK_SYSTEMCAPABILITIES_EXTRA);
+            (String) Util.getPropertyWithSubs(configProps, FelixConstants.FRAMEWORK_SYSTEMCAPABILITIES_EXTRA);
         syscaps = ((capextra == null) || (capextra.trim().length() == 0))
-            ? syscaps : syscaps + "," + capextra;
+            ? syscaps : syscaps + (capextra.trim().startsWith(",") ? capextra : "," + capextra);
         m_headerMap.put(FelixConstants.PROVIDE_CAPABILITY, syscaps);
         try
         {
```

Figure 4.6: Felix Commit: ExtensionManager.java; remaining three change parts



```
17 framework/src/main/java/org/apache/felix/framework/util/Util.java                View
        @@ -23,18 +23,21 @@
23   23   import java.io.IOException;
24   24   import java.io.InputStream;
25   25   import java.io.OutputStream;
     26  +import java.lang.reflect.Method;
26   27   import java.net.URL;
27   28   import java.security.SecureRandom;
28   29   import java.util.ArrayList;
29   30   import java.util.Collections;
30   31   import java.util.HashMap;
     32  +import java.util.Iterator;
31   33   import java.util.List;
32   34   import java.util.Map;
33   35   import java.util.Map.Entry;
34   36   import java.util.Properties;
35   37   import java.util.Set;
36   38   import java.util.concurrent.ConcurrentHashMap;
37   39
     40  +import org.apache.felix.framework.Felix;
38   41   import org.apache.felix.framework.Logger;
39   42   import org.osgi.framework.Bundle;
40   43   import org.osgi.framework.Constants;
        @@ -128,6 +131,20 @@ public static String getPropertyWithSubs(Properties props, String name)
128  131          return result;
129  132      }
130  133
     134 +    public static Properties toProperties(Map map)
     135 +    {
     136 +        Properties result = new Properties();
     137 +        for (Iterator iter = map.entrySet().iterator(); iter.hasNext();)
     138 +        {
     139 +            Entry entry = (Entry) iter.next();
     140 +            if (entry.getKey() != null && entry.getValue() != null)
     141 +            {
     142 +                result.setProperty(entry.getKey().toString(), entry.getValue().toString());
     143 +            }
     144 +        }
     145 +        return result;
     146 +    }
     147 +
```

Figure 4.7: Felix Commit: Util.java

In fact, all the change parts EM218, EM226, EM238 and EM246 contain a use of the variable *configProps*, which is declared inside the change part EM211. Change parts with no relation to the others are grouped alone in a match set.
The "*center option*" may also be applied to the change parts EM218, EM226, EM238, EM246 and EM211. Selecting the "*center in the middle*" option, EM211, the change part containing the variable definition, is positioned between EM218, EM226, EM238 and EM246. Therefore, the order produced by the tool is the following:

U26, U32, U40, U134, EM39, EM218,
EM226, **EM211**, EM238, EM246, EM228, EM248

A different order may be obtained using the "*center first*" option: it moves the change part PPU242 to be first than PPU250 and PPU261 in the order. Therefore, the tool output is:

U26, U32, U40, U134, EM39, **EM211**, EM218,
EM226, EM238, EM246, EM228, EM248

Finally, applying the "*center second*" option, change part EM211 is positioned after the first variable use (the one in EM218), but before the others. The order produced applying this option is:

U26, U32, U40, U134, EM39, EM218, **EM211**,
EM226, EM238, EM246, EM228, EM248

4.2.3 Same file and Declare-use relation

Finally, the *Same file* and *Declare-use* relations are applied together to determine a tour based on the commit. To perform this task, the tool computes the match sets for both the *Same file* and *Declare-use*. They are the following:

- **Same file**: {EM39, EM211, EM218, EM226, EM228, EM238, EM246 and EM248} and {U26, U32, U40 and U134}.

- **Declare-use**: {EM211, EM218}, {EM211, EM226}, {EM211, EM238}, {EM211, EM246}, {EM39}, {EM228}, {EM248}, {U26}, {U32}, {U40} and {U134}.

Now, the change parts are disposed in groups according to the constraints imposed by the *Same file* relation. Then, they are internally ordered obeying to the *Declare-use* relation. Differently from the previous example, in this case the *Same file* relation does not prevent the application of the center option, since the related change parts belong all to the same file: EM211-EM218, EM211-EM226, EM211-EM238 and EM211-EM246. Finally, the order produced, applying the "*center first*" option, is the following:

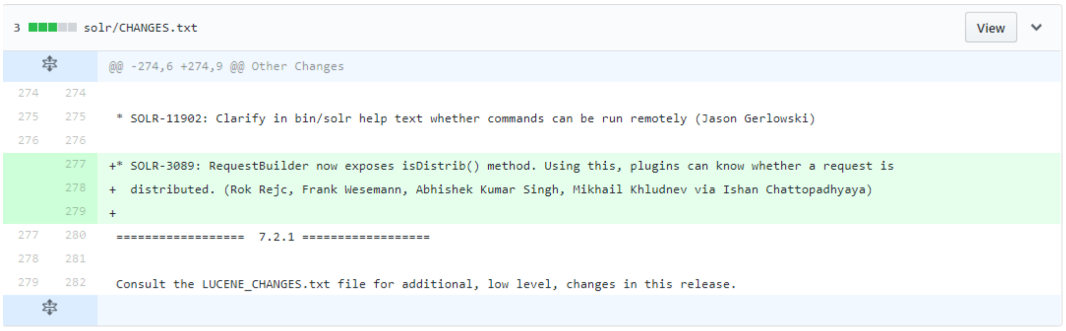
U26, U32, U40, U134, EM39, **EM211**, EM218,
EM226, EM238, EM246, EM228, EM248

While the use of the "*center in the middle*" leads to the following order:

U26, U32, U40, U134, EM39, EM218,
EM226, **EM211**, EM238, EM246, EM228, EM248

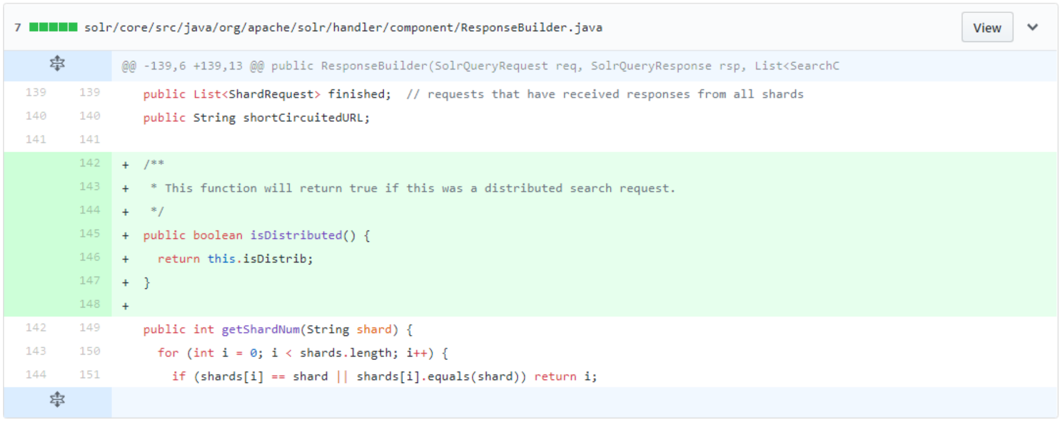
Finally, using the "*center second*" option produces as outcome:

U26, U32, U40, U134, EM39, EM218, **EM211**,
EM226, EM238, EM246, EM228, EM248



```
3 solr/CHANGES.txt                                                     View
@@ -274,6 +274,9 @@ Other Changes
274 274
275 275    * SOLR-11902: Clarify in bin/solr help text whether commands can be run remotely (Jason Gerlowski)
276 276
    277   +* SOLR-3089: RequestBuilder now exposes isDistrib() method. Using this, plugins can know whether a request is
    278   +  distributed. (Rok Rejc, Frank Wesemann, Abhishek Kumar Singh, Mikhail Khludnev via Ishan Chattopadhyaya)
    279   +
277 280    =================  7.2.1 =================
278 281
279 282    Consult the LUCENE_CHANGES.txt file for additional, low level, changes in this release.
```

Figure 4.8: Lucene Commit: CHANGES.txt



```
7 solr/core/src/java/org/apache/solr/handler/component/ResponseBuilder.java          View
@@ -139,6 +139,13 @@ public ResponseBuilder(SolrQueryRequest req, SolrQueryResponse rsp, List<SearchC
139 139      public List<ShardRequest> finished;  // requests that have received responses from all shards
140 140      public String shortCircuitedURL;
141 141
    142   +  /**
    143   +   * This function will return true if this was a distributed search request.
    144   +   */
    145   +  public boolean isDistributed() {
    146   +    return this.isDistrib;
    147   +  }
    148   +
142 149      public int getShardNum(String shard) {
143 150        for (int i = 0; i < shards.length; i++) {
144 151          if (shards[i] == shard || shards[i].equals(shard)) return i;
```

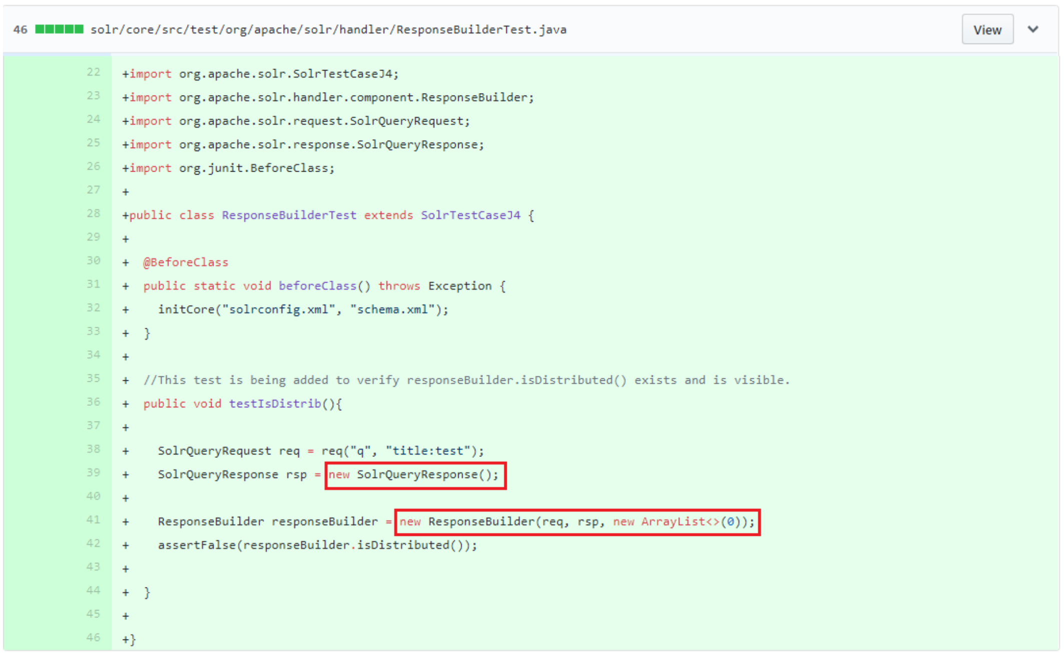
Figure 4.9: Lucene Commit: ResponseBuilder.java

4.3 Further options example

In the previous examples, it was possible to apply only the "*center option*", among the three advanced options implemented in the tool. Therefore, in this commit extracted from the Lucene-solr project the "*positioning*" and "*Add unmodified code*" options are shown, using the *New object* relation.
The considered commit contains three files: CHANGES.txt (Figure 4.8), Response-Builder.java (Figure 4.9) and ResponseBuilderTest.java (Figure 4.10). The first contains only a change part beginning at line 277 (C.txt277), while the second has a change part beginning at line 142 (RB142). Finally, the whole class ResponseBuilderTest.java has been added in this commit and, therefore, is treated as a unique change part (RBT1).

4.3.1 Positioning option

Applying the *New object* relation, the change parts RBT1 and RB142 are grouped in the same match set, since the former instantiates the class of the latter. CHANGES.txt is not a Java file, so its content is not analysed and it is grouped alone in a match set. The match sets created are the following: {RBT1, RB142} and {C.txt277}.



46 solr/core/src/test/org/apache/solr/handler/ResponseBuilderTest.java

```
+import org.apache.solr.SolrTestCaseJ4;
+import org.apache.solr.handler.component.ResponseBuilder;
+import org.apache.solr.request.SolrQueryRequest;
+import org.apache.solr.response.SolrQueryResponse;
+import org.junit.BeforeClass;
+
+public class ResponseBuilderTest extends SolrTestCaseJ4 {
+
+  @BeforeClass
+  public static void beforeClass() throws Exception {
+    initCore("solrconfig.xml", "schema.xml");
+  }
+
+  //This test is being added to verify responseBuilder.isDistributed() exists and is visible.
+  public void testIsDistrib(){
+
+    SolrQueryRequest req = req("q", "title:test");
+    SolrQueryResponse rsp = new SolrQueryResponse();
+
+    ResponseBuilder responseBuilder = new ResponseBuilder(req, rsp, new ArrayList<>(0));
+    assertFalse(responseBuilder.isDistributed());
+
+  }
+
+}
```

Figure 4.10: Lucene Commit: ResponseBuilderTest.java; the new object declarations are highlighted in red boxes.

Using the *Positioning* option, the user can decide if the change part containing the new object instantiation must be shown before of after the change part(s) belonging to the related object. Using the former possibility, the order produced by the tool is:

RBT1, RB142, C.txt277

Deciding for the latter option leads to the following order:

RB142, **RBT1**, C.txt277

4.3.2 Add unmodified files option

Change part RBT1 contains the instantiation of an object belonging to the *SolrQueryResponse* class, not modified in the commit but present in the project repository. Applying the *New object* relation and the "*Add unmodified code*" option, this class is identified in the project directory and its code is added as change part (SQR0) to the list of modifications to order. Therefore, the new match set created by the tool is: {RBT1, RB142, SQR0}. At this point, it is possible to apply the *positioning* option to decide if change part RBT1 must be positioned before or after the related change parts (RB142, SQR0) in the order.
Deciding to position the change part that contains the new object statements before, the order produced by the tool is:

RBT1, SQR0, RB142, C.txt277

On the contrary, choosing to position it after the related change parts leads to the following order:

SQR0, RB142, **RBT1**, C.txt277

4.4 Discussion

In the examples considered, a subset of the relations in the tool has been selected to show how the tool works in practice. Although it was impossible to cover all the possible scenarios, these examples are meant to showcase the procedure followed by the tool. For readability, the size of the commits considered (in terms of amount of modified files, change parts and line of code) had to be kept small. However, the same approach applies to commits of bigger size and complexity.
Moreover, in our example relations working at different granularity levels have been applied together: *Same file* and *Method call* or *Same file* and *Declare-use* relations. However, it is possible to apply together also relations working at the same granularity level: e.g., *Declare-use* and *Parameter-use*. The procedure followed by the tool is the same: the match sets for both relations are constructed and an order is produced respecting their constraints. Furthermore, the tool does not present a limit on the number of relations that can be applied at the same time.
If the commit contains files of formats different than Java (e.g., XML files), their change parts are shown in the tool order, both in the unordered and ordered one. However, their content can not be analyzed by the tool. Thus, each change part belonging to these files is grouped alone in a separate match set, when relations that require code analysis are applied. In the tour created, they will be positioned based on the number of the first line in each modification block.

Chapter 5

Testing the tool

In this section, the procedure followed to validate the tool output and its results is presented. Furthermore, an initial investigation of the impact of the order produced on code review is conducted and its results are explained.
The procedure followed is explained in Figure 5.1. First, a set of interviews has been conducted with a group of Java developers. Then, an online survey has been released to assess the usefulness of the tool orders and which factors may influence it.

5.1 External validation

To verify the correct functioning of our tool, an external validation is needed. In fact, the internal test procedure (explained in Section 3.3) may be biased due to the fact that the test set has been constructed by the same person who developed the tool. For this reason, in this procedure a new test set is constructed by a group of external developers.

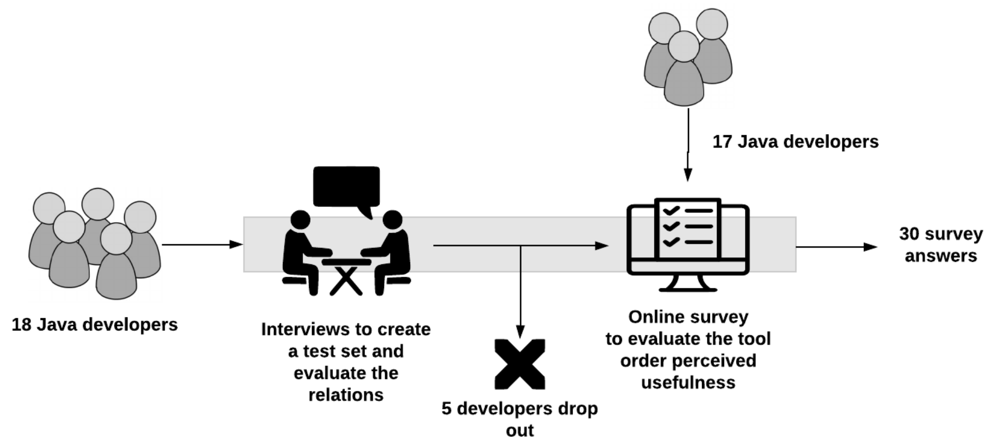

17 Java developers
18 Java developers
Interviews to create a test set and evaluate the relations
5 developers drop out
Online survey to evaluate the tool order perceived usefulness
30 survey answers

Figure 5.1: Methodology followed to test the tool and verify its usefulness

The group of developers involved in this process has been chosen among the personal contacts of the author with the only selection criterion of having programming experience in Java. This knowledge is fundamental for them to create an ordering of the change parts based on a real understanding of the code. However, it needs to be underlined that the fact that the author knows the interviewed people personally may have introduced a bias in the process.
Each developer involved was asked to participate in an interview, during which he or she had to answer a questionnaire and produce a first order for the tool test set. Furthermore, he or she was asked to produce two more orders at a later time. To ensure the correctness of the procedure, all the interviews have been recorded. Conducting face-to-face interviews had the advantage to allow us to record any comments and remarks. Furthermore, the interviewees were allowed to ask questions if any information contained in the given documents was unclear. This allowed us to reduce potential errors in the test set due to an incomplete understanding of the ordering theory and tool implementation by the developers.
During the interviews, three documents were handed to the developers (presented in Appendix B). The first one contains an explanation of the tasks that they are asked to complete (questionnaire and test cases), together with a brief description of the tool and its limits. The second file presents a description of the relations implemented by the tool. Finally, the third one shows an example of an ordering produced by the tool.

5.1.1 Questionnaire

In this procedure, the developers are first asked to answer some general questions to understand their experience in software development and, specifically, in performing code review. These questions have been asked during the afore-mentioned interviews. The set of questions asked is the following:

1. *How often do you currently do programming?*
    - *About once a day or more often*
    - *About once a week*
    - *About once a month*
    - *About once a year*
    - *Not at all*
2. *On which kinds of projects have you worked?*
    - *Company proprietary projects*
    - *Open source projects*
    - *University projects*
    - *Others*

3. *Which category does the project that you selected for this survey belong to?*
    - *Company proprietary projects*
    - *Open source projects*
    - *University projects*
    - *Others*
4. *Which is/was your predominant role in the selected project?*
    - *Developer*
    - *Reviewer*
    - *User*
    - *No role*
5. *How many years of experience do you have with Java?*
    - *1 year or less*
    - *2 years*
    - *3-5 years*
    - *6-10 years*
    - *11 years or more*
6. *How often do you perform code review tasks? Choose an answer from the following options:*
    - *About once a day or more often*
    - *About once a week*
    - *About once a month*
    - *About once a year*
    - *Not at all*
7. *Order the set of relations offered by the tool (please refer to the Relations file) in terms of importance assigning a number from 1 (most important) to 8 (least important).*

The wording of the questions has been inspired by previous surveys [8] [48] to ensure their clarity. With the first six questions, the aim is to verify the background of the person interviewed. The last one instead starts an investigation to assess the perceived usefulness of the relations implemented in the tool based on the description given to the developers

(Appendix B).

5.1.2 External test set

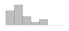
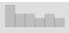
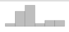
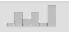
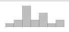
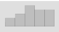
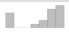
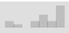
The developers are asked to select three different commits from a project of their choice. They are encouraged to select a project on which they feel confident working: possibly, a project to which they collaborated in the past. However, it must respect the following constraints:

- *Publicly available*: the source directories of the project must be available on GitHub.
- *Java-based*: the tool can not extract relations among change parts not containing Java code. Therefore, to allow meaningful test cases to be constructed, the project must contain mainly Java files.
- *JAR files available*: the JAR files of the selected projects must be publicly available (at least the ones belonging to the modules used in the selected commits). The developers are also allowed to give us the JAR files of the project, in case they can not be retrieved online.

To prepare for the possibility in which developers can not or do not want to use a project of their own, a backup project is given. The selected project is *Apache/Karaf*[1], which respects the constraints indicated above: it is Java-based and its directories and JAR files are publicly available (the project is Open-Source).
For each of the three commits, the developers must construct an ordering based on the description of the relations provided. They are asked to select only one relation for the first and second commit and a combination of relations for the third one. However, to limit the complexity, the developers can select at most four relations. Furthermore, the developers are asked to explain the reason behind the choice of a particular relation (or groups of relations), if any.
We ask the developers to produce the first ordering during the interview to take notes about any comments, remarks or difficulties encountered. However, we argue that the ordering produced during the interview may use a commit or a relation simpler to apply: in fact, the person interviewed might not have enough time to analyze a complex commit or apply a relation that requires to find many links between software entities. To address this problem, we ask the developers to produce the other two orderings later, in a situation in which they can work comfortably. We provide them with the survey files so that they may check again the tasks, the relation descriptions and the example if needed.
Furthermore, the developers are asked to select one of the three possibilities for the *Center positioning* option, when it is applicable. They are also asked to state the reason behind their choice. The goal is to understand which of the three options is perceived as the most useful and if there is a reason behind it. The three options are presented in the *Example* file in a randomized order to reduce potential bias due to the choice of the option presented first.

[1]Apache/Karaf: https://github.com/apache/karaf

Table 5.1: RELATIONS IMPORTANCE RESULTS

Rank	Relation	Mean	Variance	Histogram
1	Method call	2.33	1.55	
2	Declare-use	3.16	3.80	
3	New Object	3.27	2.31	
4	Inheritance	4	2.88	
5	Parameter-use	4	3.11	
6	Field access	4.38	2.12	
7	Same file	5.11	5.76	
8	Same format	5.47	5.24	

In this procedure, our aims are two: verifying the correct functioning of the tool and assessing if the relations implemented are understandable and useful to the developers. However, in this step we decided to put more emphasis on the first objective. Therefore, the developers are allowed to ask questions during the interview. All their comments and questions are recorded to be object of further analysis.
Differently from the internally produced test set, orders produced using the "*Add unmodified code*" option have been excluded from our analysis. This choice has been done to limit the complexity of the analysis that developers are asked to conduct, since it may have caused major imprecisions in the produced test case.

5.2 Interviews results

At the end of this step, three different results were obtained: the creation of a manually produced test set, an investigation of the developers' perception of the tool relations and an assessment of the *center option* usefulness.
A group of 18 developers was interviewed. The majority of them preferred to use commits from the default project (Apache/Karaf). Only two developers used projects of their choice. The test set created contains 29 commits (among which 23 using only one relation and 6 with multiple relations): 25 from the default project and 4 from personal projects. The complete test set is available online[2].
Moreover, the investigation of the usefulness of the relations revealed that the *Method call* relation was considered the most important, followed by *Declare-use*, *New object*, *Parameter-use* and *Inheritance* relation. On the contrary, the *Same file* and *Same format* were perceived as the least important relations to create an ordering. However, we noticed that a common trend among the developers was to consider the *Same file* relation as granted when applying a different relation to create the test order during the interview. This led us to the conclusion that, although this relation has not been perceived as useful alone, it is considered as an important prerequisite that needs to be respected in producing orders applying the other relations.

[2]https://github.com/EFregnan/Automatic-ordering-of-code-changes_related-material

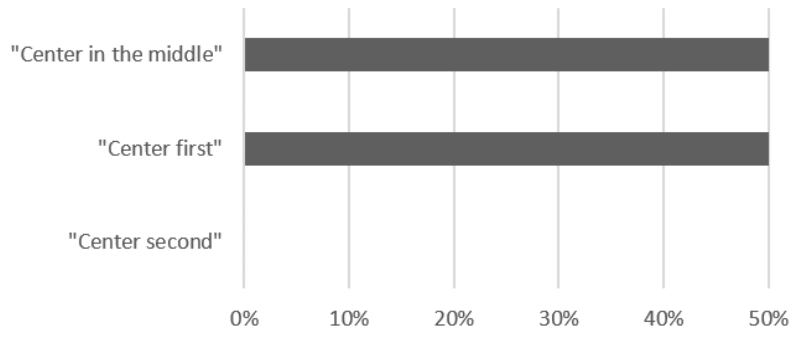


Figure 5.2: Frequency of the developers' choices for the center option

Table 5.1 ranks the relations implemented in the tool from the most to the least important based on the mean of the scores that the developers assigned. The possible values ranged from 1 (most important relation) to 8 (least important relation), therefore a lower values mean corresponds to a higher perceived importance of the relation. It is important to notice the high variance of the *Same file* and *Same format* relations. These relations have been alternatively considered as very important or not at all by the developers. On the contrary, the *Method call* relation presents a low variance, thus supporting our claim of its generally recognized usefulness.
Most of the developers did not express a reason behind the choice of a particular relation to create the test order. However, some of the few respondents stated that their choice was dictated by having identified that relation as the most important. Others stated that their decision was influenced by the constraint of having to create an order during the limited time of an interview. Therefore, they opted for relations that do not require a deep analysis of the code such as the *Same file* relation. Finally, some based their choice on the code in the commit, selecting a relation meaningful to produce a test case for the tool.
Moreover, some of the respondents expressed their remarks on new relations for the tool. One of them recognized as important to highlight the relation between the use of variables contained in an *if* statement and their related declaration. Another developer suggested to restrict the *Inheritance* relation only to change parts containing methods override. This idea is similar to the formulation of the *Class Hierarchy* relation proposed in the work conducted by Baum et al. [8] and reported in Table A.1.
The investigation of the center option revealed how the majority of developers preferred using the "*Center first*" or the "*Center in the middle*" option, while none of them recurred to the "*Center second*" option. This choice may have been caused by the major perceived usefulness of the first two center options compared to the last one. Unfortunately, none of them reported the reason behind this choice. Therefore, our analysis is based on the comments done by the developers during the interviews. Figure 5.2 shows the percentage on which the three options have been chosen in the production of the test cases.

Table 5.2: COMMITS AND RELATIONS USED IN THE SURVEY

Group	Commit id[4]	Date	Files	Change parts	Relation
Small	8217be3	13 Dec 2017	2	4	Method call
					Declare-use
Medium	76edcb8	7 Feb 2018	8	11	Method call
					Inheritance
					Method-call + Same file
Big	24fb477	24 Oct 2017	3	17	Method call
					Parameter-use
					Method call + Same file

5.3 Tool usefulness evaluation

In this step, the aim was to evaluate the perception that developers have of the order produced by the tool. Do they think it can improve their understanding of the code to review or are other orders identified as better options?
To accomplish this goal, a survey was designed and handed to a group of developers, asking them to identify which order among the ones presented was perceived as the most useful for code review. We involved in this survey all the developers who participated during the interviews in the previous phase. However, to enlarge the size of our sample, the survey was also released online. A different version of the survey was designed for this purpose, containing an initial page of preliminary questions (the same asked in the interviews). Of course, the questions regarding the project used to create the tests were removed since they were not applicable anymore.
Three commits of different sizes[3] have been selected from the example project (Apache/Karaf) used in the first part of our survey. In fact, we argue that the usefulness of an order may depend also on the number of change parts in the commit. We defined a commit as *Small* if it contains less than 10 change parts, *Medium* if it contains from 10 to 15 change parts and *Big* if it has more than 15 change parts. Furthermore, to avoid to make the survey unnecessary complex, a limit of maximum of 20 change parts was selected. Bigger commits may lead to a higher chance that the developers drop the survey or select a random answer.
Table 5.2 shows information about the commits used in the survey. Furthermore, for each commit the relations applied to construct the order are indicated. For each commit three different orders produced by the tool are included in the survey (one for each question) based on two relations and their combination with the *Same file* relation: in fact, we argue that the effectiveness of an order on the knowledge support may be increased keeping change parts from the same file together.
In each question of the survey, the respondent was asked to choose the most useful order to review the commit among the four different orderings presented. Between them, one

[3]measured in terms of number of change parts.
[4]For clarity only the first seven digits are reported.

Table 5.3: ORDERS USED IN THE SURVEY

Question	Commit	Relations	Orders
1	Small	Method call	Tool, Alph, WO, WFO
2	Small	Declare-use	Tool, Alph, WO, WFO
3	Medium	Method call	Tool, Alph, WO, WFO
4	Medium	Inheritance	Tool (CM), Tool (CF), Alph, WO
5	Medium	Method call + File	Tool, Alph, WFO-1, WFO-2
6	Big	Method call	Tool, Alph, WO, WFO
7	Big	Parameter-use	Tool (CM), Tool (CF), Alph, WO
8	Big	Method call + File	Tool, Alph, WFO-1, WFO-2

is obtained using the tool and the relations shown in Table 5.2. The *Method call* relation was identified as the most important in the previous step of our survey, therefore it was used in all the three commits. The *Declare-use*, *Inheritance* and *Parameter-use* relations have also been identified as important by the developers interviewed, so they were applied when meaningful. Moreover, our hypothesis is that keeping the file boundaries intact while applying a relation may lead to better results than an order that ignores them. For this reason, the combination of the *Same file* and *Method call* relations was also included. However, in the first commit, applying these two relations together leads to the same order produced by GitHub. Therefore, this option was not included in the survey.
The order produced by the tool was compared against three other orders. The first order, *Alph*, groups together change parts from the same file and order them based on the line number, while files are shown in alphabetical order (e.g, the order shown in GitHub). The second one is the worst order according to the formal theory presented by Baum et al. [8]. A worst order minimizes the constraints imposed by the theory's principles. Since the most important principle states to group related change parts together, a worst order (WO) contradicts this claim, positioning related change parts as distant as possible. Furthermore, two kinds of worst order exist: a general worst order that does not respect the file boundaries and a worst order that keeps in account the *Same file* relation (WFO). Both these two possibilities were used.
The survey contains a total of eight questions. Table 5.3 shows the different options given for each question, together with the commit and relation used. The options are presented in the survey in random order to reduce potential bias. Each change part is shown as composed of the name of the class on top and the modified portion of code (including a line of context before and after the modification, if applicable). This choice has been done to reduce potential bias introduced by a better graphical presentation of the changes.
According to our hypothesis, the review efficiency of the Worst Order is worse or equal than the one of the tool order and the alphabetical (*Alph*) order. Furthermore, the review efficiency of WO is worse than the one of WFO. Our aim is to compare the order produced by the tool with orders having the highest review efficiency possible to avoid a potential bias in the respondent choice: his/her answer may be driven towards the tool order if this is compared with obviously worse other options. For this reason, when possible, the WO

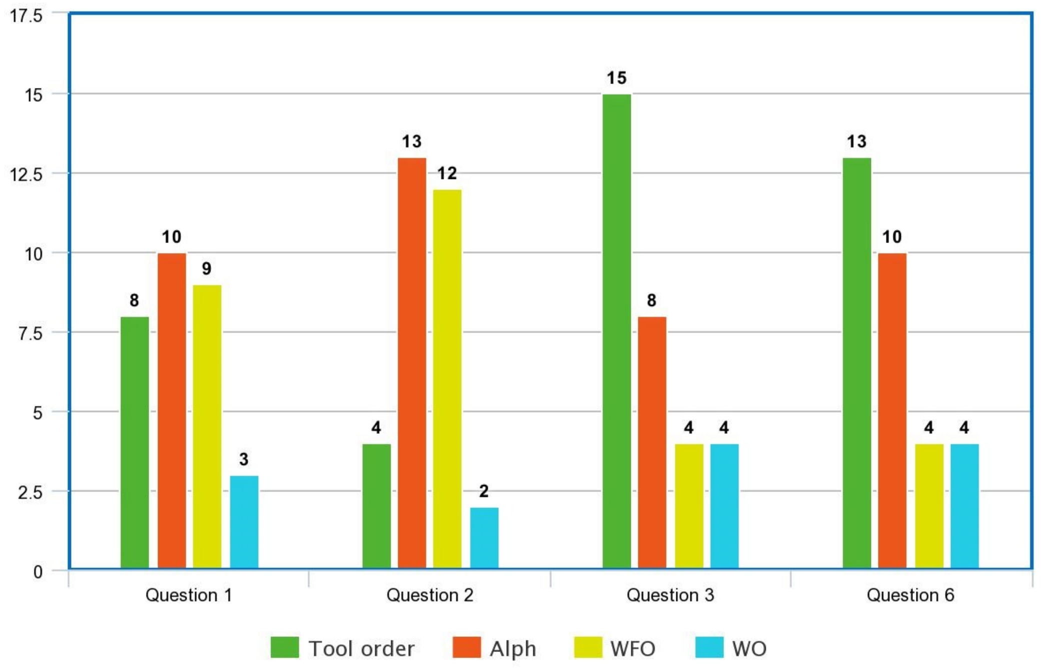


Figure 5.3: Survey results: questions 1, 2, 3 and 6 with Tool order, Alph, WFO and WO

option is replaced with the order produced by the tool using a different option for the center. Based on the results of the first part of our survey only the "*Center in the middle*" (CM) and "*Center first*" (CF) options were used, since the "*Center second*" option was not considered useful by most of the people interviewed. Furthermore, in questions 5 and 8 the order was created using the combination of *Method call* and *Same file* relation. We argue that including among the choices the Worst Order without file constraints would have been meaningless: WO would give a significant support for review compared to the other options, biasing the respondents towards immediately discarding it. Therefore, instead of WO as fourth option a second version of the WFO (Worst Order with file constraints) has been included. This is produced in a similar fashion of the first WFO, violating principle 1 (Table 2.1), but rearranging the change parts in a different order.

5.4 Survey results

30 developers took part in the online survey on the usefulness of the tool. All the people involved completed the questionnaire, except for one developer who abandoned it at question 6. Some developers indicated more than one order answering some of the questions.
In the two questions involving the small commit (question 1 and 2) there is no clear indication of a preference among the orders. In question 1, the order produced by the tool using the *Method call* relation, the *Alph* order and the WFO achieved a similar number of votes. A similar situation happened in question 2, where both the alphabetical order and WFO

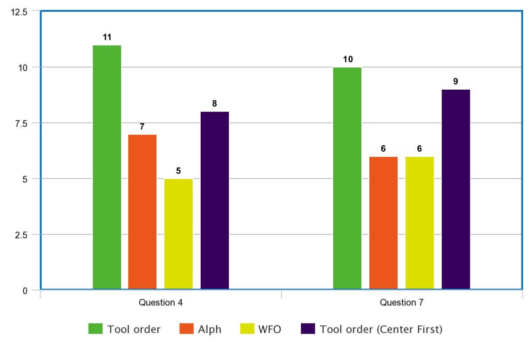


Figure 5.4 Survey results: questions 4 and 7 with Tool order, Alph, WFO and Tool order (CF)

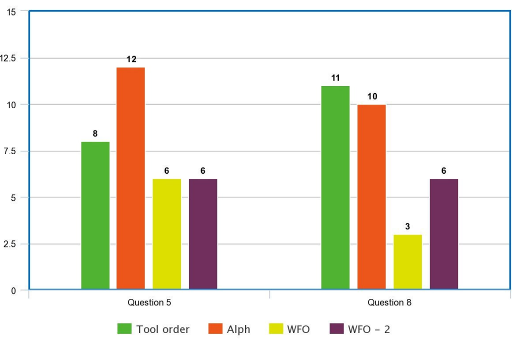


Figure 5.5 Survey results: questions 5 and 8 with Tool order, Alph and two WFO

obtained similar votes. We argue that these results are explained by the fact that having to review a small commit (in this case, the selected commit contained only 2 files and 4 change parts), the developers did not find a clear advantage in the order produced by the tool. In fact, the small size of the commit makes easy to understand relations among change parts even without grouping them using the tool ordering theory.
Questions 3, 4 and 5 involved a commit of medium size (8 files and 11 change parts). In question 3, the order produced by the tool applying the *Method call* relation has been identified as the most useful one. A similar assessment of the usefulness of the tool order is shown also in the results of question 4. In fact, in this case the last option represents the order created by the tool using the "*Center first*" option. Moreover, the option with the highest amount of votes is the tool order with the "*Center in the middle*". This shows the usefulness of the order produced by the tool and it gives an indication of the reviewers preference towards the "*Center in the middle*" instead of the "*Center first*" option. However, question 5 shows an opposite trend: in fact, the alphabetical order has been identified as more useful then the combination of the *Same file* relation and *Method call*. This may be caused by the excessive similarity between the *Alph* order and the tool order with the *Method call* and *Same file* relations.
Finally, questions 6, 7 and 8 involved orders created using a big commit (3 files and 17 change parts). In question 6, the order produced by the tool using the *Method call* relation has been identified as the most useful, followed by the alphabetical order. In question 7 both the order produced using the "*Center first*" and "*Center in the middle*" option were used. Combining the votes of these two orders, the tool order was identified as better than the alphabetical one. Finally, the respondents in question 8 showed a slight preference towards the order produced by the tool (*Method call* and *Same file* relation).
For the "*medium*" and "*big*" commits, all the results (with the only exception of question 5) show that ordering code changes using our tool is perceived useful by the developers. Although the limited number of respondents does not allow us to produce statistically significant conclusions, we argue that the trend highlighted is a good indication of the positive outcome of the tool application in real case scenarios.

Moreover, we investigated possible variations in the results dividing the respondents in groups based on their answers to the introductory questions: clusters were created based on their experience with Java and with code review. The aim was to investigate if inexperienced reviewers preferred different orders or relations compared to developers used to perform code review tasks. However, no significant trend was highlighted by this investigation.

5.5 Threats to validity

In this section, the limits of our approach are presented. First of all, the developers who took part in the first survey are personal contacts of the author. This may have introduced a bias in our results. To allow a verification of the procedure adopted in the interviews, they have been recorded and are available online[5].
The majority of the developers asked to create the test set decided to work with the backup project instead of using one of their choice. This limited the variability of the test set, reducing the amount of cases that it can cover. Moreover, a bias may be present in the orders that the developers created, due to potential mistakes in the analysis of the code of the selected commits or wrong understanding of the relations. We tried to mitigate this latter risk asking the interviewed developers to produce the first order during the interview, allowing them to ask questions when in doubt.
Another threat to the validity of this study is the number of people interviewed. We argue that the sample measured may be considered too small to obtain statistically significant results.
In each question of the online survey 4 different orders are presented. Although this number of options allowed us to reduce the risk that the tool order was selected randomly, it may require more effort to evaluate the difference between the orders. This risk needs to be taken in account especially for the questions involving the “*big*” commit.

[5]https://github.com/EFregnan/Automatic-ordering-of-code-changes_related-material

Chapter 6

Conclusions and Future Work

This chapter gives an overview of the project's contributions. After that, the results are presented and conclusions are drawn. Finally, ideas for future work are discussed.

6.1 Contributions

A tool to automatically order change parts in a commit, based on a practical implementation of the ordering theory proposed by Baum et al. [8], constitutes the main contribution of this work. Furthermore, a test set has been internally produced to verify the correctness of the tool functioning for each relation and the "*Add unmodified option*". To mitigate the potential bias presents in our first test set, a second test set has been created by a group of interviewed developers. These two sets of ordered change parts are available online and may be used for further research on the ordering theory. In this process, a study on the perceived usefulness of the relations has been conducted.
Moreover, an assessment of the usefulness of the order produced by the tool to perform code review, together with an investigation of the relations implemented, has been produced.

6.2 Conclusions

To answer the first research question (**RQ1**), a way to implement an ordering theory for code changes in a tool was investigated. This required us to take some decisions about the programming language and the relations to cover, together with the options to give to the user. We decided to restrict our tool implementation to the analysis of Java code. Moreover, based on the outcomes of a previous investigation on an ordering theory for code changes [8], we developed eight relations to be used to produce the order. A tour can be constructed applying only one of these relations or a combination of them. Furthermore, we implemented different options to allow the user to change the order produced by the tool based on his/her preferences. Two of these options, "*Positioning option*" and "*Centering option*" works modifying the relations direction, while the "*Add unmodified code*" option inserts in the final order files not included in the commit but related to the change parts under analysis.

Our second research question (**RQ2**) aimed to investigate how the tool orders could be verified. To answer it, two test sets have been produced: one by the tool developer and the other by a group of external developers. To produce this second test set, a group of 18 Java developers has been interviewed and each of them was asked to produce 3 different test cases. We compared the orders contained in the test sets to the ones automatically produced by CodeChangeOrderer to assure that they were the same. Only in a minority of cases we noticed some discrepancy between the two orders caused by the use of JAR files having a release date too old compared to the one of the commit under analysis: this led to the impossibility to solve all the relations between change parts.
Furthermore, we asked the respondents to evaluate the usefulness of the relations implemented in the tool. The *Method call* relation has been identified as the most useful, while *Same file* and *Same format* were not considered important to create an order.
Finally, the third research question (**RQ3**) aimed to assess the usefulness of the ordering to perform code review. To answer this question, an online survey was developed. The respondents were asked to choose the order of change parts that they would have used to perform code review of the given commit. The results of this investigation confirmed our claim that the tour produced by the tool is recognized useful to support developers during code review.

6.3 Discussion and lessons learned

During our investigation, different issues had to be faced on both implementing a theory to order code changes in a tool and evaluating its usefulness. This work combined engineering and research aspects: on the one hand, the need of constructing a usable and optimized tool, able to perform well in a real life scenario, on the other hand, the analysis of the ordering theory, how to implement it (e.g., which relations implement or how meaningful they are) and how to perform an initial study of its usefulness.
The results of the online form revealed that the order produced by the tool is considered as useful in most of the cases. To avoid potential bias in the decision, all the orders were presented in the same way without any explicit indication of the kind of relation used and the elements matched. However, principle 6, based on the findings of Baum et al. [8] (Table 3.1) states: "*To satisfy the other principles, use rules that the reviewer can understand.* ***Support this by making the grouping explicit to the reviewer***". Based on this, we argue that the order produced by the tool can reveal itself even more useful in a practical scenario compared to our survey, since the user has control on the relation(s) applied (which is, therefore, explicit to him). Moreover, the tool allows him or her to have access to the list of match sets constructed to have a clear understanding of the links among different change parts.

6.4 Future work

An initial assessment of the usefulness of the order produced by the tool in performing code review has been conducted by means of the questions and survey presented in Section 5.

However, to perform a complete test, the tool should be deployed in a practical scenario. For this reason, the aim of future work might be to ask developers to use this tool during their working routine. CodeChangeOrderer might be used together with the code review tool in use at a company, but we argue that the best option would be to integrate our tool with it. Of course, this puts some constraints on a future investigation since not all companies may be willing to give access to their tool code to external researchers.

If the ordering tool reveals itself useful, we can extend it with support for different programming languages. Furthermore, more relations may be investigated and included in the tool. Logical dependencies constitute a very promising group of relations to be applied to order change parts. Logical coupling relations, as proposed by Gall et al. [23] and Robbes et al. [41], consider entities that are frequently changed together in a program. For this reason, they may constitute an ideal new feature to help developers during code review: files that are changed together are likely to have a logical connection, which may not be possible to find with the relations currently implemented in the tool. Tools to extract these relations have already been proposed by the researchers: e.g., ROSE [51] or Evolution Radar [12].

Moreover, a new possible approach to the problem of grouping related code changes may be given by applying evolutionary coupling relations (as proposed by Zou et al. [52]). Information on which entities have been accessed together during the development phase may form a sound basis on which grouping together portions of code: these are likely to implement the same functionality. However, this requires having access to the data collected from the developer's IDE, e.g., integrating our tool in it.

Relations based on Semantic coupling [37] [38], based on the LSI analysis of the code elements, may also be object of further explorations. They could be a practical implementation of the *Similarity* relation (Table A.1) that was excluded in the current implementation of the tool due to its not well-defined nature.

Another promising research direction is to combine the ordering theory implemented in the tool with visualization techniques. In fact, we argue that a more immediate way to show related change parts may significantly increase the benefits of this approach. This claim is also supported by Principle 6, stated by Baum et al. [6], which argues the importance of making the grouping explicit to the reviewer (a complete formulation can be found in Table 2.1). This claim is also supported by D'Ambros et al., who stated the importance of visualization techniques as means to "*break down the complexity of information*" [12].

Bibliography

[1] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. Software inspections: an effective verification process. *IEEE Software*, 6(3):31–36, May 1989. ISSN 0740-7459. doi: 10.1109/52.28121.

[2] Jai Asundi and Rajiv Jayant. Patch review processes in open source software development communities: A comparative case study. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 166c–166c, Jan 2007.

[3] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 712–721, Piscataway, NJ, USA, 2013. IEEE Press.

[4] Mike Barnett, Christian Bird, João Brunet, and Shuvendu K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 134–144, May 2015. doi: 10.1109/ICSE.2015.35.

[5] Tobias Baum and Kurt Schneider. On the need for a new generation of code review tools. In *Product-Focused Software Process Improvement*, pages 301–308, Cham, 2016. Springer International Publishing.

[6] Tobias Baum, Olga Liskin, Kai Niklas, and Kurt Schneider. A faceted classification scheme for change-based industrial code review processes. *IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2016.

[7] Tobias Baum, Hendrik Leßmann, and Kurt Schneider. The choice of code review process: A survey on the state of the practice. In *Product-Focused Software Process Improvement*, pages 111–127, Cham, 2017. Springer International Publishing.

[8] Tobias Baum, Kurt Schneider, and Alberto Bacchelli. On the optimal order of reading source code changes for review. *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017.

[9] Mario Bernhart and Thomas Grechenig. On the understanding of programs with continuous code reviews. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 192–198, May 2013.

[10] Stefan Biffl. Analysis of the impact of reading technique and inspector capability on individual inspection performance. In *Proceedings Seventh Asia-Pacific Software Engeering Conference. APSEC 2000*, pages 136–145, 2000.

[11] David B. Bisant and James R. Lyle. A two-person inspection method to improve programming productivity. *IEEE Trans. Softw. Eng.*, 15(10):1294–1304, October 1989.

[12] Marco D'Ambros, Michele Lanza, and Mircea Lungu. Visualizing co-change information with the evolution radar. *IEEE Transactions on Software Engineering*, 35(5): 720–735, Sept 2009. ISSN 0098-5589. doi: 10.1109/TSE.2009.17.

[13] Scott Deerwester, Susan T. Dumais, George Furnas, Thomas Landauer, and Richard Harshman. Indexing by latent semantic analysis. 41:391–407, 09 1990.

[14] Martin Dias, Alberto Bacchelli, Georgios Gousios, Damien Cassou, and Stéphane Ducasse. Untangling fine-grained code changes. *CoRR*, abs/1502.06757, 2015.

[15] E. P. Doolan. Experience with fagan's inspection method. *Softw. Pract. Exper.*, 22(2): 173–182, February 1992. ISSN 0038-0644.

[16] Alastair Dunsmore, Marc Roper, and Murray Wood. The role of comprehension in software inspection. *Journal of Systems and Software*, 52(2):121 – 129, 2000.

[17] Alastair Dunsmore, Marc Roper, and Murray Wood. Systematic object-oriented inspection - an empirical study. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 135–144, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1050-7.

[18] Michael Dyer. Verification based inspection. In *Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences*, volume ii, pages 418–427 vol.2, Jan 1992.

[19] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976. ISSN 0018-8670. doi: 10.1147/sj.153.0182.

[20] Michael E. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, 12:744–751, Jul 1986.

[21] Dror G. Feitelson, Eitan. Frachtenberg, and Kent L. Beck. Development and deployment at facebook. *IEEE Internet Computing*, 17(4):8–17, July 2013.

[22] Daniel P. Freedman and Gerald M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products*. Dorset House Publishing Co., Inc., New York, NY, USA, 3rd edition, 2000.

[23] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, IWPSE '03, pages 13–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1903-2.

[24] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 121–130, Piscataway, NJ, USA, 2013. IEEE Press.

[25] Kim Herzig, Sascha Just, and Andreas Zeller. The impact of tangled code changes on defect prediction models. *Empirical Software Engineering*, 21(2):303–336, Apr 2016.

[26] Philip M. Johnson and Danu Tjahjono. Does every inspection really need a meeting? *Empirical Software Engineering*, 3(1):9–35, Mar 1998.

[27] David Kawrykow and Martin P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 351–360, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0.

[28] Lesley Pek Wee Kim, Chris Sauer, and Ross Jeffery. *A Framework for Software Development Technical Reviews*, pages 294–299. Springer US, Boston, MA, 1995.

[29] 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. ISBN 978-1-4503-2879-1.

[30] John C. Knight and E. Ann Myers. An improved inspection technique. *Commun. ACM*, 36(11):51–61, November 1993.

[31] Patrick Kreutzer, Georg Dotzler, Matthias Ring, Bjoern M. Eskofier, and Michael Philippsen. Automatic clustering of code changes. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 61–72, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4186-8.

[32] Alex Kuhn, Stéphane Ducasse, and Tudor Girba. Enriching reverse engineering with semantic clustering. In *12th Working Conference on Reverse Engineering (WCRE'05)*, pages 10 pp.–, Nov 2005.

[33] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1.

[34] Jonathan I. Maletic and Andrian Marcus. Using latent semantic analysis to identify similarities in source code to support program understanding. In *Proceedings 12th IEEE Internationals Conference on Tools with Artificial Intelligence. ICTAI 2000*, pages 46–53, 2000.

[35] Johnny Martin and Wei-Tek Tsai. N-fold inspection: A requirements analysis technique. *Commun. ACM*, 33(2):225–232, February 1990.

[36] David L. Parnas and David M. Weiss. Active design reviews: Principles and practices. *Journal of Systems and Software*, 7(4):259 – 265, 1987.

[37] Denys Poshyvanyk and Andrian Marcus. The conceptual coupling metrics for object-oriented systems. In *2006 22nd IEEE International Conference on Software Maintenance*, pages 469–478, Sept 2006.

[38] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, Feb 2009.

[39] Peter Rigby, Brendan Cleary, Frederic Painchaud, Margaret-Anne Storey, and Daniel German. Contemporary peer review in action: Lessons from open source development. *IEEE Software*, 29(6):56–61, Nov 2012.

[40] Peter C. Rigby and Christian Bird. Convergent contemporary software peer review practices. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 202–212, New York, NY, USA, 2013. ACM.

[41] Romain Robbes, Damien Pollet, and Michele Lanza. Logical coupling based on fine-grained change information. In *2008 15th Working Conference on Reverse Engineering*, pages 42–46, Oct 2008.

[42] Chris Sauer, D. Ross Jeffery, Lesley Land, and Philip Yetton. The effectiveness of software development technical reviews: a behaviorally motivated program of research. *IEEE Transactions on Software Engineering*, 26(1):1–14, Jan 2000.

[43] Forrest Shull and Carolyn Seaman. Inspecting the history of inspections: An example of evidence-based technology diffusion. *IEEE Software*, 25(1):88–90, Jan 2008.

[44] Dag I. K. Sjøberg, Tore Dybå, Bente C. D. Anda, and Jo E. Hannay. *Building Theories in Software Engineering*, pages 312–336. Springer London, London, 2008.

[45] Yida Tao and Sunghun Kim. Partitioning composite code changes to facilitate code review. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 180–190, May 2015.

[46] 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*, FSE '12, pages 51:1–51:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9.

[47] Gerald M. Weinberg and Daniel P. Freedman. Reviews, walkthroughs, and inspections. *IEEE Transactions on Software Engineering*, SE-10(1):68–72, Jan 1984.

[48] Aiko Yamashita and Leon Moonen. Do developers care about code smells? an exploratory survey. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 242–251, Oct 2013.

[49] Edward Yourdon. *Structured Walkthroughs: 4th Edition*. Yourdon Press, Upper Saddle River, NJ, USA, 1989.

[50] Tianyi Zhang, Myoungkyu Song, Joseph Pinedo, and Miryung Kim. Interactive code review for systematic changes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 111–122, Piscataway, NJ, USA, 2015. IEEE Press. ISBN 978-1-4799-1934-5.

[51] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.

[52] Lijie Zou, Michael W. Godfrey, and Ahmed E. Hassan. Detecting interaction coupling from task interaction histories. In *15th IEEE International Conference on Program Comprehension (ICPC '07)*, pages 135–144, June 2007.

Appendix A

Relations Table

Table A.1: TYPES OF RELATIONS AMONG CHANGE PARTS [8]

Type	Directed	Description
Data flow	Yes	A common example was a configuration file that was put before the code for loading the configuration. In this case data flow and call flow do not coincide, whereas they are often hard to distinguish in other cases.
Call flow	Yes	Going from a place that calls a method to the called method (top-down)or the other way around (bottom-up).
Class Hierarchy	Yes	A method overrides an operation in a superclass
Declare-Use	Yes	Something, e.g., an attribute, is declared in one change part and used in another.
File order	Yes	Change parts in the same file are related just by being in that file. Inside the file, they can be ordered by line number.
File Type	No	One of the participants talked about separating the XML file from the Java files in one of our examples.

Similarity	No	Some change parts are very similar to each other. If they are close together, the similarities and differences are easier to compare. The exact meaning of similarity has yet to be determined.
Common Identifier	No	One of the participants named a common identifier as a reason for putting change parts together. In this case this could be seen as a special case of data ow, but it could also be a special form of similarity.
Logical dependency	Yes	A logical dependency exists when some part of the code has to change when another changes. Some of the participants talked about putting change parts together because one was the reason for the other.
Development flow	Yes	Some participants stated that they would like to fol- low the development ow of the author. An optimal development ow is probably very similar to a ow based on logical dependencies.

Appendix B

Survey files

In this appendix, the three documents handled to the developers are presented. The first one is the main document, containing a brief questionnaire and an explanation of the task. The second explains the relations implemented by the tool, while the last contains a brief example of the order produced by the tool.

Thank you for finding the time to participate in this survey.

I am Enrico Fregnan, a MSc. thesis student at Delft University of Technology, currently working on code changes ordering for review under the supervision of Prof. Alberto Bacchelli (ZEST, University of Zurich) and M. Eng. Tobias Baum (Leibniz Universität Hannover). Being well aware of the difficulties of performing code review, we are building a tool to automatically order changes in a commit in the attempt to make code review easier. Our aim is to help developers to save time and effort in performing this task. Your aid will be of great importance to allow us to further improve our research.
We kindly request your help to evaluate our tool's performance.

1) First of all, **please answer to the following general questions:**

1. How often do you currently do programming?
   - ○ About once a day or more often
   - ○ About once a week
   - ○ About once a month
   - ○ About once a year
   - ○ Not at all

2. On which kind of projects have you worked?
   - ▪ Company proprietary projects
   - ▪ Open-Source projects
   - ▪ University projects
   - ▪ Others

3. Which category does the project that you selected for this survey belong to?
   - Company proprietary projects
   - Open-Source projects
   - University projects
   - Others

4. Which is/was your predominant role in the selected project?
   - ▪ Developer
   - ▪ Reviewer
   - ▪ User
   - ▪ No role

5. How many years of experience do you have with Java?
   - ○ 1 year or less
   - ○ 2 years
   - ○ 3 – 5 years
   - ○ 6 – 10 years
   - ○ 11 years or more

6. How often do you perform code review tasks? Choose an answer from the following options:
   - About once a day or more often
   - About once a week
   - About once a month
   - About once a year
   - Not at all

7. Order the set of relations offered by the tool (please refer to the "*Relations*" file) in terms of importance assigning a number from 1 (most important) to 8 (least important).

The data collected will be aggregated and treated anonymously.

2) In the second step, please select three commits from a project on which you feel comfortable working and provide us an order of the change parts in it that you consider right based on our description in the file "*Relations*". (A **change part** is a portion of code that has been modified. It can consist of either a single line or multiple lines). Please indicate the project name, a link to its repository and the code of the commits selected. If you prefer, you can instead use the following open-source project: *Apache/Karaf* (available here: https://github.com/apache/karaf).

To order a situation in which a single change part is linked to multiple others (e.g., a variable declared in a change part and then used in multiple others) you have three different options among which you can choose:

- "*Center in the middle*": the change part linked to all the others is positioned in the middle of the change parts to which is linked.
- "*Center first*": the change part linked to the others is positioned before the ones to which is linked.
- "*Center second*": the change part linked to all the others is positioned second, just after one of those to which it is linked but before the others.

Refer to the "*Example*" file for a concrete example.

We ask you to provide:

- **3 orderings (one for each commit):** for the first two you must select only one relation among the list of relations given, while for the third one you must select at least two different relations (up to a maximum of 4).

- The relations chosen for each commit and the reason behind your choice.

- The selected option among "*center in the middle*", "*center first*" and "*center second*" (if it was necessary in the chosen commits).

Constraints that need to be respected:

1) The project that you select must meet the following requirements:

- Publicly available on GitHub
- JAR files available: select a project of which the JAR files are available (or of which you can give us the JAR files).
- In case you want to use our default project (Apache/Karaf), please consider commits created no later than the end of March.

2) Keep in mind the following limitations when creating an order

- Please indicate every change part by the name of the class to which it belongs, followed by the line number of the modification in the new file (you can find a concrete example in the "*Example*" file).
- For change parts containing only deleted portions of code, please use as line number of the change part the line number of the first line after the end of the removed code block.
- When no other order applies, it is determined by the line number with which the change part begins.
- Deleted code portions are not considered for code analysis (e.g., a deleted variable declaration will not be linked to its use or a deleted class will not be linked to its parent).
- The tool currently supports only analysis of Java code: if the commit that you choose contains a file of different format, change parts belonging to it can still be ordered using the same file and same format relations. Furthermore, the order based on the line number still applies.

Let us know if you any remarks and if you want to be informed about the outcome of this research.

Thank you again for your time.

Thank you for finding the time to participate in this survey. Here, you will find an explanation of the tool's features and the relations that are currently implemented.

Please keep in mind that right now the tool can only order change parts written in Java code. Therefore, it will not be possible to apply any relation that requires the analysis of the code to different programming languages. You are not obliged to select commits that contains only Java code, but keep this limitation in mind when you are going to produce your order.

The relations that the tool supports are:

- **Same file**: change parts belonging to the same file are positioned together.
- **Same format:** change parts belonging to files having the same format are positioned close together (e.g., all change parts belonging to Java files will be in the same group)
- **New Object:** a change part which instantiates a new object will be positioned close to all the change parts belonging to the object that has been created.
- **Method Call:** a change part which calls a method will be positioned together with the change parts belonging to the method declaration. The relation holds either if the change parts are in different classes or if the change parts are contained in the same class.
- **Inheritance relation:** a child class is positioned close to its parent.
- **Declare-Use:** change parts containing variable declarations are positioned close to the ones containing the use of that variable. Please notice that this relation does not cover the case of parameter declaration and use (refer to parameter-use).
- **Parameter-Use:** change parts containing the declaration of a parameter are positioned close to the ones containing its use.
- **Field access:** a change part which uses the field of another class (or the same class) is positioned together with the field declaration.
  This relation covers cases of direct access to a field and the use of the *"this"* construct.

Multiple relations can be applied to determine an order. Note that not all the relations have the same granularity and therefore they can be applied together to determine an inner order inside a previously defined group. Please refer to the "*Example*" file attached for a practical case.

Example

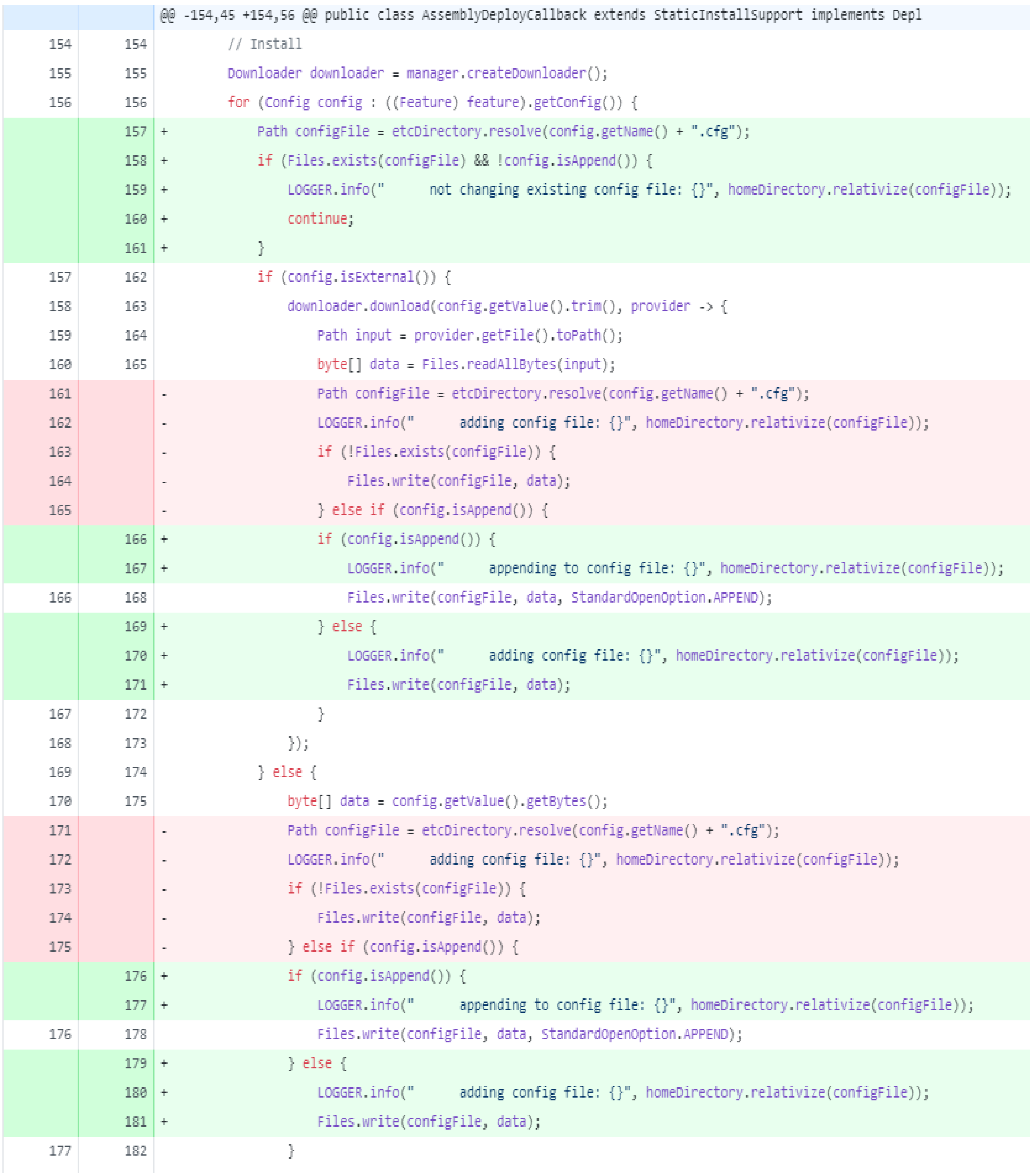
Here you can find an example of possible ordering:

Consider the default project Apache/Karaf and the commit with code:

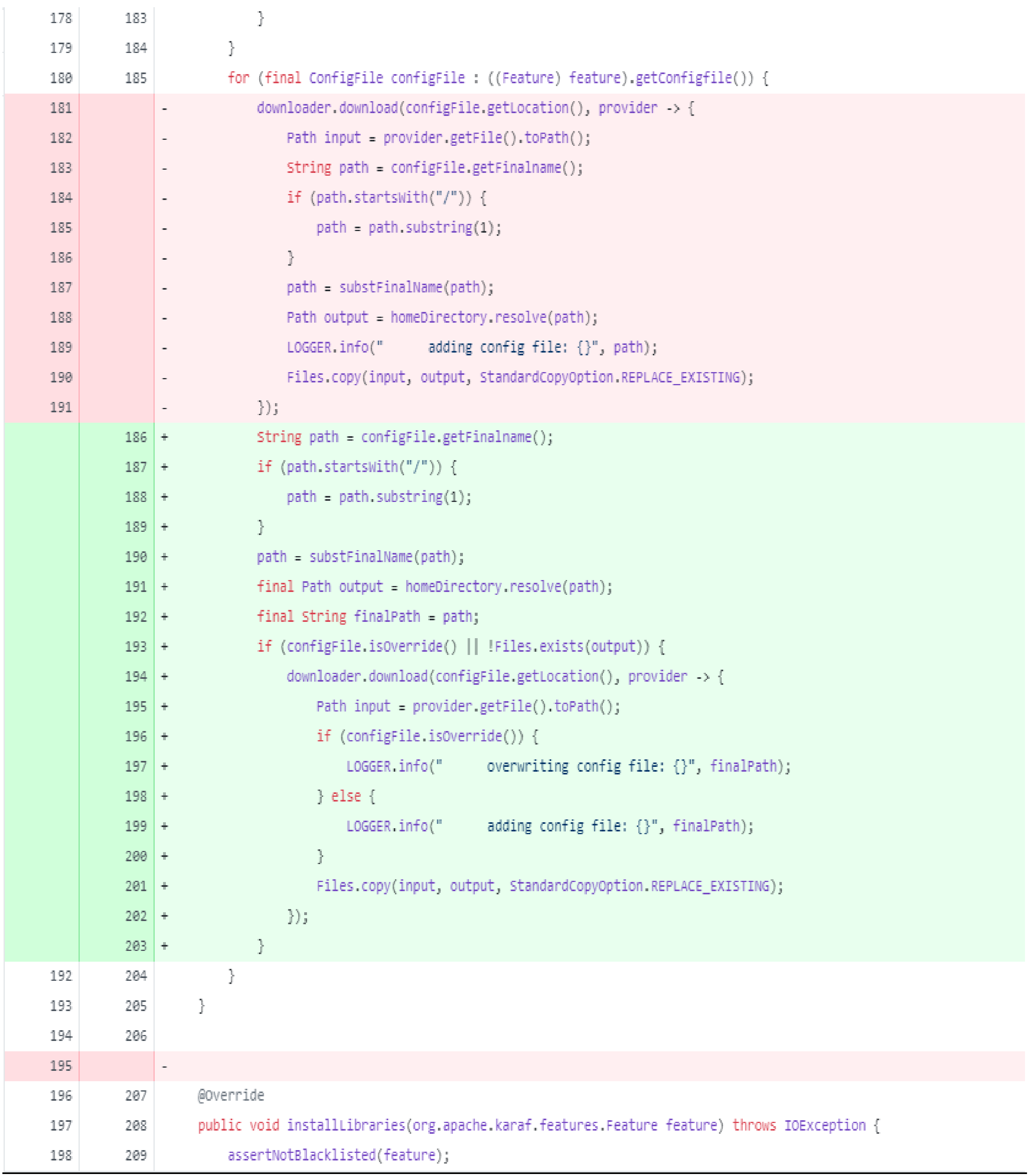
cde8c4954e04e2de7c52252c15eaace11deadd29

which contains three files:

AssemblyDeployCallback.java



```
@@ -154,45 +154,56 @@ public class AssemblyDeployCallback extends StaticInstallSupport implements Depl
          // Install
          Downloader downloader = manager.createDownloader();
          for (Config config : ((Feature) feature).getConfig()) {
+             Path configFile = etcDirectory.resolve(config.getName() + ".cfg");
+             if (Files.exists(configFile) && !config.isAppend()) {
+                 LOGGER.info("      not changing existing config file: {}", homeDirectory.relativize(configFile));
+                 continue;
+             }
              if (config.isExternal()) {
                  downloader.download(config.getValue().trim(), provider -> {
                      Path input = provider.getFile().toPath();
                      byte[] data = Files.readAllBytes(input);
-                     Path configFile = etcDirectory.resolve(config.getName() + ".cfg");
-                     LOGGER.info("      adding config file: {}", homeDirectory.relativize(configFile));
-                     if (!Files.exists(configFile)) {
-                         Files.write(configFile, data);
-                     } else if (config.isAppend()) {
+                     if (config.isAppend()) {
+                         LOGGER.info("      appending to config file: {}", homeDirectory.relativize(configFile));
                          Files.write(configFile, data, StandardOpenOption.APPEND);
+                     } else {
+                         LOGGER.info("      adding config file: {}", homeDirectory.relativize(configFile));
+                         Files.write(configFile, data);
                      }
                  });
              } else {
                  byte[] data = config.getValue().getBytes();
-                 Path configFile = etcDirectory.resolve(config.getName() + ".cfg");
-                 LOGGER.info("      adding config file: {}", homeDirectory.relativize(configFile));
-                 if (!Files.exists(configFile)) {
-                     Files.write(configFile, data);
-                 } else if (config.isAppend()) {
+                 if (config.isAppend()) {
+                     LOGGER.info("      appending to config file: {}", homeDirectory.relativize(configFile));
                      Files.write(configFile, data, StandardOpenOption.APPEND);
+                 } else {
+                     LOGGER.info("      adding config file: {}", homeDirectory.relativize(configFile));
+                     Files.write(configFile, data);
                  }
```

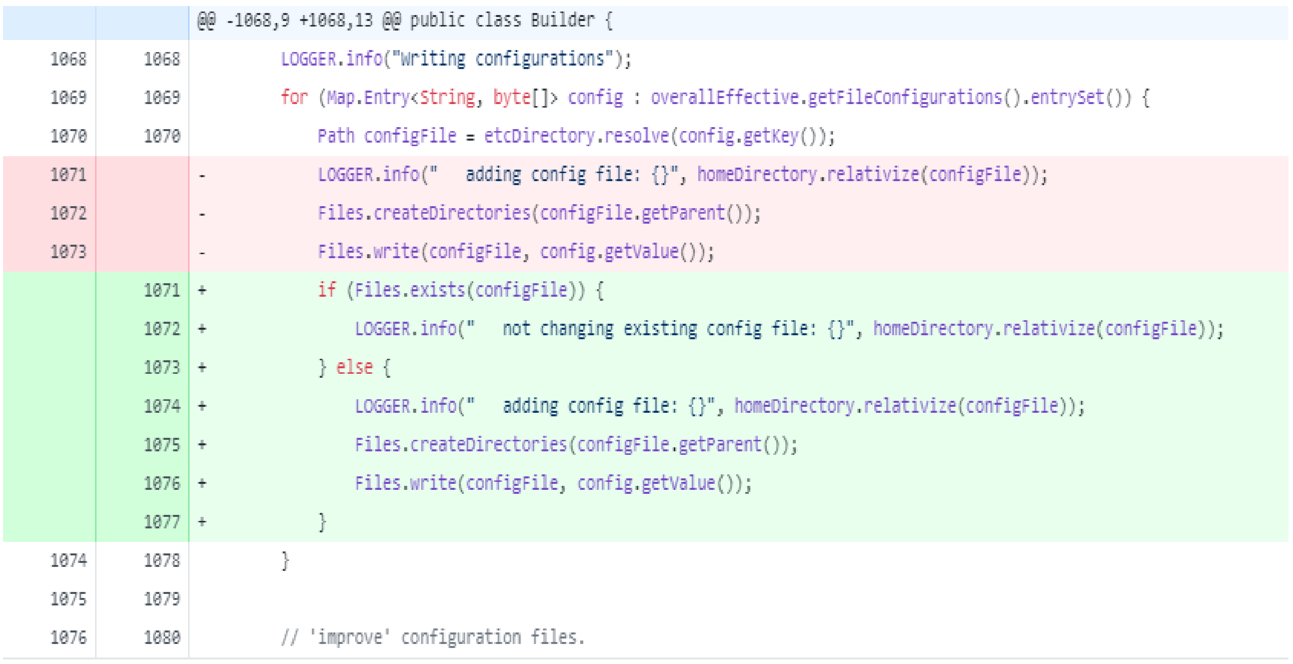


```
178  183               }
179  184           }
180  185           for (final ConfigFile configFile : ((Feature) feature).getConfigfile()) {
181       -            downloader.download(configFile.getLocation(), provider -> {
182       -                Path input = provider.getFile().toPath();
183       -                String path = configFile.getFinalname();
184       -                if (path.startsWith("/")) {
185       -                    path = path.substring(1);
186       -                }
187       -                path = substFinalName(path);
188       -                Path output = homeDirectory.resolve(path);
189       -                LOGGER.info("      adding config file: {}", path);
190       -                Files.copy(input, output, StandardCopyOption.REPLACE_EXISTING);
191       -            });
     186  +            String path = configFile.getFinalname();
     187  +            if (path.startsWith("/")) {
     188  +                path = path.substring(1);
     189  +            }
     190  +            path = substFinalName(path);
     191  +            final Path output = homeDirectory.resolve(path);
     192  +            final String finalPath = path;
     193  +            if (configFile.isOverride() || !Files.exists(output)) {
     194  +                downloader.download(configFile.getLocation(), provider -> {
     195  +                    Path input = provider.getFile().toPath();
     196  +                    if (configFile.isOverride()) {
     197  +                        LOGGER.info("      overwriting config file: {}", finalPath);
     198  +                    } else {
     199  +                        LOGGER.info("      adding config file: {}", finalPath);
     200  +                    }
     201  +                    Files.copy(input, output, StandardCopyOption.REPLACE_EXISTING);
     202  +                });
     203  +            }
192  204           }
193  205       }
194  206
195       -
196  207       @Override
197  208       public void installLibraries(org.apache.karaf.features.Feature feature) throws IOException {
198  209           assertNotBlacklisted(feature);
```

It contains the following change parts (here indicated by the initial letters of the class name and the line number on which the change part begins):

ADC157, ADC166, ADC169, ADC176, ADC179, ADC186, ADC207

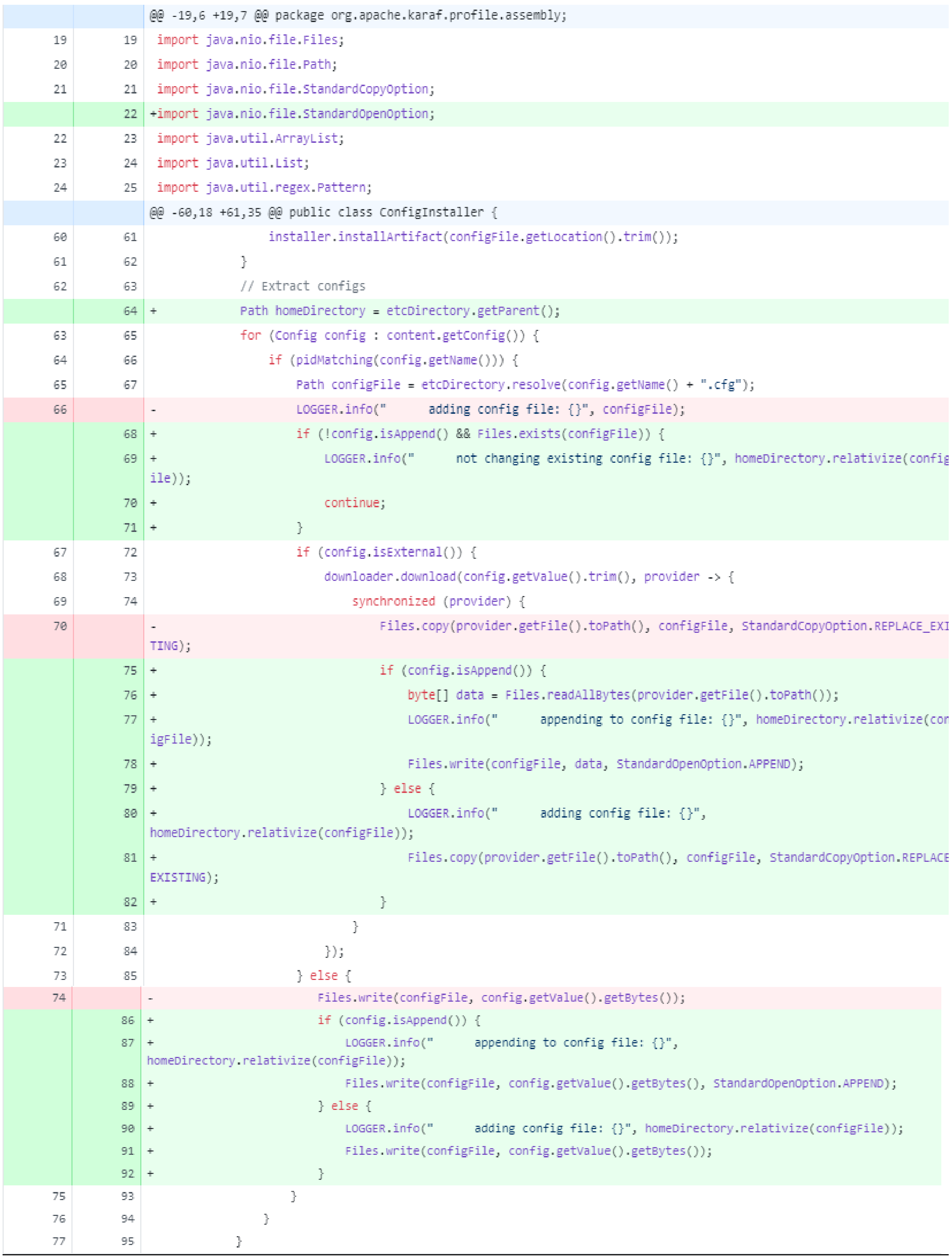
Builder.java



```
          @@ -1068,9 +1068,13 @@ public class Builder {
1068 1068          LOGGER.info("Writing configurations");
1069 1069          for (Map.Entry<String, byte[]> config : overallEffective.getFileConfigurations().entrySet()) {
1070 1070              Path configFile = etcDirectory.resolve(config.getKey());
1071      -           LOGGER.info("   adding config file: {}", homeDirectory.relativize(configFile));
1072      -           Files.createDirectories(configFile.getParent());
1073      -           Files.write(configFile, config.getValue());
     1071 +           if (Files.exists(configFile)) {
     1072 +               LOGGER.info("   not changing existing config file: {}", homeDirectory.relativize(configFile));
     1073 +           } else {
     1074 +               LOGGER.info("   adding config file: {}", homeDirectory.relativize(configFile));
     1075 +               Files.createDirectories(configFile.getParent());
     1076 +               Files.write(configFile, config.getValue());
     1077 +           }
1074 1078          }
1075 1079
1076 1080          // 'improve' configuration files.
```

This file contains only one change part: B1071

ConfigInstaller.java



```
            @@ -19,6 +19,7 @@ package org.apache.karaf.profile.assembly;
19    19     import java.nio.file.Files;
20    20     import java.nio.file.Path;
21    21     import java.nio.file.StandardCopyOption;
      22    +import java.nio.file.StandardOpenOption;
22    23     import java.util.ArrayList;
23    24     import java.util.List;
24    25     import java.util.regex.Pattern;
            @@ -60,18 +61,35 @@ public class ConfigInstaller {
60    61                     installer.installArtifact(configFile.getLocation().trim());
61    62                 }
62    63                 // Extract configs
      64    +          Path homeDirectory = etcDirectory.getParent();
63    65                 for (Config config : content.getConfig()) {
64    66                     if (pidMatching(config.getName())) {
65    67                         Path configFile = etcDirectory.resolve(config.getName() + ".cfg");
66          -                  LOGGER.info("      adding config file: {}", configFile);
      68    +                  if (!config.isAppend() && Files.exists(configFile)) {
      69    +                      LOGGER.info("      not changing existing config file: {}", homeDirectory.relativize(config
            ile));
      70    +                      continue;
      71    +                  }
67    72                         if (config.isExternal()) {
68    73                             downloader.download(config.getValue().trim(), provider -> {
69    74                                 synchronized (provider) {
70          -                                  Files.copy(provider.getFile().toPath(), configFile, StandardCopyOption.REPLACE_EXI
            TING);
      75    +                                  if (config.isAppend()) {
      76    +                                      byte[] data = Files.readAllBytes(provider.getFile().toPath());
      77    +                                      LOGGER.info("      appending to config file: {}", homeDirectory.relativize(con
            igFile));
      78    +                                      Files.write(configFile, data, StandardOpenOption.APPEND);
      79    +                                  } else {
      80    +                                      LOGGER.info("      adding config file: {}",
            homeDirectory.relativize(configFile));
      81    +                                      Files.copy(provider.getFile().toPath(), configFile, StandardCopyOption.REPLACE
            EXISTING);
      82    +                                  }
71    83                                 }
72    84                             });
73    85                         } else {
74          -                          Files.write(configFile, config.getValue().getBytes());
      86    +                          if (config.isAppend()) {
      87    +                              LOGGER.info("      appending to config file: {}",
            homeDirectory.relativize(configFile));
      88    +                              Files.write(configFile, config.getValue().getBytes(), StandardOpenOption.APPEND);
      89    +                          } else {
      90    +                              LOGGER.info("      adding config file: {}", homeDirectory.relativize(configFile));
      91    +                              Files.write(configFile, config.getValue().getBytes());
      92    +                          }
75    93                         }
76    94                     }
77    95                 }
```

It contains the following change parts:

CI22, CI64, CI68, CI75, CI86

Example ordering

For this example's sake, the two relations Same File and Declare-Use will be used.

Applying only the “*Same file”* relation we obtain:

CI22, CI64, CI68, CI75, CI86, ADC157, ADC166, ADC169, ADC176, ADC179, ADC186, ADC207, B1077

Applying only the “*Declare-use*” relation we obtain:

CI22, CI68, CI64, CI75, CI86, ADC166, ADC169, ADC157, ADC176, ADC179, ADC186, ADC207, B1077

In fact, in the class **ConfigInstaller** the variable *homedirectory* (line 64) is used in the change parts 68, 75 and 86. For this reason, the tool positions change part at line 64 (containing the definition of *homedirectory*) as close as possible to all the change parts in which this variable is used. The same applies also for the class **AssemblyDeployCallback,** where the variable *configFile* (line **157**) is used in the change parts at line 166, 169, 176, 179.

Please notice that in case of multiple change parts connected to the same one (e.g., multiple variable uses connected to the same variable declaration), you can use three different options to order them:

1. **“center in the middle”**: as reported in the example above
2. “**center first”:** the change parts connected to all the others are put before them.

CI22, **CI64**, CI68, CI75, CI86, **ADC157**, ADC166, ADC169, ADC176, ADC179, ADC186, ADC207, B1077

3. “**center second”:** the change parts connected to all the others are put in second position.

CI22, CI68, **CI64**, CI75, CI86, ADC166, **ADC157,** ADC169, ADC176, ADC179, ADC186, ADC207, B1077

To produce your order you are free to use the one that you prefer (please report which one you used)

Applying both relations at the same time, we obtain (using the “center in the middle” option):

CI22, CI68, CI64, CI75, CI86, ADC166, ADC169, ADC157, ADC176, ADC179, ADC186, ADC207, B1077

In this case, the order is the same as the one produced by the declare-use relation only because change parts from the same file were already ordered close to each other due to their line number.