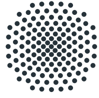

Universität Stuttgart

Mining Software Repositories for Coupled Changes

Von der Fakultät für Informatik, Elektrotechnik und Informationstechnik der Universität Stuttgart zur Erlangung der Würde eines Doktors der Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

Jasmin Ramadani

aus Tetovo, Mazedonien

Hauptberichter: Prof. Dr. Stefan Wagner

Mitberichter: Prof. Dr. Barbara Russo

Tag der mündlichen Prüfung: 19.09.2017

Institut für Softwaretechnologie

2017

Contents

1 Introduction **19**

1.1 Motivation . . . 19

1.2 Problem Statement . . . 22

1.3 Research Objective . . . 22

1.4 Contribution . . . 23

1.5 Thesis Outline . . . 25

1.5.1 Previously Published Material . . . 26

2 Background **29**

2.1 Software Repositories . . . 29

2.1.1 Version Control Systems . . . 29

2.1.2 Issue Tracking System . . . 30

2.1.3 Software Documentation . . . 31

2.2 Coupled File Changes . . . 31

2.2.1 Atomic Change Sets . . . 32

2.2.2 Related Changes . . . 32

2.2.3 Heuristics 34
2.2.4 Mapping Between Commits and Issues 35
2.3 System Packages 37
2.4 Data Mining 38
2.4.1 Mining Frequent Itemsets 39
2.4.2 FP-Growth Algorithm 40
2.4.3 Data Mining Framework 46
2.5 Developers' Feedback 46
2.5.1 Types of Feedback 47
2.5.2 Surveys 47
2.5.3 Interviews 47
2.5.4 Observation 48
2.6 Software Maintenance 48
2.6.1 Maintenance Categories 49
2.6.2 Maintenance Activities 50
2.6.3 Search for Task Relevant Information Sources . 50
2.6.4 Developer Expertise Profiles 51

3 State of the Art 53
3.1 Logical Couplings 53
3.1.1 Granularity 53
3.1.2 Data Sources 54
3.1.3 Commit Messages 54
3.1.4 Change Set Heuristics 55
3.2 Mining Repositories 56
3.2.1 Techniques 56
3.2.2 Algorithms 56
3.3 Developers' Feedback 57
3.3.1 Couplings and Feedback 57

3.3.2 Video Materials and Feedback 57
3.4 Assessment of Maintenance Tasks 58
3.5 Help Seeking During Maintenance Tasks 60
3.6 System Packages 61
3.7 Developer's Expertise 61
3.7.1 Expertise Identification Based on The Contribution 61
3.7.2 Expertise Identification Based on The Developer Roles 62
3.7.3 Expertise Identification Based on Software Repository Analysis 63
3.8 Recommender Tools 64

4 Coupled File Change Suggestions 67
4.1 Coupled File Changes 68
4.2 Obtaining Coupled File Changes 68
4.2.1 Data Extraction 69
4.2.2 Data Preparation 70
4.2.3 Mining Coupled Files 71
4.3 Obtaining Repository Attributes 71
4.3.1 Extracting Commit Attributes Set 72
4.3.2 Extracting Issue Attributes Set 73
4.3.3 Extracting Documentation Attributes Set 73
4.4 Building Coupled Changes Suggestions 74
4.4.1 Database Structure 74
4.4.2 Data Joining 74
4.4.3 Coupled Change Suggestions Example 77
4.5 Developer Expertise Profiles Based on Coupled Packages 80

5 Theory on the Use of Coupled File Change Suggestions 83
5.1 Use of Coupled File Change Suggestions 84
5.1.1 Interestingness of Coupled File Change Suggestions 84
5.1.2 Usefulness of Coupled File Change Suggestions 84
5.2 Theory Building Description 85
5.2.1 Constructs 86
5.2.2 Propositions 92
5.2.3 Explanations 94
5.2.4 Scope of the Theory 98
5.2.5 Theory Testing 98

6 Mining Related Change sets in Git: A Quasi-Experimental Study 101
6.1 Introduction 101
6.2 Experimental Design 103
6.2.1 Research Questions 103
6.2.2 Hypotheses 104
6.2.3 Experimental Variables 105
6.2.4 Experiment Design 105
6.2.5 Objects 106
6.2.6 Experiment Instruments 107
6.2.7 Data Collection Procedure 107
6.2.8 Analysis Procedure 109
6.3 Results and Discussion 113
6.3.1 Descriptive Statistics 113
6.3.2 Influence of the time between the commits and the branching on the relatedness 115

6.3.3 Influence of time between the commits on the relatedness . . . 117
6.3.4 Influence of the branching on the relatedness . 118
6.3.5 Influence of the time between the commits and branching on the relatedness across projects 119
6.4 Threats to Validity . . . 120
6.5 Conclusion . . . 122

7 Interestingness of Coupled File Changes: A Case Study 123
7.1 Introduction . . . 123
7.2 Case Study Design . . . 124
7.2.1 Research Questions . . . 124
7.2.2 Case Selection . . . 126
7.2.3 Data Collection Procedure . . . 126
7.2.4 Ethical Considerations . . . 130
7.2.5 Analysis Procedure . . . 131
7.2.6 Validity Procedure . . . 134
7.3 Results and Discussion . . . 134
7.3.1 Case Description . . . 135
7.3.2 Number of Couplings (RQ 1) . . . 136
7.3.3 Interestingness of Coupled Changes (RQ 2) . . 137
7.3.4 Influence of Developer Experience on Interestingness (RQ 3) . . . 137
7.3.5 Influence of Developer Involvement in the Project on Interestingness (RQ 4) . . . 138
7.3.6 Interestingness of Additional Information (RQ 5) . . . 139
7.3.7 Influence of Developer Experience on Interestingness of Additional Information (RQ 6) . . . 140

7.3.8 Validation and Theory 142
7.3.9 Discussion 144
7.3.10 Evaluation of Validity 147
7.4 Conclusion 148

8 Usefulness of Coupled File Changes: A Controlled Experiment Study 151
8.1 Introduction 151
8.2 Experimental Design 152
8.2.1 Study Goal 153
8.2.2 Research Questions 153
8.2.3 Hypotheses 154
8.2.4 Experiment Variables 155
8.2.5 Experiment Design 157
8.2.6 Objects 157
8.2.7 Subjects 158
8.2.8 Materials, Procedure and Environment 158
8.2.9 Selection of Change Author 160
8.2.10 Selection of Coupled Files 160
8.2.11 Classification of Issues 162
8.2.12 Definition of Tasks 163
8.2.13 Tasks and Coupled File Changes 164
8.2.14 Solution of Tasks 165
8.2.15 Maintenance Activities 166
8.2.16 Data Collection Procedure 166
8.2.17 Data Analysis Procedure 170
8.2.18 Execution Procedure 173
8.3 Results and Discussion 175
8.3.1 Participants 175

8.3.2 Issues Classification 176
8.3.3 Usefulness of Coupled File Changes 177
8.3.4 Usefulness of software repository attributes . . 185
8.3.5 Threats to Validity 189
8.4 Conclusion 192

9 Coupled File Changes Influence on Help Seeking: An Exploratory Study 193
9.1 Introduction 193
9.2 Experimental Design 194
9.2.1 Study Goal 194
9.2.2 Research Questions 194
9.2.3 Overview 195
9.2.4 Data Analysis 199
9.3 Results and Discussion 200
9.3.1 Information Sources 200
9.3.2 Influence of Coupled Change Suggestions . . . 204
9.4 Threats to Validity 206
9.5 Conclusion 206

10 Mining System Packages for Developer Expertise: An Exploratory Study 209
10.1 Introduction 209
10.2 Case Study Design 211
10.2.1 Research Questions 211
10.2.2 Case Selection 211
10.2.3 Data Collection Procedure 212
10.2.4 Analysis Procedure 212

10.3 Results and Discussion 215
10.3.1 Most frequent package couplings per developer (RQ1) 215
10.3.2 Developer profiles (RQ2) 216
10.3.3 Discussion 217
10.4 Threats to Validity 217
10.5 Conclusion 218

11 Tool Support 219
11.1 Concept and Design 219
11.1.1 Components 220
11.2 User Interface 221
11.2.1 Activation 223
11.2.2 Wizard 223
11.2.3 Views 224
11.2.4 Usage 225

12 Conclusion 227
12.1 Summary 227
12.2 Next Steps 229

Bibliography 231

List of Figures 253

List of Tables 255

ZUSAMMENFASSUNG

Software-Repositories enthalten Informationen über den Entwicklungsverlauf eines Softwaresystems, die von den Entwicklern bei Wartungsarbeiten genutzt werden können. Dazu gehören Daten im Versionsverwaltungssystem, im Issue-Tracking-System und in den Dokumentationsarchiven. Eine der meistverwendeten Techniken zur Analyse von Software-Repositories ist Data Mining, wobei die Frequent-Itemsets-Analyse häufig verwendet wird, um Gruppen von Dateien zu definieren, die in der Vergangenheit häufig zusammen geändert wurden. Diese Dateigruppen definieren wir als gekoppelte Dateien oder Coupled Files.

Die meisten Studien über gekoppelte Dateiänderungen in Software-Repositories beziehen Git-Versioning-Systeme nicht mit ein. Auch wird darin das Feedback der Entwickler bezüglich der Nützlichkeit der gekoppelten Dateiänderungen und deren Einfluss auf die Wartungsaufgaben nicht berücksichtigt.

Das Hauptziel der vorliegenden Untersuchung besteht darin, Ent-

wickler bei ihren Wartungsaufgaben zu unterstützen, und zwar in Form von Vorschlägen für mögliche Dateiänderungen, die auf früheren Änderungen im Git-Versionsverlauf der Software basieren. Wir untersuchten die Extraktion gekoppelter Dateiänderungen durch Data Mining mit Git und analysierten die Rückmeldung von Entwicklern bezüglich der Interessantheit und Nützlichkeit der Vorschläge für gekoppelte Dateiänderungen sowie deren Einfluss auf die Wartungsarbeiten.

Anhand einer Fallstudie in der Industrie extrahierten wir gekoppelte Dateiänderungen aus drei Git-Repositories. Basierend auf der in dieser Fallstudie untersuchten Interessantheit und den Erkenntnissen aus einer Reihe empirischer Studien zu Wartungsarbeiten stellten wir eine Theorie auf über die Verwendung von Vorschlägen für gekoppelte Dateiänderungen bei Wartungsarbeiten. Diese Theorie wurde in folgenden Studien getestet:

(1) Wir führten ein kontrolliertes Experiment durch, in dem wir Heuristiken zum Gruppieren verwandter Änderungssätze in Git untersuchten, aus denen wir relevante gekoppelte Dateiänderungen extrahierten. (2) In einem Quasi-Experiment untersuchten wir die Nützlichkeit gekoppelter Dateiänderungsvorschläge und deren Auswirkung auf die Korrektheit der Lösung sowie den Zeitaufwand für die Bearbeitung der Wartungsaufgaben. (3) In einer Explorationsstudie untersuchten wir, wie gekoppelte Dateiänderungsvorschläge die Strategie der Entwickler beeinflussen, mit der sie Wartungsarbeiten Hilfe suchen.

In einer Explorationsstudie erweiterten wir das Konzept gekoppelter Dateiänderungen auf Paketebene und ermittelten verschiedene Stufen von Entwickler-Kompetenz anhand der von den Entwicklern bearbeiteten Systempakete.

Wir entwickelten ein Werkzeug auf Eclipse-Basis, das gekoppelte Dateiänderungsvorschläge extrahiert, visualisiert und Entwicklern bei Wartungsaufgaben zur Verfügung stellt.

Wir haben Heuristiken definiert um verwandte Änderungen in Git zu gruppieren.

Mit der Frequent-Itemsets-Analyse gelang uns die Extraktion relativer häufiger gekoppelter Dateiänderungen aus Git.

Die an der Fallstudie zur Interessantheit gekoppelter Dateien beteiligten Entwickler zeigten sich interessiert an dieser Art von Hilfe während der Wartungsarbeiten.

Das Experiment zur Nützlichkeit von gekoppelten Dateiänderungsvorschlägen ergab, dass die Entwickler, die die Vorschläge nutzten, ihre Aufgaben erfolgreicher bewältigten als diejenigen, die es nicht taten.

Die Ergebnisse der Explorationsstudie zur Inanspruchnahme von Hilfe bei Wartungsaufgaben zeigen, dass gekoppelte Dateiänderungsvorschläge auch den Bedarf an für die Wartungsaufgaben relevanten externen Informationsquellen reduzieren und so den Wartungsprozess kompakter machen. Zudem wurde das Konzept der gekoppelten Dateiänderungen erfolgreich eingesetzt, um Kompetenzprofile mit unterschiedlichen Spezialisierungen zu erstellen, die auf den Änderungen in den gekoppelten Systempaketen basieren. Die Rückmeldungen der Entwickler zu dem Verfahren der gekoppelten Dateiänderungsvorschläge wurden als positiv identifiziert.

Unsere Theorie über den Einsatz gekoppelter Dateiänderungsvorschläge bei Wartungsaufgaben wurde erfolgreich getestet. Mit den vorgeschlagenen Heuristiken ermittelten wir, dass die Gruppierung der Änderungssätze in Git ihre Relevanz beeinflusst. Die Rückmeldungen der Entwickler zeigten, dass das Format und der Kontext

gekoppelter Dateiänderungsvorschläge sich auf deren Nützlichkeit auswirken. Die Ergebnisse zeigen auch, dass gekoppelte Dateiänderungsvorschläge die Bearbeitung der Wartungsaufgaben und die Strategien zum Suchen nach Hilfe positiv beeinflussen.

Die weitere Analyse von Kopplungen zwischen Teilen des Quellcodes anhand großer Datensätze ermöglicht es, die Auswirkungen gekoppelter Dateiänderungen auf die Wartung und die Qualität von Software besser zu verstehen.

Abstract

Software repositories contain information about the history of a software system which can be used by developers during maintenance tasks. This includes the data in the versioning system, the issue tracking system and the documentation archives. One of the most used techniques to analyze software repositories is data mining whereby frequent itemsets analysis, has often been used to define sets of files which changed frequently together in the past called coupled files.

Most of the studies on coupled changes from software repositories do not involve Git versioning history. Further, developers' feedback on the usefulness of coupled file changes and their influence on the maintenance tasks solution has been ignored.

The overall goal of this research is to help developers in their maintenance tasks by suggesting potential file changes based on previous modifications in the Git version history of a software product.

We investigated the process of extracting coupled file changes from Git using data mining and analyzed the feedback of developers on the

interestingness and the usefulness of coupled file change suggestions as well as their influence on maintenance tasks.

Using an industrial case study, we extracted coupled file changes from three Git repositories. Based on their interestingness investigated in this case study and the knowledge from a set of empirical studies related to maintenance activities, we built a theory on the use of coupled file change suggestions during maintenance tasks.

The theory we provided has been tested using following studies: (1) We performed a controlled experiment where we investigated heuristics for grouping related change sets in Git from which we extract relevant coupled file changes. (2) We used a quasi-experiment to explore the usefulness of coupled file changes suggestions and their impact on the correctness of solution and the time needed to solve maintenance tasks. (3) We performed an exploratory study on the impact of coupled file change suggestions on developers' strategy for searching help during maintenance tasks.

Performing an exploratory study, we extended the concept of coupled file changes on a package level and aggregated various levels of developer expertise based on the functionalities of the system packages they have changed.

We have developed an Eclipse based tool to implement the extraction and visualization of coupled file change suggestions.

We identified the heuristics for grouping change sets based on the developer and the time of commit influence their relatedness at most.

Implementing the frequent itemsets analysis, we successfully extracted relative frequent coupled file changes from Git.

The developers participating in the case study on the interestingness of coupled files demonstrated to be interested for this kind of help during maintenance tasks.

From the experiment on the usefulness of coupled file change suggestions results, we recognize that using coupled file changes, the developers managed to solve their tasks more successfully than those not using these suggestions.

The outcomes of the exploratory study on the search for help during maintenance tasks, show that coupled file change suggestions reduced the need of external task relevant information sources which makes the solving process compacter.

Additionally, in the exploratory study on the coupling of system packages, we successfully used the concept of coupled file changes to provide expertise profiles with different level of specialization based on the changes in the coupled packages.

The feedback on the coupled file change suggestions approach by the developers has been identified as positive.

We have successfully tested our theory on the use of coupled file change suggestions during maintenance tasks. Using the proposed heuristics we determined that the grouping of the change sets in Git influences their relatedness. The developers' feedback revealed that the format and the context of coupled files influences their usefulness. The results also show that coupled file change suggestions positively influence maintenance tasks solution and the strategy for searching help.

Further analysis of couplings between parts of the source code based on large data sets, can be used to better understand the impact of coupled file changes on the maintenance and the quality of software.

Chapter 1

Introduction

1.1 Motivation

Software development produces large amounts of data which is stored in software repositories. Over time, this data becomes an useful source of information about the software product. The rapidly increasing amount of data in the repositories makes it difficult to maintain or to extract useful information from it. This data is organized in *Software Repositories*. This term has been defined as the data which has been managed or generated by tools during software development [Moc14].

Software repositories include various data sources like the versioning system, the issue tracking system and the documentation archives of a software system.

Version control systems such as CVS[1], Subversion[2], Git[3] or Mercurial[4] store information concerning the changed files like the time of the change, the author of the changes and the file content before the change happened. In versioning control systems like Git, the source code changes are organized in a distributed fashion using commits which represent the so called atomic change sets. The structuring of changes in Git using commits as atomic change sets, the branching usage, the specific time concept as well as the possibility to map the commits with the issues, motivates us to investigate the process of obtaining of coupled file changes from Git.

Software issues like features or bugs can be tracked using issue tracking systems like JIRA[5], Redmine[6] or Bugzilla[7]. They include information about current and previous issues and maintenance tasks.

The documentation archives of software projects contain information related to the software development process, the design and the organization of the system.

Software repositories contain hidden dependencies between source code parts like classes, files or modules which are not structurally related. These dependencies found in the version history of a software product have been defined as logical dependencies [GHJ98] or evolutionary couplings [BKPS97]. During time, various parts of the source code are changed together several times across the system [GHJ98] creating patterns of hidden dependencies which are not easy to find.

[1]Free Software Foundation, http://www.nongnu.org/cvs/
[2]The apache Software Foundation http://subversion.apache.org/
[3]Git-fast-version-control https://Git-scm.com/
[4]Mercurial source control managementhttps://www.mercurial-scm.org/
[5]JIRA Software, https://www.atlassian.com/software/jira
[6]Redmine Application, http://http://www.redmine.org/
[7]Bugzilla Software, https://www.bugzilla.org/

In a situation where the developer changes a part of the source code and also changes another part shortly after, we say that we have *Coupled Changes*. The notation of change couplings was introduced in [FGP05] and later in [DGL08]. If the couplings are based on a file level, we talk about *Coupled File Changes*.

Coupled file changes suggest source code modifications the developers may need to perform in other files of the system. These modifications can contribute to solve their maintenance tasks more successfully [Par].

The authors of previous source code changes may not be available anymore. New or inexperienced developers working on maintenance tasks on the system cannot be immediately productive. Their programming experience and their knowledge of a specific system application impact the productivity [Bec09; Cha08]. Experienced developers either do not have the time or the conditions do not provide an opportunity to coach them [SH98].

Working on unfamiliar parts of the system requires an additional knowledge which can lead to a situation where developers cannot solve all tasks and need more effort to understand them [SLVA97].

Having suggestions about other previously changed files for similar issues, can help developers to solve their maintenance tasks more successfully. It would be interesting to examine the developers' feedback on the acceptance and the usefulness of coupled file change suggestions as well as their influence on maintenance tasks.

Coupled changes are not easy to recognize among the data in the software repositories. To learn from the data and find useful information in the version history, we need a technique to extract coupled changes from it. One of the most popular techniques to extract useful information from software repositories related to the changes

in the software is Data Mining. It has been defined as a process of discovering interesting patterns in large data amounts [HMP05]. Having large amounts of data in software repositories from which we can uncover useful information about the software system makes data mining suitable for extracting coupled file changes.

The term Mining Software Repositories (MSR) has been defined to describe investigations of software repositories using data mining [KCM07].

1.2 Problem Statement

Several researchers have proposed approaches to identify logical couplings from a software versioning history [BDO+13; KYM06; YMNC04; ZWDZ04].

Current studies on mining software repositories include version systems like CVS or Subversion. Git, as a very popular versioning system has not been investigated in this context.

Existing studies focus on expert findings and ignore the feedback of developers related to the acceptance and the usefulness of coupled changes during maintenance tasks. Further, the influence of coupled file change suggestions on the maintenance tasks solution and the strategy of solving these tasks has not been directly investigated using the developers' feedback.

1.3 Research Objective

The overall aim of this research is to extract coupled file changes extracted from Git which can help developers in the solving their maintenance tasks.

First of all we explore the organization of source code changes in Git, the relatedness and the grouping of changes from we can extract relevant coupled file changes.

Further, we investigate the process of extracting files being frequently changed together in the past from Git as well as the corresponding repository attributes from the version history, the issue tracking system and the project documentation of a software system.

To be able to extract coupled files from the commits in the version history using data mining, we need a technique and an algorithm suitable to work with frequently occurring events like file changes.

We continue by investigating the feedback of the developers on coupled file change suggestions. Using it, we crave to determine the acceptance of the concept of coupled file changes as an additional help during maintenance tasks. Further, we aim to determine the usefulness of coupled files as well as their impact on the activities of the developers during maintenance tasks.

1.4 Contribution

The main contributions of this research include the proposed methodology for mining coupled files from Git using frequent itemset analysis and building coupled file change suggestions from Git version histories, as well as the definition and the successful testing of the theory on the use of coupled file changes by exposing the factors that influence the acceptance of coupled file change suggestions and their influence on maintenance tasks.

We explored Git relevant issues using a quasi-experiment on the relatedness of change sets in Git [RW16d]. We use them as a source for extracting relevant coupled file changes.

We provided coupled file change suggestions using frequent itemset analysis based on the information gathered from the distributed version histories of three software projects as presented in our industrial case study [RW16b]. Here, we also explored developers' feedback on the interestingness of coupled file change suggestions.

Based on the constructs defined in the case study [RW16b] and the knowledge from a set of empirical studies related to maintenance tasks and activities, we built a theory on the use of coupled file change suggestions during maintenance tasks. We tested the theory using the following empirical studies:

(1) In [RW16d], performing a quasi-experiment, we investigated heuristics for grouping related change sets from the version history concerning important Git characteristics like the existence of commits, their timing and branching status. We exposed that besides the authorship, the commit time has a significant influence on the relatedness.

(2) In [RW16a], we used a controlled experiment to explore the usefulness of coupled change suggestions based on the participants' solution of maintenance tasks. Their feedback determines that the coupled file changes and a subset of proposed attributes are useful. The analysis of their actions addresses a significant influence of the use of coupled file change suggestions on the number of correctly solved tasks, whereby the suggestions do not significantly reduce the time needed to perform the tasks.

(3) In [RW17a], using an exploratory study, we investigated the influence of coupled file change suggestions on the strategy of solving maintenance tasks. We found that the use of coupled change suggestions influences the participants' choice of information sources to searching for help during their task solution. The choice of help

locations of the participants using coupled file change suggestions is more compact and limited to the use of internal information sources related to the IDE, they seldom use external information sources.

We extended the concept of coupled file changes on a package level using an exploratory case study [RW17b]. Here the system packages have been used to extract couplings. Based on these couplings of packages frequently changed together, we aggregated developer profiles of expertise. We have managed to identify profiles including various levels of functionalities behind the source modifications.

We also implemented the concept of coupled file change suggestions using an Eclipse based plug-in tool which includes the process of extracting commits from Git, their storage in a database, the performing of data mining to extract coupled files and the visualizing of coupled files and repository attributes. The tool has been designed and conceptualized using the methodology presented in this work and was developed as part of 5 student theses [Ala16; Cic15; Dem15; Kau17; Leh15].

1.5 Thesis Outline

Chapter 1 defines the motivation, the problem statement, the research objective and the contribution of this work.
The remaining chapters are organized as follows:
Chapter 2 describes the motivation to investigate coupled file changes as well as the background of the methodology and the techniques used to investigate coupled file change suggestions.
Chapter 3 includes the related work of this research where we cover the current state of the art in the field of the analysis of version histories and software maintenance aspects.

Chapter 4 is in account of the methodology for extracting coupled file changes and the corresponding repository attributes from Git as well as the building of coupled file change suggestions.
Chapter 5 consists of the proposed theory on the use of coupled file change suggestions covering the factors that influence the acceptance of coupled file change suggestions as well as their influence on maintenance tasks.
Chapter 6 is based on the grouping related changes in Git from which we can extract coupled file changes.
Chapter 7 concentrates on the feedback of the developers on the interestingness of coupled file change suggestions during maintenance tasks.
Chapter 8 involves the developers' feedback on the usefulness and the influence of coupled file change suggestions on maintenance tasks.
Chapter 9 covers the influence of coupled file change suggestions on the strategy for searching for help during maintenance tasks.
Chapter 10 describes an extended approach of the concept of coupled file changes using system packages to define developer expertise profiles.
Chapter 11 covers the description of an Eclipse based tool for implementation and visualization of coupled file change suggestions.
Chapter 12 encloses the conclusions of this work as well as the potential future research steps.

1.5.1 Previously Published Material

The material covered in this work is based, in part on the following publications where i am the first author:

- *Are Suggestions of Coupled File Changes Interesting?*: A case

study on the interestingness of coupled files, published in the Proceedings of the 2016 International Conference on Evaluation of Novel Software Approaches to Software Engineering (ENASE) [RW16b].

- *How Interesting Are Suggestions of Coupled File Changes for Software Developers?*: An extended version of [RW16b], published in the 2016 Springer Communications in Computer and Information Science series (CCIS) [RW16c].
- *Which Change Sets in Git Repositories Are Related?*: A quasi-experiment on the relatedness of change sets in Git repository, published in the Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS) [RW16d].
- *Are Coupled File Changes Suggestions Useful?*: A controlled experiment on the usefulness of coupled file changes during maintenance tasks, published in: PeerJ Computer Science Journal Preprint [RW16a].
- *How Do Coupled File Changes Influence How Developers Seek Help During Maintenance Tasks?*: An exploratory study on the influence of coupled file changes on the strategy of developers seeking for help during maintenance tasks, published in the Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS) [RW17a].
- *Mining Java Packages for Developer Profiles*: An exploratory study on extracting system package couplings for defining developer expertise profiles, published in the Proceedings of the Workshop on Industrial Applications of Artificial Intelligence (IAAI) at the

2017 Conference on Database Systems for Business, Technology and Web (BTW) [RW17b].

Background

2.1 Software Repositories

2.1.1 Version Control Systems

Version control systems enclose information concerning the files been changed, the time when the change happened, the author of the change and the files content before the change [HLT09]. Two main types of version control systems are used today: centralized version control systems (CVCS) like CVS or Subversion and distributed version control systems (DVCS) like Git or Mercurial. In CVCS, we have a central repository with the version database where the developers check out their projects on their local computers. DVCS have gained an increased popularity lately. In DVCS, each team member has the complete repository on his or her local machine called local repository. Git provides a copy of all files and a copy of the repository

we work with [Loe09]. If one file changes, the complete repository changes [Ott]. The developers can clone remote repository or pull data into their local repositories, commit the changes in it and then push the changes on the remote repository again.

Git organizes the changed files in commits which represent so called *atomic changes*. The commits are identified by their commit ID hashes. An important attribute of Git is the way it handles with the time of changes in the source code. Since Git does not track the timestamps when the files were originally modified, it does not have a central time concept, it tracks only the time of commit. Many version control systems have a branching functionality. Branches are very often used in Git. They allow the developers to create many lines of development. Branching is used to separate commit changes from each other. Some projects use branches for development, testing or release [RKTC16]. Other maintain branches for bug fixes or adding new features [WS08].

Versioning control systems maintain the source code history of changes but also provide additional attributes [RD04]. Git provides an extensive formatting support for the output of the log information. Among various meta-data it provides information like *commit ID*, *commit message*, *commit author*, *commit date* as well as the *paths* of the files in the commits.

2.1.2 Issue Tracking System

Software change issues like features or bugs can be tracked using issue tracking systems [FPG03b]. Issues contain specific information about the problems we need to solve like the *issue ID*, *issue description*, *issue author*, *issue type*, *issue status* and other attributes. Usually, the

issue tracking systems allow us to export the issues and their attributes for further analysis.

2.1.3 Software Documentation

During software development, a large amount of documentation is generated. The documentation can be used by developers as an information source during maintenance [Som02]. In the context of investigating coupled files, we concentrate on the product documentation which describes the developed software product. The software documentation gathered during the development process represents a rich source of information. The documentation can contain information about the system architecture, a description of the functionalities behind the classes, packages, features or application layers. It can also represent a manual for using or administrating the system.

The content and the quality of the software documentation can influence the use of coupled file changes by providing additional information related to the changes in the version history.

2.2 Coupled File Changes

Structural dependencies between software elements can be expressed using different abstraction levels. They can exist on architectural, design or implementation level [EKKM08]. However, hidden dependencies between files or other artifacts that changed together frequently also exist. For the first time they were introduced as evolutionary couplings [BKPS97] or logical dependencies [GHJ98].

Developers change various source files with different frequency during software development. The notation of change couplings was

introduced in [FGP05] and later in [DGL08]. *Coupled file changes* describe a situation where the developer changes a particular file and also changes another file afterwards. Extracting sets of files being changed together by analyzing the version history of the software can detect logical dependencies between these files. Developers can use them during maintenance tasks.

2.2.1 Atomic Change Sets

In versioning systems such as Subversion or Git, commits represent so called "atomic change sets". They include the sets of files which can be used as sources to extract coupled file changes. Older versioning systems like CVS, do not maintain this kind of change sets. The information, which software artifacts were checked in together is not available. Therefore, researchers investigate the change history using the technique of fixed or sliding time windows [Ger04; GHJ98; ZWDZ04].

2.2.2 Related Changes

To be able to extract relevant coupled file changes, firstly, we need to group the committed change sets. The same set of coupled files can be found in different commits performed on different occasions by various developers. Although we deal with the same files, the commit messages can describe different changes about unrelated functionalities which are not relevant to group them (Table 2.1). Here the commit messages describe totally different issues. This means that also unrelated sets of files happen to be grouped together as coupled file change suggestions. To avoid this, we need to identify

the related change sets out of which we can extract the coupled file changes.

So what makes file changes related? The developers want relevant suggestions. We assume that the coupled file change sets are related if they are addressing the same issue either solving the issue in many steps or repeating the change in several occasions.

The change set can have various granularity and composition across different tasks [KYM06].

Table 2.1: Unrelated changes

Commit 1	Commit 2
synchronizing layouts done refs #827	added highlighter to the connection anchors refs #347

Table 2.2: Related changes

Commit 1	Commit 2
began to adapt controller structure refs #503	Adapt controller structure refs #503
added icons to export wizard refs #868	added new icons and new png filter to export wizard refs #868

One task can be solved in several steps using more than one commit. In case a task cannot be solved in a short time, the work on it can be interrupted and continued at another time. The first example in Table 2.2 shows that the task is addressed in two steps, the developer started dealing with the controller structure in one occasion and

finished the changes later in another one.

The same change can be committed many times in different occasions. The second example in Table 2.2, is related to the change set commented as "added icons to export wizard" and the change set commented as "added new icons and new png filter to export wizard". They represent file changes on the same functionality repeated in different commits.

2.2.3 Heuristics

There are different heuristics for grouping file changes from versioning systems that support atomic change sets [KMS07; KYM06]:

- *Time*: It groups all changes committed in a given time period, usually a day. The changes in the software performed during this time period by one or many different authors have been considered to be related. All changes from the same author or different authors that have been performed before or after the defined time interval are rated as unrelated. The number of different time intervals determines the number of change set groups and by that influences the extraction of coupled file changes. Grouping change sets from different developers performed during the defined time period can result into grouping of changes that are not relevant and cover totally different functionalities or issues.
- *Developer*: This heuristic groups the changes committed by a single developer as related. This excludes the changes performed by other developers from the change set group regardless of the time of change. We have so many groups of related change sets as we have developers on the project. This type of heuristic

allows us to extract a relatively high number of frequent coupled file changes even for projects with a low number of developers.

- *Time and Developer*: This heuristics is a combination of the previous two heuristics. It rates that the changes committed by the same developer during the defined time period are related. All changes from the same developer during various time periods or in the same period but performed by different developers are not related. The number of change set groups depends on the number of developers and the time of commit. This heuristic is very restrictive and is not convenient for projects having a low number of developers or where they do not commit their changes frequently. Due to this disadvantage, the number of change sets can be insufficient to extract coupled file changes from the repository.

2.2.4 Mapping Between Commits and Issues

The comments stored as commit messages in Git, contain a description about the committed changes in the version system and describe the change purpose [MHS+12]. However, commit messages do not always deliver understandable textual content. The analysis of these messages even with the help of natural language processing is not always useful.

Every commit has a hash value which represents the commit ID. It is an unique value which identifies all the commits in the database. The issues are also identified by their keys or IDs. The use of merge points to map the commit messages and the issues from the issue tracking system is an useful practice today. Here, the commit messages contain the issue IDs which identify a particular task, a feature or a bug. We

use the issue IDs to follow down the related commits. We distinguish three cases of mapping commits and issues:

- *1-1*: The first type is a mapping between one commit and one issue (Figure 2.1). The commit and the issue included in this set are not part of any other mappings.
- *1-n*: The second type maps one commit to several issues which is a rare case where the changes in the commit are used to solve a group of similar issues (Figure 2.2).
- *n-1*: The third one is mapping many commits to one issue (Figure 2.3). This is a frequently used mapping where an issue has been solved in several commits.

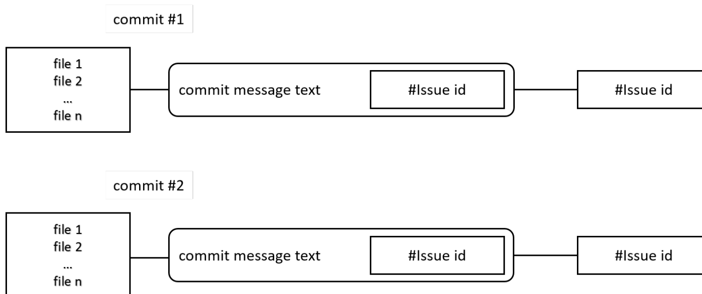

Figure 2.1: 1 commit to 1 issue

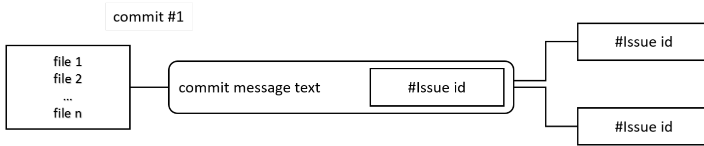

Figure 2.2: 1 commit to n issues

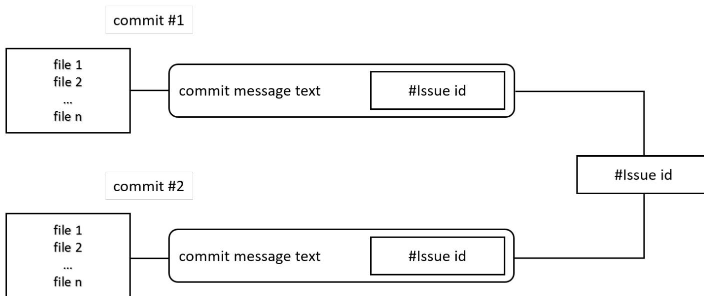

Figure 2.3: n commits to 1 issue

2.3 System Packages

Developers usually organize their source code by grouping common characteristics or functionalities. Files changing frequently together are contained in system packages. We can track the changes in the software by exploring the couplings between packages which can be useful to discover areas of specific functionalities or expertise.

The decomposition of software into modules represents a benefit in the software development [Par]. Packages have been considered as program modules [Hau02]. They are also used to prevent conflicts in the source code.

In Java, packages represent namespaces to organize classes and

interfaces as folders. The name of a package corresponds to a directory structure where the classes are stored. For example, the package *com.astpa.ui* is stored in the directory *com/astpa/ui* in the form of a relative path. Hierarchical package names can be also constructed like: *com.astpa.ui* and *com.astpa.ui.viewer*. They are treated as independent packages.

There are two fundamental approaches for creating packages in Java: packaging by feature and packaging by layer. Package by feature reflects a set of features of the software [Sys13]. It organizes all the classes related to a single feature into a single package or directory. For example the following packages:
com.example.gridview.ui and *com.example.menubar.ui* identify two features related to the user interface. Each package contains only items which are related to a specific feature but not related to other features. Package by layer reflects various application levels instead of features. For example, the following packages:
com.example.controller.actions and *com.example.dataacess.actions* represent two different application layers.

2.4 Data Mining

Data Mining is one of the most popular techniques for investigating software repositories. The term "Data Mining" has been used either as synonym for knowledge discovery or to describe one of the steps of this process. However, the broader definition of data mining is that it is a process to discover interesting knowledge from large amounts of data like databases or repositories [HMP05].

2.4.1 Mining Frequent Itemsets

The discovery of frequent item sets is a common data mining technique. In the context of this work, it is often used to extract logical couplings from the versioning history. It was originally presented to analyze transaction data of customer buying behavior in supermarkets [AIS93]. Here, frequent sets are containing products which have often been bought together [HMP05].
We use the procedure suggested in [Goe10] to formally describe frequent itemsets:
Let J represents a set of items.

The couple $T = \{t_i, I\}$, over J is a transaction identified by t_i and I is set of items from J.
We define D to be a database of transactions over J.
T supports a set X whereby, $X \subseteq I$ which contains a subset of items.
The cover of the set X in D consists of the identifiers of the transactions in D supporting X.
The number of transactions in the cover of X in D is the support of set X in D.
The frequency of X in D is the probability that X can be found in a transaction which is determined by dividing by the total number of transactions in D.
We denote the item sets to be frequent if they have a support threshold *minsup* greater than a minimum specified by the user:

$$0 \leq minsup_{abs} \leq |D| \tag{2.1}$$

If using frequencies of sets we have a relative frequency threshold:

$$0 \leq minsup_{rel} \leq 1 \tag{2.2}$$

Having a database D of transactions over a set of items J and *minsup* as minimal support threshold, we denote the collection of frequent sets as F in D with respect to *minsup* as [HMP05]:

$$F(D, minsup) := \{X \subseteq J | support(X, D) \geq minsup\} \tag{2.3}$$

Let us have the following transactions as commits containing files changed by developers as presented in Table 2.3:

Table 2.3: File Change Transactions

Transaction	Itemset
t_1	f_1,f_2,f_3
t_2	f_1,f_3,f_5
t_3	f_1,f_3,f_8
t_4	f_2,f_3,f_6

From these transactions, we see that the items f_1 and f_3 have been covered in three transactions: t_1, t_2 and t_3 which means a support level of 3 and frequency of 75%. The items f_2 and f_3 have been covered in two transactions: $t_1$and t_4, with a support level of 2 and frequency of 50%. Having an user defined minimum support threshold of 3, we conclude that the f_1 and f_3 represent a frequent itemset.

2.4.2 FP-Growth Algorithm

Various algorithms for mining frequent itemsets and association rules have been proposed in literature [AS94; GG04; HMP05]. A classical example is the Apriori method, which has been used in various studies [KYM06; ZWDZ04]. However, Apriori generates many itemset candidates and scans the database many times for longer itemsets candidates [HMP05]. Their generation and the need for multiple

database scans is considered expensive [TSK05].

As opposed to the Apriori, the FP-Growth algorithm allows frequent items discovery without candidate item set generation. This algorithm is considered to be faster and more memory efficient than the Apriori algorithm [HPYM04].

FP-Growth scans the database only twice. This methods uses partition and divide-and-conquer methods [HPYM04]. Firstly, it compresses the frequent itemsets database into a data structure called frequent pattern tree(FP-tree) and adds the relations between the elements. Afterwards, the database is divided into conditional databases related to the frequent items, whereby each of the conditional databases is mined one by one to create the final set of frequent items.

The FP-Tree construction is defined as follows [ZW14]:

1. Scan the database D and gathering the set of frequent items F.
2. Create the root of the FP-Tree *("root")*.
3. Scan D for the second time to create the transaction branches.
4. Generate the conditional pattern base and the conditional FP-Tree for each frequent item.

The FP-Tree construction algorithm description is as follows [HPYM04]:

Input: The database D in the form of the FP-tree and the minimum threshold minsup.
Output: The set of frequent patterns.
Method: Call FP-growth (Fp-tree, null)
Procedure: FP-growth (Tree, frequent itemset(α))
{

1. **if** *Tree* contains a single path *P*
2. **then for each** combination (β) of the nodes in the path *P* do
3. generate pattern $\beta \cup \alpha$ with *support= minsup of nodes in* (β);
4. **else for each** item in the path a_i in the header of *Tree* do {
5. generate pattern $\beta = a_i \cup \alpha$ with *support*=a_i*.support;*
6. construct the conditional pattern base of β and then the conditional FP-tree of β ($Tree_{\beta_i}$);
7. **if** *Tree* $\neq 0$
8. **then** call **FP-growth** **(**$Tree_{\beta_i}$**)** }

}

Let us for example create the FP-tree for the files in Table 2.4:

Table 2.4: Transaction Items

t_i	Items
t_1	f_1, f_2
t_2	f_2, f_3, f_4
t_3	f_1, f_4
t_4	f_1, f_2, f_3, f_5
t_5	f_1, f_3, f_4
t_6	f_1, f_2, f_3
t_7	f_1, f_2, f_3
t_8	f_1, f_2, f_4
t_9	f_1, f_2, f_5

- We scan the database to find the frequent items and their support. For our example we found the following set of items: $L = \{\{f_1 : 8\}, \{f_2 : 7\}, \{f_3 : 5\}, \{f_4 : 5\}, \{f_5 : 2\}\}$

- We create the "root" of the FP-tree
- For each transaction in Table 2.4, we sort the items by descending order related to their support and create a branch of the tree. Lets take the first transaction t_1. It contains the items: f_1 and f_2, which construct the first branch of the tree as presented in Figure 2.4.

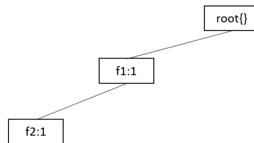

Figure 2.4: FP-Tree after t_1

- After the second transaction which contains the items f_2, f_3 and f_4, we have the tree branch as presented in Figure 2.5. We see that the node f_2 is found in both branches so we link them using a dotted line.

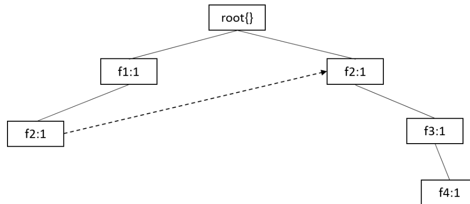

Figure 2.5: FP-Tree after t_2

- After the third transaction containing the items f_1 and f_4 we have the tree branch as in Figure 2.6. This branch contains the node f_1 which is also included in the first branch so we increase the count for this node to be 2.

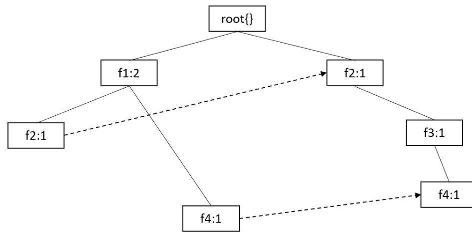

Figure 2.6: FP-Tree after t_3

The complete tree for the transactions in Table 2.4 is presented in Figure 2.7. It contains the final tree branches, the frequency of occurrence of each node and the linking between the nodes.

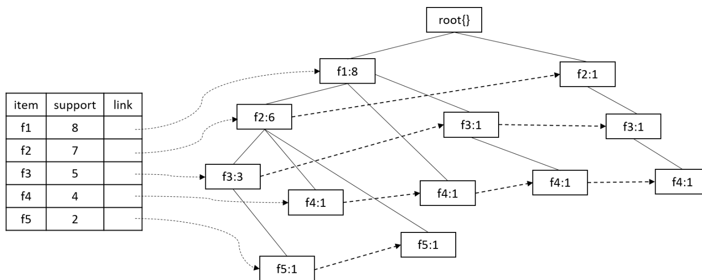

Figure 2.7: FP-Tree

We start mining the tree for each frequent pattern as starting suffix pattern, afterwards, we construct the conditional pattern base and the sub-database which contains the prefix paths which are found in the tree and occur together with the suffix pattern. Further, we construct the conditional tree and then recursively mine this tree and we concatenate the suffix patterns

and the frequent patterns from the conditional tree to perform the pattern growth[HMP05].

For example, let us take the last suffix f_5 and set a minimum support of 2. From the set of frequent items we notice that the item f_5 is included in two branches of the FP-tree. The first one consists of the elements (f_1, f_2, f_3, f_5) and the second of the elements: (f_1, f_2, f_5).
The prefixes for the f_5 suffix are: $(f_1, f_2, f_3 : 1)$ and $(f_1, f_2 : 1)$.
The conditional FP-tree has only one path $(f_1, f_2 : 1)$. We exclude the item f_3 for the reason that it has been found only once which is less than the minimum support we set to be 2.
The single path combines the following frequent patterns for f_5: $(f_1, f_5 : 2)$, $(f_2, f_5 : 2)$ and $(f_1, f_2, f_5 : 2)$. For all the suffixes we have the frequent patterns presented in (Table 2.5).

Table 2.5: Frequent Patterns

item	pattern : support
f_5	(f_1, f_5) :2, (f_2, f_5) :2, (f_1, f_2, f_5) :2
f_4	(f_1, f_4) :4, (f_1, f_2, f_4) :2, (f_2, f_4):3,
	(f_1, f_3, f_4) :2, (f_2, f_3, f_4) :2, (f_3, f_4):3
f_3	(f_1, f_3) :4, (f_1, f_2, f_3) :3, (f_2, f_4):3
f_2	(f_1,f_2) :6
f_1	(f_2,f_5) :6

FP-Growth has a higher performance then the Apriori algorithm, however the main drawbacks is that the performance depends on the compactness of the dataset [TSK05]. FP-Growth also has a large memory usage for the tree compared to the dataset size [GZ05].

2.4.3 Data Mining Framework

There are many frameworks and libraries to perform data mining on the data from transactional databases. We use an open source pattern mining framework called Sequential pattern mining framework (SPMF) [VGG+14]. It contains data mining libraries implemented in Java and includes a large set of data mining algorithms. It is specialized in pattern discovery in transactional databases like frequent itemsets, sequential patterns and association rules mining [VGG+14]. It is a well documented Java based and cross platform framework. We use the FP-Growth algorithm implementation in this framework as a basis for the extraction of coupled file changes.

2.5 Developers' Feedback

One of the main objectives of this research is to investigate the use of coupled file changes by exploring developers' feedback on the interestingness and the usefulness of coupled files. Information can be denoted as *interesting* from an objective and subjective point of view. The first one relies on the structure of the data, whereby the second depends on the user. The subjective interestingness of an information depends on that if it is novel or unexpected and if the developer can perform an action with it to his or her advantage [PM94]. *Useful* means that information can help to achieve a goal defined by the user or system [FPM92]. The *usefulness* is related to the user perception of the performance when solving a task [ANT92].

2.5.1 Types of Feedback

To gain the feedback of developers on the concept of coupled changes, we need a technique for a direct data collection. In this context, the primary research gives the possibility to directly collect the data from the source for qualitative and quantitative research. This data, called "primary data" has been defined to be unknown and directly obtained by the researcher [CPD05]. There are three common ways to conduct primary research: survey, interviews and observation [Dri10].

2.5.2 Surveys

The participants are asked about their opinion using a set of questions collected as questionnaires. Afterwards, the data can be compared based on their responses. Here, the Likert-approach is the most common used rating scale [CPD05]. The positive side of using surveys is that we can involve people which are not physically available, so the developers who worked on a project can be reached to gather their opinion. One of the greatest challenges of this method is that the participants may not understand the questions and therefore, may not answer them carefully.

2.5.3 Interviews

Here, the researcher performs a discussion based on set of questions using small groups of participants. They are asked questions to gain their feedback whereby their response is noted or recorded for further analysis. The strengths of the interview method is that interviews are useful to gather broader feedback from the participants by adjusting the questions according to the developer reaction. We can also use

interviews to validate previous responses by participants using surveys. However, the disadvantages of this method include the fact that interviews last longer than surveys and it is more difficult to analyze the answers.

2.5.4 Observation

The behavior of the participants involved in an experiment like their actions or task responses can be directly observed and measured. Usually the data is collected by noting or recording the participants' behavior or by directly taking part in the observed activities. One of the advantages of this method is that we do not rely of the answers of the participants, we extract the information by analyzing what they do. One of the challenges of this method is that it is more complex to filter the relevant information and interpret participants' feedback.

2.6 Software Maintenance

Our goal is to help developers in solving their maintenance tasks. Software maintenance represents an event driven process [KTM+99]. The basic causes for maintenance work have been defined as processing, performance or implementation failure in the software [Swa76]. It includes program or documentation changes to make the software system perform correctly or more efficiently [SCR98].

Software maintenance has been defined in the IEEE 1219 Standard for Software Maintenance [IEE98] to be a software product modification after delivery to remove faults, improve performance or adapt the environment. In the ISO/IEC 12207 Life Cycle Processes Standard [ISO95], the maintenance is described as a process where

the source code or the documentation modification is performed due to a problem or improvement.

2.6.1 Maintenance Categories

Three different categories of maintenance have been defined [Swa76]: *corrective, adaptive* and *perfective*. The ISO/IEC 14764 International Standard for Software Maintenance [ISO00] updates this list with a fourth category, the *preventive* maintenance, so we have the following maintenance categories [Pig96]:

- *Corrective Maintenance*: This type of maintenance includes corrections of errors in systems that should have never occurred [BG10]. It includes design, source code and implementation error corrections.
- *Adaptive Maintenance*: It satisfies the changes related to new requirements and includes the addition of new features to the system. Software product modifications are performed to ensure usability in changed environment.
- *Perfective Maintenance*: This involves changes in the system which influence its efficiency. Also it includes software product modifications to improve usability, maintainability or performance.
- *Preventive Maintenance*: Here, the changes periodically have been performed in the system to find problems and reduce the possibility of future failures. It includes software product modification to detect and remove failures before they become effective.

2.6.2 Maintenance Activities

The process of solving maintenance tasks includes the following activities [BBC+96; NBD11]:

- *Task understanding*: Before taking any actions, developers need to read the task description to understand the failure and prepare for the source code changes.
- *Change specification*: During this step, developers locate the source code they need to change, try to understand it and specify the code changes.
- *Change design*: In this step, the developers perform the already specified source code changes and debug the affected source code.
- *Change test*: To specify the success of the code changes, developers perform an unit test of the solution.

2.6.3 Search for Task Relevant Information Sources

Developers working on maintenance tasks need to locate the part of the code they have to edit during maintenance tasks. Facing a difficulty or needing additional information for a task solution, developers often seek for a relevant information on various locations [KMCA06]. There can be several types of relevance related to the source code. They can seek for source code to find files or code locations for further modifications [RGP11] or look for a code description to better understand a part of the code they need to edit or try to find code examples based on a similar situation or problem.

We have internal information sources like the IDE for example Eclipse, including its search function, the source code window or

the package explorer. Also there are other external sources like the software documentation, web search engines, source code examples or programming tutorials.

2.6.4 Developer Expertise Profiles

The information about the role of the developers in the development of a software project has been identified as fundamental [BNF14; Job16]. During time a software system can grows and can become difficult to identify which team member has the knowledge about specific part of the system [FOMM10]. Not every developer knows the software system in details [GKSD05].

During maintenance, the information about the authorship of software modifications can be useful. Developers working on a project can be differentiated based on their contribution to the functionalities of the software. There are developers that commit a significant number of commits to the system whereby others only contribute [ADG08]. Having this information which functionalities have been covered by a particular developer can help selecting the person whose data can be used during maintenance tasks.

Chapter 3

State of the Art

3.1 Logical Couplings

A lot of scientific work has been dedicated to the investigation of software repositories to find logical dependencies between software elements [BAY03; BKPS97; FGP05; GJK03]. For the first time, they were introduced as evolutionary couplings between classes in [BKPS97] and later as logical coupling between modules in [GHJ98].

3.1.1 Granularity

Logical couplings from version histories have been explored on various granularity levels. One group of studies use fine granular couplings where the coupled changes are identified between parts of files like classes or methods [FGP05; RPL08; ZWDZ04]. Other studies explore couplings based on a file level [KYM06; YMNC04]. Some studies

investigate couplings between features [FPG03a], modules or packages [BAY03; GHJ98].

In this work, the main approach of coupled changes relies on investigating couplings on a file level. Afterwards, we extend the scope of the couplings on a package level.

3.1.2 Data Sources

Considering the investigated type of version control systems, the majority of the studies use centralized version controlled systems (CVCS), typically CVS [CC05; ZWDZ04] or Subversion [KMS07; YMNC04] as a data source.

To our knowledge, there are few studies on mining Git repositories [BRB+09; Car13; GAH15; RW]. These studies concentrate on common challenges in mining Git and do not cover coupled changes.

Various studies combine more than one data source like version control systems and issue tracking systems [CC05; DLR09; FPG03b; WZKC11]. The data extracted from these sources is used to determine the link between the source code revisions and bugs or issues.

We combine the data from three different sources to build coupled file change suggestions: file changes and commit attributes from the Git versioning system, issue attributes from the issue tracking system and attributes from the documentation archives.

3.1.3 Commit Messages

Very few studies investigate characteristics of commit messages. Mainly, they concentrate on creating vocabulary terms [AKM08], the words that appear in the messages [DNRN13] and sentiment analysis [GAL]. We use a set of commit attributes as a basic set and extend it with the

issue attributes using the presence of a mapping between commits and issues.

3.1.4 Change Set Heuristics

Various heuristics for grouping related change sets have been proposed in the literature. Several heuristics based on the data source and pruning technique are introduced in [HH04]. Here, entity data, developer data and name similarity data are considered and pruned by their frequency and recency to group and reduce the change sets.

Kagdi et al. in [KYM06] suggests three main heuristics for grouping change sets. The time interval heuristic includes changes committed by one or different committer in a predefined time period. This heuristic organizes all the file changed during this time period in one group. It considers the changes committed in a specific time period as related. The committer heuristic groups the change sets from a single developer in one group. This heuristic identifies the changes committed by a specific developer as related. The combined time interval+committer heuristic, includes the changes performed by a single developer during a specific time period. It groups the changes committed by a specific developer with an additional limitation for the time of commit as related.

The proposed heuristics are evaluated classically using coverage, precision and recall measures without regarding the developers' feedback.

We use the *committer* as a fundamental heuristic to group the related changes to build the coupled file changes for the reason that it reported the highest coverage and precision values [KYM06]. We also investigate two relevant factors: the time of the commits and the

branching of the changes as important Git features for the relatedness of the change sets.

3.2 Mining Repositories

Analyzing the used methodology, most of the studies investigating software repositories use some kind of data mining for this purpose [Ger04; HSCS08; KYM06; RD04; SLM03; YMNC04; ZWDZ04]. We use data mining to extract frequent file couplings from Git version history for the reason that it has the capability to discover hidden patterns from data and to extract and transform them in to an useful information.

3.2.1 Techniques

Various data mining techniques have been used to investigate software repositories. Some studies include the classification which places the items into classes [SLM03]. Other use the clustering as mining technique [VT06]. Here, the item classes are not known in advantage.

Often, the frequent itemsets mining technique is used to identify changes in the version history that happened frequently together [KYM06; YMNC04; ZWDZ04]. Our investigation includes extraction of frequent sets of file changes from the software version history which conforms the use of the frequent itemsets mining technique.

3.2.2 Algorithms

The Apriori algorithm [KYM06; ZWDZ04] and the FP-Growth algorithm are among the most used algorithm to generate frequent itemsets from version histories [YMNC04]. We extract the coupled

file changes using the FP-Growth Tree algorithm which has proven to have a higher performance than the Apriori algorithm.

3.3 Developers' Feedback

3.3.1 Couplings and Feedback

To the best of our knowledge, there are very few studies investigating how couplings align with developers' opinions or feedback. A set of feature coupling metrics on the structural and the semantic level have been investigated in [RGP11]. The developers were asked if they find these metrics to be useful. They show that feature couplings on a higher level of abstraction than classes have been found to be useful.

Developers' perceptions of couplings on class level were investigated in [BDO+13]. Here the authors examine how class couplings captured by semantic or logical measures align with the developers perception of couplings. The semantic couplings have received the best rating of all types of couplings.

The feedback of developers on the clusters of co-changed classes from the version history has been investigated in [SVAA15]. They were classified regarding their projection to the package as: encapsulated, crosscutting and octopus reported as healthy designs, anomalies and normally associated class distributions.

3.3.2 Video Materials and Feedback

A number of studies involve the use of observation as a type of feedback on developer actions using captured videos. Developer actions in the investigation of help seeking during maintenance have been described in [KMCA06; LXPZ13]. Screen captured videos represent a method

to record the interaction of developers and the IDE [BLX+15]. The systematic approach of stages and concepts of selecting data from videos have been described in [RFJS07].

The characteristics and constraints in transcribing and coding videos have been investigated in [Jew12]. The application of visual grounded theory for multi-slice imaging has been presented in [Kon11]. In [Ste08], the authors identify the problems in the coding of video data using Grounded Theory for qualitative analysis. They recommend practices to support video data analysis like perspectives and concepts. We follow the suggestions and focus on the information sources before we transcribe the videos.

We focus on the developers' feedback on the interestingness and usefulness of coupled changes as well as on their impact on the task solution and the developers' strategy of seeking task relevant information. We gain their feedback using surveys and interviews and analyze it using the Grounded Theory method. We also use observation by analyzing the videos of their actions during maintenance tasks.

3.4 Assessment of Maintenance Tasks

Various studies investigate the assessment and the estimation of maintenance tasks. In [NBD11], the assessment and estimation of various types of software maintenance tasks has been investigated. In [RLR+12], they have assessed the correctness and the time duration of maintenance tasks in model driven development context. In [WA09], the researchers investigated the effect on the task order on the maintainability of object oriented software like the correctness and duration of maintenance tasks. In [SAPM14], evaluate the effects of reactive programming on program comprehension where besides

the programming skills they also explore the correctness and the time required to solve tasks. The influence on software visualization on the program comprehension based on the time and the correctness of task solutions has been investigated in [CZD11; WLR11].

We also explore the correctness of maintenance task solutions and the time of task completion for developers using coupled file change suggestions compared to the developers not using the suggestions during maintenance tasks.

The fault localization as technique for finding the location of defects given the program failures has been investigated in many occasions. In [KXLL16], the researchers explored the developers expectations on fault localization. They reported the importance of data availability, granularity, reliability efficiency as well the need of an IDE integration of fault localization techniques. The study on the use of a spectra-based fault localization techniques in [XBLL16], shows that the technique saves significantly saves debugging time. In [PO11], the authors empirically investigated the usability of a technique for spectrum based fault localization, with and without using this technique whereby the ranking and the search for defects has been identified as important.

We do not localize faults or bugs, we provide suggestions for potential file changes to solve an issue or a defined maintenance task.

The program comprehension including the understanding of the structure of a program and the functional dependencies has been investigated in [PA16]. They proposed a tool-set which improves the location of software components needed to be inspected. In [PA14], they introduced a spectrum based approach borrowed from fault localization technique. Based on test case executions, it identifies important components related to a given feature. In [LBB+10], they

investigated how developers navigate during debugging using modern programming environment proposing to include the information foraging theory when searching and fixing a bug. The use of the same theory has been proposed by in [FSP+13] to support developer activities for information seeking needed to understand software engineering tasks.

We do not involve testing or debugging, we provide the coupled files based on the changes stored in the version history.

3.5 Help Seeking During Maintenance Tasks

Several studies explored how developers seek for help during their development or maintenance tasks. Software engineering work practices of developers have been investigated in [SLVA97]. Some help seeking activities in software engineering both from static and dynamic perspective as well as relying on human factors have been identified in [LXPZ13].

The authors in [KMCA06] describe the activities performed by developers in searching for task relevant information, the perception of the information relevance relating as well as the task context variation. In [WPXZ11], the researcher investigate the feature location process during maintenance tasks by defining a three granularity levels of hierarchy: actions, phases and patterns. In [RP09], the effectiveness of ten feature location has been presented as a combination of textual, dynamic and static analysis.

In our research, we examine the influence of coupled file changes on the developers' strategy of searching for help for their maintenance tasks. We examine the used information sources, their relevance and the source patterns for the developers using versus those not using

coupled file changes suggestions.

3.6 System Packages

The information about the software components including the examined coupled file changes gives us the opportunity to investigate logical couplings on a higher level like packages or modules.

There are several studies where the package information in the project has been investigated. Robles et al. [RGMA06] studied the characteristics of software including the packages, lines of codes or the programming language. We do not describe the content of the classes in the packages, we identify the approach how the source code is divided in packages based on the features or layers.

The project analysis performed in [Hau02; SDMA07] involves dependency analysis between the packages. In [DL06], the packages and their relations have been visualized using artifacts and metrics and their structural evolution. We investigate logical couplings between the packages and not architectural dependencies.

3.7 Developer's Expertise

3.7.1 Expertise Identification Based on The Contribution

The global approach in software development makes the identification of experts for a given task to be important [TPF+14]. This includes the developers' skills and expertise levels related to a specific topic like for example Java, web or testing, as well as software components or tasks related to specific functionalities of a software product.

Various studies characterize developer's expertise based on the

authorship of the contribution in the source code. In [McD01], the researchers present a recommendation system based on the Line 10 rule. Here the developer whose name shows up on line 10 of the change log is recommended to ask for help. A developer is considered to be an expert if he or she is the last person having the file in memory [RR13].

In [GKSD05], they measure the code ownership based on the information who is the original developer. They asked how many developers worked on the system, which developer worked on which part of the system and investigated behavioral patterns of the developers.

We identify the expertise using the authorship behind the source code changes, however we do not concentrate on the files, we use the packages behind these files.

3.7.2 Expertise Identification Based on The Developer Roles

In [MFH02], they identify that a core of 10-15 developers control the code base and create more 80 % of the functionalities which leads to a strict code ownership policy. The developers have been divided in to core and non-core developers.

An empirical study about classification of developers in to core and peripheral roles has been presented in [Job16]. Here, the researchers use developer networks to model organizational structure of software projects. In [BSS13], the authors statistically explore developers' programming activity to create additional categories of developers next to the core developers like active, occasional or rare developers in order to understand the participation of different developers and the structure of the development.

In [NYN+02], the researchers examine the evolution of an open

source system, the project communities as well as their relations to create evolution patterns. They also describe the roles of the contributing developers to the project like passive, active, core and other. In [BNF14], the authors create developer roles based on the bugs or the source code contribution.

We do not classify specific developer roles, we aggregate their expertise based on the functionalities of the changes in the source code.

3.7.3 Expertise Identification Based on Software Repository Analysis

Developer profiles based on the developers' technical knowledge, organizational information and communication networks have been proposed in [YR14]. Using data from versioning systems, interaction history and issue tracking systems, the profiles have been determined to be useful to collect information for recommending developers to help on a task.

In [MM07], an approach for developer expertise using data from versioning system has been presented. Here, the authors explore how often the developer changed a particular file to show the structure of the development team to make the communication easier.

In [SZ08], an approach has been proposed that recommends experts using mining of version archives based on the changes in the methods.

In [ADG08] they use a classification of the source code tree in CVS as a path to identify and visualize developers expertise for further exploration of the repository. Another approach, presented in [AM07] defines a three-way approach analyzing the committer of the code in the source repository check-in logs, the author of the entries in

the bug reports and the developers working on the modules which contain changes on a file or package level.

We use the version history of the project, which in our case is the Git repository to extract couplings between system packages to be able to identify developers' expertise which can be useful for solving maintenance tasks.

3.8 Recommender Tools

There are several tools implementing logical couplings from versioning histories using various levels of granularity and data sources. They usually offer recommendations or warnings to the developers about related or predicted changes.

ROSE [ZWDZ04] is an Eclipse based tool that recommends change couplings from an existing CSV version archive. It operates at fine granularity like functions or variables but also uses files. The tool uses the Apriori algorithm to find frequent changes in the history to recommend changes to the developer. The recommendations and corresponding actions are produced automatically in Eclipse. In case that the developer does not follow the recommended change, the tool issues a warning.

The tool presented in [KYM06] extracts the changesets from the repository logs. They have been grouped into sequences according to a specific heuristic related to the time and the authorship of a commit.

The main use of SOFTCHANGE, a tool proposed in [Ger04] is to help developers to understand the evolution of a software product. The tool supports queries on changes in the past based on a file level. It extracts the software changes from the version control system (CVS), the issue tracking system (Bugzilla) and mailing lists. It provides

graphical and statistical representations of the authors of the changes and the files being modified together.

HIPIKAT [ČM03] is an Eclipse based tool that offers recommendations from a project's development history for new developers relevant to their tasks. This tool gathers information about the relations between documents of same or different type. It uses data sources like mail archives and on-line documentation. It supports CVS, Bugzilla, Newsgroups, Mailing Lists and the Project Web Site.

A tool for predicting source code changes by mining revision history association rule mining based on CVS version archives has been proposed in [YMNC04]. It concentrates on the evaluation of the usefulness of recommendations based on a file level.

Our tool implementation represents an Eclipse based plug-in which operates on a file level scope and extracts coupled file changes from Git as well as additional set of repository attributes related to these couplings. It mines the file changes and recommends suggestions about other files to be modified based on the versioning history.

It executes the frequent itemset analysis and presents the results to the developer by visualizing the coupled file changes, the commit, issue and documentation attributes as a part of the Eclipse IDE.

Coupled File Change Suggestions

Couple file changes extracted from the version history represent the basic elements of this research. The process of building coupled file suggestions includes two main parts: obtaining coupled file changes (Figure 4.1) and obtaining additional repository attributes (Figure 4.2). We work with Git versioning systems which means that we need to consider the specific organization of the source code changes as well as the available options for extracting and formatting the log data.

4.1 Coupled File Changes

Coupled file changes describe a situation where some developer changed a particular file and also changed another file afterwards.

Let us have the following three commits: $C_1 = \{f_1, f_2, f_3, f_6\}$, $C_2 = \{f_1, f_3, f_5, f_7\}$ and $C_3 = \{f_1, f_2, f_3, f_4\}$. From these commits, we can determine that f_1 and f_3 are found together. This means that when the developers changed file f_1, they also changed file f_3. If these files are found together frequently, it can help other persons by suggesting that if they change f_1, they should also change f_3.

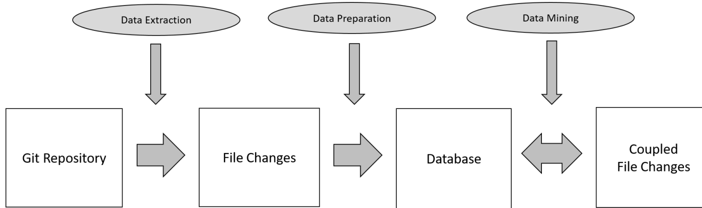

Figure 4.1: Obtaining Coupled File Changes

4.2 Obtaining Coupled File Changes

The process of obtaining coupled file changes consists of the extraction and grouping of change sets from Git, the data preparation for database storage and the execution of the data mining algorithm to extract the coupled file changes.

4.2.1 Data Extraction

4.2.1.1 Changeset Grouping Heuristics

We use a "developer based heuristic" to group file changes as proposed in [KYM06]. It considers the file changes performed by a single committer to be related. Using this heuristic, we group the sets of files we use as a data source to extract the coupled files. We account this heuristic for the reason it was defined for a repository that supports atomic change sets [KYM06] and provided good coverage and precision values. It also delivers a high number of coupled file changes from projects not having many developers or when developers do not commit their changes frequently.

We also investigated two additional heuristics in Chapter 6: "time between the commits" and "branching status of the commits". However, for smaller projects and projects where developers do not often commit, they would significantly decrease the number of change sets we can work with. This can make very difficult to extract coupled file changes from the repository.

4.2.1.2 Identifying Relevant Developers

We extract the information related to the changes in Git using appropriate log commands[1]. They allow us to adjust the format of the Git log output according to our needs. However, before executing the log commands for extracting the changed files and the repository attributes, we need to identify which developers' data we include as our source for the coupled file changes. For this purpose we filter the developers having a minimum number of commits in Git.

[1] Git-Documentation https://Git-scm.com/docs/Git-log/

We have set a rule that we include only developers having 50 or more commits. This user set threshold is important to be able to generate relative frequent file change sets. This avoids reporting non frequent itemsets or failures in the execution of the mining algorithm. We do not include in the mining process developers having less than the specified minimal number of commits.

For example, using the following command we can enlist the developers on the projects and the number of their commits:

```
1 Git shortlog -s -n --format='%aN' |sort -u
```

Here, the *shortlog* command summarizes the Git output using the "*-s*" option, the "*-n*" option sorts the entries according to the number of commits, the format option "*%An*" prints the author names and sort "*-u*" includes only unique authors. This way we identify the developers having committed enough commits to be included in the analysis.

After these adjustments are done, we can extract the commits containing the changed files for each of the developers. For example for the developer "John Doe", we can use the following Git log command to extract the names of the committed files:

```
1 Git log --author="John Doe" --name-only
```

Here, we use the author and the name-only option which include only the developer and the file name paths in their commits. The file path represents the information we use to represent the coupled file changes.

4.2.2 Data Preparation

After the data from the Git log has been generated, we have to prepare the data for the mining process. The first step is to remove the outliers

in the change sets. This includes a removal of empty commits and commits having only one entry. This will avoid algorithm execution problems and does not corrupt the frequency value of the couplings.

4.2.3 Mining Coupled Files

For the process of extracting coupled file changes from Git, we have chosen the frequent itemsets as data mining technique and use the FP-Growth algorithm, a fast algorithm for frequent itemsets mining without candidate generation. The implementation of this algorithm is based on the SPMF mining framework defined in [VGG+14]. However, this framework uses text files as input and output. The size of Git repositories can rise very fast, so this can represent a risk so the analysis can be slow or unable to be performed on an ordinary PC.

To improve the performance and extensibility for larger data sets, we have provided a solution where the data is exported from Git directly in a relational database. The data mining process is performed on the database entries. The output of the data mining process is also stored in a database to avoid the performance pitfall of using text files. The tool implementation of the coupled file changes approach is presented in Chapter 11.

4.3 Obtaining Repository Attributes

Besides obtaining coupled file changes from Git, we provide a set of repository attributes from three different data sources: commit attributes from the Git version history, issue attributes from the issue tracking system and system structure information from the documentation archives.

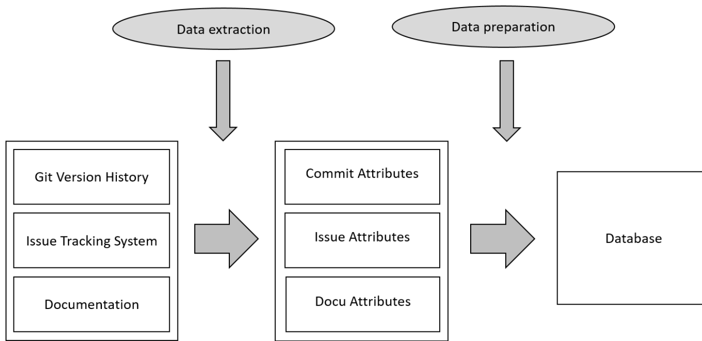

Figure 4.2: Obtaining Repository Attributes

4.3.1 Extracting Commit Attributes Set

Along with the extraction of the change sets from Git, we also perform the extraction of commit attributes. We use a list of commit attributes used in various versioning systems and describe the committed files in Git: *commit id*, *commit message*, *commit date* and *commit author*.

The following command extracts all the commits, their attributes and the changed files by a particular developer.

```
1 Git log --author="John Doe" --pretty=format:"@%h
    ##%an##%ad##%s\a" --name-only
```

In this command, we use the *pretty=format* option to be able to format the output of the Git log according to our needs. This options include the following user defined separators: "α" for the start of the commit, "##" for the delimiters between the attributes and "\a" for the end of the transaction. The last part of the command is the option which enlists the file names in the commits. This command produces for example the output presented in Figure 4.3.

```
@5486558##John Doe##Mon Dec 1 12:13:00 2014 +0100
##created astpa.extension\a
astpa.extension/.project
astpa.extension/META-INF/MANIFEST.MF
astpa.extension/build.properties
astpa.extension/plugin.xml
astpa.extension/pom.xml
astpa.extension/schema/stepedProcess.exsd
```

Figure 4.3: Commit Export from Git

4.3.2 Extracting Issue Attributes Set

To enlist the issues and their attributes, we use a csv export from the issue tracking system. The issues are described using a set of issue attributes used in various issue tracking systems: *issue ID*, *issue description*, *issue author* and *issue time*. An example is presented in Table 4.1.

Table 4.1: Issues Export

Issue ID	Issue Description	Issue Author	Issue Date
#533	The splash screen should include a progress bar to show the data loading	GA	19.06.2013
#869	All Tables in the PDF export should have a consistent design	BZ	12.06.2013

4.3.3 Extracting Documentation Attributes Set

In software product archives, we can find information describing the structure of the product. We can add the *file description* as attribute to describe the functionalities of the files or packages behind the file paths. We add these attributes manually in a csv file and import them

in the database.

4.4 Building Coupled Changes Suggestions

4.4.1 Database Structure

Coupled file change suggestions consist of coupled files from the Git versioning history and the available set of repository attributes. The extracted data from the repository will be stored in a relational database to make it available for further operations.

We define the structure of the database to include the files included in the changed files, the commit, the issue and the documentation attributes. In Figure 4.4, we present an entity relationship diagram which represents the files, commits, issues and documentation as entities. We have separated the commit attributes from the files. The commits are mapped to the issues and the documentation describes the files. The results of the mining process is stored in a database table which includes the files and the involved commits identified by their IDs.

4.4.2 Data Joining

After performing the mining process, we join the software repository attributes to build the coupled file change suggestions (Figure 4.5). The number and type of attributes depends on their availability and the structure of the commits. Using specific Git log commands, we can easily extract the commit attributes from the version control system and join them to the coupled files based on the commit ID value mapping.

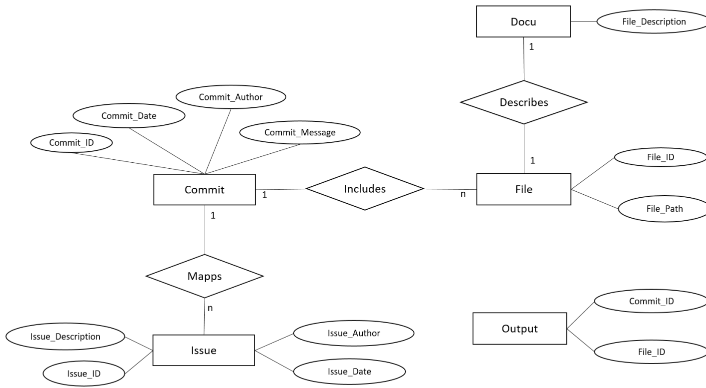

Figure 4.4: Coupled Changes ER-Diagram

Assuming that the entries of the issue tracking system are available and the commits are mapped with the issue IDs, we can join them to the coupled files and the commit attributes to enrich the coupled file change suggestions. To maintain the relation between the commits

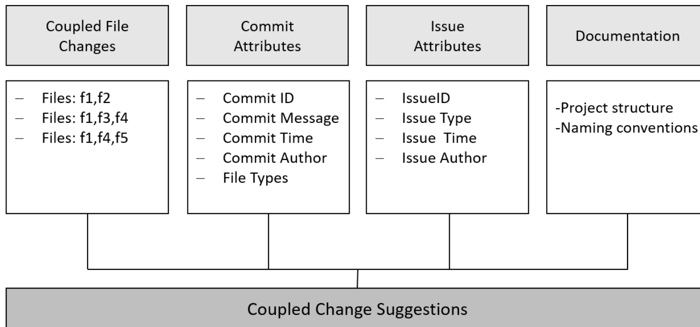

Figure 4.5: Building Coupled File Change Suggestions

and the issues, we map them using the issue IDs found in the commit messages. This mapping is based on the so called *smart commits* which represent an useful and widely spread practice in the software development. Many projects which use this kind of mapping can be found on GitHub[1] or other repository collections.

Let us take as example the commits and the issue in Figure 4.6. Here, both commits are dealing with a similar functionality described in both commit messages. However, not always the developer leave useful or logical description of the commits, so using a mapping of both commits to the issue number#513, we establish a relation to be added in the coupled file change suggestions which gives the information which issue these commits are solving.

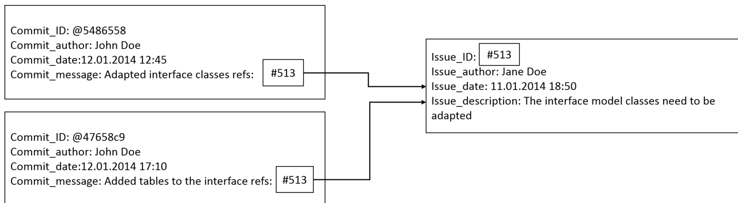

Figure 4.6: Smart Commits

If the documentation archives are available and provide an information related to the system structure like package or class description, we can also add them to the coupled file changes and the other attributes to build more complete suggestions.

For example an interesting information would be the package description for the files in the couplings to show their functionality as presented in Table 4.2.

[1] GitHub https://Github.com/

Table 4.2: Package description

Package	Package Description
astpa/controlstructure/figure	Classes related to the control diagram visualization
astpa/model/hazacc/	Classes related to the hazard action model

By joining all the repository attributes we create coupled file suggestions to be presented to the developers working on maintenance tasks.

4.4.3 Coupled Change Suggestions Example

An example of a simplified coupled file change suggestion is presented in Table 4.3. The coupled change suggestion in this example offers the following information enlisted in the sections 1-10.
(1) Provides a set of three files which represent the coupled files. This set has been built based on the frequency of their co-occurrence in the version history. We can see that the files: *Accident.java*, *Hazard.java* and *Actions.java* were changed together.
Besides the set of coupled files, this coupled change suggestions offer additional information based on the repository attributes provided in the sections 2-10.
Sections 2-5 declare that the set of coupled files has been changed in two commits by reporting their IDs (2), their author (3) and the commit time (4). Both commit messages (5) include mapping to the same issue ID meaning that they are related to this issue.
Based on this mapping we provide set of attributes of the issue (sections 6-9). From (6) we can read the ID of the related issue, the

description is provided in (7), the author is declared in (8) and the date of the issues can be found in (9). In this example, we see that the commits are related to an issue of modifying the copyright header in the software system views.
The information in (10) describes the functionality of the classes for the files in the coupling. This informs the developer that they are dealing with the model views in the user interface of the software system.

We expect that the coupled files and the additional attributes will be useful for the developers in the solving of their maintenance tasks.

A theory on the use of coupled file change suggestions during maintenance tasks is presented in Chapter 5.

Table 4.3: Coupled Change Suggestion

Coupled Files (1)	model/view/ Accident.java	model/view/ Hazzard.java	model/view/ ActionView.java
Commit ID (2)	688a81a	d3eed2e	
Commit Author (3)	MD		
Commit Date (4)	01.09.2013: 22:44:56	01.09.2013: 12:03:10	
Commit Message (5)	Copyright header in View Classes #462	Copyright header in the Views #462	
Issue ID (6)	#462		
Issue Description (7)	Insert Team Information in the copyright header for the views in the UI		
Issue Author (8)	LB		
Issue Date (9)	01.09.2013:09:34:20		
File Description (10)	Class of the model view	Class of the model view	Class of the model view

4.5 Developer Expertise Profiles Based on Coupled Packages

Developers working on their maintenance tasks can use data from other developers who worked previously on the system. They can select whose data matches their tasks the best given their implementation expertise. Using profiles we can identify their expertise so the developers working on this tasks can decide which other developers' data can be useful. Defining developer profiles, we capture the characteristics of their changes according to their contribution in the source code.

During the process of extracting the coupled files, we can acquire the information in which packages the changed files appear using the content of the file paths. This allows us to extend the coupled file changes approach on a package level and create package couplings as presented in Table 4.4.

Instead of simply measuring the frequency of the changed packages by the developer to define the expertise behind the source code changes, we dynamically investigate the couplings between system packages. This way we can identify expertise profiles based on the various sets of source code functionalities changed frequently together.

Table 4.4: Packages couplings

coupling 1	coupling 2
controller=>	commands/policy=>
(1) commands/create (2) components figure (3) controller/editPart/ commands/policy	(1) controlstructure

The process of aggregating developer expertise profiles has been previously defined in [SZ08]. Similarly, we use the changed packages from the version history to aggregate expertise profiles of the developers using the coupled file changes approach as previously presented in Figure 4.5.

Here we can identify which packages the developer changed most frequently together. We rank the coupled packages and aggregate the common functionalities behind them to build the expertise profile.

Table 4.5: Developer profiles

Developer1 packages	profile
controlstructure/controller/ controlstructure/create controlstructure/command/ controlstructure/controller/editParts/ controlstructure/policy/	control structure
Developer2 packages	
ui/interfaces/ ui/grid components/sds/	user interface+components

For example, lets suppose that a developer has a maintenance task related to a change in the *control structure* of the application. He or she looks up at the developers profiles to find the developer whose most frequent package couplings are relevant for his or her task.

In Table 4.5, we have two developers, their most frequently changed packages and their developer profiles. We see that the coupled packages for the first developer cover features related to various sub-packages like *controller*, *command* or *policy*. This information does not deliver a common functionality. Moving one level up, we see that all these sub-packages belong to the same package, the *control struc-*

ture. Having this common functionality, we aggregate the features covered by the profile called *control structure*. We define the name of the profile using the current package name.

The second developer profile describes the most frequently co-changed packages related to the user interface which in this case are not relevant for the specific task. Here we have two different functionalities, *user interface* and *components*. For the first two sub-packages in the second coupling presented in Table 4.5 called interfaces and grid, we have a common packages called *ui*. This identifies that they cover a common functionality related to the user interface. The third package is related to the application component and identifies the functionality called *components*. The first option to provide an expertise profile is to use the common functionality for the first two packages and define the profile *user interface*. Another possibility is from both functionalities covered in all three packages to define the *user interface + components* profile.

The profiles can identify which developer contributed on related source code modifications and functionalities. Furthermore, the expertise profiles can be used as precedent step to enlist which developers' data is suitable to be used to extract coupled file changes relevant for specific maintenance tasks.

Chapter 5

Theory on the Use of Coupled File Change Suggestions

Besides extracting coupled file changes from Git repositories and building coupled file change suggestions to be delivered to the developers during maintenance, the objective of our research is to investigate their use during maintenance tasks. For this purpose we build a *theory on the use of coupled file change suggestions* using the feedback of developers on the acceptance and the usefulness of coupled file change suggestions as well as their influence on maintenance tasks.

5.1 Use of Coupled File Change Suggestions

Coupled changes suggestions can be delivered to the developers with an intention to help them by proposing hints about other file changes they need to perform, based on previous changes in the version system. However, there is no guarantee thay will use this kind of help.

5.1.1 Interestingness of Coupled File Change Suggestions

The use of coupled file change suggestions depends on that if they are accepted by the developers. Before we investigate their use, we determine the level of acceptance of the concept of coupled files using developers' feedback on their interestingness. We define the interestingness as a subjective measure derived from user expectations [McG05]. The subjective interestingness of an information relies on that if it is novel or unexpected so that it brings an advantage for the developer [PM94].

The interestingness of coupled file changes can be expressed using the developers' feedback. In our case study [RW16b], we performed a survey with the developers to collect the feedback about their background and experience as well as their attitude towards the presented concept of coupled file changes. Furthermore, in this case study, we validated the outcomes of the surveys using interviews which provide a possibility to transfer additional information from the developers related to the acceptance of the coupled file changes concept.

5.1.2 Usefulness of Coupled File Change Suggestions

After determining the acceptance of coupled file change suggestions, we explore if the developers have a benefit using them when solving

maintenance tasks. The usefulness is related to the performance perception by the user when solving a task [ANT92]. An information is useful if it can help to achieve an user or system defined goal [FPM92]. We operationalize the usefulness of coupled file changes by exploring their benefits or influence on maintenance tasks using a controlled experiment [RW16a]. Here, the basic indicators of their usefulness are defined using a survey with the developers and an observation of their actions during maintenance task solution.

Observing their maintenance activities we can directly measure the influence of coupled file changes by evaluating the solution of the maintenance tasks. Using observation as a direct method for collecting information, we avoid the influence of the developers' willingness to provide their answers objectively.

5.2 Theory Building Description

Before we start building our theory, we need to describe the theory building process. For that purpose we use the approach presented in [SDAH08]. It characterizes the basic elements of the theory derived from four archetype classes: *Actor, Technology, Activity* and *System*. The theory building process is presented in Figure 5.1 and includes the following steps:

- *Defining of constructs:* The first step involves the definition of the main elements called constructs as well as the determination of their sources.
- *Establishing propositions:* The second step includes the building of relations between the constructs called propositions. Together with the constructs, the propositions represent the basic

elements of the theory.

- *Constructing explanations*: The third step engages the constructing of the explanations to clarify the propositions and determine the theory scope.
- *Theory testing*: Finally, after performing the steps above, we describe the testability of the theory using empirical research.

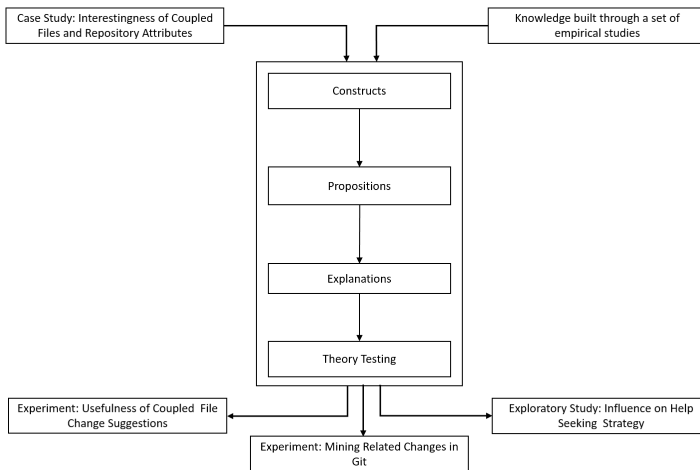

Figure 5.1: Theory Building Steps

5.2.1 Constructs

The first step of the theory building involves the definition of the constructs as basic entities of our theory [SDAH08]. We build the theory using the inductive approach whereby the constructs have been defined using the knowledge from our case study [RW16b] presented in

Chapter 7 and a set of empirical studies as information sources related to: coupled files [KMS07; KYM06; YMNC04; ZWDZ04]; grouping of change sets [ABCO98; PSW11; PTL+11; WS02]; maintenance tasks [CZD11; NBD11; RLR+12; WA09; WLR11] and help seeking during maintenance [KMCA06; LXPZ13]. The sources of the constructs are presented in Figure 5.2.

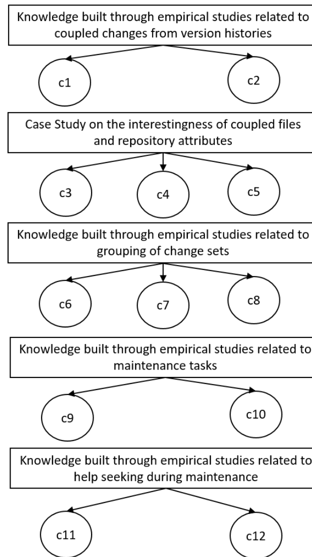

Figure 5.2: Construct Sources

This approach of building constructs from multiple research studies is inspired by the framework presented in [BSL99]. It defines the building of a knowledge based on families of experiments. We use phenomena investigated in various empirical studies to support the validity and the relevance of the constructs and their relations in the context of coupled file change suggestions.

We propose the following list of constructs which represent the basic elements of the theory including the factors that influence the use of coupled file change suggestions as well as the maintenance related aspects the suggestions are affecting:

C1 Coupled File Change Suggestions

C2 Coupled Files

C3 Format

C4 Context

C5 Repository Attributes

C6 Developer Heuristic

C7 Time Heuristic

C8 Branching Heuristic

C9 Task Solution Correctness

C10 Task Solution Time

C11 Help Location

C12 Help Searching Pattern

The first two constructs: *Coupled File Change Suggestions* (C1) and *Coupled Files* (C2), have been investigated in previous empirical studies in the literature [KYM06; YMNC04; ZWDZ04].

Coupled file changes have been defined as sets of files that have been previously changed together frequently. They represent logical couplings on a file level. In Git, the commits represent the change sets used to organize the changes from which we extract the coupled files.

Coupled file change suggestions represent warnings to be delivered to the developers when they edit a file included in a set of files being changed together. Besides the coupled files, the suggestions also include additional information extracted from software repositories.

The *Format* (C3) describes the content of coupled file change suggestions. This involves the aspect how the information is organized and presented to the users. For example, the simplest way is to provide a list containing the coupled files which however does not include a high usability especially if we have large coupled files including many files or when we have numerous couplings.

The *Context* (C4) of coupled file change suggestions depicts the functionality these changes are related to. For example, besides the set of changed files, we can provide an information about the involved functionality by mapping a link between the commits containing the coupled files and the issue to be solved. This way, the developers receive a description to determine the background of the changes. The context of the coupled files can be determined from the content of the commit messages in Git or the textual description of a related issue in the issue tracking system.

The *Repository attributes* (C5) represent a set of well-known attributes from the versioning system, the issue tracking system and the documentation archives. There are many repository attributes provided in various versioning systems. In our case study [RW16b], we have enlisted a set of common attributes used in Git: *commit id*, *commit message*, *commit date* and *commit author*. We use issue attributes found in various issue tracking systems: *issue ID*, *issue description*, *issue author* and *issue time*. We also include the *class*, *file* or *package description* as attributes which describe the functionalities of the files. The participants in the case study reported a subset of the

most interesting attributes to be added to the coupled files as part of the coupled file change suggestions.

These three constructs (C3, C4 and C5) have been defined in our case study on the interestingness of coupled file change suggestions [RW16b] by analyzing the questionnaires and the interviews with developers. The study is described in Chapter 7.

For the reason Git is a distributed versioning system which uses commits as atomic change sets, has a specific time management and supports branching, we determine the constructs (C6, C7 and C8) related to the grouping of change sets. The grouping process is important in order to provide related changes meaning they deal with common functionalities. This way we avoid grouping changes having different context. For example related change sets can describe several phases of a task solution whereby the developers split their works in several commits. They can also include multiple changes on the same set of files in several occasions until the requirements are not satisfied and the developers deliver an acceptable solution.

The developer heuristic (C6) groups the change sets committed by a single developer. The commit time heuristic groups the changes from various developers performed during a predefined time period (C7). Both heuristics support atomic change sets and have been proposed in [KMS07; KYM06].

We add a third heuristic (C8) based on the branching status of the commits in the versioning system. The branching patterns have been previously investigated in [ABCO98; WS02]. In [PSW11; PTL+11], the authors provide an overview of branching practices in software development. This heuristic groups change sets as related if they belong to the same commit branch in the versioning system.

The *Correctness* of maintenance task solutions (C9) and the *Time*

needed to solve them (C10) have been previously investigated in a number of empirical studies using various scope like: comparing different types of maintenance tasks [NBD11], model driven development [RLR+12], ordering of maintenance tasks [WA09] and visualization of software systems [CZD11; WLR11]. The correctness measures how well the developer solved the task. Usually, the researchers use a scale which defines if the developer solved the task completely, partially or did not solve it. The time needed to solve the task is usually measured using time units like minutes or seconds.

We transfer the concepts of *correctness* and the *time* duration of solving maintenance tasks in the context of the *use of coupled file change suggestions*. For example, the researchers in [NBD11] explored the correctness and the time of task solution to compare the results for different types of maintenance tasks. We use the same variables to compare the results between the developers using coupled file change suggestions and those not using coupled file change suggestions.

In [LXPZ13], the researchers define task relevant information sources and describe the process of help seeking during maintenance tasks. The authors of another research [KMCA06], investigate where the developers search for task relevant information and how they navigate between these sources.

Based on these use of various task relevant information sources by the developers during maintenance tasks, we define the constructs: *Help Location* (C11) and *Help Searching Patterns* (C12) in the context of the use of coupled file changes during maintenance tasks. Here, the help location identifies the source of task relevant information related to the functionality or the source code location of the needed modification. We have internal and external information sources. Internal information sources include the IDE functionalities which

deliver information related to the source code the developer need to change like the project explorer, the search functionalities and the source code editor. External information sources represent the documentation of the software product or external web sources like search engines or web sites.

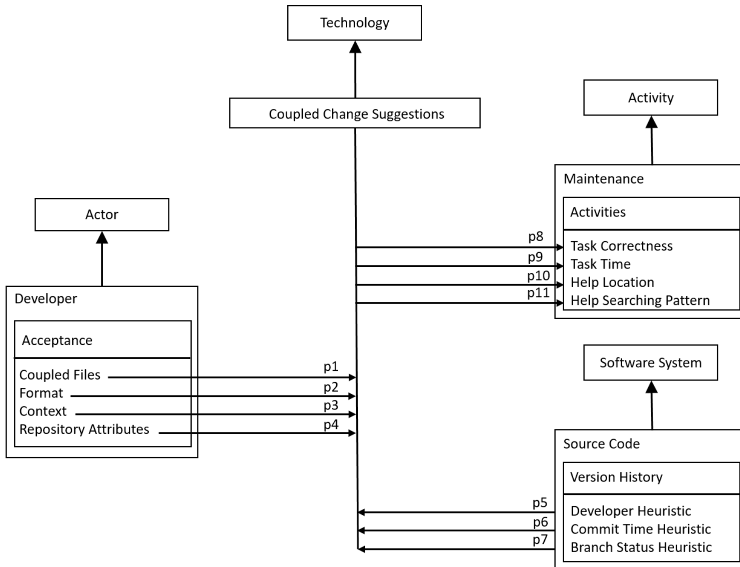

Figure 5.3: A Theory on the use of Coupled File Change Suggestions

5.2.2 Propositions

The second step in the theory building is the specification of propositions. The propositions represent the relationships between the constructs [SDAH08]. They depict which factors influence the use of coupled file change suggestions and how the use of these suggestions

influences maintenance task solutions (Figure 5.3). We define the following propositions:

- **P1** The acceptance of the concept of coupled files influences the use of coupled file change suggestions
- **P2** The acceptance of the provided format of coupled file change suggestions influences their use.
- **P3** The acceptance of the provided context of coupled file change suggestions influences their use.
- **P4** The acceptance of the provided set of repository attributes influences the use of coupled change suggestions
- **P5** The grouping of change sets based on the developer influences the relevance of coupled change suggestions
- **P6** The grouping of change sets based on the time of commit influences the relevance coupled change suggestions
- **P7** The grouping of change sets based on the branching status influences the relevance coupled change suggestions
- **P8** Coupled file change suggestions increase the correctness of maintenance tasks
- **P9** Coupled file change suggestions reduce the time needed to solve maintenance tasks
- **P10** Coupled file change suggestions influence the choice of help search locations during maintenance tasks
- **P11** Coupled change suggestions influence the help searching pattern during maintenance tasks

5.2.3 Explanations

The third part of the theory building process is to explain the provided constructs and propositions [SDAH08]. Using explanations, we clarify the meaning of the constructs and describe the propositions. We present the following explanations for the constructs and proposition defined above:

- **E1** The proposition P1 defines the relation between the constructs: *Coupled Files* (C2) and *Coupled File Change Suggestions* (C1). The acceptance of the coupled files concept is of fundamental importance for the use of coupled file change suggestions. If developers find coupled file changes interesting and they recognize potential benefits from it during maintenance tasks, we can expect that they will use the assistance of suggestions about other potential files to be changed.
- **E2** The proposition P2 relates the constructs: *Format* (C3) and *Coupled File Change Suggestions* (C1). It identifies that the use of coupled file changes depends on the manner how they are brought to the developers. The formatting of the provided content of the information and the visualization directly influences if the developers will accept and use the suggestions.
- **E3** The proposition P3 links the constructs: *Context* (C4) and *Coupled File Change Suggestions* (C1). Providing coupled file changes related to a particular issue or the part of the source code we needed to edit increases the possibility to be accepted. Knowing the nature of the performed changes in the source code can help the developers to determine if this changes are relevant for their tasks.

E4 The proposition P4 involves the constructs: *Repository Attributes* (C5) and *Coupled File Change Suggestions* (C1). In our research, we use a number of common attributes from three information sources: versioning system, issue tracking system and documentation. These sources provides various meta-data or attributes. However, not every attribute is relevant and can be useful to help the developers. Having large set of attributes can overwhelm the developer and decrease the usefulness. Flooding the developer with information can be counterproductive. However, presenting a small or inappropriate set of attributes can make the coupled file change suggestions not interesting.

E5 The proposition P5 depicts the relation between the constructs: *Developer Heuristic* (C6) and *Coupled File Change Suggestions* (C1). Grouping the change sets by their author allows us to define change set groups whereby each group includes commits performed by the same author. Here, all changes performed by the same developer are considered to be related. Grouping the change sets by their author, can influence their relatedness. We suppose that a single developer worked on a smaller diapason of functionalities, compared to a set of different developers.

E6 The proposition P6 describes the relation between the constructs: *Time Heuristic* (C7) and *Coupled File Change Suggestions* (C1). Using the time based heuristic, we can group the files committed by different developers in a predefined time period. All change sets performed by various developer during a time period, are considered as related. Grouping source code changes performed by various developers, can include changes which do not have common functionalities. A combined heuristic, relates

the changes performed by the same developer during a time period. This combination can reduce the possibility of grouping unrelated changes. However, it is very restrictive for smaller projects and limits the number of coupled file changes we can extract from the versioning history.

E7 The proposition P7 represents the relation between the constructs: *Branching Heuristic* (C8) and *Coupled File Change Suggestions* (C1). This heuristic allows us to group the change sets using the branching status. Branches are very often used in Git. They allow the developers to separate the commit changes from each other. This way, we can group the changes based on the branches covering various features or issues. All change sets which belong to the same branch, are considered to be related. Presuming that branches include common functionalities, this kind of grouping can increase their relatedness. However, having various branching concepts and strategies, can deliver different levels of relatedness.

E8 The proposition P8 expresses the relation between the constructs: *Task Solution Correctness* (C9) and *Coupled File Change Suggestions* (C1). This relation reveals that coupled file changes influence the level of successfulness of the maintenance task solutions. Offering hints about other possible parts of the system where the developers need to perform their changes, can increase the number of successfully solved tasks. It can lead to more complete task solutions by reducing the chance to miss a related change in another file which we need to modify in order to solve their tasks.

E9 The proposition P9 describes the relation between the con-

structs: *Task Solution Time* (C10) and *Coupled File Change Suggestions* (C1). It determines that the developers using coupled file change suggestions solve their tasks faster. Using the provided coupled file change suggestions, the developers can reduce the time needed to search for the files needed to be modified in order to solve their tasks. This way, the suggestions can reduce the time needed for delivering the solutions.

E10 The proposition P10 relates the constructs: *Help Location* (C11) and *Coupled File Change Suggestions* (C1). Holding the information about potential change locations in the software delivered by the coupled file change suggestions, developers can select different sources to search for help for their tasks than the developers not using coupled file change suggestions. The suggestions can influence where the developer looks for a task relevant information. This can include searching for code examples or code explanations to help him or her to solve the maintenance task. The suggestions can affect the need to use IDE internal or external information to solve the tasks.

E11 The proposition P11 links the constructs: *Help Searching Pattern* (C12) and *Coupled File Change Suggestions* (C1). Using coupled file change suggestions, developers can use different patterns of help sources. This means that coupled file change suggestions can affect the sequence of actions to locate relevant information during maintenance tasks. The searching sequence consisted of various information sources, can lead us to the information what kind of strategy the developers use to solve their tasks.

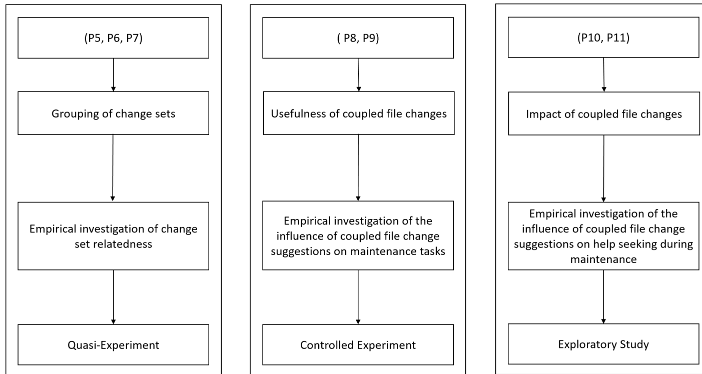

Figure 5.4: Theory Testing

5.2.4 Scope of the Theory

Step four in the theory building is to determine the scope of the theory for the defined archetype classes [SDAH08].

Regarding the technology, the scope of the theory is on the use of coupled file change suggestions for developers working on maintenance tasks. Related to the Actor archetype class, the scope is on inexperienced developers on the project which need to perform a modification on the system. The software system scope of the theory relies on the source code stored in the Git version system where the coupled files have been extracted. The scope on the activity includes maintenance activities related to the task solution.

5.2.5 Theory Testing

After the theory has been built it needs to be tested. This includes the process of empirical validation of the theory [SDAH08]. Our overall

empirical research strategy involves both quantitative and qualitative research. Using a mixed method we perform empirical studies to evaluate the propositions of our theory. The main parts of the testing process are presented in Figure 5.4 are as follows:

- *Grouping of change sets:* The first part concentrates on the propositions about the relatedness of the change sets as sources for extracting coupled files. We tested it using an empirical investigation of the propositions regarding the constructs for grouping change sets. We performed an experiment where we explored the relatedness of the change sets grouped using the developer, the time and the branching based heuristics performing logistic regression. The experiment is presented in Chapter 6.
- *Usefulness of coupled file change suggestions:* We went beyond the interestingness investigated in [RW16b] and we examined in [RW16a] the usefulness of couple file change suggestions. For this purpose we conducted a controlled experiment which is described in Chapter 8. Here, we explored the feedback of developers on the usefulness of coupled changes. We performed a survey and analyzed the developers' opinion on the coupled files and each of the proposed attributes.

  We have presented them coupled file change suggestions consisted of coupled files and a set of repository attributes we defined in the interestingness case study [RW16b]. We captured their activities performed to solve a set of maintenance tasks by recording their computer screens. We used the Grounded Theory method to analyze the recorded videos, to determine the influence of coupled file change suggestions on the task solutions. The developers worked on real issues on the project

used in the experiment.

Further, in this experiment, we tested how the use of coupled change suggestions influences the number of successfully solved tasks by dividing the developers in two groups: one using coupled file change suggestions and another not using this kind of help. Also, the time needed to solve the tasks has been compared between both groups. The experiment is described in Chapter 8.

- *Impact of coupled file change suggestions:* Using an exploratory study [RW17a], we examined how coupled file change suggestions influence the strategy the developers use to search for help during maintenance. We investigated which information sources the developers used by observing their actions. We compared two groups: one that uses coupled file change suggestions and another one not using these suggestions. We also explored the produced patterns of both groups involving the information sources used to find the relevant information for the tasks. This study is presented in Chapter 9.

Chapter 6

Mining Related Change sets in Git: A Quasi-Experimental Study

6.1 Introduction

In this study, we test the part of our theory on the use of coupled file change suggestions related to heuristics for grouping of change sets in Git.

Source code modifications can be performed on several occasions to solve different issues or tasks. These changes in Git are organized

in change sets called commits. Coupled file changes can be extracted from different change sets in the version history. However, not all of them are related. We need to define related change sets to use them as a data source to extract coupled files. To the best of our knowledge, the relatedness of change sets from Git repositories has not been previously investigated. The specific time concept for tracking the changes and the branching as important Git features, have not been previously investigated together to find related file change sets.

Although various CVCS support commits change sets having a time and branching management of the committed changes like Git, the fact that the developers in Git as DVCS work locally and then commit and push their changes on a remote server motivates us to investigate how these factors influence the relatedness of change sets in Git. The relatedness of change sets is important for the reason that grouping of unrelated change set can lead to irrelevant coupled file changes.

The aim of this research is to examine heuristics to group related file change sets in a Git repository which are used to extracting coupled file changes. We include Git characteristics like the time management of the commits and their branch location.

We present a quasi-experiment where we extract related change sets from Git repositories. Here, the basic idea is that change sets are related if they are associated with the same functionality or issue. To establish a relation between the commits and the issues we use a mapping of commit messages with the issue IDs.

We measure the time between the commits and their branching status. We investigate the influence of these two factors on their relatedness using logistic regression.

We use Git repositories from five open source software project repositories. Two of the projects have been developed at the University

of Stuttgart. The other projects have been found on GitHub[1].

6.2 Experimental Design

We select our metrics using the GQM approach [BCR94] and its MEDEA extension [BMB02]. Our *goal* is to define a heuristic for related change sets in Git. Our *objective* is to determine the relation of the commit time and branching towards the relatedness of change sets. The *purpose* is to measure the relatedness for different time commit and branching values. Our *viewpoint* is as software developers and the targeted *environment* is open source systems.

6.2.1 Research Questions

RQ1: Is there an influence of the time between the commits and the existence of branching on the relatedness of file change sets? This question is relevant to investigate since Git maintains the time of commit of file changes and supports local and remote branching in the development. This is our main research question. We investigate the combination of these two factors on the relatedness of change sets which leads us to the formulation of the heuristic we proposed.

RQ2: Is there an influence of the time between the commits on the relatedness of file change sets? Here we refer to the first individual factor, the commit time. We investigate different time periods between the commits to find out if this influences their relatedness.

[1] https://github.com/

RQ3: Is there an influence of the existence on branching on the relatedness of file change sets? We concentrate on the second factor, the branching location. We investigate how the commit location in the same or different branches takes effect on their relatedness.

RQ4: Is there any difference in the relatedness of the time between the commits and the branching on the relatedness of change sets across projects. We investigate the spread out of the relatedness for every repository individually to explore how it varies across the projects.

6.2.2 Hypotheses

For **RQ1** we define the following hypotheses:

$\mathbf{H_{0.1}}$: There is no influence of the time between the commits and branching on the relatedness of file change sets.
$\mathbf{H_{A.1}}$: There is an influence of the time between the commits and branching on the relatedness of file change sets.

To answer **RQ2** we formulate these hypotheses:

$\mathbf{H_{0.2}}$: There is no influence of the time between the commits on the relatedness of file change sets.
$\mathbf{H_{A.2}}$: There is an influence of the time between the commits on the relatedness of file change sets.

For **RQ3** we derive the hypotheses as follows:

$\mathbf{H_{0.3}}$: There is no influence of the branching on the relatedness of file changes-sets
$\mathbf{H_{A.3}}$: There is an influence of the branching on the relatedness of file change sets.

6.2.3 Experimental Variables

6.2.3.1 Independent Variables

In this experiment, we define two independent variables: *time between commits* and *branching*. The first independent variable, represents a continuous numerical variable, measuring the time between a pair of commits. We use calendar days as measure for this variable. The second independent variable is dichotomous, having two categorical states representing the branching status: are the commits in the same branch or not.

6.2.3.2 Dependent Variables

There is one dependent variable, the *relatedness of file changes*. This variable is also dichotomous and has two categorical values: related and not related.

6.2.4 Experiment Design

The specific type of the variables directly influences the type of our experiment design. We investigate the effects of two independent variables (continuous and categorical) on a single dependent categorical variable as presented in Figure 6.1. Having either continuous and/or categorical independent variables, we need a *regression* layout [QK02]. If the dependent variable is categorical we use *logistic*

regression [GE04]. We have multiple independent variables measured along with a single dependent variable so we use *multiple logistic regression* for the analysis.

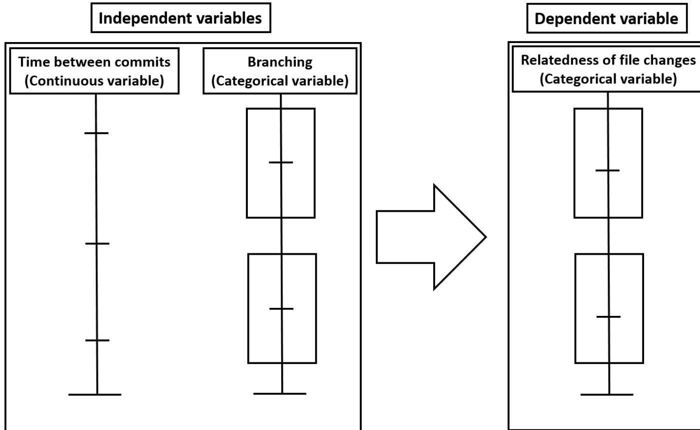

Figure 6.1: Experimental variables

6.2.5 Objects

The prerequisite to include the repositories for analysis is that most of their commits must contain mappings to the issue IDs. We use Git repositories of five different open-source software projects. The first project, ASTPA[1], is an Eclipse RCP application. The second, RIOT[2] is a Java web services and Android project. Project number three is Metrics[3], a JavaScript library for visualizing time-series data. The

[1] http://sourceforge.net/projects/astpa/
[2] https://github.com/SE-Stuttgart/riot/
[3] https://github.com/mozilla/metrics-graphics

fourth project is Akka[1], a toolkit for message-driven applications on the JVM. Project number five is an Add-on builder for Firefox[2]. The first two projects were found in the local GitLab on the University of Stuttgart. The other projects are popular projects hosted on GitHub.

6.2.6 Experiment Instruments

We collect the log data from Git repositories using a self-developed program written in Java. We use it to automatically extract the commits, their attributes, like commit messages and commit times, as well as the committed file. This data is stored in a relational database. We use the SPMF[3] data mining framework to generate the coupled file changes. We have adjusted the framework to work with databases instead of text files. The data analysis is performed using the statistical software SPSS[4].

6.2.7 Data Collection Procedure

We collect the information from the repository logs about the commits, extract the coupled change files and store them in a relational database. In the database, we enlist the coupled file changes for all the developers in the projects. We choose randomly one set of coupled file changes per developer and join the attributes like the commit IDs, the time of commit and the commit message.

- *Data extraction:* First of all we extract the logs from the repository. We gather the commits, the files changed and the attributes

[1] https://github.com/akka/akka
[2] https://github.com/mozilla/addon-sdk
[3] http://www.philippe-fournier-viger.com/spmf/
[4] http://www-01.ibm.com/software/analytics/spss/products/statistics/

for all the developers who committed the file changes and store them in the database.

- *Data Preparation:* To make sure we have relative frequent couplings and to avoid failures of the mining algorithm we need to prepare the data. We exclude the empty commits, the commits containing single entries and the commits which do not contain references to the issue IDs in the commit messages. We do not consider the data from those developers who did less than 50 commits. This filtering is performed to set a rule for the minimum frequency of coupled file changes. In this case, we set the minimum frequency to be 5. We set a minimum support level of 10% meaning that we do not consider the file changes found in less than 5 commits. This way we have a minimum user-set degree of the frequency of coupled changes.
- *Coupled changes randomization:* For every developer we choose a random set of coupled file changes. We join the appropriate set of attributes to the coupled file changes. Using our scripts for automated Git log extraction, we enlist the information about the committer, commit time, commit message and files changed.
- *Commit pairing:* We take the chosen coupled file change and pair all the commits where this change was detected. The pairs are generated by combining all the change sets for a specific coupled change. We start with the latest commits and continue with previous commits. For example, if a set of files changed together was found in the commits with the IDs: *6c08c5a, e7c56dd, cfc90b3* and *9d29bd5*, we will examine the relatedness of all the combined pairs of commits as presented in Table 6.1.

Table 6.1: Commit Combinations

Commit1	Commit2
6c08c5a	e7c56dd
6c08c5a	cfc90b4
6c08c5a	8d29bd5
e7c56dd	cfc90b4
e7c56dd	8d29bd5
cfc90b4	8d29bd5

6.2.8 Analysis Procedure

6.2.8.1 Commit Time Analysis

Every commit in Git has its own time stamp marking the time of commit in the repository. We calculate the difference of time between the paired commits for the investigated set of coupled file changes. The time difference is stored in a data sheet for every commit pair entry included in the analysis. We use calendar days as time units for this measurement.

6.2.8.2 Branching Analysis

We analyze the placement of the commits considering their branching. We leverage the branch of the investigated commit and compare its position with the second commit in the pair. If they are found on the same branch, we set the value of the branching variable to yes, otherwise if they are committed in different branches, we denote it as no.

6.2.8.3 Relatedness Analysis

To find related change sets, we analyze the messages content for all possible pairs of commits where the file change coupling was found. We parse the commit message text for mappings with issue IDs. For the first commit in the pair, we look up the issue id in the commit message text. Next, we repeat this for the second commit in the pair. To determine if these two commits are related, we compare their IDs. If in both commit messages, the references to the issues match, we denote them as related. Commit pairs with different issue references are classified as not related.

6.2.8.4 Logistic Regression Analysis

The analysis outcome of logistic regression is often coded as 0 or 1, where 1 means that the outcome we are interested in is present, 0 that it is not present. If p represents the probability of outcome to be 1, the multiple regression model can be written as follows:

$$\hat{p} = \frac{exp(b_0 + b_1X_1 + b_2X_2 + +... + b_pX_p)}{1 + exp(b_0 + b_1X_1 + b_2X_2 + +... + b_pX_p)} \tag{6.1}$$

here, $\hat{p}$ is the expected probability that the outcome is present; X_1-X_p are distinct independent variables and b_0-b_p are the regression coefficients. The logistic regression model can be also written as follows:

$$ln(\frac{\hat{p}}{1-\hat{p}}) = b_0 + b_1X_1 + b_2X_2 + +... + b_pX_p \tag{6.2}$$

where the outcome is the expected log of the odds that the outcome is present.

We follow the procedure for statistical analysis with logistic regression presented by Schwab in [Sch03] which includes the determining of the sample size, possible numerical problems, the relationship between the combination of the independent variables and the dependent variable, the relationship between the individual independent variables and the dependent variables, the strength of logistic regression relationship and the logistic regression model validation.

- *Sample size*: The first step before we start with the regression analysis is to determine the sample size requirements. The minimum number of cases per independent variable is 10 using the guideline provided by Hosmer and Lemeshow. Their work presented in [HL00] represents the main source for logistic regression.
- *Numerical problems*: To determine if we have numerical problems, we use the standard error value. A standard error larger than 2, indicates numerical problems, such as multicollinearity among the independent variables [Sch03].
- *Relationship between the combination of the independent variables and the dependent variable*: This relationship is based on the reduction on the likelihood for a model that does not contain independent variables and a model that contains independent variables [Sch03]. This difference in the likeliness is called the model chi-square.

  The significance test for the model chi-square is the statistical evidence of the relationship between the combination of the independent variables and the dependent variable. It determines if the null hypothesis for the combination of the predictors should be accepted or rejected. Values equal or less than our

predetermined alpha level of significance, in our case set to 0.05, reject the null hypothesis.

- *Relationship of individual independent variables to dependent variables*: This relationship is based on the statistical significance of the Wald statistics called the Wald test. It determines whether or not the value of the independent variable is statistically significant. Values lower than a 0.05 support threshold for the relationship. This will reject the null hypothesis for the individual independent variable. The value of the B coefficient represents the change in the odds for an event for an one-unit change of the independent variable. We distinguish positive and negative values. Positive values show that the event is more likely to happen, while negative values decrease the odds. The odds ratio identified by the Exp (B) is the exponentiation of the B coefficient. We chose the Exp (B) representation for the reason that the odds ratios can be interpreted easier than the B coefficient, which is set in log units [Sch03].
- *Strength of logistic regression relationship*: It measures the by chance accuracy of the regression. It is computed by calculating the proportion of each of the cases in each group for the dependent variable. We square and sum the proportions of cases in each group to calculate the proportional by chance accuracy rate. An improvement of 25% over the rate of accuracy achievable by chance, satisfies the classification accuracy [Sch03] of the relationship.
- *Logistic regression model validation*: This represent the validation strategy for the regression. We use 80-20 cross validation strategy where we divide randomly the cases in two subsets.

the first subset is the training sample containing 80 % of the cases and an holdout sample containing 20 % of the cases. It is required that the significance of the overall relationship and the relationships of the individual predicting variables for the training sample matches the significance results for the full data set model.

6.3 Results and Discussion

The data study results are available at http://dx.doi.org/10.5281/zenodo.49187.

6.3.1 Descriptive Statistics

The summary of the descriptive statistics for the experiment is presented in Table 6.2. From 1641 commit pairs from all five projects, after the removal of commit messages without issue references, there are 1218 left for analysis. We have 136 related and 1082 unrelated pairs of commits. We also report the minimum, maximum, mean, mode and standard deviation for the time between the related commits. The minimum commit time difference between the related change sets is 0 or in a single day. The maximum varies over the 5 projects between 4 and 25 days, whereby these values are very rare and extreme. Calculating the mean, we found that the average time difference between two related commits varies between 0.35 and 3.33 days. The overall value is around 2 days. The value of the standard deviation varies between 0.933 and 6.457 days depending on the project. The standard deviation value for all the related change sets is around 3.17. Hence, we have a low spread of the commit dates

for the related change sets. The mode shows that the most frequent difference in the time of change for the related change sets is between 0 and 1 days. For the complete data set it is 0 days, which means that most of the related change sets were committed together during one day.

The relatedness distribution is presented in Table 6.3. For the time between the commits, we have created two groups based on the mean value which is 2 days. The first group includes the change sets where the commit time difference is less or equal than 2 days. The second group includes the change set where the commit time difference is more than two days.

Table 6.2: Descriptive Statistics

Project	ASTPA	RIOT	Metrics	Akka	Mozilla Addon	All
Commit pairs	520	149	70	201	365	1218
Commit pairs related	36	48	20	20	12	136
Commit pairs unrelated	484	191	50	81	266	1082
Time between the commits in days						
min	0	0	0	0	0	0
max	15	25	11	4	22	25
mean	1.94	3.19	2.1	0.35	3.33	2.13
st.dev	2.714	4.088	3.478	0.933	6.457	3.17
mode	1	1	1	0	0	0

From 136 pairs of related change sets, 97 commits or 71.3% were committed in less than 2 days, 39 commits or 28.6% of them were committed in more than 2 days. Here we see that most of the related change-sets were committed in less than two days difference.

Table 6.3: Relatedness distributions

Relatedness	Related	Not related
Time between commits		
≤ 2 days	97 (71.3%)	64 (5.9%)
> 2 days	39 (28.6%)	1018 (94.1%)
Commits in the same branch		
yes	55 (40.4%)	49 (4.6%)
no	81 (59.6%)	1033 (95.4%)

The relatedness distribution of the unrelated change sets, 64 or 5.9% of the unrelated commits happened in less than 2 days, 1018 or 94.1% were committed in more than 2 days. Almost 95 percent of the unrelated change sets were committed in more than two days difference.

Considering the branching of related change sets, 55 commits or 40% are in the same branch, 80 or near 60% are not in the same branch. In both cases, in the same and in different branches, we have related change sets. For the unrelated changes, 49 of them or 4.6% are in the same branch, 1033 commits or 95.4% are not. Most of the unrelated change sets were found in different branches.

6.3.2 Influence of the time between the commits and the branching on the relatedness

To answer RQ_1, we test our main hypothesis investigating if the combination of time and the branching influences the relatedness of the coupled changes. The regression results are presented in Table 6.4.

The average sample size for all data sets after down-sampling is

Table 6.4: Regression Results

Statistic	Average (validated)	Average
Sample Size	270	227
Model Chi-Square	0.000	0.000
Standard error Time	0.043	0.048
Standard error Branch	0.441	0.488
B Coeff. (Time)	-0.300	-0.291
B Coeff. (Branch)	-0.607	-0.566
p Wald (Time)	0.000	0.000
p Wald (Branch)	0.194	0.221
Exp (B) (Time)	0.741	0.742
Exp (B) (Branch)	0.553	0.761
By chance Accuracy	66.5%	66%
Model Accuracy	90.4%	91.6%

270. This value is much larger than the minimum number of 10 cases per variable, which satisfies the requirements for the sample size. The average value of the standard error for both of the independent variables is lower than the 2.0 threshold which reports that there are no numerical problems in the analysis.

The presence of a relationship between the combination of the independent variables and the dependent variable is based on the statistical significance of the model-chi square. Our analysis shows that the model chi-square value is 0.000 which is less than the 0.05 threshold. Therefore, the null hypothesis is rejected, meaning that the combination of time between the commits and the branching has an influence on the relatedness of the change sets. The values of the model chi-square statistical significance presented in Table 6.4 for the validated subsets are very close to the values for the full data set. This satisfies the classification accuracy of our regression model.

The values in Table 6.3 indicate that related change sets were found in both groups for the time between the commits. Also they were found both in the same or in different branches. This means that a combination of these two factors influences the relatedness. According to the results, the possibility to have related change sets drops with the rise of the time of commit and the placement of the commits in different branches.

6.3.3 Influence of time between the commits on the relatedness

For RQ_2, we test our second hypothesis where we examine the influence of the time between commits on the relatedness of change sets. The average value for the Wald test for the time variable as independent variable is 0.000 which is lower than the 0.05 threshold. This result delivers a significant presence of a relationship between the individual independent variable (time between the commits) and the dependent variable (relatedness), rejecting the null hypothesis in this case.

The average B coefficient value is negative, meaning that one unit change in the time has a negative influence on the relatedness odds. The average value of the exponent of the B coefficient Exp (B) for the time of commit is 0.741 (0.742 for the validated data set) which means that a change of one unit in the commit time when the other independent variable is constant is going to decrease the odds to have related change sets by 26%.

The results in table 6.3 show that the average commit time period between related change sets is two days, whereby most frequently, related change sets were committed during one day. The frequency of the commit time differences between related change sets is presented

in Figure 6.2. Here we can see that the number of related change sets cases drops with the increase of the time. This confirms the high influence of the time between the commits on the relatedness of change sets.

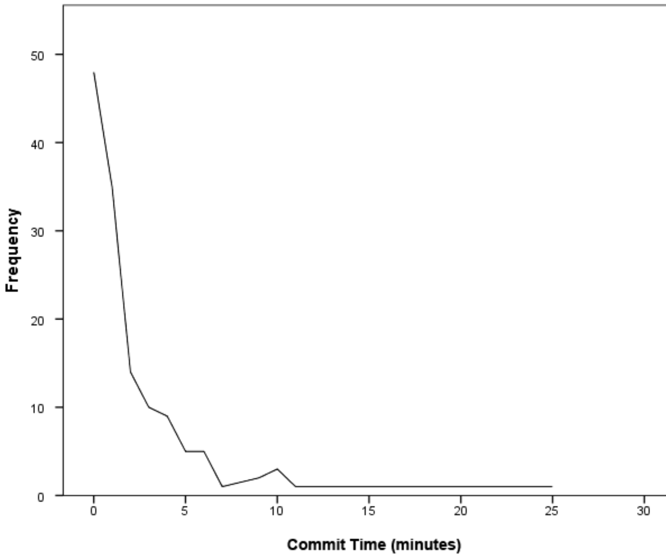

Figure 6.2: Related change sets time distribution

6.3.4 Influence of the branching on the relatedness

For RQ_3, we test the null hypothesis about the influence on the branching on the relatedness of change sets. The average value of the Wald test statistics for the data set is 0.194 and for the validated subsets is 0.221. These values are larger than 0.05. Therefore, the null hypothe-

sis in this case is not rejected. This means that individually measured, the branching does not influence the relatedness of change sets when the other variable is constant.

The results show that both in the same or in different branches we have related and unrelated changes as presented in table 6.3. There is no clear distribution between the related and unrelated change sets considering the branch location of the commits.

6.3.5 Influence of the time between the commits and branching on the relatedness across projects

Regarding RQ_4, we explore if there is a difference in the influence of these two factors on the relatedness across the five project repositories. We use the mean and the standard deviation values to investigate the spread of the Wald Statistic values for all five projects. The results presented in Table 6.5 show that the data from the first three projects delivers useful data. The last two projects do not deliver data ready for analysis. The Wald statistic value which is equal to 0 for all three projects, reports that there is a strong relation between the time of commit and the relatedness. The value of the Wald statistics for the branching in all three projects is above the 0.05 threshold which identifies that there is no significant influence of the branching on the relatedness of the change sets.

The mean value of the Wald Statistics for the commit is 0.000 for all projects. This shows that across all projects, the time has a significant influence on the relatedness of the change sets. For all projects, the value of the Wald Test for the branching is larger than 0.05. This means that in all five projects, the branching does not influence the relatedness of the change sets. The standard deviation for both factors

identifies low spread meaning that there is no a significant difference in the influence across the projects.

The last two projects do not deliver measurable interaction between the time of commit and the branching because that there is no variance in the branching. In these projects, all unrelated cases of change sets are found in different branches. There is not a single case of an unrelated change set where the commits are in the same branch. This make the regression analysis not possible. A possible reason for this situation could be that these two projects include heavy branching. This may influence the way the related change sets are organized in branches. To be able to further investigate this kind of projects, we need to analyze a larger number of change sets per project.

Table 6.5: Influence on relatedness across project

Project	Wald Test (Time)	Wald Test (Branch)
ASTPA	0.000	0.396
RIOT	0.000	0.157
Metrics	0.000	0.310
Akka	-	-
Firefox Addon	-	-
Mean	0	0.287
St. Deviation	0	0.157

6.4 Threats to Validity

The mapping between the commits and the issues represents a central construct validity threat for our study whereby the developers could provide false references. Using data from different projects and various developers having high rate of commit mappings decreases the possibility for this threat.

The relatively high data mining support threshold of 10% excludes a number of coupled changes and commits. However, this threshold ensures a relatively high level of frequency of the reported coupled file changes which avoids the possibility to have changes that could happen by chance.

The influence of the experimenter during the execution of the experiment could affect the internal validity of the study. The experimenter needs to define the relatedness of the change sets by manually examining the commit messages. We use an additional review of a sample of the relatedness of the file changes by a second person.

A potential internal threat could also be the influence of particular developer data on the change sets for the analysis. The commit behavior of the developers and the discipline in the referencing of the commits with the issue can influence the truthfulness of this relation. We minimize the influence on the results by randomly selecting a single set of commits per developer before the logistic regression analysis has been performed.

An external validity threat is the limitation of our analysis generalization on other projects. The analysis is performed on projects which include mapping between the commits and the issues. However, on GitHub there are many projects where this mapping is implemented. We have also used different project repositories developed in various environments. Although we have used relatively small repositories, the use of well-known analysis methods ensures that we can repeat the analysis on larger projects.

6.5 Conclusion

In this study, we have tested the part of our theory on the use of coupled file change suggestions which is related to the influence of the grouping of change sets on their relatedness.

The results give evidence that the combination of the time of commit and the branching status influences the relatedness of change sets. Having most of the related changes sets committed in a day and stored in the same branch supports the heuristic we proposed. We have extracted a number of commits performed in various time intervals which allowed us to investigate the relatedness of the time between the commits and their relatedness. However, the lack of heavier branching in the examined projects limits the outcomes related to the branching status of the commits.

These results show that to build relevant coupled file changes and provide the context of these changes we have to consider the committing time to avoid delivering unrelated file change suggestions. The next steps in our research is to investigate larger and heavier branched repositories and examine higher number of change sets for a deeper investigation on the relatedness of change sets.

Interestingness of Coupled File Changes: A Case Study

7.1 Introduction

Suggestion about files that were frequently changed together can support developers during maintenance tasks, when the developer is new on the project or does not have much experience in software development. Existing studies [KYM06; YMNC04; ZWDZ04], focus on coupled changes from software repositories. They use expert findings and ignore the feedback of developers.

One of the main goals of this thesis is to investigate the use of

coupled file changes during maintenance tasks by gathering their feedback. For this purpose we explore the acceptance of the coupled file changes and common repository attributes related to the file changes.

In this study, we concentrate on applying *Software Repository Mining* to provide suggestions for likely changes so that we can investigate how interesting the suggestions are for the developers and what further information besides version histories might increase the interestingness.

We present an industrial case study on the interestingness of coupled change suggestions. We identify frequent couplings between file changes based on the information gathered from three software project repositories. We use the version control system, the issue tracking system and the project documentation archives as data sources for additional repository attributes we join to the coupled changes we discover. In particular, we investigate the feedback of the developers about the interestingness of our findings by conducting a survey. We evaluate the answers by performing additional interviews and analyze them using the *Grounded Theory* method.

7.2 Case Study Design

The structure of this case study is based on existing guidelines proposed in [RH09].

7.2.1 Research Questions

RQ1: How many coupled changes can we extract from software repositories? This research question provides the basis for our re-

search. It is relevant to investigate for the reason that the number of coupled changes affects the outcome of the repository data analysis by influencing the interestingness of the concept of coupled files.

RQ2: How interesting are coupled change suggestions for developers? This is the central question of this study which decides if developers would like to use the suggested couplings.

RQ3: Does the experience of developers influence the interestingness of coupled changes? We expect that inexperienced developers would be more interested in coupled file suggestions considering their possible problems understanding the system [Say+11]. Therefore, we investigate the developer's programming and project experience.

RQ4: Does the involvement in the project of developers influences the interestingness of coupled changes? We include both developers who were involved and those not involved in the development of the software products used in the case study. Although our goal is to support inexperienced or developers not involved in the projects, we expand the investigation on developers which were included in the software products, we want to get their feedback on the coupled changes in order to determine if their involvement in the projects affects the acceptance of coupled changes.

RQ5: How interesting is additional information from other related project artifacts? After we determine the interestingness of the couplings, we will investigate if adding additional data sources influences the interestingness. First, we examine the version control system that is related to the changes, e.g. commit IDs where the

couplings were found, commit messages, commit dates and authors of the commits. Second, the information stored in the issue tracking system is investigated, attributes like issue description, issue date and issue status. Third, we look into the project documentation archive for information about the project structure and naming conventions.

RQ6: Does the experience of developers influences the interestingness of additional information from other related project artifacts? We investigate if the choice of the attributes from the version control system and the issue tracking system depends on the developer's programming experience.

7.2.2 Case Selection

The case selection is based on their availability and the suitability for our research. We select cases from industry as part of our cooperation with our industrial partners as well as from the available open source projects developed at the University of Stuttgart. Hence, our subjects will be practitioners as well as students.

7.2.3 Data Collection Procedure

The case study uses two main data sources to investigate the coupled file changes. As first data source, we use the artifacts from the software product development archived in software repositories. We did not have any direct contact with the development process of the product. Instead, we examine the repositories of the software product being developed or maintained. The second data source consists of surveys and interviews with the project stakeholders providing direct information. We divide the data collection procedure into five parts.

7.2.3.1 Version Control System

The first unit of data we use is the log data from the version control system. Two software projects used Git, while the third project uses Mercurial as a control management tool. Both are distributed version control systems allowing the developers to maintain their local versions of source code.

The data collection from the version control system consists of four steps which lead to the extraction of the information we need.

- *Log Extraction:* We extract the information from the log file containing the committed file changes and the commit attributes. The log data is exported as text file.
- *Data preprocessing*: After the text files with the log data have been generated, we continue with the preparation of the data for data mining. Various data mining frameworks use their own format, so the input for the data mining algorithm and framework needs to be adjusted.
- *Identifying atomic change sets:* We divide the data into a collection of atomic change sets. Version control systems deal with this issue differently. In our case, the version control systems preserve the possibility to group changes into a single change set or a so-called *atomic commit*. It represents an atomic changeset regardless of the number of files changed. A commit snapshot represents the total set of modified files and directories [Loe09]. We organize the data in a transaction form where every transaction represents a set of files which changed together in a single commit.
- *Data filtering:* We filter the file names and the following commit

attributes: *commit id, commit message, commit date* and *commit author*. We deal with empty entries and outliers and we prepare the log entries for data mining.

- *Change grouping heuristic:* There are different heuristics proposed for grouping file changes [KYM06]. We use a heuristic considering the file changes done by a single committer as related. We group the transactions of files committed only by a particular author. We do not relate the changes done by other committers. This heuristic is suitable for smaller projects or those having lower number of developers for the reason that each developer represents a group of change sets we can use to extract coupled files.

7.2.3.2 Issue Tracking System

Issue tracking systems store important information about the software changes or problems. In our case, the companies chose to use JIRA and Redmine as issue tracking systems. The students also track their issues using Redmine. We investigate the following issue attributes: *issue titles*, *issue descriptions* and *issue messages*. The issue tracking systems support spreadsheet export containing the considered issue attributes.

7.2.3.3 Project Documentation

The software documentation gathered during the development process represents a rich source of data. The documentation consists of file naming conventions, directory paths and the package structure description. From these documents, we discover the project structure.

For example in the last project, the subproject containing the files described by the path `astpa/controlstructure/figure/` contains the Java classes responsible for the control diagram figures of this software.

7.2.3.4 Joining Collected Data

After the mining process is finished and we have identified the coupled changes, we join them with the attributes from the version control system, the issue tracker and the project documentation. In [FPG03b], the authors create a release history database where they import the data from the version control systems and the issue tracking systems. Similarly, we create a database containing all file changes and the corresponding attributes from the repositories.

Every commit has it own hash value which represents the commit id. It is an unique value which identifies all the commits in the database. The issues are identified by their keys. We use the issue keys to follow down the commit where the change took place by using merge points of the issues and the commit messages. We use the path information of the changed files to enlist the sub-projects. As a result we have a list of the most frequently changed files accompanied by the information about the commit attributes, issue attributes and the project structure.

7.2.3.5 Survey and Interviews

We investigate the developers' feedback on the interestingness of coupled changes and the additional attributes by conducting a survey and performing interviews[1] with the developers.

---

[1] All questions are available on `http://dx.doi.org/10.5281/zenodo.15065`

- *Survey:* The developers answer a list of multiple-choice questions on-line. We investigate the background of the developers by asking their programming and project experience. The developers give us feedback on the concept of coupled changes, not on particular couplings. We choose this setup to get as many opinions as possible. Only few developers were available for in-depth interviews on specific findings. The developer can choose between: *interesting*, *neutral* and *not interesting* to evaluate the interestingness of coupled changes and repository attributes.
- *Interviews:* We perform semi-structured interviews to get more in-depth feedback from the developers. This way, we ensure that the developers did not answer the surveys by randomly choosing the options. We ask the available developers who worked on the projects and other uninvolved developers about the interestingness of the file changes and the attributes. We present them actual coupled file changes extracted from the repositories.

7.2.4 Ethical Considerations

The data delivered by the companies is confidential. Therefore, we preserve the anonymity of the stakeholders and the companies during this study. The confidentiality and the publication is regulated by a non-disclosure agreement between the researchers and the companies. All personal information extracted from the repositories, the survey and the interviews is anonymized and is not presented in the study.

7.2.5 Analysis Procedure

The data analysis is a combination of quantitative and qualitative methods. We use quantitative methods to find the number of couplings. We augment the results with a qualitative and quantitative analysis of the survey and the interviews with the developers.

7.2.5.1 Analysis of Repository Data

We analyze the repository data to answer RQ1. We run the mining algorithm to discover frequently coupled file changes. We investigate the additional attributes we gather from the commit logs, the issue tracking export and the project documentation.

- *Data Mining Algorithm:* Various algorithms for mining frequent itemsets and association rules have been proposed in literature [AS94; GG04; HPYM04]. We use the FP-Tree-Growth algorithm to find the frequent change patterns. As opposed to the Apriori algorithm [AS94] which uses a bottom up generation of frequent itemset combinations, the FP-Tree algorithm uses partition and divide-and-conquer methods [GG04]. This algorithm is faster and more memory efficient than the Apriori algorithm used in other studies and allows frequent itemset discovery without candidate itemset generation.
- *Support level:* We analyze the coupled changes by defining the threshold value of the support for the frequent itemset algorithm. We use the thresholds that give us a frequent yet still manageable number of couplings. This threshold is normally defined by the user. We use the technique proposed by Fournier-Viger presented in [Fou13] to identify the support level. These

values vary from developer to developer, so we test the highest possible value that delivers frequent itemsets.

If for a particular developer, the support value does not bring any useful results, we continue dropping the value of the threshold. We did not consider itemsets with a support below 0.2 for the first two projects and 0.1 for the third project. There is a variety of commercial and open-source products offering data mining techniques and algorithms. For the analysis, we use an open-source framework specialized on mining frequent itemsets and association rules called the *SPMF-Framework*.[1] It consists of a large collection of algorithms supported by appropriate documentation.

7.2.5.2 Analysis of Questionnaires and Interviews

To answer RQ2–RQ6, we analyze the questionnaires and the outcomes of the interviews.

We start by investigating the background of the developers by checking their answers about their programming and project experience. We analyze the answers of the questionnaire by calculating the distribution of the frequency of their answers. We put the focus on the answers of the participants about the interestingness of coupled changes and the answers about the additional attributes.

We examine the interviews with the developers to validate the outcomes of the questionnaires and to understand the context of their answers.

We analyze the interviews by using *Grounded Theory* [GS67; SC98]. It has been described to be a methodology used to develop induc-

[1] http://www.philippe-fournier-viger.com/spmf

tive theories for systematically gathering and analyzing data [Bit05]. Grounded Theory is considered to be appropriate for investigating human aspects of Software Engineering [HNM11]. The goal of the Grounded Theory is to generate a theory that emerges from the data being comparatively analyzed.

To analyze the data and build the theory, we use the following types of coding activities in sequence: open, axial and selective coding [SC98]. After these codings, we perform the theoretical coding and create the conceptual model. We use the analysis software *Atlas.ti*[1] to link the codes and create a network diagram.

- *Open coding:* In the open coding, we have a line-by-line examination of the interview transcripts to identify the main concepts and categories together with their dimensions and properties. We code the data from interview answers with a set of open codes derived from our research questions. Before we continue, we write a memo consisting of the hypotheses and ideas noted during the analysis.
- *Axial coding:* After the open coding is performed, we continue with the axial coding where we relate the categories, concepts and codes by identifying the relations among them. This is done using the paradigm model [SC98] and considering the relationships between contexts, interactions, conditions and consequences.
- *Selective coding:* The selective coding formulates a core category to which all other categories and codes can be related and includes all the data.

[1] http://www.atlasti.com/index.html

- *Theoretical coding*: After finishing the open and axial coding, this coding involves the relationships between categories and subcategories and gives meaning to the theory.
- *Conceptual mapping and model:* We express the concepts of our theory and present their relations. We draw a category map which emerges from the analysis.

7.2.6 Validity Procedure

We use well-known techniques and algorithms for repository mining. We extract data from a repository systems used among a high number of companies. We analyze the data from the software repository, perform a survey among the developers and we validate the answers given in the questionnaires by interviewing developers. We collect the answers and compare the results related to the research questions to identify if these reflect the investigated information [RH09]. This way we avoid relying on a possible lack of precision in the answers on the questionnaires by the developers concerning the interestingness.

We choose representative cases with high standards considering software development and standardized development techniques. We use an independent party to record the memos for the interviews and code the information to increase the objectivity of the analysis results.

7.3 Results and Discussion

We report the results of the analysis of the software repository data, the questionnaires and the interviews in relation to the interestingness

of coupled changes and attributes.[1] We discuss the analysis outcomes and evaluate the validity of our results by taking into account the feedback from the developers.

7.3.1 Case Description

The cases in this study are three software projects. The first two projects were provided by IT companies from the area of Stuttgart, Germany. The third one is an open-source project developed at the University of Stuttgart. The first project is a web-based software written in Java and supplied by an industrial partner. The repository of this project contains 1,610 commits performed by 26 developers during 2 years of development. The software changes are stored in Git and the issues are tracked using JIRA.

The second project is a C# software supplied by another partner from the IT industry. the repository contains 159 commits performed by 5 developers during 1 year of development. The project used Mercurial as version control tool and Redmine for issues management.

The third project is a Java open source software which was developed at the University of Stuttgart by student developers. The repository contains 752 commits, committed by 9 developers during 1 year. It uses Git for versioning and Redmine as issue tracking system. Certain project documentation archives of the projects were available from where we extract the information about the software structure and the naming conventions.

[1] The analysis results are available at http://dx.doi.org/10.5281/zenodo.15065

Table 7.1: Results based on repository analysis

	Project1	Project2	Project3
No. of relev. dev.	22	4	9
No. of commits	1610	138	752
No. of couplings	205	13	200
Freq. itemset supp.	0.2	0.2	0.1

Table 7.2: Interestingness of coupled changes

	Involved	Not involved	All
Interesting	2	2	4
Neutral	9	10	19
Not interesting	0	0	0
Sum	11	12	23

7.3.2 Number of Couplings (RQ 1)

In Table 7.1, we summarize the analyzed information from the repositories. Referring to the first project, the data from 22 out of 26 developers was relevant for the study. For the second project, the data from 4 out of 5 developers was taken into account. For the third project, the data committed by all 9 developers was suitable for analysis. The rest of the developers reported a low number of commits so we did not consider their change commits. We excluded their commits as unsuitable for the reason that they did not reach the minimum support for the frequency of the changes we defined previously.

The number of commits represents the size of the projects followed by the number of change couplings we have extracted. The number of coupled changes represents the basis of our analysis. We were able to extract 205 couplings from the first repository. From the second,

a smaller repository, we report only 13 coupled changes. The third repository delivered 200 coupled changes. These results show that we need larger project repositories containing high number of commits to be able to deliver a high number of couplings.

7.3.3 Interestingness of Coupled Changes (RQ 2)

The participants were asked to give their feedback on how interesting coupled changes for maintenance tasks are. Most of the developers (19 of 23) reported a neutral opinion for the concept of coupled changes. A small group of four participants noted coupled changes as interesting. None of the developers rejected the idea as not interesting (Table 7.2).

The fact that the developers did not reject coupled changes allows us to continue our analysis. These results allow us to continue investigating the next research questions. We proceed our analysis and investigate how coupled changes is influenced by the developers' programming and project experience. Taking into account our small sample size, we refrain from formal hypotheses testing.

7.3.4 Influence of Developer Experience on Interestingness (RQ 3)

Both experienced and inexperienced developers were similarly interested in coupled changes which is in contrast to our expectations. In Table 7.3, we present the distribution of the interestingness of coupled changes in relation to the programming experience of the developers. What we can see is that regardless of their expertise level, none of the developers rejected the coupled changes. Very few developers have accepted the coupled changes as interesting, yet most of the developers took a neutral position toward the coupled change suggestions.

7.3.5 Influence of Developer Involvement in the Project on Interestingness (RQ 4)

The results in Table 7.2 show that there is no difference based on the involvement of the developers in the projects. Both involved and uninvolved developers did not reject coupled changes. Continuing with the developers involved in the project development, we group their answers based on their project experience. Table 7.4 shows the distribution of the developers by their project experience. Again in all three groups from beginners to developers knowing the system, most of them have answered neutrally, they did not reject the concept coupled change suggestions, some of them even answered that they find them interesting.

Table 7.3: Couplings and developer's experience

Programming Experience	Freq.	Freq. [%]	Interesting	Neutral	Not interesting
<1 year	2	9	0	2	0
1–3 years	4	17	2	2	0
3–5 years	9	39	1	8	0
>5 years	8	35	1	7	0

Table 7.4: Couplings and developer's project involvement

Project Involvement	Freq.	Freq. [%]	Interesting	Neutral	Not interesting
<6 mo.–1 year	3	27	0	3	0
1–2 years	3	27	1	2	0
>2 years	5	46	1	4	0

Table 7.5: Interesting attributes

Attribute	Frequency	Frequency [%]
Commit message	22	95
File name	18	78
File type	9	39
Commit time	8	34
Commiter	6	26
Commit id	2	9
Issue title	21	91
Issue status	15	65
Issue type	14	60
Issue time	6	26
Project structure	20	86
Naming conventions	15	65

7.3.6 Interestingness of Additional Information (RQ 5)

After the investigation of the coupled changes, we continued examining the interestingness of the repository attributes we have joined to the coupled files presented in Table 7.5. To support the coupled changes, we reported a set of common meta-data attributes [SZ13] which allow us to find more information about the commits, the issues and the product itself. The repositories offer various attributes related to the committed changes, the issues found and the project structure. We asked the participants about their feedback on the interestingness of each of the provided repository attributes. The results show that most of the offered attributes were rated by the developers as interesting.

Considering the commit related attributes, most of the developers found the commit message to be the most interesting attribute followed by the file name. The developers did not show much interest

for the commit time, the committer and the file type. The commit ID as attribute, did not attract the developers' interest.

Regarding the issue related attributes, most of the developers were interested in the issue description. Some developers also found the issue status and type to be interesting. The issue time was not interesting for the developers.

From the documentation related attributes, the developers reported that both naming convention and the project structure information are interesting.

7.3.7 Influence of Developer Experience on Interestingness of Additional Information (RQ 6)

We examined the distribution of interestingness of the repository attributes according to developers' experience level. Based on this distribution we created two general groups of developers in this context: the first group called experienced, includes the developers having more than 5 years experience and the second group called inexperienced, includes developers having less than 5 years of experience. The results show that the experienced developers have a more clear picture of the set of interesting repository attributes. They have chosen a lower number of attributes compared to the inexperienced developers. The inexperienced developers have marked various commit and issue attributes being interesting for them. The more experienced developers' choice is more narrow than the one for the inexperienced ones. The distribution of commit attributes is shown in Figure 7.1. The distribution of issue attributes is presented in Figure 7.2.

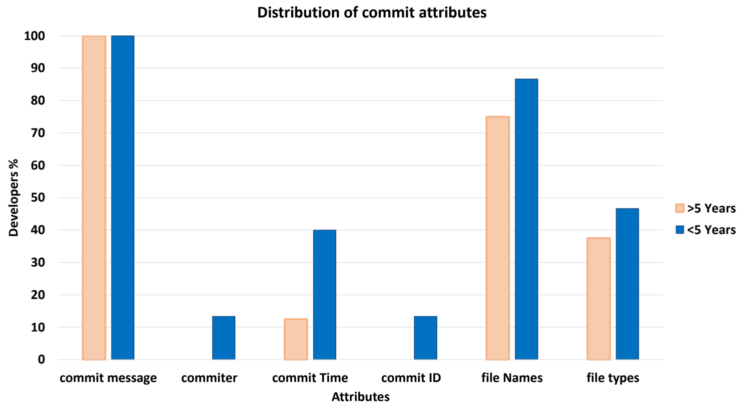

Figure 7.1: Commit attributes and experience

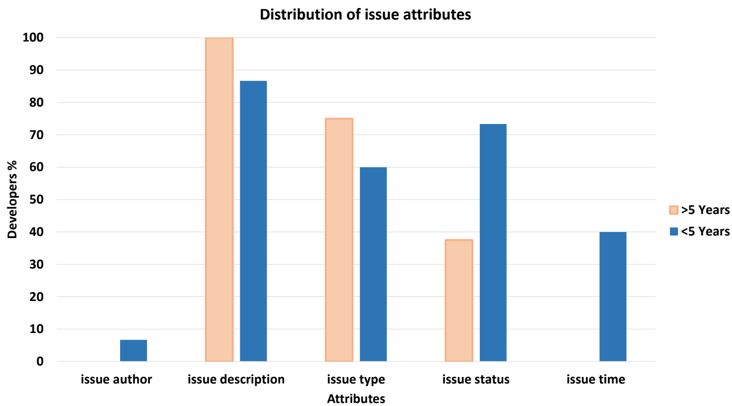

Figure 7.2: Issue attributes and experience

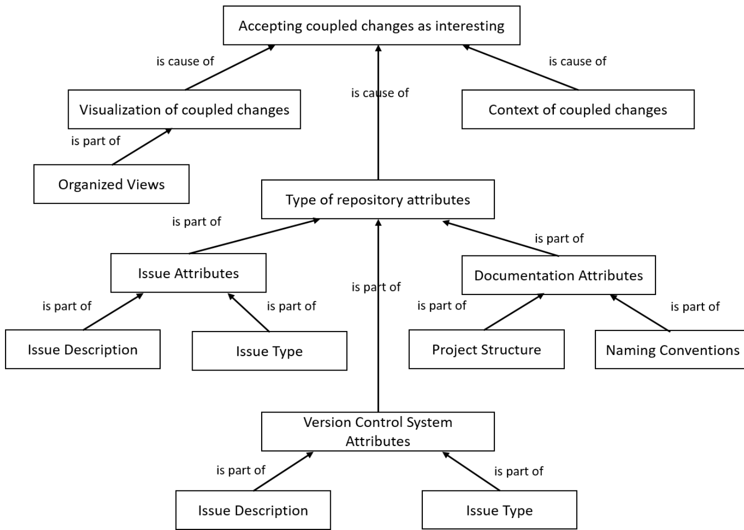

Figure 7.3: Theoretical Framework

7.3.8 Validation and Theory

After the data mining analysis, we performed the interviews with developers who were active on the projects. For the first project, we managed to enlist 2 of the developers for interviewing. For the second project, we interviewed 2 developers and from the third project, we interviewed 4 out of 9 developers. They had been involved in the project from the beginning and have the most knowledge about the software. We also interviewed 4 developers not involved in any of the projects.

Using Grounded Theory analysis on the interview transcripts, we derived a corresponding theory. We created the codes using an open

coding procedure of the memos we created. They represent the answers of our participants to interview questions. We extracted the codes by identifying common issues in their answers.

We continued with the axial coding where we identified several categories as presented in Figure 7.3. The core category we identified after the selective coding is *Accepting coupled changes as interesting*. The results from the theoretical code show the core category, the sub-categories and the relationships presented as a diagram in Figure 7.3. We have categories covering the attributes we found to be interesting: version control, issue and software documentation attributes. They are respectively divided in these subcategories: commit message, file names, issue titles, issue types, project structure and naming conventions. They represent the most interesting attributes which affect the interestingness of coupled changes.

The next categories are the visualization of coupled changes, consisting of the sub-category *organized view*, and the category *context of coupled changes*. The last two categories represent an additional feedback given by the interviewed developers where they would like to see an organized representation of changed files with a possibility to filter the information about them. They would also like to have information about the context of the changes. We present the key concepts of the theory together with their relations in Figure 7.4. We see that the interestingness of the coupled changes also depends on the chosen repository attributes. Furthermore, it is also important to develop an organized presentation of coupled changes to the developers and to describe the context of these changes.

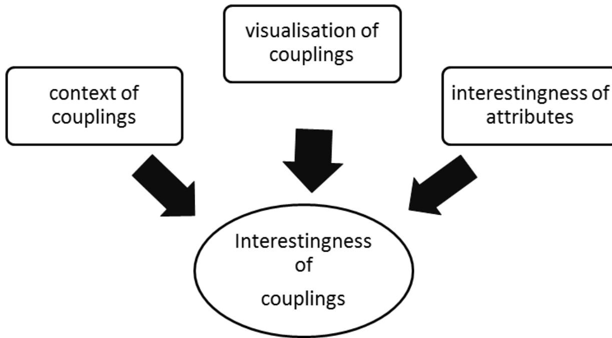

Figure 7.4: Conceptual Model from Grounded Theory

7.3.9 Discussion

The results related to RQ1 show that large repositories deliver more couplings compared to the smaller or younger repositories. Projects with a low number of commits do not provide enough data for a broader analysis. The number of commits and their size limit the output of our analysis. Our results lead to the conclusion that we need a relatively high number of couplings to be able to present a more exhaustive support for the developers in their tasks. Still, the setup of our analysis identifies a number of strongly coupled changes which limits the possibility they have happened by chance. We could reduce the support level of the data mining algorithm to provide a higher number of coupled changes, however, this could produce a threat for their accuracy.

The results for RQ2 report that the developers weakly support that coupled changes are interesting. The general concept of coupled

changes was received mostly as neutral. The developers did not judge the coupled change suggestions very positive for the reason that they were not solving real maintenance tasks. We believe that working with coupled change suggestions related to real maintenance tasks would increase the acceptance of coupled file changes.

The fact that none of the developers rejected the coupled changes, gave us an impulse to investigate other attributes related to the coupled changes. We proceeded with the analysis of the interestingness based on the developers' experience. During the interviews, actual examples of coupled changes were presented to the developers which increased their acceptance.

Considering RQ3, we expected that the coupled changes would be interesting for developers having a lack of programming experience. Our results at contrary show that also the experienced developers are similarly interested in coupled changes. The developers higher experience does not eliminate the possibility that the coupled suggestions could be helpful when working on an unknown source code, software structure or on older project. The fact they did not reject the coupled changes reports that the benefit from them is not limited on novice developers which makes the coupled changes attractive for a broader audience.

According to the results related to RQ4, both uninvolved developers in the project development of the investigated software products and those who worked on the projects provided a neutral feedback. They fact that they did not reject the coupled changes increases the target group for our coupled changes suggestions. These unexpected results show that also the developers working on a particular part of the source code could use some help when working on other parts of the system. These findings encourage us to include the coupled changes

as a part of an integrated tool support for developers.

Answering RQ5, our results show that most of the attributes from the provided set were interesting for the developers. These results were also validated by the interviews. Using the commits, the questionnaire and the interviews, we reported that the commit message and the file names are the most interesting attributes. This shows that the developers found the information about the files being changed and the description of these changes to be interesting.

For the issue attributes, the developers reported that the issue description and the issue type are interesting, meaning that they were looking for the information which describes the problem to be solved and the importance of the issue.

For the documentation attributes, the project structure and the naming convention were both interesting for the developers. This shows that they were looking for the information that could help them find the location in the system to begin with their source code changes.

We reported a set of repository attributes used by well-known versioning and issue tracking systems involved in the projects. The attributes we defined are known and common in software development. During the analysis of the interviews, however, we found that the developers want a clear graphical representation of the coupled changes. They also reported that they would like to see the context of the coupled changes. This brings additional aspects to be considered in further research about coupled changes.

The results for RQ6 show that experienced developers know well what kind of repository attributes they want to see. Their choice is more precise compared to the inexperienced developers. The inexperienced developers did not have a clear picture which attributes to choose from the provided set. The fact that developers with different pro-

gramming experience considered various attributes to be interesting brings us to the conclusion that we should not make a fixed choice of attributes for all developers. We can offer a flexible way for the developers to choose the attributes individually. This way, we support the developers which are not experienced and would like to have an overview of the provided set of repository attributes. On the other side we would like to offer the experienced developers to hide the unnecessary information including the not interesting attributes during maintenance tasks.

The results of the grounded theory show that the interestingness of coupled file changes is influenced by their presentation form and the related information such as the description of the change context. Providing a good visual concept is inevitable for a successful visual representation. Also the repository attributes influence their interestingness. Choosing wrong or not useful attributes can drop the acceptance of coupled change suggestions.

7.3.10 Evaluation of Validity

We validated the results of our study by checking all the steps in the procedure of gathering and transforming the data from the repository, the analysis methods and the results. In our study, we used a single data mining technique for the reason that the frequent itemsets technique is most appropriate for investigating frequent couplings. We investigated products built with common technologies and the repositories are maintained by well-known and commonly used products.
We tested different threshold values for the support and the confidence of the algorithm to produce a sufficient number of frequent itemsets. The relatively low support threshold signalizes that there is

not much space for a greater reduction of the value. However, it also reports a relatively low number of frequent couplings which reduces the possibility that these couplings happened by chance.
We validated the outcomes of the questionnaire answers by asking the developers again in the interviews about the interestingness of the couplings and attributes. The interview transcript was coded by two persons after we compared the notes. This way we checked whether we understood the developer's answers correctly. We interviewed both involved and not involved developers on the projects. We also performed double checks of the coding and the outcomes of the Grounded Theory analysis.

7.4 Conclusion

The results of this case study represent the basic source for the constructs related to the acceptance of coupled files: the format, the context and the chosen repository attributes. They represent the fundamentals of our theory on the use of coupled file suggestions.

The feedback of developers on the interestingness of coupled changes is mostly neutral. Our results lead to the conclusion that the couplings were weakly accepted by developers having various programming experience and level of involvement in the project. The developers accepted most of the proposed software repository attributes joined to the couplings as interesting. Experienced developers report a narrower or more individual choice of attributes as opposed to the inexperienced developers. The Grounded Theory shows that the set of repository attributes influences the interestingness of coupled changes.

Although we provided a number of repository attributes, the devel-

opers suggested additional aspects concerning the coupled change suggestions and the repository attributes. They would like to see more information about the change context and the visual presentation of the coupled changes.

Working on real maintenance tasks would increase the acceptance of coupled change suggestions. We need to develop a visualization concept for the coupled change suggestions and provide the possibility that the developers can individually adjust their choice of repository attributes.

The next step is to perform an experiment to investigate coupled changes by directly observing their use for a real maintenance tasks. They could be visualized in a tool to present these changes to the developers.

CHAPTER 8

Usefulness of Coupled File Changes: A Controlled Experiment Study

8.1 Introduction

We perform this study to test the part of our theory on the use of coupled file change suggestions related to their influence on maintenance tasks.

Existing studies about coupled change recommendations [BDO+13; KYM06; YMNC04; ZWDZ04] focus on the presentation of the mining

results and expert investigations and neglect the feedback of developers on the findings as well as the influence of coupled changes on the performance on maintenance tasks.

The aim of this study is to investigate the usefulness of coupled file change suggestions in supporting inexperienced developers in their maintenance tasks. We identify sets of files being frequently changed together using the information gathered from the software repository. We use the version control system, the issue tracking system and the project documentation archives as data sources for additional attributes. We join this information to the coupled changes we discover. The usefulness of coupled file changes is determined by analyzing the developers' feedback, their influence on the correctness of the task solutions and the time spent for solving the maintenance tasks.

We present a controlled experiment on the usefulness of coupled change suggestions where each of the 36 participants try to solve 4 different perfective maintenance tasks and report their feedback on the usefulness of the repository attributes. We use this experiment to test the propositions of our theory on the use of coupled change suggestions by investigating their influence on the number of successfully solved tasks and the time needed to solve the tasks.

8.2 Experimental Design

In this section we define the research questions, hypotheses and metrics used in our analysis.

8.2.1 Study Goal

We use the GQM approach [BCR94] and its MEDEA extension [BMB02] to define the study goal. The *goal* of the study is to analyze the usefulness of coupled file change suggestions. The *objective* is to compare the correctness of the solution and the time needed for a set of maintenance tasks between the group using coupled change suggestions and the group that does not use this kind of help. The *purpose* is to evaluate how effective coupled file change suggestions are regarding the correctness of the modified source code and the time required to perform the maintenance tasks. The *viewpoint* is that of a software developers' and the targeted environment is open source systems.

8.2.2 Research Questions

We investigate the usefulness of coupled file change suggestions and the corresponding repository attributes. In this study, we concentrate on perfective maintenance to have a similar set of tasks. For that purpose we define the following research questions:

RQ1: How useful are coupled file change suggestions in solving perfective maintenance tasks?
To determine the usefulness of the coupled file changes concept we define the following sub-questions:

RQ1.1: Do coupled file change suggestions influence the correctness of perfective maintenance tasks?
We investigate if there is any difference in the correctness of the maintenance task solutions between the group of developers who used coupled file change suggestions and the group not using them.

RQ1.2: Do coupled file change suggestions influence the time needed to solve perfective maintenance tasks?
We explore if the time the developers need to complete the maintenance tasks differs between the group using coupled change suggestions and the group not using these suggestions. We consider two scenarios: The first one includes only the time needed to solve the tasks, the second one also includes the time needed to select relevant coupled file changes.

RQ2: How useful are the attributes from the software repository in solving perfective maintenance tasks?
The second research question deals with the attributes from the versioning system, the issue tracking system and the documentation. We investigate the perceived usefulness of each attribute in the proposed set to understand which attributes are good candidates to be provided to the developers.

8.2.3 Hypotheses

We formulate the following hypotheses to answer the research questions in our study.

For **RQ1.1** we define the following hypotheses:

$\mathbf{H}_{0.1.1}$: There is no significant difference in the correctness of perfective maintenance task solutions between the developers using coupled file change suggestions and those not using these suggestions.
$\mathbf{H}_{A.1.1}$: There is a significant difference in the correctness of perfec-

tive maintenance task between the developers who used coupled file change suggestions and those not using these suggestions.

For **RQ1.2** we address the following hypotheses:

$\mathbf{H_{0.1.2}}$: There is no significant difference in the time required to solve perfective maintenance tasks between the developers who used coupled file change suggestions and the developers not using these suggestions.
$\mathbf{H_{A.1.2}}$: There is a significant difference in the time required to solve perfective maintenance tasks between the developers who used coupled file change suggestions and those not using these suggestions.

To answer **RQ2** we formulate the following hypotheses:

$\mathbf{H_{0.2}}$: There is no significant difference in the perceived usefulness among the attributes from the software repository in the current set.
$\mathbf{H_{A.2}}$: There is a significant difference in the perceived usefulness among the attributes from the software repository in the current set.

8.2.4 Experiment Variables

We define the following dependent variables: the correctness of the solution after the execution of the maintenance task, the time spent to perform the maintenance task and the usefulness of the repository attributes. For the first variable, the correctness of the task solution, we assign scores to each developer's solution of the maintenance tasks.

Our approach is similar to the one presented by [RLR+12] where the correctness of the solution of the maintenance task is manually assessed by defining scores from totally incorrect to completely correct task solution. We define three scores: 0 if the developers did not execute or did not solve the task at all, 1 if the task was partially solved and 2 if the developer performed a complete solution of the maintenance task. The solutions are tested using unit tests to ensure the correctness of the edited source code.

The second variable, the time required for executing the maintenance tasks is measured by examining the screen recordings. We mark the start time and the end time for every task. We calculate the difference to compute the total time needed to solve each task. We differentiate the time needed only to solve the tasks t_s and the time needed to determine the relatedness of the coupled files t_r. For the third variable, the usefulness of the repository attributes, we use an ordinal scale to identify the feedback of the developers. The participants can choose between the following options for each attribute: very useful, somewhat useful, neutral, not particularly useful and not useful. We code the usefulness feedback using the scoring presented in Table 8.1.

Table 8.1: Usefulness score

very useful	somewhat useful	neutral	not particularly useful	not useful
5	4	3	2	1

8.2.5 Experiment Design

We distinguish two cases for the maintenance tasks: the first one includes tasks executed on Java Code in the Eclipse IDE without any suggestions and the second one includes tasks executed with additional coupled files suggestions and corresponding attributes from the repositories. We use a similar approach to the one presented by [RLR+12] and define two values: − for Eclipse only and + for the coupled file suggestions.

We use a counterbalanced experiment design as described in Table 9.1. This ensures that all subjects work with both treatments: without and with coupled change suggestions. We split the subjects randomly into two groups working in two lab sessions of two hours each. In each session, the participants work on two tasks using only the task description and on two tasks using coupled file change suggestions and their related attributes. The participants in the second lab swapped the order of the tasks in the first lab.

Table 8.2: Experiment Design

Lab	Tasks	
Lab 1	Tasks 1-2 (–)	Tasks 3-4 (+)
Lab 2	Tasks 1-2 (+)	Tasks 3-4 (–)

8.2.6 Objects

The object of the study is an open source Java software called A-STPA. The source code and the repository were downloaded from

SourceForge.[1] The system was built mainly in Java by 12 developers at the University of Stuttgart during a software project between 2013 and 2014. It represents an Eclipse-based tool for hazard analysis. The source code contains 16012 lines of code and 178 classes organized in 37 packages. The Git repository of the project contains 1106 commits from which we extracted 205 coupled file changes.

8.2.7 Subjects

The experiment participants are 36 students from the Software Engineering course in their second semester at the University of Stuttgart (Germany). The students have basic Java programming and Eclipse knowledge and have not been related in any way with the software system investigated in the experiment.

8.2.8 Materials, Procedure and Environment

All subjects received the following materials which can be found in the supplemental material of this paper.

- *Tools and code*: The participants received the Eclipse IDE to work with, the screen capturing tool and the source code they need to edit.

- *Questionnaires*: The first questionnaire is filled in at the start of the experiment and it is related to their programming background. The second questionnaire performed at the end of the experiment is about their feedback on the usefulness of coupled changes and the additional set of repository attributes.

[1] https://sourceforge.net/projects/astpa/

- *Software documentation*: We provided the technical documentation for the software system including the architecture description covering the sub-projects, the overview of the classes in the data model, the application controllers, the graphic editor and the package descriptions.
- *Setup instructions*: The participants received the instruction steps how to prepare the environment, where to find the IDE, the source code and how to perform the experiment.
- *Maintenance tasks and description*: Every participant received spreadsheets with four maintenance tasks and their free-text description. The maintenance tasks represent quick program fixes that should be performed by the participants according to the maintenance requests [Bas90]. The maintenance tasks used in the experiment require the participants to add various enhancements to the system. The changes do not influence the structure or the functionalities of the application. The tasks are related to simple changes of the user interface of the system. All four maintenance tasks are perfective and have been assigned to the participants in both groups.
- *Set of coupled files:* The files changed together frequently used to solve a similar tasks have been provided to the group that uses coupled file changes.
- *Repository Attributes:* The attribute set from the versioning system, the issue tracking system and the documentation about similar tasks performed in the system. They have been joined to the coupled files using a mapping between the commits containing the coupled files and the issues using their issue IDs.

The environment for the experiment tasks was Eclipse IDE on a Windows PC in both treatments. For each lab, we prepared an Eclipse project containing the Java source code of the A-STPA system. The project materials were made available to the subjects on a flash drive. The participants had a maximum of two hours to fill the questionnaires and perform the maintenance tasks.

8.2.9 Selection of Change Author

According to the used heuristic for grouping the change sets in the versioning history, we need to select the authors of the changes whose data will be included in the analysis.

The selection process of the developers as authors of the source code changes is presented in Figure 8.1. Out of 12 developers who worked on the A-STPA software, after performing the frequent itemset analysis, we have 8 developers left whose entries in the repository delivered coupled files.

We have 4 maintenance tasks to be solved in the experiment. For each of the tasks we use commits from a different developer to avoid the influence of the authorship of the commits on the tasks. Out of the 8 developers we need to select 4, one for each maintenance task.

8.2.10 Selection of Coupled Files

After selecting the developers, we continue with the selection of the coupled files. The process includes the selection of the most frequent coupled files followed by the selection of relevant coupled files as presented in Figure 8.1.

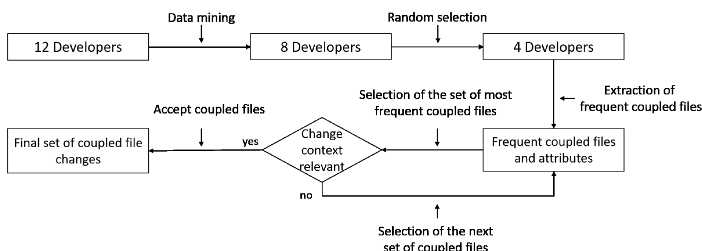

Figure 8.1: Changes Selection

8.2.10.1 Selection of the Most Frequent Coupled Files

We need to select the coupled files which we will include in the suggestions for the developers in the experiment. For each of the 4 developers we list the most frequent coupled files we have extracted. We sort the sets of coupled file changes by their frequency in descending order, so on top of the list we have the most frequent set of coupled files. We start selecting the sets of coupled files from the top of the list.

We do this for two main reasons: (1) To avoid a potential subjectivity in the selection of the coupled files. (2) We want to use the strongest couplings, meaning the coupled files which are frequent and did not happen by chance.

8.2.10.2 Selection of Relevant Coupled Files

After identifying the most frequent coupled files, we examine their broader change context. This means that we need to determine if they fulfill the requirements to be: (1) of perfective nature and (2) related to modifications in the user interface of the application.

We determine this change context using a manual analysis of the content of the commit messages where the coupled file changes were included as well as the description of the related issues. To perform this, we use the mappings between the commit messages and the issue IDs provided as part of the corresponding repository attributes we added to the coupled file changes.

8.2.11 Classification of Issues

We classified the issues for the examined software systems using the approach proposed in [HGH08]. We determine the following classes of issues:

- *Corrective*: These issues cover failures related to the processing or performance of the software.
- *Adaptive*: These changes include modifications related to the data and the processing environment.
- *Perfective*: The changes include modifications related to performance, maintenance or usability improvements.
- *Implementation*: These tasks include new requirements for the software system.
- *Other*: These include changes that are not functionally related to the system like copyright or control version system related issues.

We go further and classify the perfective changes based on the most frequently involved system functionalities. For example, we want to know how many perfective issues have been defined for the user interface of the application and what are the main parts of this interface

addressed in these issues. This way we expose the representativeness of the selected coupled file changes and the defined tasks for the software system we examine.

8.2.12 Definition of Tasks

After we determined the sets of coupled file changes which fulfill the requirements of the experiment, we continue with the definition of the tasks the participants need to solve.

Firstly, we determine the change context of the selected coupled file sets more precisely by looking up repeatedly in the related commit messages and the issue description. This identifies the functionality the file changes are related to. We use the mapping of the issue IDs and the commit messages to follow up this information. After we have identifies the issues related to sets of relevant coupled file changes, we define perfective maintenance tasks related to similar functionalities covered in these issues. For example, in Table 8.3 we have an issue extracted from the issue tracking system of the A-STPA product which defines that a new item in the application view should be created using a keyboard shortcut. The commit message for the changes solving this task represents the comment of the developer who placed the shortcut. Considering the described functionality, we create a task where the developer needs to create a new shortcut combination for that purpose. In the same manner, we repeat the procedure for each of the relevant coupled files we have selected and define 4 tasks.

The content of the text description of the tasks is related to the content of the issues we extracted from the issue tracking system. We keep the content of the task definitions very simple. They contain the functionality or the part of the system which has to be changed

Table 8.3: Task Information and Coupled File Changes

User Task	Task Solution File Set
Change the shortcut for adding new items in all the user interface views from "SWT.KeyDown and ’n’" into "SWT.KeyUp and ’y’"	ControlActionView.java SystemGoalView.java DesignRequirementView.java SafetyConstraintView.java HazardsView.java AccidentsView.java
Related Commit	**Suggested Coupled Changes**
I have set a simple shortcut for new items to be "n", which can be quickly changed if needed.	ControlActionView.java DesignRequirementView.java SafetyConstraintView.java AccidentsView.java
Related Issue	
Using a keyboard shortcut, a new item should be created in the application views.	

and the action to be performed. This makes it easier to replicate the process using other software products and their repositories.

8.2.13 Tasks and Coupled File Changes

Our goal is for each of the tasks to provide coupled file changes related to their context. This feature is of great importance for the study. Offering unrelated coupled file can be misleading and confusing for the developers.

We can extend the examination of the commit messages content and the issue descriptions to determine the change context as a part of a tool using natural-language processing techniques. We can compare the content of the user input or the issue content with the comments in the commit messages or issue description we mapped to the coupled file change sets. However, this exceeds the scope of this study and can be considered as future work.

8.2.14 Solution of Tasks

The complete list of files included in the task solutions are defined manually by analyzing the solutions of the related issues and evaluated by an independent party.

An example of the relation between the files included in the solution for a particular maintenance task and the set of coupled file changes is presented in Table 8.3. Here, we can see that to be able to solve the mentioned task, the developer needs to change 6 files which are related to the views of the application.

The coupled change suggestion based on an issue related to the defined task recommends 4 files to be changed. These files were extracted from the version history have been changed frequently together in the past.

We would like to point out that the file change suggestions do not represent the solutions for a particular task in the experiment. The solution usually contains more files than the provided suggestions. Although the provided suggestions contain a subset of the solution set, the developers still need to find the location in the source code meaning the method or the class they need to modify in order to solve the tasks. This is finer grained information we do not provide

in our coupled files. The developers still have to read the repository attributes and decide if they want to follow the coupled file change suggestions.

8.2.15 Maintenance Activities

After receiving the task description, the participants investigate the source code of the application, identify the files where the change is needed and perform the change according to the requirement. The scenario for solving the provided maintenance tasks includes the following activities [NBD11]:

- *Task understanding*: First of all, the participants need to read the task description and the instructions and prepare for the changes. They can ask if they need some clarification about the settings and the instructions.
- *Change specification*: During this step, the participants locate the source code they need to change, try to understand and specify the code change.
- *Change design*: This step includes the execution of the already specified source code changes and debugging the affected source code.
- *Change test*: To specify the successfulness of the performed code changes, a unit test needs to be performed. This step is performed by the experiment organizers after the lab sessions.

8.2.16 Data Collection Procedure

We collect data from several sources: the software repository of the system, the questionnaires, the provided task solutions and the screen

capture recordings.

8.2.16.1 Software Repositories

- *Version Control System*: The first data source we use is the log data from the version control system. The investigated project uses Git as a control management tool. It is a distributed versioning system allowing the developers to maintain their local versions of source code. This version control system preserves the possibility to group changes into a single change set or a so-called *atomic commit* regardless of the number of directories, files or lines of code that change. A commit snapshot represents the total set of modified files and directories [Loe09]. We organize the data in a transaction form where every transaction represents the files which changed together in a single commit. From this data source we extract the coupled file changes and the commit related attributes.

Table 8.4: Repository Attributes Description

Attribute Name	Attribute Description
Commit ID	Unique ID of Git commit
Commit Message	Free-text comment of the commit in Git
Commit Time	Time-stamp of committed change in Git
Commit Author	Person who executed the commit
Issue Descr.	Free-text comment on issue to be solved
Issue Type	Type of the issue: bug, feature
Issue Author	Person who created the issue to be solved
Package Descr.	Text description of the package: layer, feature

- *Issue Tracking System:* It stores important information about the software changes or problems. In our case, the developers

used JIRA as issue tracking systems. This data source is used to extract the issue-related attributes.

- *Project Documentation:* The software documentation gathered during the development process represents a rich source of data. The documentation contains the data model and code descriptions. From these documents, we discover the project structure. For example, in the investigated project, the package containing the files described by the following path: *astpa/controlstructure/figure/*, contains the Java classes responsible for the control diagram figures of this software. We use the documentation to identify the package description.

The complete set of attributes we extract from the software repository is presented in Table 8.4.

8.2.16.2 Questionnaire

The developers answer a number of multiple-choice questions. Using the first questionnaire, we investigate the developers' programming background. We use a second questionnaire after the tasks are solved in order to gather the feedback on the usefulness of coupled changes and the additional attributes[1].

8.2.16.3 Tasks completion

Similarly to other studies [Cha08; NBD11; RLR+12], we define two factors which represent the completion of the maintenance tasks:

[1] The questionnaires are available in the supplemental material of this paper

- Correctness of solution: We determine the correctness of the solution by examining the changed source code if the solution satisfies the change requirements. We use the scoring presented previously where we summarize the points each developer gathers for each of the four tasks. The score is added next to each of the participants for both treatments, with and without using coupled file changes.
- Time of task completion (t_s): This represents the time measured in minutes the developers spent to solve the maintenance tasks. Having a scenario where the developers only need to solve the tasks, the selection of the coupled files is not included in the total time for the tasks. It does not include the time needed to determine the relatedness of the coupled files for a specific task. The completion time could be automatically determined using a tool implementation or as part of an analysis procedure and does not represent part of the developer task solution. We use a screen capturing device to record the time that each participant spends solving each of the four tasks. We record the time needed for each task in both treatments.
- Time required to determine the relevance of the coupled files (t_r): This represents the time needed to determine the change context of each of the coupled files related to the tasks. Considering a worst-case scenario, the selection of the coupled files has to be performed by the developers and the time needs to be calculated for the group using coupled file change suggestions. In this case, the total time needed for each of the tasks is the sum ($t_r + t_s$) of the time needed to select the coupled files and the time to solve the task.

Given the task list, the coupled files list and the issue list, we record the time the developers need to go through the process of determining the change context of the coupled files we examine for a given task. We use three additional developers to measure the time required to determine the context of each of the coupled file changes related to the tasks.

8.2.17 Data Analysis Procedure

To be able to test our hypotheses, we need to analyze the usefulness of the coupled file changes and the usefulness of the attributes from the software repository. We perform the analysis using SPSS statistical software.

8.2.17.1 Usefulness of Coupled File Changes

The main part of the analysis is the investigation of the usefulness of the coupled changes. For this purpose we compare the scores of each task solution and the amount of time needed for solving the tasks in both groups: without using coupled file suggestions and with using of coupled file suggestions.

For the time needed for the solution, we only use the values for the accomplished tasks. This way we assure that the values for the unsolved tasks do not corrupt the overall values for the time needed to successfully solve the tasks.

Here we have two main scenarios. The first one includes only the time the developers need to solve the tasks. The second scenario also includes the time needed to select the coupled files set related to a specific maintenance task. We calculate the mean time for a particular task. Furthermore, we repeat the calculation for each participant on

the task. At last, we determine the grand mean as the average of all the means of the time values for each of the tasks determined by the participants, weighted by the sample size. In our case this is the number of coupled files.
Having k populations or tasks, the ith observation is t_{ri} which is the $j(i)$th coupled files set. We write $j(i)$ to indicate the group associated with the observation i. Let i vary from 1 to n, which is the total number of samples, in our case, these are the coupled files, j varies from 1 to k, the total number of tasks. There are a total of n observations with n_j observations in sample j: $n = n_1 + ... + n_k$
The grand mean of all observations is calculated using the formula:

$$\overline{t_r} = \sum_{j=1}^{k} \left(\frac{n_j}{n}\right) \overline{t_{rj}} \tag{8.1}$$

here, $\overline{t_r}$ is the average of the sample means, weighted by the sample size [HL11].

To determine the usefulness of coupled file changes, we test the overall difference in the correctness of solving the tasks using the two-tailed Mann-Whitney U test. It is used to test hypotheses where two samples from the same population have the same medians or that one of them has larger values, so we test the statistical significance of difference between two value sets.

Determining an appropriate significance threshold defines whether the null hypothesis must be rejected [Nac08]. If the p-value is small, the null hypothesis can be rejected meaning that the value sets are different. If the p-value is large, the values do not differ. Usually a 0.05-level of significance is used as threshold. The p-value is not enough

to determine the strength of the relationship between variables. For that purpose we report the effect size estimate [TT14].

We use a conservative approach where we test the difference in the correctness of our tasks. Without differentiating the tasks, we compare all the solutions of the tasks using coupled file changes and the tasks performed without any suggestion. We repeat the same approach to test the overall difference between the time needed to solve the tasks using coupled change suggestions against the tasks solved without the help of coupled file changes.

We use the SPSS statistical software and its typical output for the Mann-Whitney U Test whereby the p-value of the statistical significance in the difference between the two groups is reported. The mean ranking determines how each group scored in the test. To support statistical difference between the samples, we calculate the r-value of the effect size proposed in [Coh77] using the z value from the SPSS output [FMR12]. A value of 0.5 determines a large effect, 0.3 means medium effect and 0.1 identifies a small effect [CT13]. Given that we have a study which is restricted to a small number of comparisons, we do not adjust the p-value using a procedure like the Bonferroni correction [Arm14].

8.2.17.2 Usefulness of Attributes

We analyze the feedback from the questionnaire investigating which attributes are useful. We investigate every attribute in the set extracted from the versioning system, the issue tracking system and the documentation as previously presented. For that purpose we use the Kruskal-Wallis H test, an extension of the Mann-Whitney U test. Using this test, we determine if there are statistically significant differences

between the medians of more than two independent groups. We test the statistical significance between more than two value sets. The significance level determines if we can reject the null hypothesis. p-values bellow 0.05 mean that there is a significant difference between the groups [Poh14]. To determine the effect size for the Kruskal-Wallis H test, we calculate the effect sizes for the pairwise Mann-Whitney U tests for each of the attributes using the z statistic. We individually calculate the r-value for the effect size for each pair comparison. The r-value is calculated using the following formula:

$$r = \frac{z}{\sqrt{N}} \tag{8.2}$$

Our approach tests the differences in the feedback about the usefulness between all the attributes for all 36 participants. This way we identify which attributes we should offer to the participants when solving their tasks together with the coupled file change suggestions. Using SPSS, we provide the statistical significance values of the difference between all eight attributes. The ranking of the means for the feedback on the usefulness values determine the most useful attributes.

8.2.18 Execution Procedure

- *Log Extraction:* We extract the information from the Git log containing the committed file changes and the attributes. The log data is exported as a text file and the output is managed using proper log commands.
- *Data preprocessing:* After the text files with the log data have been generated, we continue with the preparation of the data for

mining. Various data mining frameworks use their own format, so the input for the data mining algorithm and framework needs to be adjusted.

- *Support threshold*: To be able to begin the investigation, we need to extract coupled file changes from the software repository. We extract the coupled changes by defining the threshold value of the support for the frequent item set algorithm. We use the thresholds that give us a frequent yet still manageable number of couplings. This threshold is normally defined by the user. We use the technique presented in [Fou13] to identify the support level. These values vary from developer to developer, so we test the highest possible value that delivers frequent item sets. If the support value does not yield any useful results for a particular developer, we drop the value of the threshold. We did not consider item sets with a support rate below 0.2 meaning the coupled changes should have been found in 20 percent of the commits.

- *Mining Framework*: There is a variety of commercial and open-source products offering data mining techniques and algorithms. For the analysis, we use an open-source framework specializing in mining frequent item sets and association rules called the *SPMF-Framework*.[1] It consists of a large collection of algorithms supported by appropriate documentation.

- *Experiment preparation*: We prepare the environment by setting up the source code and the Eclipse where the participants will work on the tasks. We define the maintenance tasks and provide the free text description. We prepare the coupled file changes

[1] http://www.philippe-fournier-viger.com/spmf

and the attributes from the software repository to be presented to the participants in the experiment.

- *Solving tasks*: The participants in both groups work for two hours in two labs and provide solutions for the maintenance tasks. The solution and the screen recording are saved for further analysis.
- *Material gathering*: We gather the questionnaires, the edited source codes and the video files of the participants screens for further analysis.
- *Solution analysis*: We analyze the scores for the correctness of the maintenance tasks, calculate the time needed for solving the tasks and determine the influence of the coupled file changes on the task solution.

8.3 Results and Discussion

The complete list of the maintenance tasks, the coupled file changes, the software repository attributes, the questionnaires and the analysis results can be found in the supplemental material of this paper.

8.3.1 Participants

The participants' feedback about their background reports that most of them have around one year of programming experience and less than 1 year experience with versioning and issue tracking systems. None of them answered to be in any way involved on the A-STPA project.

8.3.2 Issues Classification

Based on the proposed classification from [HGH08], we classified the issues from the issue tracking system related to commits in the Git version history as presented in Table 8.5. Here, we see that most changes of the system are corrective, implementation and perfective issues.

Table 8.5: Issue Classification

Issue Category	Frequency	%
Corrective	217	31.77
Implementation	169	24.74
Perfective	146	21.38
Adaptive	85	12.45
Other	66	9.66

Table 8.6: Perfective Issues

Change category	Frequency	%
Views	74	49,01
Control Structure	34	22,52
Menus	22	14,57
Non functional source changes	13	8,61

Further, we examined the perfective issues in more detail to determine to which parts of the system they are related. We have identified several classes of perfective issues related to the main functionalities of the system we investigated in the experiment as presented in Table 8.6.

The most frequent perfective issues are related to changes to the view elements of the system user interface responsible for the visualization of the hazard analysis steps including their layout, tables, grids, text fields, buttons, icons and labels. These changes have been organized

in the class called *Views*.
The next class called *Control Structure* is composed by the issues which handle the changes related to the control structure functionality of the user interface. It is responsible for drawing the diagrams, connects the layout components and includes changes on the diagram elements like objects, labels and connections.
The following class we call *Menus* is related to the issues associated to the user interface menus which are used to manipulate the creating and editing of project elements including changes in the wizards, the actions, labels and icons.
The last class includes the issues covering non-functional changes in the source code like cleanups, refactoring or formatting.
The task distribution in the experiment corresponds to this classification. We have defined two tasks for the application views, one task for the menus and one task for the control structure of the user interface.

Table 8.7: Statistical Significance (Coupled changes)

Depend. Variable	p-value	r-value
Correctness	0.000	0.448
Time Effort	0.041	0.259

8.3.3 Usefulness of Coupled File Changes

As we already explained, we operationalize the usefulness of coupled file changes by their influence on the correctness of the solutions and the time needed to solve the tasks.

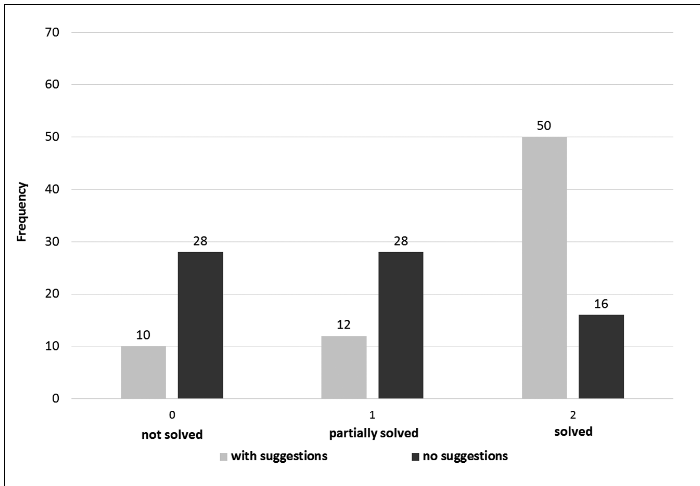

Figure 8.2: Task Correctness Distribution

8.3.3.1 Correctness

We summarize the correctness distribution as presented in Figure 8.2. On the y-axis we have the frequency of occurrence and on the x-axis the score of solving of the tasks. Here, the observations are grouped based on the presence of coupled change suggestions during the maintenance task solution. From this figure we see that the participants achieved better scores using the coupled file change suggestions we provided.

We investigate the correctness difference of both groups by testing the first null hypothesis of the first research question claiming that there is no significant difference in the correctness of the task solutions.

Applying the Mann-Whitney U Test results in a p-value of 0.000 as

presented in Table 8.7. This result has to be lower than the threshold of 0.05, so this null hypothesis can be rejected. This means that there is a statistically significant difference in the correctness of the solution for the provided tasks when using coupled file change suggestions against the correctness of the solutions only using the provided task description. The r-value of the effect size for the correctness is 0.448 which describes a strong statistical difference in the correctness of the maintenance task solutions between the groups that did or did not coupled change suggestions.

In Table 8.8, we represent the descriptive statistics for the correctness of the task solutions. The participants which used the suggestions solved 63.8% of the tasks completely, whereby the participants not using suggestions solved only 22% of the tasks. This shows an significantly higher score for the group using coupled change suggestions.

The median absolute deviation (MAD) value for the group using coupled changes is 0, whereby the value for the group not using coupled changes is 1. These values show that the correctness score is spread very close to the median for the score of the first group. The statistical results provide evidence that the coupled file changes

Table 8.8: Descriptive statistics for the correctness of the tasks

Without Suggestions (–)			With suggestions (+)		
Completely solved tasks	Median	MAD	Completely solved tasks	Median	MAD
22 %	1	1	63.8 %	2	0

significantly influenced the correctness of the maintenance tasks in the experiment. Inexperienced developers solved more tasks when using our suggestions which means they used the benefit of hints related to similar tasks. The coupled change suggestions allow the

developer to follow a set of files and remind him/her that similar tasks include changes in various locations in the source code.

The improvement in the number of solved tasks for the group using the coupled change suggestions shows that developers have used the benefits of additional help in locating the features and the files to be modified to solve their tasks successfully. The group that did not use this kind of help did not succeed to solve the same or a higher number of tasks which points to the usefulness of our approach. The use of

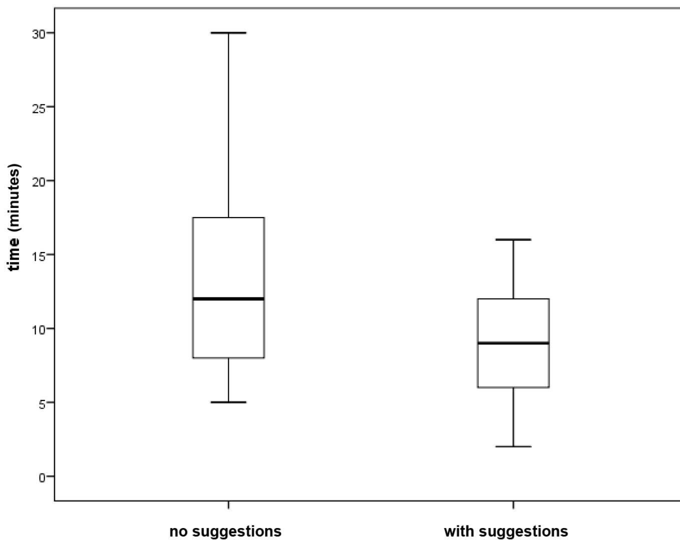

Figure 8.3: Time Boxplots (t_s)

coupled file changes has been especially noticed in cases where the developer needs to perform similar changes in several locations, like editing different views of the application GUI. Here, the developers not

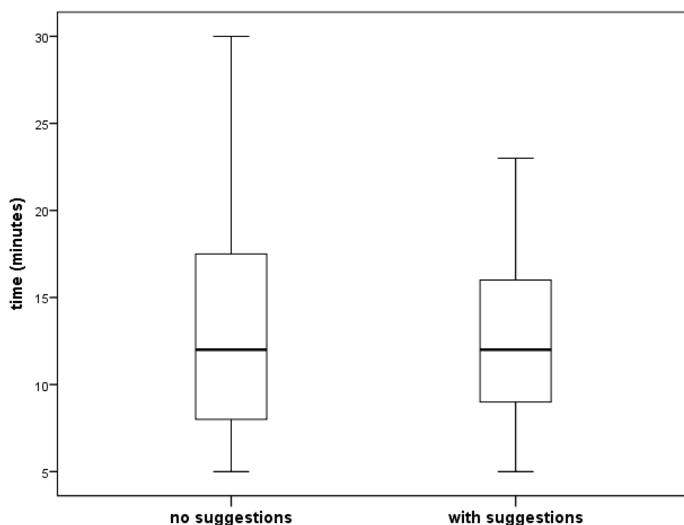

Figure 8.4: Time Boxplots ($t_r + t_s$)

using coupled change suggestions missed implementing the change in all the files where the change should have been performed. Coupled file change suggestions help the developers not to miss other source code locations they need for their task.

8.3.3.2 Time

We have analyzed the influence of using coupled file change suggestions on the time needed to successfully perform the tasks versus not using them. Many participants used split-screen and kept the documentation window open so we were not able to subtract the time

spent reading the documentation from the total amount needed to solve the tasks.

The distribution of the values for the time needed to solve the tasks is presented in Figure 8.3. We see that the distributions are similar with a slight tendency to more time needed to solve the tasks without suggestions.

We test the second null hypothesis which claims that there is no influence of the coupled file changes on the time needed to solve the tasks.

The distribution including the time to determine the relatedness of the coupled files is presented in Figure 8.4. Considering only the time needed to solve the tasks (t_s), the p-value for the two tailed test is 0.041. This value is slightly below the 0.05 threshold for the significance of the difference in the time needed to solve the tasks by the group using coupled file changes versus the group that didn't. Therefore, we reject the null hypothesis. The r-value for the time needed to solve the maintenance tasks is 0.259 which shows a relatively small statistical difference between the group that used coupled change suggestions and the group that didn't.

Considering the case we include the time to select the coupled files to the time needed to solve the tasks ($t_r + t_s$), we can see that there is almost no difference in the time measured for the group not using the coupled files and the group using coupled files. Here, the p-value for the total time is 0.987, which means that in this case the null hypothesis cannot not be rejected.

The r-value for the total time is 0.02, which emphasizes this small difference between using and not using coupled file change suggestions.

After calculating the grand mean for t_r, we added 3 more minutes

to the amount of time for the task solution and included it in the analysis of the difference between both groups regarding the use of coupled file change suggestions. The time needed to determine the related coupled files for the additional participants is presented in Table 8.9.

For the total time including the time needed to select the coupled files, we add the number of considered coupled files per task and the mean time the developers needed to select the coupled files for the particular task.

The descriptive statistics in Table 8.10 for the time needed to solve the tasks report a decrease in the means for the time needed to solve the tasks by 26% for the group using coupled change suggestions. The means ranking reports slightly better results for the group using coupled file changes, which means that the participants of this group solved their tasks slightly faster. The standard deviation for the group using coupled changes is twice lower than for the group not using coupled changes which shows a higher spread-out for the first group. Including the time needed to select the coupled files, the values are almost the same for both groups.

From the results, we can see that in this case, because of the additional time we added for each of the participants, there is almost no difference between the mean values which tells us that the group using coupled files did not manage to solve the tasks faster.

The results related to the task selection time show a small improvement for the time needed to solve the tasks. The developers using coupled change suggestions needed less time to find the files to be changed. Without coupled file changes, they would need to search for the features and files in the source code they need to edit.

The improvement in the time needed to solve the tasks for the group

Table 8.9: Time to determine related coupled files

	Time (minutes)			
Participant 1	Task 1	Task 2	Task 3	Task 4
Coupled Files 1	3	4	3	3
Coupled Files 2	5	3	2	2
Coupled Files 3	3	2	-	-
Participant 2	Task 1	Task 2	Task 3	Task 4
Coupled Files 1	2	2	2	3
Coupled Files 2	2	2	2	2
Coupled Files 3	2	2	-	-
Participant 3	Task 1	Task 2	Task 3	Task 4
Coupled Files 1	2	4	5	2
Coupled Files 2	2	4	2	2
Coupled Files 3	2	2	-	-
All Participants				
Mean (Coupled Files)	2.55	2.55	2.66	2.33
Grand Mean (Tasks)	3.41			

Table 8.10: Descriptive statistics for the time needed in minutes

	Median	Mean	Stand. Dev.
Without suggestions	12	13.50	7.403
With suggestions (t_s)	9	9.11	3.837
With suggestions ($t_r + t_s$)	12	12.33	4.158

using the coupled file changes is not as strong as the improvement in the correctness of the task solutions. It does not eliminate the time the developers need to understand the features and the changes they need to perform in the source code. They still need time to organize this information and use it. Furthermore, they need to read and understand the suggestions. Coupled file change suggestions do not automatically provide a solution for solving their tasks.

If we include time needed to select the coupled files, the results show that there is no improvement for the group using the coupled file change suggestions. If the coupled files need to be determined by the developers as a part of the task solution procedure, the small advantage for the groups using the suggestions disappears. An automated extraction of coupled file change suggestions including the determination of their relatedness could therefore be beneficial.

8.3.4 Usefulness of software repository attributes

Table 8.11: Descriptive Statistics (Attributes Usefulness)

Attribute	Median	MAD
Package Description	4	1
Issue Description	4	1
Commit Message	4	1
Issue Type	3	1
Commit ID	3	1
Commit Author	3	1
Issue Author	3	1
Commit Time	3	1

The distribution of the usefulness of each repository attribute is presented in Figure 8.5. The mean values for the usefulness of each of the repository attributes have been determined using the feedback

of all participants in the experiment.

We test the third null hypothesis which claims that there is no difference in the usefulness between the attributes using the p-value of the Kruskal-Wallis H Test. In our case, the p-value for this test is 0.000 which is lower than the 0.05 threshold. This result leads us to reject the null hypothesis. This means that the alternative hypothesis claiming that there is a significant difference in the perceived usefulness among the attributes from the software repository is true.

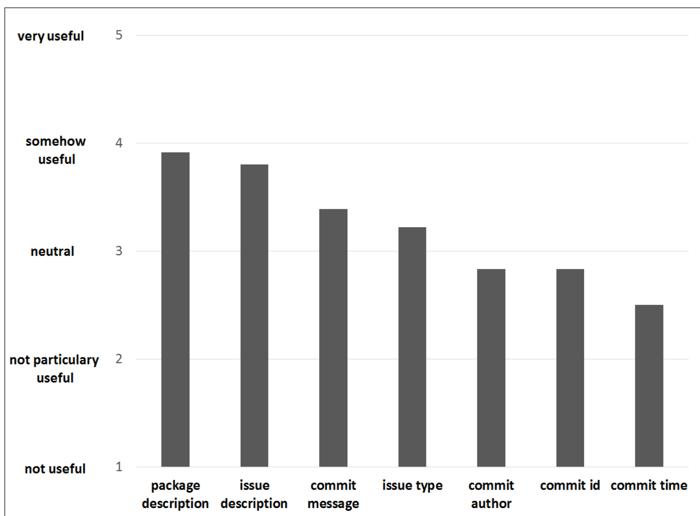

Figure 8.5: Usefulness of Attributes

We reported a set of various software attributes from the software repository. The participants reported their feedback on their usefulness at the end of the experiment lab after the tasks had been performed.

We gathered the descriptive statistics for the participants' feedback

Table 8.12: Statistical Significance (Coupled changes)

p-value	r-value	Repository Attribute pairs	
0.180	0.279	Commit ID	Commit Message
0.972	0.004	Commit ID	Commit Author
0.249	0.136	Commit ID	Commit Time
0.000	0.467	Commit ID	Issue Description
0.108	0.190	Commit ID	Issue Type
0.624	0.058	Commit ID	Issue Author
0.000	0.465	Commit ID	Package Description
0.022	0.270	Commit Message	Commit Author
0.001	0.400	Commit Message	Commit Time
0.048	0.233	Commit Message	Issue Description
0.582	0.065	Commit Message	Issue Type
0.004	0.336	Commit Message	Issue Author
0.220	0.269	Commit Message	Package Description
0.228	0.142	Commit Author	Commit Time
0.000	0.459	Commit Author	Issue Description
0.122	0.182	Commit Author	Issue Type
0.599	0.062	Commit Author	Issue Author
0.000	0.464	Commit Author	Package Description
0.000	0.566	Commit Time	Issue Description
0.008	0.311	Commit Time	Issue Type
0.476	0.084	Commit Time	Issue Author
0.000	0.557	Commit Time	Package Description
0.118	0.279	Issue Description	Issue Type
0.000	0.526	Issue Description	Issue Author
0.530	0.074	Issue Description	Package Description
0.039	0.244	Issue Type	Issue Author
0.009	0.308	Issue Type	Package Description
0.000	0.515	Issue Author	Package Description

on the usefulness of each attribute presented in Table 8.11. The median values vary from 3 for the commit ID, the commit author, the commit time, the issue author and the issue time, to 4 for the commit message and the package description. This places the cutoff between

“neutral” and “somewhat interesting” for most of the attributes. The MAD value for all attributes is 1, which shows a low spread out of the usefulness values around the median.

We calculated the r-value of the size effect for the repository attributes by creating pairs of each of the attributes where we determined the z-value of the Mann-Whitney test for each pair as presented in Table 8.12. We have 28 pairs of attributes.

The greatest difference in the usefulness is between the commit time and the issue description where the r-value is 0.566, followed by the difference between the commit time and the package description with an r-value of 0.557. This indicates a high statistical significance between these pairs of attributes. The lowest difference is between the commit ID and the commit author, here the r-value is 0.004, followed by the difference between the commit ID and the issue author with an r-value of 0.058. This shows that there are significant differences in the usefulness between individual attributes.

We determined that the attributes have different usefulness using the feedback of the participants. The median ranking defines which of the attributes are most useful. As the most useful attribute we identify the package description followed by the issue description and the commit message. This leads us to the conclusion that the inexperienced developers seek for help about the features of the source code they need to edit and the task they have to complete.

The issue type and the commit time are in the middle of the list. The most useless attribute is the commit author followed by the issue author and the commit id. Here, we suppose that the developers are not interested in the information who performed the changes because they do not know this person. This could change if the developers were included in the project for a longer time.

Although we produced a list of typical repository attributes, the participants have identified a smaller set of attributes to be useful for them than we provided in this experiment. This means that we don't have to present all the attributes to the developers together with the coupled files for the reason that different developers can happen to find some attributes as obsolete to be included in the coupled file change suggestions. An individual choice of useful attributes can avoid confusion and increase the acceptance of the coupled file change suggestions concept.

8.3.5 Threats to Validity

- *Internal Validity:* Potential internal validity threats can rise from the experiment design. To limit the learning effect, we use a counterbalanced design where every developer solves four different tasks where each of them solves two tasks without and two tasks using coupled change suggestions. This way the results are not directly influenced by the task supported by the coupled file suggestions.

  Other validity threats related to the experiment design are the selection of the coupled file changes, the creation of the maintenance tasks as well as their definition and solution.

  We extracted coupled files using a relatively high threshold which limits the possibility to provide suggestions for coupled changes that happened by chance.

  We selected the most frequent coupled files for each of the developers to avoid subjective interference. We also avoided delivering unrelated changes in order not to confuse the developer by providing suggestions out of the context.

The maintenance tasks were constructed manually. However, they are related to issues from the issue tracking system and fulfill the conditions set in the experiment to be perfective and related to changes of the user interface.

We classified the issues on the system based on the maintenance categories to show the representativeness of our maintenance tasks. The content includes a simple description of the functionalities and the required actions in order not to overwhelm the inexperienced developers by providing unnecessary information.

The set of files included in the solution of the tasks was provided by manually analyzing related issue solutions. We validated the task solutions using a third party.

The judgment of correctness of the developers' task solutions represents another internal threat whereby we test the solutions to determine the level of correctness.

The time needed to determine the relatedness of the coupled files can differ. To avoid an influence by particular tasks, we calculate the average time per coupled file set and calculate the grand mean for all tasks. We used independent student participants for the measurement of the time needed to select the related coupled files.

Also the metrics we used to determine the usefulness can represent a threat. The subjective usefulness rating represents another construct validity whereby we evaluate the provided task solutions pairwise to minimize the errors in conducting the score distribution. For the time needed to solve the tasks, we play the captured screens of the participants to calculate the

time the developers needed to solve the tasks.

- *External Validity*: The external validity threat concerns the generalization of the experiment. The main threats here are related to the choice of the coupled file changes, the type and description of the maintenance tasks as well as the participants and the system we investigate.

  We used a data mining technique that can be easily performed on other Git repositories to extract coupled file changes. Our approach uses mapping between the commits and the issues which excludes the projects not using them. However, this practice is used very often today. We can find many projects in various on-line software repository collections like GitHub using this kind of mapping and providing issue and project description.

  We chose simple perfective tasks that can be easily replicated and do not require large changes in the source code. The description of the tasks is simple and includes the source code functionalities to be changed and the activities without any specific format or structure. This way we maintain the possibility to repeat the process for other projects and limit the possibility of creating artificial conditions specially tailored for our experiment. Yet, it is not clear whether the results can be generalized for other types of maintenance tasks.

  The student participants in the experiment have basic programming experience which corresponds to the target group of our study to address inexperienced developers.

  The system we used for the experiment is an open source Java project with a clear project structure and repository. It does not

contain specific information that can challenge the replication of the analysis

8.4 Conclusion

In this experiment, we successfully tested the part of our theory on the use of coupled change suggestions related to their influence on the maintenance tasks solution.

Coupled file change suggestions are useful for inexperienced developers working on maintenance tasks. The influence is positive on the correctness level of the tasks solutions, meaning that it helps them to solve their tasks more successfully. The influence of the coupled change suggestions on the time effort for solving the tasks is not significant. We extended the findings of [RW16b] where the participants in their feedback reported the coupled file changes and the attributes as neutral to use in maintenance tasks. Our experiments outcomes are more positive compared to the results of [RW16b]. Working on real maintenance tasks using the tasks of the working software product increases the usefulness of coupled change suggestions by the developers. We rounded up the set of useful attributes based on the set presented in this study.

The next steps would be to transform the results and the findings in a tool implementation to support the developers working on maintenance tasks using visual presentation of suggestions which set of files they should also change. Also it would be interesting to follow the influence of coupled changes on the strategy of solving maintenance tasks.

Chapter 9

Coupled File Changes Influence on Help Seeking: An Exploratory Study

9.1 Introduction

Using this exploratory study, we test the part of our theory on the use of coupled file change suggestions related to their influence how developers seek for help during maintenance tasks.

We investigate the developers' information sources using the Grounded Theory method on video recordings on the developer's screen. We

explore which information sources are used by the developers and what kind of information both the group not using coupled change suggestions and the group using this kind of help seek for to solve the tasks. We present an exploratory study based on the data from a controlled experiment where each of the 36 participants try to solve 4 different maintenance tasks. The original experiment analysis is available in Chapter 8.

9.2 Experimental Design

9.2.1 Study Goal

We perform our exploratory study using a mixed-method approach on top of a maintenance tasks experiment where the impact of coupled change suggestions on the time and the effort needed to solve maintenance tasks has been investigated [RW16a]. We define the *goal* of the study using the GQM approach [BCR94] and its MEDEA extension [BMB02] which is analyzing the influence of coupled file change suggestions on where and why developers search for help. The *objective* is to compare the information sources the developers access for the tasks using coupled change suggestions and the tasks without using them. The *purpose* is to evaluate how effective are coupled file change suggestions related to the location of the task relevant information, the relevance of the information they look for, the frequency and the patterns of information sources.

9.2.2 Research Questions

RQ1: Where do developers look for task relevant information? We want to identify the sources for the information used for a particular

maintenance tasks solution. This will show what are the usual locations to seek for help during the tasks.

RQ2: What kind of relevance has the source of information for their task? The answer of this research question will show how information sources contribute to the tasks solution.

RQ3: How do coupled file changes influence the search for task relevant information during maintenance tasks with and without using coupled change suggestions? We investigate their influence related to the following subquestions:

RQ3.1: How frequently do developers use the sources of information? We identify the most popular sources for seeking help used by the developers in this study to identify common information sources.

RQ3.2: Are there any information sources patterns the developers use to find task relevant information? We explore there are some sequence patterns of information sources to identify the difference in the strategy how developers seek for help.

9.2.3 Overview

9.2.3.1 Experiment Design

We use a counterbalanced experiment design similar to the one presented by Ricca et al. [RLR+12] which ensures that all subjects work on tasks without and with coupled change suggestions. We split the subjects randomly in two lab sessions having a maximum of two hours to solve the tasks. We distinguish two groups of maintenance tasks:

the first one includes tasks executed in Eclipse IDE without using suggestions and the second scenario includes additional coupled files suggestions and corresponding attributes from the repositories for similar tasks.

In each session, the subjects work on two tasks without coupled suggestions using only the task description and on two tasks with the coupled file changes suggestions and the related attributes delivered together with the issue description. The participants in the second lab swap the order of the tasks used during the first lab.

Table 9.1: Experiment Design

Lab	Tasks	
Lab 1	Tasks 1-2 (without suggestions)	Tasks 3-4 (with suggestions)
Lab 2	Tasks 1-2 (with suggestions)	Tasks 3-4 (without suggestions)

9.2.3.2 Objects

The study object is A-STPA, an open source Eclipse based Java tool for hazard analysis built at the University of Stuttgart[1] in 2013. It has been chosen for the analysis because of the availability of the source code, the git repository, the complete list of the issues and the project documentation. The source code contains 16012 lines of code and 178 classes organized in 37 packages. The Git repository of the

[1] https://sourceforge.net/projects/astpa/

project contains 1106 commits from which we have extracted 205 coupled changes.

9.2.3.3 Subjects

The participants are 36 undergraduate students from the Introduction to Software Engineering course in their 1st and 2nd study year at the University of Stuttgart. The students have already passed the Java programming course and have basic Java and Eclipse knowledge.

9.2.3.4 Environment, Materials

The participants worked on a Windows PC with an Eclipse IDE. Their actions were recorded using a full screen video capturing tool. We have provided the source code, the technical documentation for the software system as pdf document including the data model and package descriptions, the instructions for the experiment and the maintenance tasks free-text description.

9.2.3.5 Tasks

The maintenance tasks represent program fixes needed to be performed by the participants according to the maintenance requests [Bas90]. All four maintenance tasks are perfective and enhance the software usability without influencing the system structure. The tasks are related to simple changes of the user interface of the system. The complete set of tasks and coupled changes is available on-line[1].

[1] https://peerj.com/preprints/2492/

9.2.3.6 Coupled Files

We have provided a set of files which changed together frequently to the group which uses coupled file change suggestions. These coupled files do not represent the solutions for a particular task in the experiment and usually contain a subset of the solution file set. We have joined to the coupled changes a set of attributes from the versioning system, the issue tracking system and the project documentation to build the suggestions. Using the related information about the context of the change, the developers can decide if the coupled files are suitable for their tasks.

9.2.3.7 Maintenance Activities

The maintenance tasks solving process includes the following activities: task understanding where the participants read the task description and the instructions, change specification where they locate the source code to be changed, change design where they perform the modification and change testing where the successfulness of the changes is tested.

9.2.3.8 Data Collection

We have collected the data about that how developers search for help using the full screen video capturing of their actions during the experiments including the mouse movement and the keyboard input they performed. We have recorded the screen including their actions when using the IDE they work with as well as the applications the

developers used or opened on the screen.

9.2.4 Data Analysis

We analyze the captured videos to follow the sources of information developers used during their search for help.

9.2.4.1 Information Sources

Before we use the Grounded Theory method, we define the focus involving the locations the developers use to search for help to avoid confusion during the coding. We transcribe the captured videos to isolate the locations as concepts. After this we start with the coding process to identify the information sources.

9.2.4.2 Relevance of the Information

The relevance of the information is important to maintain the information what is the purpose of the help search. We want to see how the task-relevant information contributes to the solution. We differentiate between resolving the source code location and the source code description.

9.2.4.3 Frequency of Using Information Sources

We explore how popular various information sources are. We analyze the frequency of use of a particular source of help to see what the developers find most useful as a task relevant information source.

9.2.4.4 Patterns of Information Sources

Sequential pattern mining is used to discover interesting subsequence in a set of sequences. One of the measures for their interestingness is the occurrence frequency of the ordered elements [FLKK17]. For this purpose we use the Prefix-Span algorithm, a fast and and memory efficient sequential pattern mining algorithms [PHM+04]. It uses the pattern-grown paradigm and outperforms fast algorithms like GSM or SPADE as well as the BIDE algorithm, for greater support levels [WH04].

Using sequential pattern mining, we explore common patterns in the information sources. For this purpose we use the Prefix-Span algorithm, one of the fastest sequential pattern mining algorithms [PHM+04]. We take all unique information sources for a particular task as items and order them in subsets whereby the items in a subset are not repeating. All subsets of tasks for a particular task build a transaction. Using all transactions we explore the most frequent patterns of information sources.

9.3 Results and Discussion[1]

9.3.1 Information Sources

9.3.1.1 Identified Sources of Information

We identified a number of task relevant information sources using open coding whereby the main concepts were coded using a set of open codes. We continued with the axial coding to relate the concepts

[1]The study results are also available at: http://dx.doi.org/10.5281/zenodo.291865.

and their categories as well as the relations among them. Further, using the theoretical coding, we established the relationships between categories and subcategories and delivered the information sources as concepts for the theory as presented in Figure 9.1. We identified a number of information sources categorized in two categories: internal locations including the project explorer, the source code and the search window in Eclipse and external locations like the task description, the documentation and the web search.

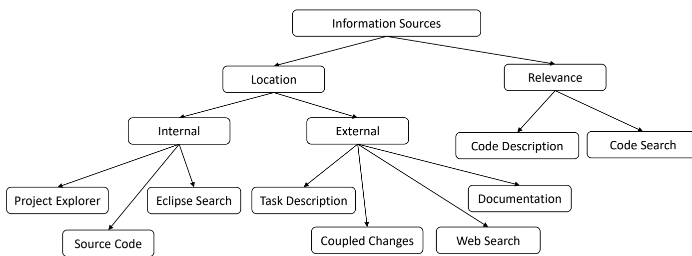

Figure 9.1: Information Sources

Table 9.2: Information Sources Frequency

Information Source	Source ID	No Suggestions		With Suggestions	
		Frequency	%	Frequency	%
Task description	1	65	100	67	100
Project explorer	2	65	100	67	100
Documentation	3	21	32	7	10.4
Eclipse search	4	26	40	28	41.8
Web search	5	11	17	1	1.5
Source code	6	65	100	67	100

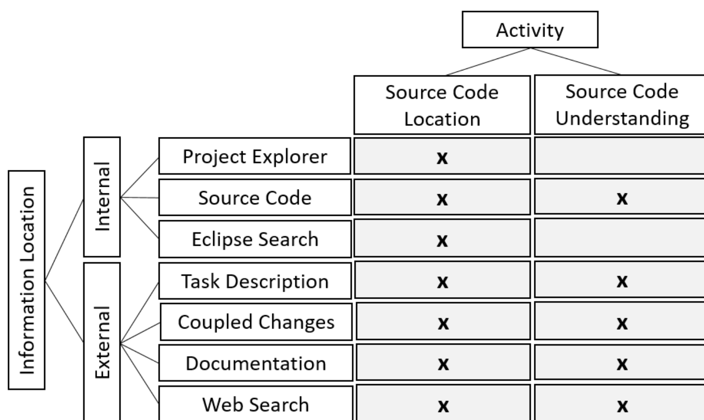

Figure 9.2: Information Sources Relevance

Table 9.3: information Source patterns

Support	Sequence patterns without suggestions (Source ID)
0.1	1 2, 1 2 4, 1 2 4 6, 1 2 3, 1 2 6, 1 3, 1 2 3 4, 1 2 3 4 6, 1 2 3 6, 1 2 5, 1 4, 1 4 6, 1 3, 1 3 4, 1 3 4 6, 1 6, 2 4, 2 3, 2 3 4, 2 3 4 6, 2 3 6, 2 5, 2 6, 2 4 6, 2 5 6, 3 6, 3 4 6, 4 6, 4 5, 5 6,
0.2	1 2, 1 2 6, 1 2 4, 1 2 4 6, 1 2 4, 1 6, 1 4, 1 4 6, 1 5, 2 4, 2 6, 2 4 6, 2 5, 3 6, 4 6
0.4	1 2, 1 2 6, 1 6, 1 4, 2 6

9.3.1.2 Relevance of Information Sources

Using the Grounded Theory method, we have defined two general categories of information relevance: source code location, where the developers try to locate the source code they have to edit and

Table 9.4: information Source patterns

Support	Sequence patterns using suggestions (Source ID)
0.1	1 2, 1 2 4, 1 2 4 6, 1 2 3, 1 2 6, 1 4, 1 4 6, 1 6, 2 6, 3 6, 4 6
0.2	1 2, 1 2 6, 1 4, 1 4 6, 1 6, 2 4, 2 6
0.4	1 2, 1 2 6, 1 6, 2 6

source code description where they try to find an information to understand the source code. We use a relevance matrix (Figure 9.2) to demonstrate the relation of the information sources and their relevance for the task solution similar to the maintenance matrix presented in [DWP+07]. We can see that the internal task relevant information sources like the project explorer and the Eclipse search are used to locate the source code location to be modified, except the source code windows which can be also used to find some comments or other source code parts to find a description or an additional information for the task.

The external sources of information like the documentation and the web search have been used in both cases: to find the source code location to be edited and to find additional information about it. The task description and the coupled changes are used to describe the tasks, which in the case of using coupled change suggestions can also be used to identify locations where similar code modifications need to be performed. This shows that the developers use the various information sources for their strategy, by excavating the search for additional information and the search for the source code to modify.

9.3.2 Influence of Coupled Change Suggestions

9.3.2.1 Frequency of Use of Information Sources

Table 9.2 presents how frequently each of the information sources was used both for the group using coupled file changes and the group without any additional help. We see that the task description, the project explorer and the source code has been used in all tasks in the experiment in both groups. The documentation has been used three times more for the tasks in the group not using coupled change suggestions. Eclipse search has been almost equally used in both groups. The web search has been used ten times more in the group without using coupled change suggestions versus the group not using them.

These results show that we have two different situations describing the influence of coupled change suggestions with respect to which task relevant information sources have been used during the maintenance tasks. The first one shows that coupled change suggestions do not influence the use of the internal information sources, both groups used almost equally the Eclipse tool properties. Also very frequently they used the task description. This means that coupled change suggestions do not affect how the developers use the IDE and do not reduce the need for the task description. Moreover, the tendency of the group without coupled change suggestions is to use more external help like the documentation or web search on pages like Google search, Stack Overflow, tutorials and videos.

The developers tried to find some help outside their workspace. For the tasks with coupled change suggestions, developers used mostly the internal source code locations and made very little use of the documentation and almost no use of web search for their tasks. This

indicates that using coupled change suggestions, the developers do not spread out their search for help, meaning the use of coupled change suggestions reduces the need to search for external source code location.

9.3.2.2 Patterns of Information Sources

We have extended our investigation of the used task relevant information sources by looking for possible patterns of information sources using coupled change suggestions vs. not using coupled change suggestions. The results in Table 9.3 and Table 9.4 show the pattern sequences for both groups and for various levels of frequency of occurrence of these patterns. Here, the source code locations are represented by their IDs as described in Table 9.2. We can see that for the highest support value of 0.4 including the patterns repeating in 40% of the tasks, a typical pattern is starting with the task description, continuing with the project explorer and Eclipse search and ending with the source code window.

For the lower support values including 20% and 10% of the tasks respectively, we have patterns in the group not using coupled change suggestions where they start with the task description, look up in the project explorer and jump for the web search or use the documentation before they edit the code in the source code window.

In summary, these results show that the most frequent patterns are similar for both of the groups. In some cases their strategy for looking for help during their maintenance tasks varies. Some developers not using coupled change suggestions extend their pattern of help seeking strategy with the external sources before accessing the source code they need to edit. This shows that without the coupled change

suggestions, the developers need more sources and have a different strategy in help seeking than the group which uses coupled change suggestions which concentrates on the IDE elements to accomplish the source code modification.

9.4 Threats to Validity

The main internal validity threat is that most of the analysis in this study is based on the subjective actions by the researchers. Transcribing the videos and the coding process can be error prone whereby the researchers can drop some actions by the developers. For that reason we include a third party in the transcribing of the videos for a random set of videos.

The work in a controlled experiment using inexperienced 1st and 2nd year students instead of a real development process in company represents a major threat to the external validity. Maybe, developers with more experience show other patterns of help seeking. Especially developers familiar with a system might show a different behavior. We have designed simple maintenance tasks to increase the generalization possibility. We used an open source project for the study and a well-known data mining technique which can be performed on other repositories.

9.5 Conclusion

We have successfully tested our theory on the use of coupled file change suggestions in the part related to the propositions about the influence of the suggestions on the help seeking process during maintenance. The use of coupled file change suggestions influenced the

strategy of inexperienced developers seeking for help using task relevant information sources by reducing the need of different information sources.

Our study shows that developers using coupled change suggestions mostly concentrate on the IDE and use its features and windows like the project explorer, the source code window and the Eclipse search. Without using this kind of help, developers accessed external information sources like the product documentation or used search on websites for code examples or code descriptions to find the needed information.

The main effect of coupled change suggestions is that it makes the process of help seeking shorter and compacter by reducing the need for additional sources of information related to the concept location and the source code comprehension. They influence the strategy of using various information sources and help the developers save time and effort for solving their maintenance tasks.

Chapter 10

Mining System Packages for Developer Expertise: An Exploratory Study

10.1 Introduction

The contribution of developers to a project consists of the combination of their actions performed during software development [GKS08]. This kind of expertise is called *implementation expertise* [SZ08]. As the development team grows, it becomes more complicated to allocate the developers according to their expertise considering their contribution

to the project. One of the approaches defines that the developers who changed a file are considered to have the expertise for that file [SZ08]. This information can be found by mining changes in the software repositories.

In Java projects, packages are important to group source code based on its functionality. By scanning the changes in the versioning system, we can extract the packages of files being changed. For the reason the package structure is more stable and do not often change, we use the packages instead of classes or methods to define developer expertise profiles. Sets of files changed together frequently are called *coupled file changes* and can be offered to the developers as recommendations when solving some maintenance task. We use this technique to extract the packages which changed most frequently together to define developer profiles. The profiles can be used to identify who worked on similar issues on the project.

Developers working on maintenance task could use the information who worked on which part of the source code. Having many developers on the project, it could be awkward to identify the experts working on topics related for their tasks.

The aim of our study is to identify the developer's expertise profile based on the project package structure. We use the package organization of the source code because it reflects the grouping of the system functionalities and defines the fundamental features or layers of the system. Using data mining we investigate the software repositories to extract the sets of packages that were most frequently changed together by a given developer during software development. We present an exploratory case study where we define developer profiles of expertise based on the aggregated information about the system packages that were most frequently changed together. By differenti-

ating developer profiles, the effort for the analysis drops because the users involved in maintenance do not have to examine the data from all developers on the project. They can choose the data from those developers who implemented changes in the system relevant to their tasks.

10.2 Case Study Design

10.2.1 Research Questions

RQ1:Which package couplings are most frequent per developers? We examine which packages are most frequently changed together by particular developer. We use this information to investigate the functionalities the developers were mostly involved into during the software development.

RQ2: What kind of developer profiles can we define based on the packages? Based on the changed packages and their functionalities, we aggregate them to define their profiles related to their expertise on the system software. Using this information developers can explicitly identify the software changes related to their task.

10.2.2 Case Selection

For our study, we use three open source projects: *ASTPA*, an Eclipse based Java project, *RIOT*, Java and Android based software and *VITA*, Java based text analysis software. They were all developed at the University of Stuttgart and were found on the local GitLab. The projects have been selected based on the intensive use of packages and their availability for analysis.

10.2.3 Data Collection Procedure

We extract the Git log from the project repositories and format the output to separate the commits according to the developer who has done the changes. We enlist the commits with all changed files represented by the file names containing the relative file paths including the packages and sub-packages of the project. To prepare the data for analysis, we remove empty or commits having single entry. We group the commits based on the developer heuristics. We create data sets of commits for every developer who contributed to the repository.

10.2.4 Analysis Procedure

- *Extracting the packages from the commits:* We extract all names of the files changed in a commit. They include the relative file path on the system which includes the names of the package and sub-packages. We scan the file paths from the back and remove the filenames from the path. The resulting string identifies the name of the package or sub package.
- *Mining packages:* We perform a frequent item sets analysis on the packages to extract the most frequent couplings from the repository. Due to the high number of file couplings, we use a relatively high user-defined support level for the frequent item sets analysis. This way we include only the coupled packages which happened frequently and not by chance.
- *Define developer profiles:* We rank all the coupled packages for a developer starting with the most frequently changed package to identify the most frequent ones. After the ranking of the packages, we join the group of most frequent packages to the

developer. This will identify the expertise of the developer marking the functionalities he or she was most contributed. We look up the features that are involved in the files behind this packages. We aggregate developers expertise profile based on the most frequently changed packages.

Table 10.1: Most frequent packages in the couplings

ASTPA		
	packages	profiles
Dev.1	astpa.src.astpa. controller.editParts	control structure
Dev.2	astpa.src.astpa.controlstructure astpa.src.astpa.controlstructure. controller.editParts astpa.src.astpa.controlstructure.figure	control structure
Dev.3	astpa.src.astpa.controlstructure. controller.editParts astpa.src.astpa.model.interfaces astpa.src.astpa.ui.common.grid	control structure+ user inter-face+model
Dev.4	astpa.src.astpa.model.interfaces	user inter-face+model
Dev.5	astpa.astpa.intro.graphics.icons	graphics
Dev.6	astpa.src.astpa.ui.sds, astpa.src.astpa.ui.acchaz	user interface
Dev.7	astpa.src.astpa.ui.common.grid	user interface
Dev.8	astpa.icons.buttons.systemdescr	graphics

Table 10.2: Most frequent packages in the couplings

RIOT		
	packages	profiles
Dev1	android.riot.res.layout, android.riot.res.drawable-xhdpi android.riot.res.drawable-mdpi, android.riot.res.drawable-hdpi, android.riot	android layout
Dev2	android.res.layout android.res.layout android.src.main.java.de.uni_stuttgart. riot.android.management	android layout
Dev3	commons.src.main.java.de.uni_stuttgart. riot.server.commons.db	database
Dev4	android.riot, android.riot.settings, android.riot.res.drawable-hdpi android.riot.res.drawable-mdpi, android.riot.res.drawable-xhdpi android.riot.res.drawable-xxhdpi, android.riot.res.layout android.riot.res.menu, android.riot.res.values-de	android layout
Dev5	android.src.main.java.de.uni_stuttgart. riot.android.account	android account
Dev6	android.src.main.java.de. uni_stuttgart.riot.android.idea.libraries .idea.libraries	android libraries
Dev7	usermanagement.src.main.java.de. uni_stuttgart. riot. usermanagement.data.sqlQueryDao.impl	database
Dev8	commons.src.main.java.de. uni_stuttgart. riot.thing	riot things
Dev9	webapp	webapps
Dev10	webapp	webapps

10.3 Results and Discussion[1]

We have extracted the most frequent package couplings for the developers in all three projects. We use average support level for the data mining of 80 % for the first project, 60% for the second and 40% for the third project. Our results show that the number of different coupled packages vary from developer to developer and from project to project. These values are mainly influenced by the number of functionalities the developers have been involved into.

Table 10.3: Most frequent packages in the couplings

VITA		
	packages	profiles
Dev1	src.main.resources.gate_home. plugins.annie.resources.gazetteer	ANN plugins
Dev2	src.main.java.de.unistuttgart.vis. vita.services	services
Dev3	src.main.front-end.app.partials	frontend
Dev4	src.main.java.de.unistuttgart.vis. vita.analysis	analysis
Dev5	src.main.java.de.unistuttgart.vis. vita.importer.epub	importer
Dev6	src.main.java.de.unistuttgart.vis. vita.importer.epub	importer
Dev7	src.main.front-end.app.partials	frontend

10.3.1 Most frequent package couplings per developer (RQ1)

For every developer, we have extracted a list of package couplings covering various software features. The first developer on the *ASTPA*

[1] The complete list of all couplings and profiles are available at: http://dx.doi.org/10.5281/zenodo.51302.

project changed mostly files in the *controller.editParts* sub-package, whereby another one mostly changed functionalities in two *ui* packages (Table 10.1).

The first developer on the *RIOT* project, worked in several packages with very similar functionalities on the android layout (Table 10.2).

The enlisted developers on the *VITA* project worked on the services and the front-end (Table 10.3). The results show that some of them changed functionalities in files that belong to the same package, whereby others worked on different packages. Some developers share the packages meaning that two or more developers worked on the same packages. In other cases, the developers split their work and contributed into totally different packages.

10.3.2 Developer profiles (RQ2)

Based on most frequent package couplings we have identified a number of developer profiles. For the first project we identify profiles of developers working on the *control structure, user interface* and *graphics*. For the second project we define profiles for developers working on the *android layout, database, account, libraries, android functionalities* and *web apps*. For the developers in the third projects we define profiles including involvement in the *plug-ins, services, front end, analysis* and *import features*. The difference in the profiles is directly influenced by the parts of the code the developers were working on and the project organization. Instead of generating coupled file changes for every developer, the user working on the web application in the RIOT project can narrow the analysis on the data of the two last developers in Table 10.2.

10.3.3 Discussion

Using the most frequently changed packages, various functionality topics in the source code structure have appeared. For some users we have limited set of changed system functionalities which gives clear information what were they mainly working on. This way, the profiles related to a low number of functionalities show a more precise developer expertise. This is more useful to define the developers' expertise profile. The users can easily identify he or she covers changes related to his tasks.

Other package changes show developers with various topics in the systems. They show more general expertise working on different areas of the system. Their profile does not clearly represent a clear picture of their changes which could identify them as experts for a specific functionality. This information does not appear to represent a high value for an automatized expertise profile analysis.

10.4 Threats to Validity

A construct threat could be that developers create packages and influence the organization of the code. The names of the packages could lead to a group of classes which are not related or similar. We look up in the classes and other files in the packages to inspect manually if they are related to the package name.

As internal threat we can mention the relative high support level for the data mining algorithms provides relative small number of package couplings. However, this ensures that these couplings happened frequently and not by chance.

The generalization of our approach represents an external threat

because it is limited and we investigate a small number of Java projects. However, it is possible to implement on projects having clear package structure and having many developers working on it.

10.5 Conclusion

We conclude that we can successfully define developer profiles based on the packages most frequently changed together. There are developers working on similar system functionalities which can lead to developer profiles can be useful for the users because. This can limit the number of data sets for data mining, which decreases the time efforts for the analysis and can be very helpful in projects with large number of developers.

However, for the developers working on various or totally different parts of the code, the profiles do not show a clear expertise. The resulting profiles differentiate a number of functionalities which can help the users working on their maintenance tasks to choose the most relevant implementation.

The next steps would be to formalize the automation of the expertise analysis process of the profile description according to the package structure of the system.

CHAPTER 11

TOOL SUPPORT

Although many tools exist for extracting logical couplings from version histories, the fact we work with Git and we use various data sources lead us to the decision to design and implement our own tool.

11.1 Concept and Design

To implement our coupled file change suggestions, we have designed a Java based software tool which integrates the data mining process in the Eclipse IDE when working on maintenance tasks [1].

Based on the coupled files concept presented in this thesis, we provided the requirements and the design of the tool. It was then implemented and optimized as part of five student theses [Ala16; Cic15; Dem15; Kau17; Leh15].

[1] SRM Tool, http://www.mining-repos.com

The tool architecture is presented in Figure 11.1. The software tool for extracting file change suggestion is called *SRM (Software Repository Miner)* and involves of three main components: *ATSR (Automatic Transformation of Software Repositories)* plugin, *FPGA (Frequent Pattern Growth Algorithm)* plugin and *SRMP (Software Repository Mining)* plugin.

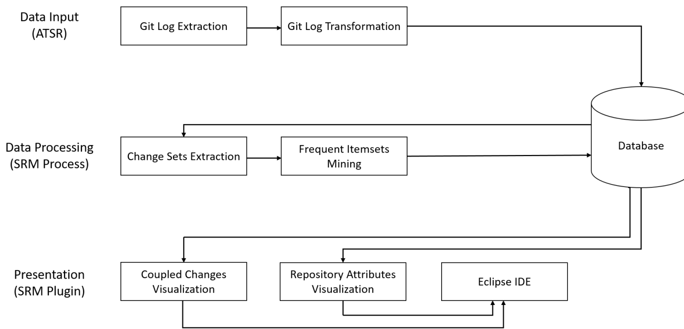

Figure 11.1: Tool Design

11.1.1 Components

- *ATSR*: This component implements the data extraction including the file change sets and the commit attributes from Git as well as the issue and documentation attributes. It also includes the preparation of the data and their storage in the database.
- *FPGA*: It performs the frequent sets mining performed on the change sets in the database. This component includes the data mining algorithm implementation based on the SPMF framework[VGG+14] whereby, instead of using text files, the input and output is performed using a database.

- *SRMP*: It selects the coupled files and the repository attributes from the database and visualizes them in the IDE views in Eclipse.

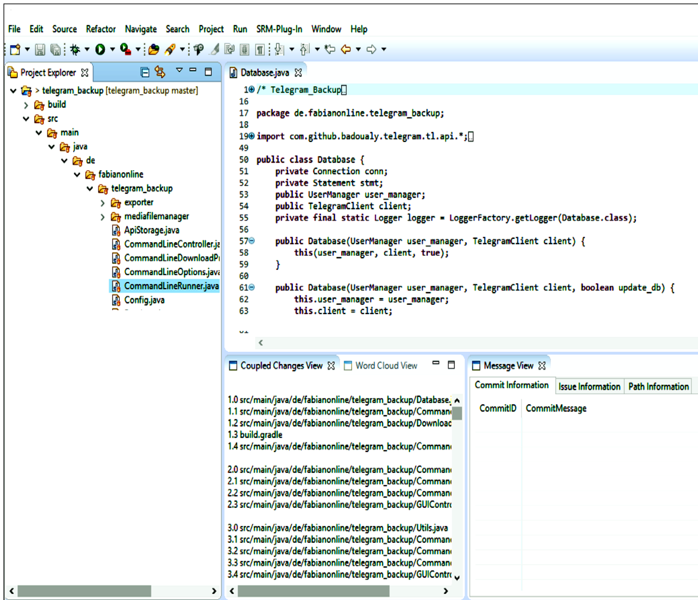

Figure 11.2: Eclipse IDE

11.2 User Interface

The user interface views are presented in Figure 11.2. The main parts of the user interface are: the button for starting the view perspective, the wizard for selecting the repository and the developers and perform the mining process, as well as the views to visualize the coupled files

and the repository attributes.

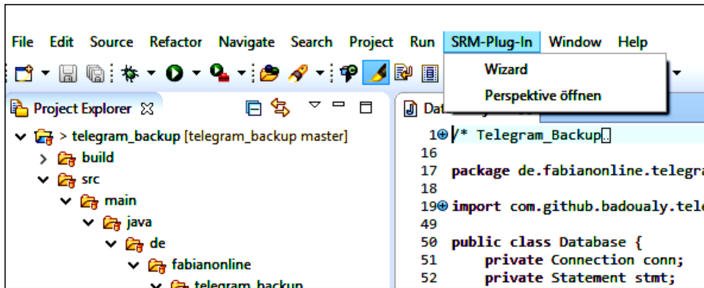

Figure 11.3: Tool Start

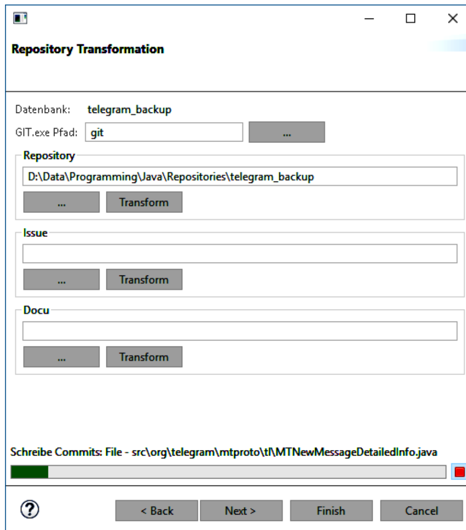

Figure 11.4: Wizard Sources

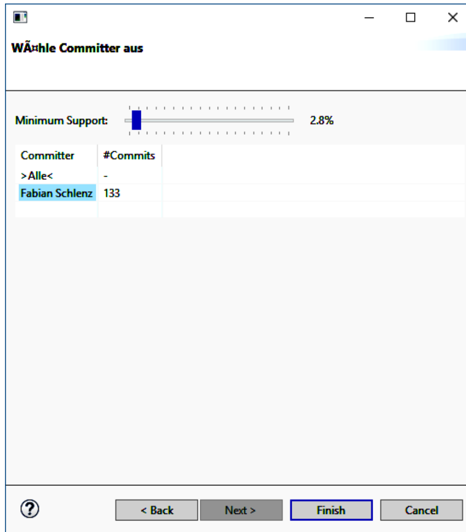

Figure 11.5: Wizard Developers

11.2.1 Activation

To activate the Eclipse plugin and start the mining process for a particular project we have incorporated a button on the Eclipse menu which enables the views for the coupled files and the repository attributes (Figure 11.3).

11.2.2 Wizard

The wizard contains two main steps and enables the user to select the data sources and the developers' data for the mining process.

- *Repository Choice*: The first part of the wizard (Figure 11.4) includes the feature which enables us to select the Git reposi-

tory to be analyzed. Additionally we have the issue and docu import functionalities in this wizard for the repository attributes extraction.

- *Developers and Support Choice*: After selecting the repository, the second step in the wizard (Figure 11.5) includes the choice of the developers whose commits we want to use as change sets for mining coupled files. We can select the complete repository to be used or to select a particular developer to use his or her data for the analysis according to the developer heuristic. Also the frequency of the coupled changes can be set using the slider for the support threshold.

11.2.3 Views

For the visualization of coupled changes and repository attributes, we have created three views in the Eclipse IDE: *coupled changes view*, *word cloud view* and *message view*.

- *Coupled Changes View*: This view (Figure 11.6) enlists the couplings for a particular file selected in the project explorer and included in a set of coupled files.
- *Message View*: This view (Figure 11.7) represents a tabbed representation of three sub-views where the developer can read three type of attributes for the selected file coupling in the coupled file changes view. The first tab enlists the commit attributes, the second the attributes for the related issue and the third one is displaying the description of the files in the couplings.

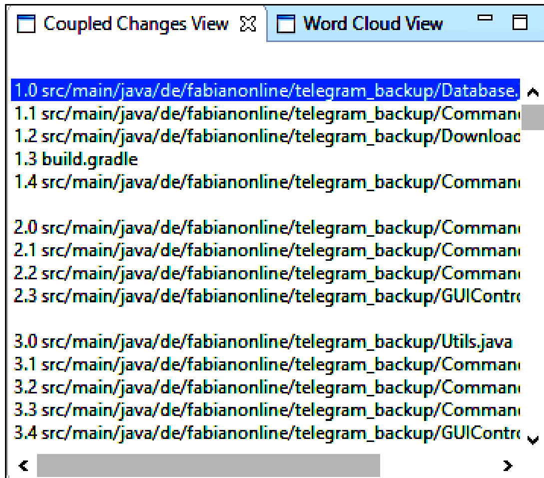

Figure 11.6: View Coupling Lists

11.2.4 Usage

The tool can be used to investigate the version history of a software when working on maintenance tasks in Eclipse. The user locates the

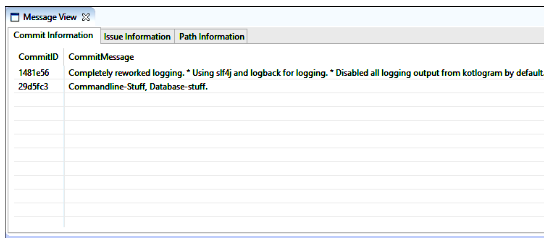

Figure 11.7: Message View

Git repository and extracts the coupled changes and the repository attributes. Working on an issue, the user selects the file to be modified in the project explorer of Eclipse, the software checks the database for potential coupled file changes including this file.

If the developer selects a file being part of coupled file changes,the views automatically display the coupled files, the commits where this file was found, information about the issues and the files included in the couplings. The developers can use the couplings in order not to miss other files needed to be modified and inform themselves about the background of previous changes on similar issues or tasks.

CHAPTER 12

CONCLUSION

12.1 Summary

We summarize the main contributions of this thesis to the concept of coupled file change suggestions.

We performed an industrial case study on the interestingness of coupled file changes where using data mining, precisely using frequent itemsets analysis, we extracted coupled file changes from Git version repositories. Additionally, we provided a set of repository attributes to build coupled file change suggestions for developers working on maintenance tasks. The developers reported in their feedback the concept of coupled files and the attributes to be interesting. They also provided additional issues to be included like the context and the visualization of coupled file change suggestions.

The outcomes of the case study expose that the implemented min-

ing technique is suitable to extract coupled files from Git. Based on this case study and a set of studies related to maintenance tasks and activities, we provided a set of constructs and their relations to define a theory on the use of coupled change suggestions during maintenance tasks. We managed to test the theory on the use coupled file change suggestions during maintenance tasks by performing empirical studies.

Using a controlled experiment, we tested the part of the theory related to the grouping of changes sets in Git. We have determined the heuristics for grouping related change sets from Git repositories which was not examined previously. Grouping the commits by their author allows extracting coupled file change sets even for projects not having many developers or commits. The heuristic based on the time between the commits reported that most of the related changesets have been committed in a short time of period between them, usually in a day. This heuristic, however, significantly reduces the number of change sets and the frequency of possible file couplings. The experiment results reveal that the proposed heuristics significantly influence the relatedness of change sets in Git and the extraction of relevant coupled files.

In a quasi-experiment, we tested the part of the theory related to the influence of the coupled files on the maintenance task solution. We reported that coupled file changes have positive influence on the correctness of the task solution has been found to be positive. The outcomes reveal that the coupled file change suggestions to be useful. However, they do not provide magical solution for the tasks and do not substitute the effort of the developers to search for the source code they need to modify.

We also tested the part of the theory related to the influence on the

strategies of the developers for searching for help during maintenance using an exploratory study. Here, the developers using coupled file change suggestions, concentrated on IDE internal task relevant information sources, whereby the developers not using these suggestions accessed more often the documentation and searched on the web for help. The study results show that the use of coupled files influences the developers' strategy for searching help during maintenance tasks.

The concept of coupled file changes was successfully extended on a package level using an exploratory study. Using logical dependencies between system packages, based on their frequency, we aggregated developer profile showing their expertise which reduces the number of developers data to be analyzed for particular tasks. The results of this study reveal that the concept of coupled files can be useful to help developers to recognize whose data they would like to analyze for their tasks.

We have implemented the concept of coupled file change suggestions in an Eclipse based tool to perform the data mining process and visualize the couplings and the additional repository attributes. The tool successfully extracts coupled files from various Git repositories.

12.2 Next Steps

The data mining methodology for extracting coupled files from Git version histories, has been reported to be useful. However, the next steps in the research could include larger data sets in the scope of big data. This can provide additional insights in the dependencies between different artifacts besides the source code files. Providing suggestions based on logical couplings which are not limited to the repository of the specific project, we can spread out the analysis on

other project repositories.

Recommendations about source code modification based on classes of commits or issues as well as on aspects like common libraries, APIs or package structures, could provide useful information. These recommendations can be used to help developers to increase the quality of the development and maintenance of their software products.

Bibliography

[ABCO98] B. Appleton, S. Berczuk, R. Cabrera, R. Orenstein. “Streamed lines: Branching patterns for parallel software development.” In: *In Proceedings of PloP*. 1998 (cit. on pp. 87, 90).

[ADG08] O. Alonso, P. T. Devanbu, M. Gertz. “Expertise Identification and Visualization from CVS.” In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. MSR ’08. 2008, pp. 125–128 (cit. on pp. 51, 63).

[AIS93] R. Agrawal, T. Imielinski, A. N. Swami. “Mining Association Rules between Sets of Items in Large Databases.” In: *SIGMOD*. 1993, pp. 207–216 (cit. on p. 39).

[AKM08] A. Alali, H. Kagdi, J. I. Maletic. “What’s a Typical Commit? A Characterization of Open Source Software Repositories.” In: *International Conference on Program Comprehension* (2008), pp. 182–191 (cit. on p. 54).

[Ala16] D. Alakus. “Integration of Data Mining in Eclipse Plugin.” Bachelor Thesis. University of Stuttgart, July 2016 (cit. on pp. 25, 219).

[AM07] J. Anvik, G. C. Murphy. “Determining Implementation Expertise from Bug Reports.” In: *Proceedings of the Fourth International*

Workshop on Mining Software Repositories. MSR '07. 2007 (cit. on p. 63).

[ANT92] D. A. Adams, R. R. Nelson, P. A. Todd. "Perceived Usefulness, Ease of Use, and Usage of Information Technology: A Replication." In: *MIS Q.* 16.2 (June 1992), pp. 227–247 (cit. on pp. 46, 85).

[Arm14] R. A. Armstrong. "When to use the Bonferroni correction." In: *Ophthalmic and Physiological Optics* 34.5 (2014), pp. 502–508 (cit. on p. 172).

[AS94] R. Agrawal, R. Srikant. "Fast Algorithms for Mining Association Rules in Large Databases." In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. 1994, pp. 487–499 (cit. on pp. 40, 131).

[Bas90] V. R. Basili. "Viewing Maintenance As Reuse-Oriented Software Development." In: *IEEE Softw.* 7.1 (Jan. 1990), pp. 19–25 (cit. on pp. 159, 197).

[BAY03] J. Bieman, A. Andrews, H. Yang. "Understanding change-proneness in OO software through visualization." In: *Program Comprehension, 2003. 11th IEEE International Workshop on*. 2003, pp. 44–53 (cit. on pp. 53, 54).

[BBC+96] V. Basili, L. Briand, S. Condon, Y.-M. Kim, W. L. Melo, J. D. Valett. "Understanding and Predicting the Process of Software Maintenance Release." In: *Proceedings of the 18th International Conference on Software Engineering*. ICSE '96. 1996, pp. 464–474 (cit. on p. 50).

[BCR94] V. R. Basili, G. Caldiera, H. D. Rombach. *The Goal Question Metric Approach*. Wiley, 1994 (cit. on pp. 103, 153, 194).

[BDO+13] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, A. De Lucia. "An Empirical Study on the Developers Perception of Software Coupling." In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE. 2013, pp. 692–701 (cit. on pp. 22, 57, 151).

[Bec09] G. Becker. *Human Capital: A Theoretical and Empirical Analysis, with Special Reference to Education*. Midway reprint. University of Chicago Press, 2009 (cit. on p. 21).

[BG10] J. G. Burch, F. h. Grupe. *A Systems Approach to Software Maintenance*. 2010 (cit. on p. 49).

[Bit05] V. Bitsch. "Qualitative Research: A Grounded Theory Example and Evaluation Criteria." In: *Journal of Agribusiness* 23.1 (2005) (cit. on p. 133).

[BKPS97] T. Ball, J.-m. Kim, A. A. Porter, H. P. Siy. *If Your Version Control System Could Talk...* 1997 (cit. on pp. 20, 31, 53).

[BLX+15] L. Bao, J. Li, Z. Xing, X. Wang, B. Zhou. "Reverse engineering time-series interaction data from screen-captured videos." In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 2015, pp. 399–408 (cit. on p. 58).

[BMB02] L. Briand, S. Morasca, V. Basili. "An operational process for goal-driven definition of measures." In: *IEEE Transactions on Software Engineering* 28 (12 2002), pp. 1106–1125 (cit. on pp. 103, 153, 194).

[BNF14] P. Bhattacharya, I. Neamtiu, M. Faloutsos. "Determining Developers' Expertise and Role: A Graph Hierarchy-Based Approach." In: *2014 IEEE International Conference on Software Maintenance and Evolution*. 2014, pp. 11–20 (cit. on pp. 51, 63).

[BRB+09] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, P. T. Devanbu. "The promises and perils of mining git." In: *MSR*. 2009, pp. 1–10 (cit. on p. 54).

[BSL99] V. R. Basili, F. Shull, F. Lanubile. "Building Knowledge Through Families of Experiments." In: *IEEE Trans. Softw. Eng.* 25.4 (July 1999), pp. 456–473 (cit. on p. 87).

[BSS13] E. di Bella, A. Sillitti, G. Succi. "A multivariate classification of open source developers." In: *Information Sciences* 221 (2013), pp. 72–83 (cit. on p. 62).

[Car13] E. Carlsson. *Mining Git Repositories : An introduction to repository mining*. 2013 (cit. on p. 54).

[CC05] G. Canfora, L. Cerulo. "Impact analysis by mining software and change request repositories." In: *Software Metrics, 2005. 11th IEEE International Symposium*. 2005, pp. 9–29 (cit. on p. 54).

[Cha08] T. Chan. "Impact of programming and application-specific knowledge on maintenance effort:A hazard rate model." In: *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*. 2008, pp. 47–56 (cit. on pp. 21, 168).

[Cic15] F. Cicek. "Presentation of Software Repository Mining in Eclipse." Bachelor Thesis. University of Stuttgart, Apr. 2015 (cit. on pp. 25, 219).

[ČM03] D. Čubranić, G. C. Murphy. "Hipikat: Recommending Pertinent Software Development Artifacts." In: *Proceedings of the 25th International Conference on Software Engineering*. 2003, pp. 408–418 (cit. on p. 65).

[Coh77] J. Cohen. In: *Statistical Power Analysis for the Behavioral Sciences*. Revised Edition. Academic Press, 1977, pp. 469–474 (cit. on p. 172).

[CPD05] D. Currie, C. I. of Personnel, Development. *Developing and Applying Study Skills: Writing Assignments, Dissertations and Management Reports*. Cipd Publications. Chartered Institute of Personnel and Development, 2005 (cit. on p. 47).

[CT13] H. Coolican, F. Taylor. *Research methods and statistics in psychology*. Routledge, 2013 (cit. on p. 172).

[CZD11] B. Cornelissen, A. Zaidman, A. van Deursen. "A Controlled Experiment for Program Comprehension through Trace Visualization." In: *IEEE Transactions on Software Engineering* 37.3 (2011), pp. 341–355 (cit. on pp. 59, 87, 91).

[Dem15] Y. Demir. "Visualization Optimization of Repository Data in Eclipse." Bachelor Thesis. University of Stuttgart, Dec. 2015 (cit. on pp. 25, 219).

[DGL08] M. D'Ambros, H. Gall, M. Lanza. "Analyzing software repositories to understand software evolution." In: *Software Evolution*. Ed. by T. Mens, S. Demeyer. Springer, 2008. Chap. 3, pp. 39–70 (cit. on pp. 21, 32).

[DL06] M. D'Ambros, M. Lanza. "Reverse Engineering with Logical Coupling." In: *2006 13th Working Conference on Reverse Engineering*. 2006, pp. 189–198 (cit. on p. 61).

[DLR09] M. D'Ambros, M. Lanza, R. Robbes. "On the Relationship Between Change Coupling and Software Defects." In: *WCRE*. 2009, pp. 135–144 (cit. on p. 54).

[DNRN13] R. Dyer, H. A. Nguyen, H. Rajan, T. N. Nguyen. "Boa: a language and infrastructure for analyzing ultra-large-scale software repositories." In: *ICSE*. 2013, pp. 422–431 (cit. on p. 54).

[Dri10] D. L. Driscoll. *Introduction to primary research: Observations, surveys, and interviews*. Writing Spaces, Readings on Writing. Parlor Press, 2010 (cit. on p. 47).

[DWP+07] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, J. F. Girard. "An Activity-Based Quality Model for Maintainability." In: *2007 IEEE International Conference on Software Maintenance*. 2007, pp. 184–193 (cit. on p. 203).

[EKKM08] M. Eichberg, S. Kloppenburg, K. Klose, M. Mezini. "Defining and Continuous Checking of Structural Program Dependencies." In: *Proceedings of the 30th International Conference on Software Engineering*. ICSE '08. 2008, pp. 391–400 (cit. on p. 31).

[FGP05] B. Fluri, H. Gall, M. Pinzger. "Fine-grained analysis of change couplings." In: *Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on*. 2005, pp. 66–74 (cit. on pp. 21, 32, 53).

[FLKK17] P. Fournier-Viger, J. C.-W. Lin, R. U. Kiran, Y. S. Koh. "A Survey of Sequential Pattern Mining." In: *Data Science and Pattern Recognition* 1.1 (2017), pp. 54–77 (cit. on p. 200).

[FMR12] C. O. Fritz, P. E. Morris, J. J. Richler. "Effect size estimates: Current use, calculations, and interpretation." In: *Journal of Experimental Psychology : General* 141.1 (Feb. 2012), pp. 2–18 (cit. on p. 172).

[FOMM10] T. Fritz, J. Ou, G. C. Murphy, E. Murphy-Hill. "A Degree-of-knowledge Model to Capture Source Code Familiarity." In: *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. 2010, pp. 385–394 (cit. on p. 51).

[Fou13] P. Fournier-Viger. *How to auto-adjust the minimum support threshold according to the data size*. http://data-mining.philippe-fournier-viger.com. 2013 (cit. on pp. 131, 174).

[FPG03a] M. Fischer, M. Pinzger, H. Gall. "Analyzing and Relating Bug Report Data for Feature Tracking." In: *Proceedings of the 10th Working Conference on Reverse Engineering*. WCRE '03. 2003, pp. 90– (cit. on p. 54).

[FPG03b] M. Fischer, M. Pinzger, H. Gall. "Populating a Release History Database from Version Control and Bug Tracking Systems." In: *Proceedings of the International Conference on Software Maintenance*. ICSM '03. 2003, pp. 23– (cit. on pp. 30, 54, 129).

[FPM92] W. J. Frawley, G. Piatetsky-shapiro, C. J. Matheus. *Knowledge Discovery in Databases: an Overview*. 1992 (cit. on pp. 46, 85).

[FSP+13] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, I. Kwan. "An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks." In: *ACM Transactions on Software Engineering and Methodology* 22 (2013), 14:1–14:41 (cit. on p. 60).

[GAH15] D. German, B. Adams, A. Hassan. "Continuously mining distributed version control systems: an empirical study of how Linux uses Git." In: *Empirical Software Engineering* (2015), pp. 1–40 (cit. on p. 54).

[GAL] E. Guzman, D. Azócar, Y. Li. "Sentiment Analysis of Commit Comments in GitHub: An Empirical Study." In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014, pp. 352–355 (cit. on p. 54).

[GE04] N. J. Gotelli, A. M. Ellison. *A Primer of Ecological Statistics*. Sinauer Associates, 2004 (cit. on p. 106).

[Ger04] D. M. German. "Mining CVS repositories, the softChange experience." In: *1st International Workshop on Mining Software Repositories*. 2004, pp. 17–21 (cit. on pp. 32, 56, 64).

[GG04] C. Győrödi, R. Győrödi. *A Comparative Study of Association Rules Mining Algorithms*. 2004 (cit. on pp. 40, 131).

[GHJ98] H. Gall, K. Hajek, M. Jazayeri. "Detection of Logical Coupling Based on Product Release History." In: *Proceedings of the International Conference on Software Maintenance*. ICSM '98. 1998, pp. 190– (cit. on pp. 20, 31, 32, 53, 54).

[GJK03] H. Gall, M. Jazayeri, J. Krajewski. "CVS release history data for detecting logical couplings." In: *Software Evolution, 2003. Proceedings. Sixth International Workshop on Principles of*. 2003, pp. 13–23 (cit. on p. 53).

[GKS08] G. Gousios, E. Kalliamvakou, D. Spinellis. "Measuring Developer Contribution from Software Repository Data." In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. MSR '08. 2008, pp. 129–132 (cit. on p. 209).

[GKSD05] T. Girba, A. Kuhn, M. Seeberger, S. Ducasse. "How Developers Drive Software Evolution." In: *Proceedings of the Eighth International Workshop on Principles of Software Evolution*. IWPSE '05. 2005, pp. 113–122 (cit. on pp. 51, 62).

[Goe10] B. Goethals. "Frequent Set Mining." In: *Data Mining and Knowledge Discovery Handbook*. Springer, 2010, pp. 321–338 (cit. on p. 39).

[GS67] B. Glaser, A. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Observations (Chicago, Ill.) Aldine Publishing Company, 1967 (cit. on p. 132).

[GZ05] G. Grahne, J. Zhu. "Fast Algorithms for Frequent Itemset Mining Using FP-Trees." In: *IEEE Trans. on Knowl. and Data Eng.* 17.10 (Oct. 2005), pp. 1347–1362 (cit. on p. 45).

[Hau02] E. Hautus. *Improving Java Software Through Package Structure Analysis*. `http://ehautus.home.xs4all.nl/papers/PASTA.pdf`. 2002 (cit. on pp. 37, 61).

[HGH08] A. Hindle, D. M. German, R. Holt. “What Do Large Commits Tell Us?: A Taxonomical Study of Large Commits.” In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. 2008, pp. 99–108 (cit. on pp. 162, 176).

[HH04] A. E. Hassan, R. C. Holt. “Predicting Change Propagation in Software Systems.” In: *Proceedings of the 20th IEEE International Conference on Software Maintenance*. ICSM ’04. 2004, pp. 284–293 (cit. on p. 55).

[HL00] D. W. Hosmer, S. Lemeshow. *Applied logistic regression*. Wiley series in probability and statistics. John Wiley & Sons, Inc. A Wiley-Interscience Publication, 2000 (cit. on p. 111).

[HL11] B. Hanlon, B. Larget. *Analysis of Variance*. `http://www.stat.wisc.edu/~st571-1/13-anova-4.pdf`. 2011 (cit. on p. 171).

[HLT09] K. Hinsen, K. Laeufer, G. K. Thiruvathukal. “Essential Tools: Version Control Systems.” In: *Computing in Science and Engineering* 11.6 (Dec. 6, 2009), pp. 84–91 (cit. on p. 29).

[HMP05] J. Han, K. Micheline, J. Pei. *Data Mining: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005 (cit. on pp. 22, 38–40, 45).

[HNM11] R. Hoda, J. Noble, S. Marshall. “Grounded Theory for Geeks.” In: *Proceedings of the 18th Conference on Pattern Languages of Programs*. PLoP ’11. 2011, 24:1–24:17 (cit. on p. 133).

[HPYM04] J. Han, J. Pei, Y. Yin, R. Mao. “Mining Frequent Patterns Without Candidate Generation: A Frequent-Pattern Tree Approach.” In: *Data Min. Knowl. Discov.* 8.1 (Jan. 2004), pp. 53–87 (cit. on pp. 41, 131).

[HSCS08] L. Hattori, G. dos Santos Jr, F. Cardoso, M. Sampaio. "Mining Software Repositories for Software Change Impact Analysis: A Case Study." In: *Proceedings of the 23rd Brazilian Symposium on Databases*. SBBD '08. 2008, pp. 210–223 (cit. on p. 56).

[IEE98] IEEE. "STD 1219: Standard for Software Maintenance." In: 1998 (cit. on p. 48).

[ISO00] ISO/IEC. *14764: Software Engineering-Software Maintenance*. 2000 (cit. on p. 49).

[ISO95] ISO/IEC. *12207: Information Technology-Software life cycle processes*. 1995 (cit. on p. 48).

[Jew12] C. Jewitt. *An introduction to using video for research*. 2012 (cit. on p. 58).

[Job16] M. Joblin. *Classifying Developers into Core and Peripheral: An Empirical Study on Count and Network Metrics*. `http://arxiv.org/abs/1604.00830`. 2016 (cit. on pp. 51, 62).

[Kau17] Y. Kaupe. "Visualization of Data Mining in an Eclipse Plugin using Word Clouds." Bachelor Thesis. University of Stuttgart, Apr. 2017 (cit. on pp. 25, 219).

[KCM07] H. Kagdi, M. L. Collard, J. I. Maletic. "A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution." In: *J. Softw. Maint. Evol.* 19.2 (Mar. 2007), pp. 77–131 (cit. on p. 22).

[KMCA06] A. J. Ko, B. A. Myers, M. J. Coblenz, H. H. Aung. "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information During Software Maintenance Tasks." In: *IEEE Trans. Softw. Eng.* 32.12 (Dec. 2006), pp. 971–987 (cit. on pp. 50, 57, 60, 87, 91).

[KMS07] H. Kagdi, J. I. Maletic, B. Sharif. “Mining software repositories for traceability links.” In: *15th IEEE International Conference on Program Comprehension (ICPC ’07)*. 2007, pp. 145–154 (cit. on pp. 34, 54, 87, 90).

[Kon11] K. T. Konecki. *Visual Grounded Theory: A Methodological Outline and Examples from Empirical Work*. 2011 (cit. on p. 58).

[KTM+99] B. A. Kitchenham, G. H. Travassos, A. von Mayrhauser, F. Niessink, N. F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, H. Yang. “Towards an ontology of software maintenance.” In: *Journal of Software Maintenance* 11.6 (1999), pp. 365–389 (cit. on p. 48).

[KXLL16] P. S. Kochhar, X. Xia, D. Lo, S. Li. “Practitioners’ Expectations on Automated Fault Localization.” In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. 2016, pp. 165–176 (cit. on p. 59).

[KYM06] H. Kagdi, S. Yusuf, J. I. Maletic. “Mining Sequences of Changed-files from Version Histories.” In: *Proceedings of the 2006 International Workshop on Mining Software Repositories*. MSR ’06. 2006 (cit. on pp. 22, 33, 34, 40, 53, 55, 56, 64, 69, 87, 88, 90, 123, 128, 151).

[LBB+10] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, S. D. Fleming. “How Programmers Debug, Revisited: An Information Foraging Theory Perspective.” In: *IEEE Transactions on Knowledge and Data Engineering* 39 (2010), pp. 197–215 (cit. on p. 59).

[Leh15] S. Lehmann. “Automatic Transformation of Data from SOftware Repositories and their Preparation for Data Mining.” Bachelor Thesis. University of Stuttgart, Nov. 2015 (cit. on pp. 25, 219).

[Loe09] J. Loeliger. *Version Control with Git - Powerful techniques for centralized and distributed project management.* O’Reilly, 2009, pp. I–XV, 1–310 (cit. on pp. 30, 127, 167).

[LXPZ13] H. Li, Z. Xing, X. Peng, W. Zhao. "What help do developers seek, when and how?" In: *2013 20th Working Conference on Reverse Engineering (WCRE)* (2013), pp. 142–151 (cit. on pp. 57, 60, 87, 91).

[McD01] D. W. McDonald. "Evaluating Expertise Recommendations." In: *Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*. GROUP '01. 2001, pp. 214–223 (cit. on p. 62).

[McG05] K. McGarry. "A Survey of Interestingness Measures for Knowledge Discovery." In: *Knowl. Eng. Rev.* 20.1 (Mar. 2005), pp. 39–61 (cit. on p. 84).

[MFH02] A. Mockus, R. T. Fielding, J. D. Herbsleb. "Two Case Studies of Open Source Software Development: Apache and Mozilla." In: *ACM Trans. Softw. Eng. Methodol.* 11.3 (July 2002), pp. 309–346 (cit. on p. 62).

[MHS+12] A. Mauczka, M. Huber, C. Schanes, W. Schramm, M. Bernhart, T. Grechenig. "Tracing Your Maintenance Work - A Cross-Project Validation of an Automated Classification Dictionary for Commit Messages." In: *FASE*. Lecture Notes in Computer Science. 2012, pp. 301–315 (cit. on p. 35).

[MM07] S. Minto, G. C. Murphy. "Recommending Emergent Teams." In: *Fourth International Workshop on Mining Software Repositories (MSR'07:ICSE Workshops 2007)*. 2007, pp. 5–5 (cit. on p. 63).

[Moc14] A. Mockus. "Is Mining Software Repositories Data Science? (Keynote)." In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. 2014 (cit. on p. 19).

[Nac08] N. Nachar. "The Mann-Whitney U: A Test for Assessing Whether Two Independent Samples Come from the Same Distribution." In: *Tutorials in Quantitative Methods for Psychology* 4.1 (2008), pp. 13–20 (cit. on p. 171).

[NBD11] V. Nguyen, B. Boehm, P. Danphitsanuphan. "A controlled experiment in assessing and estimating software maintenance tasks." In: *Inf. Softw. Technol.* 53.6 (June 2011), pp. 682–691 (cit. on pp. 50, 58, 87, 91, 166, 168).

[NYN+02] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, Y. Ye. "Evolution Patterns of Open-source Software Systems and Communities." In: *Proceedings of the International Workshop on Principles of Software Evolution*. IWPSE '02. 2002, pp. 76–85 (cit. on p. 62).

[Ott] S. Otte. *Version Control Systems* (cit. on p. 30).

[PA14] A. Perez, R. Abreu. "A Diagnosis-based Approach to Software Comprehension." In: *Proceedings of the 22Nd International Conference on Program Comprehension*. ICPC 2014. 2014, pp. 37–47 (cit. on p. 59).

[PA16] A. Perez, R. Abreu. "Framing program comprehension as fault localization." In: *Journal of Software Evolution and Process* 28 (2016), pp. 840–862 (cit. on p. 59).

[Par] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules." In: *Commun. ACM* 15.12 (), pp. 1053–1058 (cit. on pp. 21, 37).

[PHM+04] J. Pei, J. Han, B. Mortazavi-Asl, J. Wang, H. Pinto, Q. Chen, U. Dayal, M.-C. Hsu. "Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach." In: *IEEE Trans. on Knowl. and Data Eng.* 16.11 (Nov. 2004), pp. 1424–1440 (cit. on p. 200).

[Pig96] T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. 1st. Wiley Publishing, 1996 (cit. on p. 49).

[PM94] G. Piatetsky-Shapiro, C. J. Matheus. “The Interestingness of Deviations.” In: *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*. AAAIWS’94. 1994, pp. 25–36 (cit. on pp. 46, 84).

[PO11] C. Parnin, A. Orso. “Are Automated Debugging Techniques Actually Helping Programmers?” In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA ’11. 2011, pp. 199–209 (cit. on p. 59).

[Poh14] T. Pohlert. *The Pairwise Multiple Comparison of Mean Ranks Package (PMCMR)*. R package. 2014 (cit. on p. 173).

[PSW11] S. Phillips, J. Sillito, R. Walker. “Branching and merging: an investigation into current version control practices.” In: *In International workshop on Cooperative and human aspects of software engineering, CHASE ’11, ACM*. 2011, pp. 9–15 (cit. on pp. 87, 90).

[PTL+11] R. Premraj, A. Tang, N. Linssen, H. Geraats, H. van Vliet. “To Branch or Not to Branch?” In: *Proceedings of the 2011 International Conference on Software and Systems Process*. 2011, pp. 81–90 (cit. on pp. 87, 90).

[QK02] G. P. Quinn, M. J. Keough. *Experimental design and data analysis for biologists*. Cambridge University Press, 2002 (cit. on p. 105).

[RD04] F. van Rysselberghe, S. Demeyer. “Mining Version Control Systems for FACs (frequently Applied changes).” In: *the International Workshop on Mining Repositories*. Edinburgh, Scotland, UK, 2004 (cit. on pp. 30, 56).

[RFJS07] G. RICKI, E. FREDERICK, L. JAY, D. SHARON J. “Selecting in Video.” In: (2007). http://drdc.uchicago.edu/what/video-research-guidelines.pdf#page=1&view=fitV (cit. on p. 58).

[RGMA06] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, J. J. Amor. "Mining Large Software Compilations over Time: Another Perspective of Software Evolution." In: *Proceedings of the International Workshop on Mining Software Repositories (MSR 2006)*. 2006, pp. 3–9 (cit. on p. 61).

[RGP11] M. Revelle, M. Gethers, D. Poshyvanyk. "Using Structural and Textual Information to Capture Feature Coupling in Object-oriented Software." In: *Empirical Softw. Engg.* 16.6 (Dec. 2011), pp. 773–811 (cit. on pp. 50, 57).

[RH09] P. Runeson, M. Höst. "Guidelines for Conducting and Reporting Case Study Research in Software Engineering." In: *Empirical Softw. Engg.* 14.2 (Apr. 2009), pp. 131–164 (cit. on pp. 124, 134).

[RKTC16] R. B. Rayana, S. Killian, N. Trangez, A. Calmettes. "GitWaterFlow: A Successful Branching Model and Tooling, for Achieving Continuous Delivery with Multiple Version Branches." In: *Proceedings of the 4th International Workshop on Release Engineering*. 2016, pp. 17–20 (cit. on p. 30).

[RLR+12] F. Ricca, M. Leotta, G. Reggio, A. Tiso, G. Guerrini, M. Torchiano. "Using UniMod for maintenance tasks: an experimental assessment in the context of model driven development." In: *Proceedings of the 4th International Workshop on Modeling in Software Engineering, MiSE 2012, Zurich, Switzerland, June 2-3, 2012*. 2012, pp. 77–83 (cit. on pp. 58, 87, 91, 156, 157, 168, 195).

[RP09] M. Revelle, D. Poshyvanyk. "An exploratory study on assessing feature location techniques." In: *ICPC*. 2009, pp. 218–222 (cit. on p. 60).

[RPL08] R. Robbes, D. Pollet, M. Lanza. “Logical Coupling Based on Fine-Grained Change Information.” In: *2008 15th Working Conference on Reverse Engineering*. 2008, pp. 42–46 (cit. on p. 53).

[RR13] R. Robbes, D. Rothlisberger. “Using developer interaction data to compare expertise metrics.” In: Los Alamitos, CA, USA, 2013, pp. 297–300 (cit. on p. 62).

[RW] M. E. Rikard Andersson, A. Wingkvist†. “Mining Relations from Git Commit Messages — an Experience Report.” In: *The Fifth Swedish Language Technology Conference*. SLTC 2014 (cit. on p. 54).

[RW16a] J. Ramadani, S. Wagner. “Are coupled file changes suggestions useful?” In: (2016). `https://peerj.com/preprints/2492/` (cit. on pp. 24, 27, 85, 99, 194).

[RW16b] J. Ramadani, S. Wagner. “Are Suggestions of Coupled File Changes Interesting?” In: *In Proceedings of the 11th International Conference on Evaluation of Novel Software Approaches to Software Engineering*. 2016, pp. 15–26 (cit. on pp. 24, 27, 84, 86, 89, 90, 99, 192).

[RW16c] J. Ramadani, S. Wagner. “How Interesting Are Suggestions of Coupled File Changes for Software Developers?” In: *Evaluation of Novel Approaches to Software Engineering: 11th International Conference, ENASE, Revised Selected Papers*. Ed. by L. A. Maciaszek, J. Filipe. Springer International Publishing, 2016, pp. 201–221 (cit. on p. 27).

[RW16d] J. Ramadani, S. Wagner. “Which Change Sets in Git Repositories Are Related?” In: *In Proceedings of the International Conference on Software Quality, Reliability and Security (QRS)*. 2016 (cit. on pp. 23, 24, 27).

[RW17a] J. Ramadani, S. Wagner. “How Do Coupled File Changes Influence How Developers Seek Help During Maintenance Tasks?” In: *In Proceedings of the International Conference on Software Quality, Reliability and Security (QRS)*. 2017 (cit. on pp. 24, 27, 100).

[RW17b] J. Ramadani, S. Wagner. “Mining Java Packages for Developer Profiles: An Exploratory Study.” In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017), 17. Fachtagung des GI-Fachbereichs ‚Datenbanken und Informationssysteme" (DBIS), 6.-10. März 2017, Stuttgart, Germany, Workshopband*. 2017, pp. 143–152 (cit. on pp. 25, 28).

[SAPM14] G. Salvaneschi, S. Amann, S. Proksch, M. Mezini. “An Empirical Study on Program Comprehension with Reactive Programming.” In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. 2014, pp. 564–575 (cit. on p. 58).

[Say+11] J. Sayles et al. *zOS Traditional Application Maintenance and Support*. IBM Redbooks, 2011 (cit. on p. 125).

[SC98] A. Strauss, J. M. Corbin. *Basics of Qualitative Research : Techniques and Procedures for Developing Grounded Theory*. SAGE Publications, 1998 (cit. on pp. 132, 133).

[Sch03] J. Schwab. *Logistic Regression - Complete Problems*. University Lecture. 2003 (cit. on pp. 111, 112).

[SCR98] G. B. Shelly, T. J. Cashman, H. J. Rosenblatt. *Systems Analysis and Design*. 1998 (cit. on p. 48).

[SDAH08] D. I. Sjøberg, T. Dybå, B. C. Anda, J. E. Hannay. “Building theories in software engineering.” In: *Guide to advanced empirical software engineering*. Springer London, 2008, pp. 312–336 (cit. on pp. 85, 86, 92, 94, 98).

[SDMA07] S.Ducasse, D.Pollet, M.Suen, H. andI. Alloui. "Package Surface Blueprints: Visually Supporting the Understanding of Package Relationships." In: *2007 IEEE International Conference on Software Maintenance*. 2007, pp. 94–103 (cit. on p. 61).

[SH98] S. E. Sim, R. C. Holt. "The ramp-up problem in software projects: A case study of how software immigrants naturalize." In: *Proceedings of the 1998 International Conference on Software Engineering*. IEEE. 1998, pp. 361–370 (cit. on p. 21).

[SLM03] J. Shirabad, T. Lethbridge, S. Matwin. "Mining the maintenance history of a legacy software system." In: *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. 2003, pp. 95–104 (cit. on p. 56).

[SLVA97] J. Singer, T. Lethbridge, N. Vinson, N. Anquetil. "An Examination of Software Engineering Work Practices." In: *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '97. 1997, pp. 21– (cit. on pp. 21, 60).

[Som02] I. Sommerville. "Software documentation." In: *In Software Engineering, vol 2: The supporting Processes. R.H. Thayer and M.I. Christensen (eds), Willey-IEEE*. Press, 2002 (cit. on p. 31).

[Ste08] L. P. Stephan Salinger Laura Plonka. "The Mann-Whitney U: A Test for Assessing Whether Two Independent Samples Come from the Same Distribution." In: *Human Technology: An Interdisciplinary Journal on Humans in ICT Environments* 4.1 (2008), pp. 9–25 (cit. on p. 58).

[SVAA15] L. L. Silva, M. T. Valente, M. de A. Maia, N. Anquetil. "Developers' perception of co-change patterns: An empirical study." In: *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2015, pp. 21–30 (cit. on p. 57).

[Swa76] E. B. Swanson. "The Dimensions of Maintenance." In: *Proceedings of the 2Nd International Conference on Software Engineering*. ICSE '76. 1976, pp. 492–497 (cit. on pp. 48, 49).

[Sys13] H. Systems. *Package by feature, not layer* . http://www.javapractices.com/topic/TopicAction.do?Id=205. 2013 (cit. on p. 38).

[SZ08] D. Schuler, T. Zimmermann. "Mining Usage Expertise from Version Archives." In: *Proceedings of the 2008 International Working Conference on Mining Software Repositories*. MSR '08. 2008, pp. 121–124 (cit. on pp. 63, 81, 209, 210).

[SZ13] J. Steven, W. Zach. *Bad Commit Smells*. http://pages.cs.wisc.edu/~sjj/docs/commits.pdf. 2013 (cit. on p. 139).

[TPF+14] C. Teyton, M. Palyart, J.-R. Falleri, F. Morandat, X. Blanc. "Automatic Extraction of Developer Expertise." In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE '14. 2014, 8:1–8:10 (cit. on p. 61).

[TSK05] P.-N. Tan, M. Steinbach, V. Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2005 (cit. on pp. 41, 45).

[TT14] M. Tomczak, E. Tomczak. "The need to report effect size estimates revisited. An overview of some recommended measures of effect size." In: *The need to report effect size estimates revisited. An overview of some recommended measures of effect size* TRENDS in Sport Sciences (2014), pp. 19–25 (cit. on p. 172).

[VGG+14] P. F. Viger, A. Gomariz, T. Gueniche, A. Soltani, C.-W. Wu, V. S. Tseng. "SPMF: A Java Open-Source Pattern Mining Library." In: *Journal of Machine Learning Research* 15 (2014). http://www.philippe-fournier-viger.com/spmf/, pp. 3389–3393 (cit. on pp. 46, 71, 220).

[VT06] L. Voinea, A. Telea. “Mining Software Repositories with CVSgrab.” In: *Proceedings of the 2006 International Workshop on Mining Software Repositories*. 2006, pp. 167–168 (cit. on p. 56).

[WA09] A. I. Wang, E. Arisholm. “The Effect of Task Order on the Maintainability of Object-oriented Software.” In: *Inf. Softw. Technol.* 51.2 (Feb. 2009), pp. 293–305 (cit. on pp. 58, 87, 91).

[WH04] J. Wang, J. Han. “BIDE: Efficient Mining of Frequent Closed Sequences.” In: *Proceedings of the 20th International Conference on Data Engineering*. ICDE ’04. 2004, pp. 79– (cit. on p. 200).

[WLR11] R. Wettel, M. Lanza, R. Robbes. “Software Systems As Cities: A Controlled Experiment.” In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE ’11. 2011, pp. 551–560 (cit. on pp. 59, 87, 91).

[WPXZ11] J. Wang, X. Peng, Z. Xing, W. Zhao. “An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions.” In: *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE. 2011 (cit. on p. 60).

[WS02] C. Walrad, D. Strom. “The Importance of Branching Models in SCM.” In: *Computer* 35.9 (2002), pp. 31–38 (cit. on pp. 87, 90).

[WS08] C. C. Williams, J. W. Spacco. “Branching and Merging in the Repository.” In: *MSR*. New York, NY, USA, 2008, pp. 19–22 (cit. on p. 30).

[WZKC11] R. Wu, H. Zhang, S. Kim, S.-C. Cheung. “ReLink: Recovering Links Between Bugs and Changes.” In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE ’11. 2011, pp. 15–25 (cit. on p. 54).

[XBLL16] X. Xia, L. Bao, D. Lo, S. Li. "Automated Debugging Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems." In: *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution*. 2016, pp. 267–278 (cit. on p. 59).

[YMNC04] A. T. T. Ying, G. C. Murphy, R. T. Ng, M. Chu-Carroll. "Predicting Source Code Changes by Mining Change History." In: *IEEE Transactions on Software Engineering* 30.9 (2004), pp. 574–586 (cit. on pp. 22, 53, 54, 56, 65, 87, 88, 123, 151).

[YR14] A. T. T. Ying, M. P. Robillard. "Developer Profiles for Recommendation Systems." In: *Recommendation Systems in Software Engineering*. Ed. by M. P. Robillard, W. Maalej, R. J. Walker, T. Zimmermann. 2014, pp. 199–222 (cit. on p. 63).

[ZW14] L. Zhou, X. Wang. "Research of the FP-Growth Algorithm Based on Cloud Environments." In: *JSW* 9 (2014), pp. 676–683 (cit. on p. 41).

[ZWDZ04] T. Zimmermann, P. Weisgerber, S. Diehl, A. Zeller. "Mining Version Histories to Guide Software Changes." In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE '04. 2004, pp. 563–572 (cit. on pp. 22, 32, 40, 53, 54, 56, 64, 87, 88, 123, 151).

All URLs were checked at 19.09.2017.

List of Figures

2.1 1 commit to 1 issue 36
2.2 1 commit to n issues 37
2.3 n commits to 1 issue 37
2.4 FP-Tree after t_1 43
2.5 FP-Tree after t_2 43
2.6 FP-Tree after t_3 44
2.7 FP-Tree 44

4.1 Obtaining Couplings 68
4.2 Obtaining Attributes 72
4.3 Commit Export from Git 73
4.4 Coupled Changes ER-Diagram 75
4.5 Building Coupled File Change Suggestions 75
4.6 Smart Commits 76

5.1 Building Theory 86

5.2 Construct Sources 87
5.3 A Theory for Change Suggestions 92
5.4 Theory Testing 98

6.1 Experimental variables 106
6.2 Related change sets time distribution 118

7.1 Commit attributes and experience 141
7.2 Issue attributes and experience 141
7.3 Theoretical Framework 142
7.4 Conceptual Model from Grounded Theory 144

8.1 Changes Selection 161
8.2 Task Correctness Distribution 178
8.3 Time Boxplots (t_s) 180
8.4 Time Boxplots ($t_r + t_s$) 181
8.5 Usefulness of Attributes 186

9.1 Information Sources 201
9.2 Information Sources Relevance 202

11.1 Tool Architecture 220
11.2 Eclipse IDE 221
11.3 Tool Start 222
11.4 Wizard Sources 222
11.5 Wizard Developers 223
11.6 View Lists 225
11.7 Attributes View 225

List of Tables

2.1 Unrelated changes 33
2.2 Related changes 33
2.3 File Change Transactions 40
2.4 Transaction Items 42
2.5 Frequent Patterns 45

4.1 Issues Export 73
4.2 Package description 77
4.3 Coupled Change Suggestion 79
4.4 Packages couplings 80
4.5 Developer profiles 81

6.1 Commit Combinations 109
6.2 Descriptive Statistics 114
6.3 Relatedness distributions 115
6.4 Regression Results 116

6.5 Influence on relatedness across project 120

7.1 Results based on repository analysis 136
7.2 Interestingness of coupled changes 136
7.3 Couplings and developer's experience 138
7.4 Couplings and developer's project involvement 138
7.5 Interesting attributes 139

8.1 Usefulness score 156
8.2 Experiment Design 157
8.3 Task Information and Coupled File Changes 164
8.4 Repository Attributes Description 167
8.5 Issue Classification 176
8.6 Perfective Issues 176
8.7 Statistical Significance (Coupled changes) 177
8.8 Descriptive statistics for the correctness of the tasks . . 179
8.9 Time to determine related coupled files 184
8.10 Descriptive statistics for the time needed in minutes . . 184
8.11 Descriptive Statistics (Attributes Usefulness) 185
8.12 Statistical Significance (Coupled changes) 187

9.1 Experiment Design 196
9.2 Information Sources Frequency 201
9.3 information Source patterns 202
9.4 information Source patterns 203

10.1 Most frequent packages in the couplings 213
10.2 Most frequent packages in the couplings 214
10.3 Most frequent packages in the couplings 215