Studying Fine-Grained Co-Evolution Patterns of Production and Test Code

Master's Thesis

Cosmin Marsavina

Studying Fine-Grained Co-Evolution Patterns of Production and Test Code

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Cosmin Marsavina
born in Resita, Romania

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

© 2014 Cosmin Marsavina.

Studying Fine-Grained Co-Evolution Patterns of Production and Test Code

Author:	Cosmin Marsavina
Student id:	4259734
Email:	`cmarsavina@student.tudelft.nl`

Abstract

Numerous software development practices suggest updating the test code whenever the production code is changed. However, previous studies have shown that co-evolving test and production code is generally a difficult task that needs to be thoroughly investigated.

In this thesis we perform a study that, following a mixed methods approach, investigates fine-grained co-evolution patterns of production and test code. First, we mine fine-grained changes from the evolution of 5 open-source systems. Then, we use an association rule mining algorithm to generate the co-evolution patterns. Finally, we interpret the obtained patterns by performing a qualitative analysis.

The results show 6 co-evolution patterns and provide insights into their appearance along the history of the analyzed software systems. Besides providing a better understanding of how test code evolves, these findings also help identify gaps in the test code thereby assisting both researchers and developers.

Thesis Committee:

Chair:	Dr. A. Zaidman, Faculty EEMCS, TU Delft
University supervisor:	D. Romano, Faculty EEMCS, TU Delft
Committee Member:	Prof. Dr. D. H. J. Epema, Faculty EEMCS, TU Delft
Committee Member:	Dr. G. Wachsmuth, Faculty EEMCS, TU Delft

Preface

The thesis was written as part of my Master of Science degree at Delft University of Technology. Before I began working on this master's thesis I was unaware of the effects that it will have on my personal and professional development. While working on it I have learned a lot of things both in terms of software evolution and regarding the specific requirements of working in an academic environment. Even though there where times when I struggled, they proved to be just small parts of an experience that was very enjoyable overall. At the start of the thesis I was just pondering the idea of pursuing a career in research. However, after finishing the work, I am quite certain that I want to continue along this path. This is why I would like to thank a number of people for helping me through this process.

First of all, I want to express my gratitude to Dr. Andy Zaidman for allowing me to work on this project as part of the Software Engineering Research Group. He has provided me with invaluable feedback both on the thesis and on the paper that was written based on it. I have appreciated the fact that he was always available for discussion no matter how serious the problem was. The second person to whom I owe a big thank you is Daniele Romano. He has been my daily supervisor and has guided me all throughout the thesis. It was very reassuring to know that I could rely on his expertise whenever I encountered an issue. Both of them have spent a lot of time aiding me in completing this thesis.

Next I would like to thank the anonymous reviewers from the International Working Conference on Source Code Analysis and Manipulation for appreciating my work and providing me with additional feedback that was integrated in the thesis. I also want to thank the conference's organizing committee for awarding me a travel grant so that I can attend the conference and present the work.

My thanks also go to Dr. Radu Marinescu who has taken the time to proofread the thesis, thus providing an external opinion on my work. I would also like to thank my friends, Mircea Voda and Mircea Cadariu, for allowing me to consult with them on thesis related subjects while also supplying me with insightful ideas. Finally, I want to thank my family and especially my parents, Liviu and Dalila, as their continuous support has helped me achieve my goals.

Cosmin Marsavina
Delft, the Netherlands
August 19, 2014

Contents

List of Figures

Chapter 1

Introduction

This chapters provides an introduction to the work that will be presented in the following chapters of the thesis. It first discusses the problem at hand and the research questions formulated in order to address it. Afterwards, the relevance of our work is explained. The last part of the chapter presents an outline of the remainder of this document.

1.1 Problem Statement

Lehman has taught us that a software system must evolve, or it becomes progressively less useful [22]. During this evolution, the system's source code continuously changes to cope with new requirements or possible issues that might arise. However, software is multi-dimensional, because in order to develop high-quality systems other artifacts need to be taken into account, such as requirements, tests and documentation [25]. Therefore, these artifacts should *co-evolve* gracefully alongside the production code that is being written.

One of the artifacts that is of particular importance in the software development process is the developer test, which was defined by [26] as "a codified unit or integration test written by developers". Unit tests have become increasingly popular over the last years, as many programming languages support their creation through testing frameworks (*e.g.,* JUnit for Java or NUnit for C#). The importance of these tests resides in the fact that they can provide immediate feedback to the developers [37] and aid in identifying bugs [1], while also increasing program comprehension [24]. In addition to these aspects, the tests can be used to pinpoint the exact location where a defect occurs [27]. They also increase the confidence of developers in modifying the production code without causing any kinds of issues within the developed software system [4]. Moreover, when a software system evolves (e.g., through refactoring), developers should run them as persistence tests to verify whether the external behavior is preserved [10]. In this context, Moonen *et al.* have shown that even though refactorings are behavior preserving, they can invalidate tests [28]. In the same vein, Elbaum *et al.* have concluded that even minor changes in the production code can significantly affect test coverage [11].

Based on these findings, there clearly is a need for tests to evolve alongside the production code they are covering in order to obtain high-quality systems. However, creating and maintaining tests are expensive tasks. Previous studies have shown that 30 to 50% of the effort spent on a project is dedicated to testing [12]. On the same note, Zaidman *et al.* have shown that developing test code that co-evolves gracefully with the production classes it addresses is generally a difficult endeavour [47].

1.2 Research Questions

In this study we try to identify fine-grained co-evolution patterns between production and test code. These patterns consist of changes that occur in the test code when changes are made to the production code. It is also likely that some co-evolution patterns appear more frequently for particular software systems. Hence, besides identifying these patterns, we aim at correlating them with the testing effort spent for each of the analyzed systems. Finally, we want to understand how the co-evolution between production and test code takes place and to find the reasons why co-evolution does not happen in some specific cases. This leads us to our research questions:

RQ1 What kind of fine-grained co-evolution patterns between production and test code can be identified?

RQ2 Does the testing effort have an impact on the observed co-evolution patterns?

RQ3 How does the co-evolution between production and test code happen?

The first research question directly addresses production and test code co-evolution. It is concerned with identifying specific patterns that can be observed as the production and test classes evolve. Such patterns consist of changes that are made in the test code when a particular change occurs in a production class. Unlike previous work that has been done thus far in this area of the software engineering field, the source code changes are studied at a fine-grained level, thereby allowing for a deeper understanding of the co-evolution between the production and the test code of a software project.

For **RQ2** we want to compare the patterns observed for multiple projects (with different testing efforts) in order to determined whether or not this factor has an influence on the co-evolution of production and test code. More specifically, we try to prove that different patterns are encountered for projects that are extensively tested compared to the ones obtained for projects with low testing effort.

The third research question is aimed at understanding the way in which this co-evolution occurs. We are interested in determining whether the production and the test code of a software system evolve synchronous or if the tests are updated at a later point in time (several versions after the production classes they are addressing are changed). In addition to this, in the cases when the test code does not co-evolve alongside the production code it covers, we would like to identify the reasons why such situations are encountered. This allows for a thorough understanding of the nature of the co-evolution between production and test code.

We answer our research questions by following a mixed methods approach [9] that combines quantitative and qualitative analyses. First, we use an association rule mining algorithm to identify co-evolution patterns. Then, we refine these quantitative results through a qualitative analysis aimed at manually interpreting the patterns obtained. The results show: (1) 6 co-evolution patterns mined for 5 case study systems, (2) how they occur, and (3) whether the testing effort has an impact on them.

1.3 Relevance

From a research perspective, getting insight into these co-evolution patterns is particularly useful to check whether specific changes in the production code should also have consequences in the test code of a software system. By identifying the test changes that have to be done when a specific production change is performed we can determine the parts of the production code that still need to be addressed. This would also allow for an assessment of the gravity of the situation when certain changes are not made in the test code in order to cover a specific change in a production class. Furthermore, by inspecting the moments when the test changes are made, we can uncover cases in which they should have been done earlier (*e.g.*, in the same commit as the production change that triggered them). This might lead to better tool support, thereby assisting developers in designing higher quality test code.

The main contributions of this thesis are as follows:

1. A method to collect and relate fine-grained source code changes that co-occur in the production and the test code of a software system; this method can be utilized to: (1) extract the fine-grained production and test code changes and (2) link the test cases in which changes occur with the production classes where the changes that triggered them were made.

2. An empirical study to investigate the co-evolution between production and test code for 5 open-source systems; the study comprises a quantitative analysis during which a number of co-evolution patterns have been uncovered and a more in-depth qualitative analysis with anecdotal evidence on each of the patterns.

3. We enrich the software engineering body of knowledge with regard to production and test code co-evolution; by studying this topic we gain a deeper understanding of: the co-evolution patters observed for a specific project, how production and test code co-evolution happens, and whether or not the testing effort spent on a project has an impact on the co-evolution patterns. In addition to this, we provide explanations as to why co-evolution occurs in a specific way and reasons why sometimes co-evolution is absent.

1.4 Outline

The remainder of this document is structured as follows. In Chapter 2 we describe the approach adopted to collect the data for our analyses. Chapter 3 discusses the design of the studies that were conducted. Results are reported in Chapters 4 and 5 that present respectively the quantitative and qualitative analyses. In Chapter 6 we revisit the research questions and discuss threats to validity. Chapter 7 details on related work. Finally, we conclude the document and present future work direction in Chapter 8.

Chapter 2

Approach

This chapter describes the approach developed in order to be able to collect the data necessary for studying the co-evolution between production and test code. We first present the approach at a conceptual level. Then, we provide the concrete implementation details. The structure of the obtained dataset is discussed in the last part of the chapter.

2.1 Data collection procedure

As discussed in the previous chapter, there is a need within the scientific community to examine and understand the co-evolution between the production and the test code of a software project. The main goal of this study is to identify a series of patterns consisting of changes that occur in the test code when the production code evolves. We expect these co-evolution patterns to vary from one project to another, because of the different working styles of the development teams or due to different priorities with regards to testing activities. Therefore, while performing our analyses, we also assess the testing effort put into each of the projects under study. Furthermore, besides uncovering the patterns, we also inspect the source code to find and understand concrete examples that help in interpreting the obtained co-evolution patterns.

In the following subsections we describe the approach adopted to: (1) extract fine-grained changes and (2) link production and test code.

2.1.1 Change Extraction

In order to collect relevant data for studying the co-evolution of production and test code, we first obtain all the versions of a project. We mine Git as this facilitates the access to the repositories of a large variety of software projects. Moreover, it provides functionalities to compute high-level differences (*e.g.,* addition and deletion of classes) between one version of a project and another.

However, these differences are not detailed enough to allow for an in-depth analysis of the co-evolution between production and test code. For this reason we extract fine-grained source code changes between different versions using ChangeDistiller [13]. Table 2.1 details all the change categories along with the specific changes that ChangeDis-

Change category	Change
ADDED_CLASS	ADDITIONAL_CLASS
REMOVED_CLASS	REMOVED_CLASS
CLASS_DECLARATION	CLASS_RENAMING,
	PARENT_CLASS_CHANGE,
	PARENT_CLASS_DELETE,
	PARENT_CLASS_INSERT,
	PARENT_INTERFACE_CHANGE,
	PARENT_INTERFACE_DELETE,
	PARENT_INTERFACE_INSERT,
	REMOVED_FUNCTIONALITY,
	ADDITIONAL_FUNCTIONALITY
METHOD_DECLARATION	RETURN_TYPE_CHANGE,
	RETURN_TYPE_DELETE,
	RETURN_TYPE_INSERT,
	METHOD_RENAMING,
	PARAMETER_DELETE,
	PARAMETER_INSERT,
	PARAMETER_ORDERING_CHANGE,
	PARAMETER_RENAMING,
	PARAMETER_TYPE_CHANGE
ATTRIBUTE_DECLARATION	ATTRIBUTE_RENAMING,
	ATTRIBUTE_TYPE_CHANGE,
	ADDING_ATTRIBUTE_MODIFIABILITY,
	REMOVING_ATTRIBUTE_MODIFIABILITY,
	ADDITIONAL_OBJECT_STATE,
	REMOVED_OBJECT_STATE
BODY_STATEMENTS	STATEMENT_DELETE,
	STATEMENT_INSERT,
	STATEMENT_ORDERING_CHANGE,
	STATEMENT_PARENT_CHANGE,
	STATEMENT_UPDATE
BODY_CONDITIONS	CONDITION_EXPRESSION_CHANGE,
	ALTERNATIVE_PART_DELETE,
	ALTERNATIVE_PART_INSERT
COMMENTS	COMMENT_DELETE,
	COMMENT_INSERT,
	COMMENT_MOVE,
	COMMENT_UPDATE
DOCUMENTATION	DOC_DELETE,
	DOC_INSERT,
	DOC_UPDATE
OTHERS	UNCLASSIFIED_CHANGE,
	DECREASING_ACCESSIBILITY_CHANGE,
	INCREASING_ACCESSIBILITY_CHANGE,
	ADDING_CLASS_DERIVABILITY,
	ADDING_METHOD_OVERRIDABILITY,
	REMOVING_CLASS_DERIVABILITY,
	REMOVING_METHOD_OVERRIDABILITY

Table 2.1: Categories of changes retrieved with ChangeDistiller.

tiller can detect. For example, the METHOD_DECLARATION category is comprised of 9 separate changes, namely: RETURN_TYPE_CHANGE, RETURN_TYPE_DELETER, RETURN_TYPE_INSERT, METHOD_RENAMING, PARAMETER_DELETE, PARAMETER_INSERT, PARAMETER_ORDER_CHANGE, PARAMETER_RENAMING, PARAMETER_TYPE_CHANGE.

We have extracted these source code changes both from the production and from the test code. In order to make the dataset as comprehensive as possible, we have included additional information such as: the class in which the change occurred, the version when the change was made along with its timestamp, and the exact source code entity that was modified.

2.1.2 Linking production and test code

Once we have the fine-grained changes, we link the test cases to the production code they cover. We prefer a dynamic solution over a static analysis approach because it is more precise as pointed out by Van Rompaey and Demeyer [42]. The key idea behind our approach is to run each test case separately, thereby identifying all the entities from the production code addressed by the test, similarly to the approach used in [19]. To retrieve the covered entities (*e.g.,* Java classes) we process test coverage information gathered with Cobertura[1]. Cobertura is utilized because it produces more accurate results compared to other code coverage tools, such as EMMA or Clover [6].

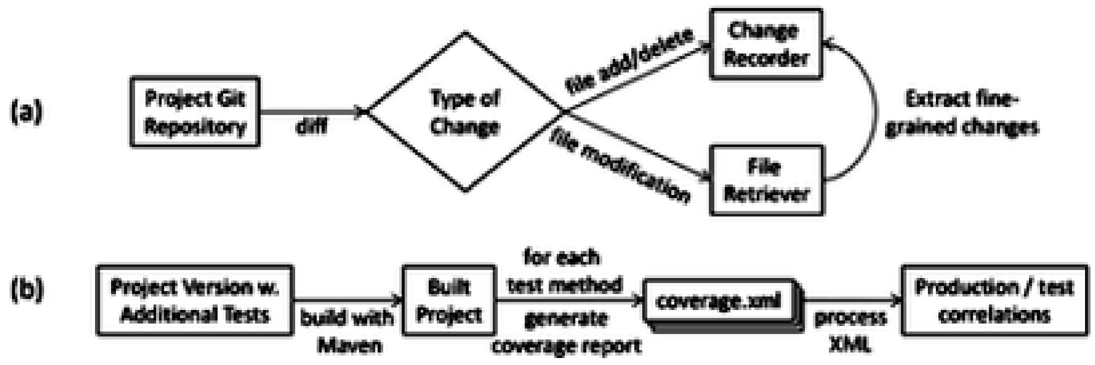


Figure 2.1: Overview of the data collection process.

2.2 Implementation

To implement our approach we use a process consisting of two steps that is described in Figure 2.1.

As a first step (see Figure 2.1(a)), we use the jGit API[2] to retrieve the software project's source code from the corresponding Git repository. Then, we compute the differences be-

[1]`http://cobertura.github.io/cobertura/` — last visited June 13th, 2014.
[2]`http://eclipse.org/jgit/` — last visited June 19th, 2014

tween two consecutive versions of the system using the same API. Based on the types of the changes retrieved between versions, one of the following two approaches is selected:

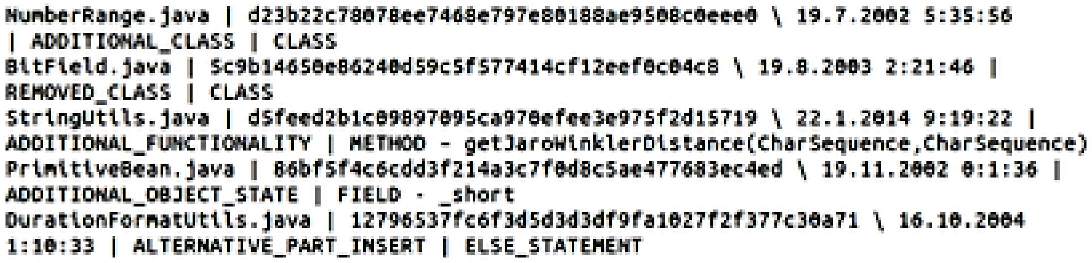
1. When entire Java classes are added or deleted, the names of the fields and the methods declared in those classes are recorded. A specialized parser[3] is used to extract these names from the corresponding class files.

2. Otherwise, ChangeDistiller is utilized to extract the fine-grained changes.

A specific procedure is applied to each project version for which the test code has been modified (shown in Figure 2.1(b)). We first compile the production code using Maven in order to ensure that it does not contain any errors. If the compilation is successful, the test cases of that version are run separately. For each test case, we let Cobertura generate a coverage report file. This file is then parsed with the jDom API[4] to identify the methods from the production code covered by the respective test method. We record these results which are used afterwards to determine the links between production classes and test cases.

2.3 Dataset structure

The obtained dataset consists of two main parts. The first one is related to the source code changes that are extracted. Figure 2.2 illustrates the structure of the data files containing production code changes. The data files with test changes have exactly the same structure. For each change, we record: (1) the class in which it occurs; (2) the commit during which the change is done along with its timestamp; (3) the type of the change; (4) the name and the type of the source code entity that is modified. We have made the dataset so extensive in order to obtain as much information as possible. For example, we have included the timestamp of each commit to be able to group commits together (*e.g.,* between two releases of a system). We have also added the exact code entity that is changed, thereby allowing for an easier identification of the parts of the source code in which a production change and its associated test changes occur.



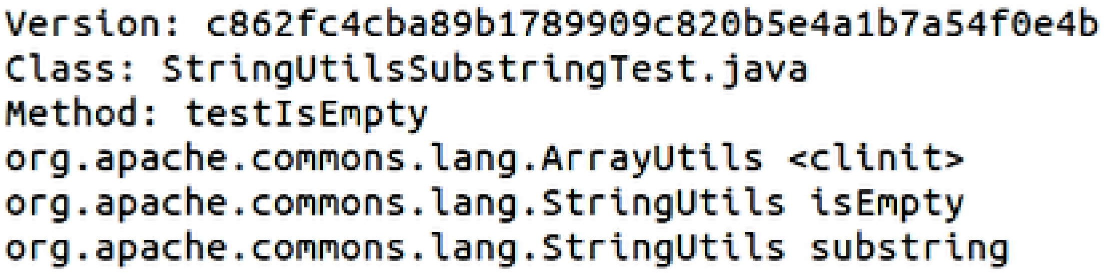
```
NumberRange.java | d23b22c78078ee7468e797e80188ae9508c0eee0 \ 19.7.2002 5:35:56
| ADDITIONAL_CLASS | CLASS
BitField.java | 5c9b14650e86240d59c5f577414cf12eef0c04c8 \ 19.8.2003 2:21:46 |
REMOVED_CLASS | CLASS
StringUtils.java | d5feed2b1c09897095ca970efee3e975f2d15719 \ 22.1.2014 9:19:22 |
ADDITIONAL_FUNCTIONALITY | METHOD - getJaroWinklerDistance(CharSequence,CharSequence)
PrimitiveBean.java | 86bf5f4c6cdd3f214a3c7f0d8c5ae477683ec4ed \ 19.11.2002 0:1:36 |
ADDITIONAL_OBJECT_STATE | FIELD - _short
DurationFormatUtils.java | 12796537fc6f3d5d3d3df9fa1027f2f377c30a71 \ 16.10.2004
1:10:33 | ALTERNATIVE_PART_INSERT | ELSE_STATEMENT
```

Figure 2.2: Dataset structure production code changes.

[3] `http://code.google.com/p/javaparser/wiki/UsingThisParseris` — Last visited June 19th, 2014.

[4] `http://www.jdom.org/` — Last visited June 19th, 2014.

The second part of the dataset comprises the methods from the production code that are covered by a specific test case. For each test case that was modified during the lifespan of the project, the following data is collected: (1) the name of the test case; (2) the test class of which it is part of; (3) the commit during which it was changed; (4) all the production methods and constructors that are addressed by the test case. This information can easily be used to establish links between the respective test case and the production code it covers. A sample of the data gathered for this part of the dataset is presented in Figure 2.3.



```
Version: c862fc4cba89b1789909c820b5e4a1b7a54f0e4b
Class: StringUtilsSubstringTest.java
Method: testIsEmpty
org.apache.commons.lang.ArrayUtils <clinit>
org.apache.commons.lang.StringUtils isEmpty
org.apache.commons.lang.StringUtils substring
```

Figure 2.3: Dataset structure production and test code links.

Chapter 3

Design of Empirical Study

This chapter describes the design of the empirical study that was conducted. It first discusses the goal of the study and the hypotheses that were formulated. Afterwards, the independent and dependent variables considered are explained, along with the procedures that were used to measure them. In the second part of the chapter, we present the approach taken when performing the quantitative and qualitative analyses.

3.1 Goal of Experiment

As explained in Chapter 2, the goal of this thesis is to study the co-evolution between the production and the test code of a software system. In order to achieve it, a series of research questions were formulated, namely:

- **RQ1:** What kind of fine-grained co-evolution patterns between production and test code can be identified?
- **RQ2:** Does the testing effort have an impact on the observed co-evolution patterns?
- **RQ3:** How does the co-evolution between production and test code happen?

The main goal of the experiment is to provide answers to these research questions. In order to do this, a mixed methods approach is followed [9]. First, we use an association rule mining algorithm to identify the co-evolution patterns of production and test code. Then, we interpret the obtained patterns by performing a qualitative analysis. The exact steps that were followed for each of the analyses are presented in the last two sections of this chapter.

3.2 Formulated Hypotheses

As discussed in Section 1.2, we made a series of assumptions with regard to production and test code co-evolution. The first major assumption was that there is a correlation between a change in a production class and one or more specific changes in the test code. If this assumption holds, then a number of fine-grained co-evolution patterns can be identified for a software system. The second assumption states that the effort put into testing for a project has an effect on the observed patterns. In order to determine whether these two assumptions are true or not, hypotheses corresponding to each of them were formulated:

Hypothesis 1

- **Null hypothesis ($\mathbf{H1_{null}}$):** There is no correlation between a change in the production code and any of the changes in the test classes.
- **Alternative hypothesis ($\mathbf{H1_{alt}}$):** A change in the production code triggers one or more specific changes in the test classes.

Hypothesis 2

- **Null hypothesis ($\mathbf{H2_{null}}$):** The testing effort put into a project has no impact on the co-evolution patterns.
- **Alternative hypothesis ($\mathbf{H2_{alt}}$):** Different co-evolution patterns can be observed for the projects that are thoroughly tested compared to the ones found for projects for which the testing effort is low.

The first hypothesis addresses **RQ1**, while the second one covers **RQ2**. These two hypotheses are the main focus of the empirical study. Determining whether the null or the alternative hypothesis holds in each of the cases is the primary objective of the experiments that are described in the following sections.

3.3 Independent and Dependent Variables

We have identified the independent and dependent variables for each of the hypotheses and developed methods through which they can be measured. Table 3.1 contains an overview of these two types of variables per hypothesis, along with their measurement procedures.

Hypothesis	Independent Variable	Procedure	Dependent Variable	Procedure
H1	Production change	Section 2.1.1	Test changes	Section 2.1.1
H2	Testing effort	Section 3.3.2	Observed co-evolution patterns	Section 3.3.2

Table 3.1: Independent and dependent variables.

3.3.1 Hypothesis 1

For **H1** the independent variables are the changes from the production code. We want to establish if they have an impact on the dependent variables, the test code changes. Both variables were measured following the procedure discussed in Section 2.1.1. For each production class from a version of the system, the changes that occur in that class and the specific changes they trigger in the corresponding test cases are recorded. These changes are part of one of the 7 categories of source code changes considered, as discussed in Section 2.1.1. Therefore, the outcome of the procedure is represented by the number of production changes from one category along with the number of changes that are triggered in the test code from each category of test changes; this is reported per production class for each version of the system.

3.3.2 Hypothesis 2

The independent variable for **H2** is the testing effort spent on a software project. The effort is estimated by combining a number of metrics that were collected for each of the projects under study. This can be regarded as a preliminary analysis of the systems included in the experiment. Four perspectives have been considered: (1) changes that occurred in the production / test code, (2) branch coverage obtained during the lifespan of the project, (3) number of versions that did not compile because of test failures, and (4) ratio between the amount of test code and production code. An overview of this preliminary analysis is shown in Table 3.2.

Project	Branch Coverage	Number of Non-Building Versions due to Test Failures	$\Sigma_{\forall v_i} LOC_{test}$ / $\Sigma_{\forall v_i} LOC_{prod}$
PMD	0.51418	369	0.130
CommonsLang	0.90678	54	0.442
CommonsMath	0.80254	131	0.366
JFreeChart	0.49274	17	0.219
Gson	0.66233	12	0.287

Table 3.2: Testing effort measurements.

The reported branch coverage has been recorded for the last version that we have considered (column Branch Coverage) and it was determined with Cobertura. We have also collected coverage information for each release of a project and observed that the overall branch coverage remains relatively stable.

The table also includes the number of versions that raised problems during compilation because of test failures (column Number of Non-Building Versions due to Test Failures). We have relied on Maven to compile the projects and recorded all the situations in which not every test from a version passed. Finally, we have calculated the ratio between the lines of test code and production code lines (globally, for all versions combined) as a measurement for quantifying the volume of testing that has been done for a system (column $\Sigma_{\forall v_i} LOC_{test}$ / $\Sigma_{\forall v_i} LOC_{prod}$).

Subsequently, we have analyzed the software projects to determine which types of changes occur in their production and test code. Table 3.3 contains an overview of these changes grouped into 10 categories corresponding to the 10 major types of changes identified by ChangeDistiller. For our analyses we only consider the first 7 categories of changes, as the last 3 are not related to the source code of a system. The total number of production and test code changes per software project has been calculated. In order to get an indication of testing “effort”, we have also determined the percentage of test code changes from the

ChangeDistiller category	PMD		CommonsLang		CommonsMath		JFreeChart		Gson	
	Prod	Test	Prod	Test	Prod	Test	Prod	Test	Prod	Test
ADDED_CLASS	4690	599	679	410	3074	1172	929	1128	130	124
REMOVED_CLASS	4993	9	591	2	1605	39	1339	0	108	11
CLASS_DECLAR	8742	1207	2396	2179	7379	4007	1777	847	542	460
METHOD_DECLAR	3038	169	1146	376	2730	709	641	399	286	64
ATTRIBUTE_DECLAR	7558	307	795	198	2787	746	890	33	330	27
BODY_STATEMENTS	107831	8179	12933	15924	44098	28260	16266	17705	4947	1134
BODY_CONDITIONS	16507	58	1466	85	2365	284	774	1	500	9
COMMENTS	2285	99	709	527	2762	1015	406	224	70	10
DOCUMENTATION	2363	88	3534	513	9145	468	449	401	212	15
OTHERS	1621	136	246	101	1051	236	14	394	114	97
TOTAL	159628	10851	24495	20315	76996	36936	23485	21132	7239	1961
$\text{Total}_{Test}/\text{Total}_{Test+Prod}$	6.37%		45.33%		41.10%		47.36%		21.32%	

Table 3.3: Total number of changes in the production / test code per ChangeDistiller change category.

total number of changes. This is depicted in the final row of Table 3.3. A visual representation of the data from Tables 3.2 and 3.3 is provided in Figure 3.1. A separate chart was created for each of the aspects considered when measuring testing effort. Figures a, b and c correspond to the columns of Table 3.2 while Figure d addresses the last row from Table 3.3.

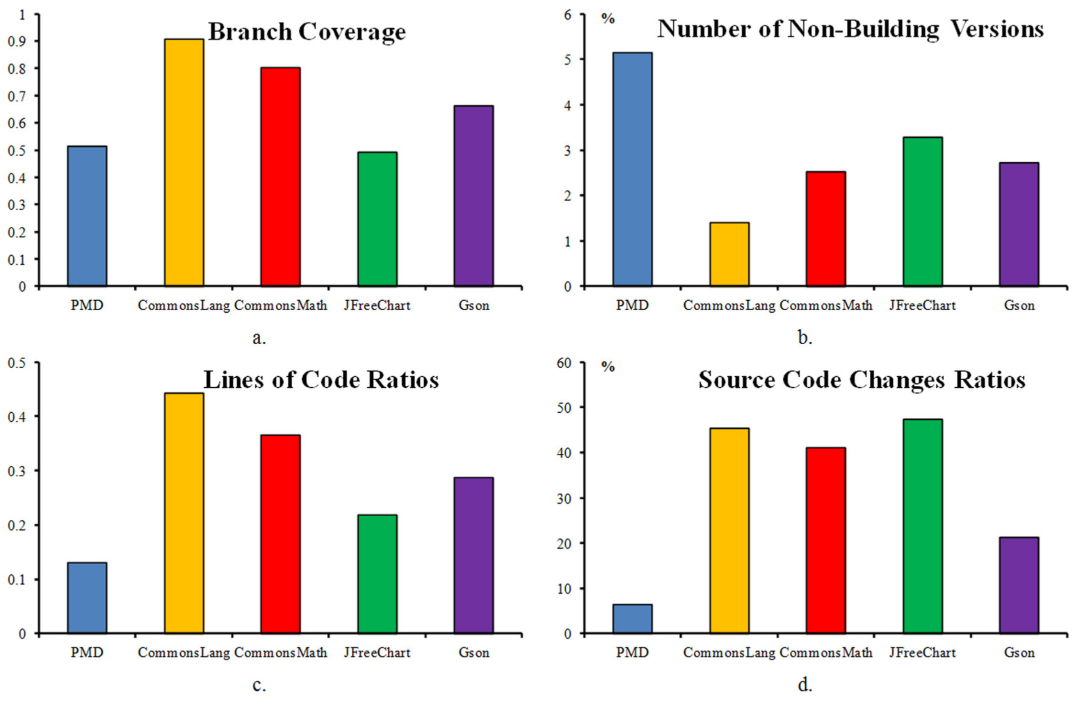


Figure 3.1: Testing effort measurements per aspect.

The following thresholds have been selected: 1) percentage of test changes - over 33%; 2) branch coverage - over 0.67; 3) percentage of non-building versions - less than 2.5%; 4) test code lines - over 0.25. A system is considered properly tested if at least 3 of the thresholds are met. Based on the above information which can be considered an indicator of testing effort, we classify the projects as extensively tested (*CommonsLang*), relatively well tested (*CommonsMath*, *Gson*) and rather poorly tested (*PMD*, *JFreeChart*).

The dependent variables in this case are the types of co-evolution patterns observed. Depending on the testing effort spent on a project, we expect different types of co-evolution patterns to appear. Therefore, based on the 3 categories of projects mentioned above, the co-evolution between production and test code can either be observed (*e.g.*, for the thoroughly and adequately tested projects) or not. For example, for the systems that are properly tested we anticipate that the creation of a production method will often trigger the addition of at least one test case, thus obtaining a pattern between the insertion of methods in the production code and test cases in the test classes. For the projects that are poorly tested we expect that this situation will not occur.

Confounding factors

For the second hypothesis, we want to determine whether different co-evolution patterns appear for systems with high testing effort compared to the ones encountered for poorly tested systems. It is clear however that testing effort is not the only factor that has an impact on the co-evolution patterns. There are a series of other factors that influence they way in which production and test code co-evolve. Even though it is difficult to identify and control all of them, we have tried to take into account a number of factors when conducting our analyses. The following two categories of confounding factors are considered:

- **Development practices:** The co-evolution patterns may differ depending on the development practices followed by the programmers that are working on a system. For example, there might be cases in which production code development happens for long periods of time which are immediately followed by sprints dedicated to testing. Distinct patterns are also obtained if integration testing is being done instead of unit testing; different association rules are generated if each of the methods added in a production class is covered by a separate test case compared to a case in which a single integration test is created to address all the production methods.

- **Project characteristics:** The specific characteristics of a software system also have an impact on the co-evolution between production and test code. As an example, the popularity of a particular project can have an influence on the way in which testing is done. If a project is popular more users provide feedback on the issues encounter while utilizing the system, thereby determining the developers to address them faster and create tests to ensure that they are fixed. Another confounding factor from this category is the specific type of the project. Different patterns might be observed for open-source systems compared to the ones encountered in commercial ones.

Project		First version				Final version considered			
	# Versions	# Classes	# Prod. Methods	# Test Methods	Release	# Classes	# Prod. Methods	# Test Methods	Release
PMD	7165	316	1846	340	11/2002	822	4418	1340	12/2013
CommonsLang	3856	31	373	318	12/2002	177	2442	2851	02/2014
CommonsMath	5174	83	758	501	12/2004	985	6548	6201	02/2014
JFreeChart	519	423	5790	1297	11/2006	701	7776	2403	03/2014
Gson	322	73	414	131	05/2008	142	719	1010	08/2012

Table 3.4: Overview of selected projects.

3.4 Project Selection

We have chosen 5 projects on which we conduct our empirical study. We rely on the criteria set by Pinto *et al.* in [33] to select the projects, namely:

- (1) a large number of versions
- (2) considerable size (in terms of production classes and methods)
- (3) an extensive JUnit test suite
- (4) be in active maintenance

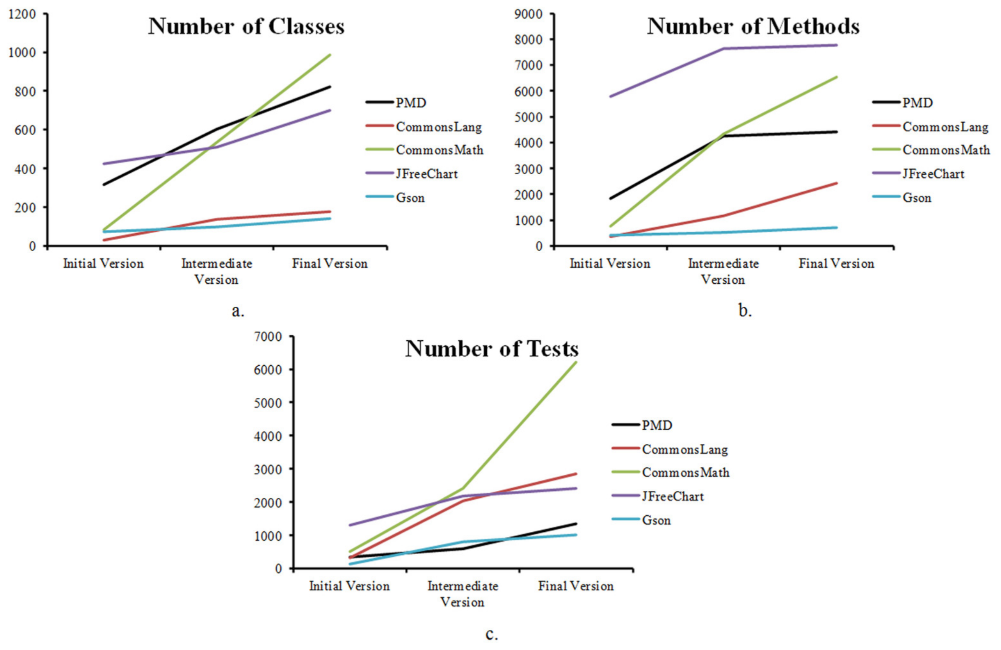


Figure 3.2: Project growth statistics.

For each of the systems, all their versions were included in the analysis. To the best of our knowledge, no study of this magnitude has ever been performed regarding the fine-grained co-evolution of production and test code.

An overview of the main characteristics of the 5 projects is presented in Table 3.4; it contains the total number of versions studied and shows metrics gathered for the first version of a project and the last version considered. The metrics collected are represented graphically in Figure 3.2. Besides the initial and the final version of a system, an intermediate version was also included when trying to illustrate how the 5 systems have grown in terms of number of classes, methods and test cases.

3.5 Analyses performed

We have performed our study following a mixed methods approach [9] that combines quantitative and qualitative analyses as described in the following subsections.

3.5.1 Quantitative Analysis

We first identify frequently occurring fine-grained co-evolution patterns between production and test code. The spmf[1] tool is used to generate a series of association rules. We have configured the Apriori algorithm with support and confidence values of 50% and 60%, respectively.

The following steps have been applied to obtain the rules. For each version of a system, all the changes that occur in the production code and in the associated tests are recorded per production class using a bucket list representation. Instead of the actual values, we use discrete values to quantify the number of changes that occur from each of the categories. We did this in order to facilitate the generation of the association rules, as it would not be possible to obtain rules with the specified support and confidence if numerical values were used. In the cases of class additions and removals, only YES and NO values have been utilized, as these kinds of changes can either happen or not. For the other types of changes, one of the following 5 values is assigned: NONE, LOW, MED_LOW, MED_HIGH or HIGH. In order to assign these values, the set containing the number of occurrences of the respective change in each class in which it was made (for all the versions of a system) has been constructed; extreme values have been filtered out, to prevent the results from being skewed. The last 4 discrete values (LOW, MED_LOW, MED_HIGH and HIGH) correspond to the (0%-25%], (25%-50%], (50%-75%], [75%-100%] intervals of values from this set. After we put the data in the appropriate format (version — production class — changes in class / associated test classes = value), the association rules are computed.

3.5.2 Qualitative Analysis

To refine the results from the quantitative analysis, we perform a qualitative study in order to better understand some of the co-evolution patterns that have been obtained during the previous analysis. We concentrate on the following 5 categories of production code changes:

[1]`http://www.philippe-fournier-viger.com/spmf/` — Last visited June 19th, 2014

added class, removed class, class declaration change, attribute declaration change, and body condition change. We disregard the other 2 categories because (1) a large variety of fine-grained changes are part of the METHOD_DECLARATION category, therefore it was difficult to find a substantial number of examples for each of them and (2) BODY_STATEMENT changes occur in almost every commit, thus making it hard to separate them from the other types of changes.

We carry out this qualitative analysis by studying concrete examples of test code changes that occur as a result of a particular change in the production code. From each category of production changes (see Table 2.1), we investigate every type of change in depth. For each occurrence of the change in a production class, all the changes it has triggered in the test code are recorded and analyzed. In order to ensure that there is indeed a connection between the production and the test code changes, the links between the respective production class and the corresponding test cases are inspected, along with the actual source code and the commit message of the project version under consideration. After gathering these examples, we make a series of observations based on them regarding (1) how co-evolution happens together with (2) an interpretation of the co-evolution patterns identified during the quantitative analysis.

Chapter 4

Quantitative Analysis

This chapter presents the results of the quantitative analysis. First, we discuss the obtained association rules. Afterwards, we explain the co-evolution patterns that can be inferred from these association rules for each project under analysis.

4.1 Association rules

A number of association rules have been generated for each category of production code changes. Figure 4.1 illustrates the structure of these rules. The antecedent is represented by the production change category that is addressed by the respective association rule along with the number of changes that occurred from that category. The consequent consists of a category of test code changes that is correlated with that particular production change category and the number of test changes encountered. The last part of the rule contains the support and confidence values obtained.

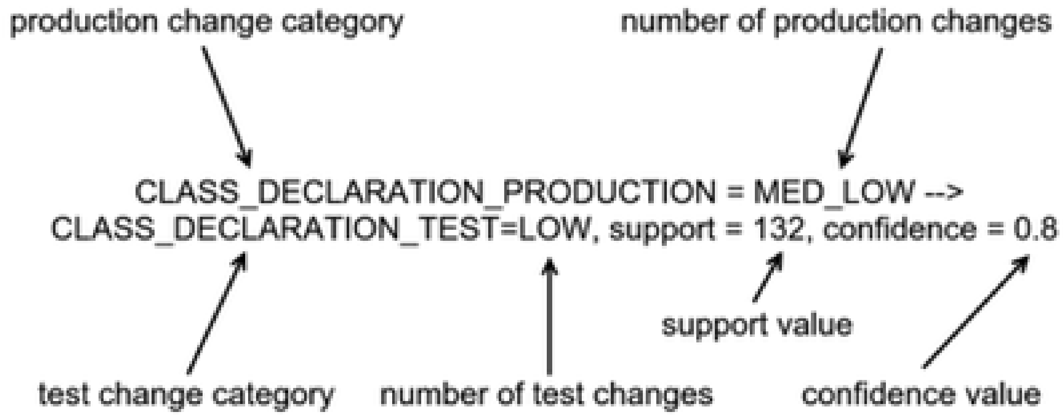

production change category
number of production changes
CLASS_DECLARATION_PRODUCTION = MED_LOW -->
CLASS_DECLARATION_TEST=LOW, support = 132, confidence = 0.8
support value
test change category
number of test changes
confidence value

Figure 4.1: Association rule format.

Table 4.2 provides an overview of the association rules for each of the 5 systems along with their support and confidence values. The second column (*i.e., Association Rule*) contains the retrieved association rule; however, the value of the consequent is missing in this column. This value can be found in the subsequent columns that are specific to each project under analysis. These columns contain the support and the confidence of the rule together with the value of the consequent (*e.g, YES*, *NO*, *SOMETHING*). For instance, for rule 1, the column *PMD* shows that no test classes are added (*i.e., NO*) when new production classes are created; the rule has a support value of 4161 and a confidence value of 0.906 for the respective project. Therefore, the complete set of association rules for a project is obtained by concatenating the second column with the specific column for that project.

In some cases an association rule has not been generated between a production and a test code change. This is caused by the fact that the changes in the production code are dispersed over a number of intervals, *i.e.* (*LOW*, *MED-LOW*, *MED-HIGH*, or *HIGH*), see for example the antecedents of rules 7 through 9 for *CommonsLang*. Because of this dispersion the threshold value for confidence might not be met, which in turn means that an association rule is not generated. However, if there was no link between the production and the test code change, we would expect that an association rule containing NONE as the value of the consequent would have been generated, thus indicating the lack of connection. The fact that this association rule with NONE is not produced suggests that there might still be a (weak) link, thing that we marked with the keyword *SOMETHING* in the consequent.

Table 4.2 also has some empty cells. This happens for some of the rules that have MED_LOW as the value for the production change. It is caused by the fact that the respective production changes generally occur only once for a class in a commit, therefore the intervals corresponding to (0%-25%] (LOW) and (25%-50%] (MED_LOW) of the values are identical (contain only 1 values).

As an example of mined association rules, consider the following two rules that address the addition of production classes for the *CommonsLang* project:

Association rule 1.1

ADDED_CLASS_PRODUCTION=YES → ADDED_CLASS_TEST=YES

support: 412, confidence: 0.64375

This first association rule indicates that for *CommonsLang*, a project that has been categorized as extensively tested, the creation of a new production class leads to the addition of a corresponding test class in around 64% of the cases.

Association rule 1.2

ADDED_CLASS_PRODUCTION=YES → CLASS_DECLARATION_TEST=NONE

support: 557, confidence: 0.87031

The second rule reveals that sometimes when a production class is created additional test cases are developed in the already existing test classes in order to cover it. Even though the value of CLASS_DECLARATION_TEST is NONE, the confidence of this rule indicates that in roughly 13% of the cases a different value than NONE was registered; therefore, in these situations at least one test case is created when the new production class is added.

The rest of the association rules are discussed in Appendix A, at the end of this document. That section contains a detailed analysis of each of the association rules and also goes more in depth with regard to their implications. Furthermore, several association rules that have not materialized into patterns are also included.

4.2 Co-evolution patterns

By inspecting the association rules from Table 4.2 we have noticed a number of interesting differences between the systems under analysis. We have identified 6 co-evolution patterns (shown in Table 4.1) that we distilled from Table 4.2 by generalizing what has been observed for the 5 projects. These 6 co-evolution patterns are further subdivided into two categories: *positive*, marked by the *a* suffix, and *negative* as exemplified by the *b* suffix. The positive patterns reflect co-evolution, while the negative ones point towards a lack of co-evolution.

Pattern	Explanation	CommonsLang	CommonsMath	PMD	Gson	JFreeChart
1a	When a new production class is added, an associated test class is also created	√	√		√	
1b	When a production class is created, no new class is added in the test code			√		√
2a	Upon the deletion of a production class, its associated test class is also removed	√	√	√	√	√
2b	When a class from the production code is removed, the test class covering it is not deleted					
3a	When a production method is created / deleted, one or more test cases addressing it are also developed / discarded	√	√		√	
3b	Upon the addition / removal of a method in the production code, no test cases are created / deleted			√		√
4a	When method-related changes occur in the production code, the tests are updated accordingly	√	√		√	
4b	When changes are made to the signature or return type of a production method, no changes occur in the test code			√		√
5a	When a field is added / removed in the production code, the existing test cases are updated in order to address this change	√	√		√	
5b	When a new production field is added / removed, no modifications occur in the test code			√		√
6a	Upon modifying conditional statements in methods from the production code, test cases are created / deleted to cover each possible path throughout the respective method	√	√		√	√
6b	When conditions are changed in production methods, no test cases are added / removed			√		

Table 4.1: Co-evolution patterns for each system under study.

We will now discuss these co-evolution patterns per project.

CommonsLang

In the case of *CommonsLang*, an extensively tested project, a series of co-evolution patterns (we are referring to the numbering of Table 4.1) have been observed, namely:

Pattern 1a The generated association rule shows that corresponding test classes are indeed created when new production classes are developed (confer rule 1 in Table 4.2, CONF = 0.643), suggesting that the developers actually test the production code they write.

Pattern 2a Another rule has uncovered that in most of the cases test classes are removed (rule 2, CONF = 0.998) when the production classes they cover are deleted, indicating that the programmers are careful not to leave non-compiling test classes in the system.

Pattern 3a The rules highlight that when a certain number of methods are added / removed from production classes corresponding test cases are also created / deleted (rules 4–6).

Patterns 4a and 5a They also show that test cases are updated accordingly when attribute or method related changes are made in the production code (rules 7-14).

Pattern 6a Finally, the association rules uncovered that test cases are created / removed when conditional statements are changed in the production classes (rules 16-18).

The patterns presented above indicate that thorough testing has been done for *CommonsLang*, which is in concordance with the initial observations that we made regarding this system.

CommonsMath

In general, the association rules generated for *CommonsMath* resemble the ones obtained for *CommonsLang*; however, from the confidence values retrieved, it can be observed that less emphasis has been put on testing for this project.

Patterns 1a and 3a For example, when new production classes / methods are developed corresponding test classes / cases are created, but their number is slightly lower than in the *CommonsLang* case (rules 1 and 4–6 respectively).

Pattern 2a Associated test classes are removed when production classes are deleted (rule 2, CONF=0.974).

Patterns 4a and 5a Test cases are altered accordingly in situations when the attributes / methods of a production class are modified (rules 9–10 and 13–14).

Pattern 6a Tests are written / dropped when conditions are changed in the production code (rules 15–18).

Even though *CommonsMath* is not tested in such detail as *CommonsLang*, the project can still be considered adequately tested as only positive co-evolution patterns occur.

PMD

Most of the association rules that were generated for *PMD* are negative (*i.e.,* have NONE as the value for the test related changes).

Pattern 1b We see strong indication that for *PMD* test classes are not developed when production classes are created (rule 1, CONF=0.906).

Patterns 2a In the few cases in which a production class has an associated test class, the two classes are deleted together (rule 2, CONF=0.998).

Patterns 3b Test cases are rarely created when production methods are added (rules 3 and 5-6). In about 80% of the cases the development / removal of a method from a production class does not trigger any changes in the test code.

Pattern 4b and 5b In the cases when attributes or methods from the production classes are modified tests are rarely changed; only a limited amount of updating is done in the test code to ensure that the test cases still compile (rules 7, 9-10, 11, and 13-14).

Pattern 6b Also, test cases are not created / deleted when conditional statements are modified in production methods (rules 15 and 17-18).

From the patterns that were inferred, it is clear that *PMD* does not have a structured approach to co-evolving production and test code. This observation is in-line with our initial assessment that *PMD* is a poorly tested project.

Gson

In most cases, the rules generated for *Gson* are similar to the ones obtained for *CommonsLang* and *CommonsMath*.

Patterns 1a The creation of a new production class is strongly correlated with the addition of an associated class in the test code (rule 1, CONF=0.772).

Patterns 2a Similarly, the deletion of a production class is immediately followed by the removal of its associated test class (rule 2, CONF=0.936).

Pattern 3a In contrast to the aforementioned *CommonsLang* and *CommonsMath*, for *Gson* we have found that when methods are added / removed from production classes, the number of test cases created / deleted is significantly lower in comparison to the other two projects (rules 5-6); nevertheless, a positive sub-pattern was still detected.

Patterns 4a and 5a In terms of changes concerning production methods or attributes, the appropriate updates are made to the existing test cases (rules 9-10 and 13-14).

Pattern 6a Also contrasting *CommonsLang* and *CommonsMath*, only when numerous condition related changes are made in the production methods, test cases are created / deleted in order to address the additional / removed branches (rule 18).

We conclude that *Gson* can be regarded as a well-tested project as most of the changes in the production code are accompanied by changes in the test classes.

JFreeChart

JFreeChart is a project that is not tested as extensively as *CommonsLang*, *CommonsMath* or *Gson*. Generally, the association rules that have been obtained in this case resemble the ones that were generated for *PMD*.

Patterns 1b and 3b Even though new production classes / methods are not backed up by additional test classes (rule 1) / cases (rules 3 and 5–6), we still see that the testing effort put into *JFreeChart* is higher compared to *PMD*'s case, because the negative association rules have a lower value for confidence.

Patterns 2a As for all the other analyzed projects, the removal of a production class triggers the deletion of its associated test class if such a class exists (rule 2, CONF=0.954).

Patterns 4b and 5b We observe that test cases are rarely updated when changes related to attributes or methods are made in the production code (rules 7, 9–10, 11, 13–14).

Pattern 6a In several cases we have noticed that test methods are created / deleted when conditional statements are altered in the production classes (rule 18).

The amount of testing that has been done while developing *JFreeChart* is on the low side, as indicated by the numerous negative association rules that were obtained. However, the testing effort is still considerably higher compared to the *PMD* case; in some situations the positive sub-patterns were encountered, and even when the negative ones were detected the confidence values of the association rules are lower than for the PMD rules.

Id	Association Rule	PMD	CommonsLang	CommonsMath	JFreeChart	Gson
1	ADDED_CLASS_PRODUCTION=YES → ADDED_CLASS_TEST	NO 4161/0.906	YES 412/0.643	SOMETHING -/-	NO 832/0.805	YES 85/0.772
2	REMOVED_CLASS_PRODUCTION=YES → REMOVED_CLASS_TEST	YES 4926/0.998	YES 569/0.998	YES 1554/0.974	YES 1331/0.954	YES 89/0.936
3	CLASS_DECLARATION_PRODUCTION=LOW → CLASS_DECLARATION_TEST	NONE 1767/0.998	NONE 244/0.953	NONE 1129/0.981	NONE 360/0.954	NONE 159/0.975
4	CLASS_DECLARATION_PRODUCTION =MED_LOW → CLASS_DECLARATION_TEST	-	LOW 132/0.8	SOMETHING -/-	-	-
5	CLASS_DECLARATION_PRODUCTION =MED_HIGH → CLASS_DECLARATION_TEST	NONE 855/0.808	SOMETHING -/-	SOMETHING -/-	NONE 174/0.754	LOW 43/0.651
6	CLASS_DECLARATION_PRODUCTION =HIGH → CLASS_DECLARATION_TEST	NONE 503/0.797	HIGH 85/0.658	SOMETHING -/-	NONE 102/0.743	SOMETHING -/-
7	METHOD_DECLARATION_PRODUCTION =LOW → BODY_STATEMENTS_TEST	NONE 634/0.725	SOMETHING -/-	NONE 331/0.614	NONE 134/0.814	NONE 60/0.833
8	METHOD_DECLARATION_PRODUCTION =MED_LOW → BODY_STATEMENTS_TEST	-	SOMETHING -/-	-	-	-
9	METHOD_DECLARATION_PRODUCTION =MED_HIGH → BODY_STATEMENTS_TEST	NONE 309/0.887	SOMETHING -/-	SOMETHING -/-	NONE 65/0.893	SOMETHING -/-
10	METHOD_DECLARATION_PRODUCTION =HIGH → BODY_STATEMENTS_TEST	NONE 234/0.823	MED-HIGH 37/0.616	SOMETHING -/-	NONE 49/0.871	SOMETHING -/-
11	ATTRIBUTE_DECLARATION=LOW → BODY_STATEMENTS_TEST	NONE 1244/0.737	SOMETHING -/-	NONE 558/0.722	NONE 148/0.833	NONE 82/0.828
12	ATTRIBUTE_DECLARATION=MED_LOW → BODY_STATEMENTS_TEST	-	SOMETHING -/-	-	-	-
13	ATTRIBUTE_DECLARATION=MED_HIGH → BODY_STATEMENTS_TEST	NONE 528/0.907	SOMETHING -/-	SOMETHING -/-	NONE 62/0.853	SOMETHING -/-
14	ATTRIBUTE_DECLARATION=HIGH → BODY_STATEMENTS_TEST	NONE 628/0.834	SOMETHING -/-	SOMETHING -/-	NONE 74/0.822	SOMETHING -/-
15	BODY_CONDITIONS_PRODUCTION=LOW → CLASS_DECLARATION_TEST	NONE 1044/0.976	NONE 126/0.670	SOMETHING -/-	NONE 94/0.853	NONE 72/0.9
16	BODY_CONDITIONS_PRODUCTION =MED_LOW → CLASS_DECLARATION_TEST	-	SOMETHING -/-	-	-	-
17	BODY_CONDITIONS_PRODUCTION =MED_HIGH → CLASS_DECLARATION_TEST	NONE 357/0.952	SOMETHING -/-	SOMETHING -/-	NONE 134/0.763	NONE 37/0.822
18	BODY_CONDITIONS_PRODUCTION=HIGH → CLASS_DECLARATION_TEST	NONE 430/0.926	SOMETHING -/-	SOMETHING -/-	SOMETHING -/-	SOMETHING -/-

Table 4.2: Associations rules mined from the evolution of the analyzed projects.

Chapter 5

Qualitative Analysis

The quantitative analysis has provided insight into the co-evolution of production and test code: we have identified 6 fine-grained co-evolution patterns for the 5 projects under analysis. We now turn towards a qualitative analysis that is aimed at 1) manually investigating how co-evolution happens and 2) interpreting the observed co-evolution patterns. Examples of test code changes that occur when specific changes are made in the production code have been manually analyzed. They provide in depth knowledge on the way in which co-evolution takes place and also aid in understanding the reasons why co-evolution is lacking in some cases.

5.1 How Co-Evolution Happens

This part of the qualitative analysis is concerned with determining whether the production and test code co-evolve simultaneously, if the test code is modified at a later time, or if no changes occur in the test classes. This aspect has been studied for all 5 production changes considered for the qualitative study, and different observations have been made for each of them. We have focused on the projects for which positive co-evolution patterns have been obtained (*CommonsLang*, *CommonsMath*, *Gson*), as they offer more information to study in comparison to the other two systems. Nonetheless, a series of insightful facts have been recorded for the poorly tested projects as well. In the following subsections, we present how co-evolution happens when a particular change is made in the production code. We discuss this in detail (with examples) for *CommonsLang* and summarize our findings for the other systems.

5.1.1 Class addition

In terms of production class additions, distinct things have been observed for the 5 projects under study. In particular, for *CommonsLang* we have determined that in most cases a new test class is indeed created when a production class is developed. We have come across the following 4 scenarios:
1) Occurs in the same commit: The test class is generally added during the same commit (in roughly 90% of the cases), thus suggesting that the developers actually test the

new production code before committing it. It has also been observed that the production class and its associated test class follow a specific naming pattern. As an example, in commit d23b22c78078ee7468e797e80188ae9508c0eee0 we have found that the classes `NumberRange` and `NumberRangeTest` are added together. Most of the examples gathered for this system are similar to the one illustrated above.
2) Occurs in a following commit: We have noticed situations in which the corresponding test class is developed during a following commit. This indicates that even though the production class was not tested at the time of its creation, the respective production code is still covered (at a later time). For example, we have determined that the addition of the production class `HashCodeUtils` was done in commit 7459257c3cd0f8dbc066520e825d995af332 5f6e on August 1st, 2002 at 22:15:43, while the corresponding unit test class was committed on August 10th, 2002 at 14:12:49.
3) Does not occur, but a different type of change is made in the test code: We have also identified cases in which a multitude of different types of test changes occur when a new production class is created. This corresponds to a scenario in which the developers update the already existing tests instead of developing a separate test class to address the production class that was added. We have observed the following changes in the test classes: the creation of test cases (corresponding to association rule 1a2), the insertion of statements containing method calls and the addition of catch blocks. A concrete example is the addition of the `TypeUtils` class in the production code, which triggers the creation of a test case called `testParameterize()` in the `UtilsTest` test class during the same commit (26d158fb242f45af145c43a70af01ded8673f5f4).
4) Does not occur: In some cases we have witnessed that a test class is not added when a production class is developed (in about 35% of the total number of cases). When such a situation occurs, the production code corresponds to either a mock class, an abstract class / an interface, or a class that is reimplemented (for which a test class does exist). For example, the creation of the `AnotherChild` mock class in commit d96fe035d3ac4df9a8aaef99124bd0 eff6430927 does not cause any changes in the test code. Cases in which important production classes were not covered by tests have rarely been seen for *CommonsLang*.
CommonsMath and Gson: For these two systems, when production classes are added the co-evolution takes place similarly to the CommonsLang case. The corresponding test classes are usually added during the same commit. In most of the cases in which a test class is not created, other types of changes are observed in the test code, including additional test cases and statement inserts. Situations in which important production classes are not covered by tests have rarely been encountered. However, they do appear more frequently for these two projects than for *CommonsLang*.
PMD and JFreeChart: On the other hand, for the systems that are tested less extensively, corresponding test classes are not added when production classes are created. In the cases in which new test classes are developed, they cover either a single production class or contain a number of integration tests. There are more situations in which these test classes are added in subsequent commits compared to the 3 projects that were previously discussed.

5.1.2 Class removal

1) Occurs in the same commit: In most of the cases (association rule 2a1) the removal of a production class triggers the deletion of its associated test class. This generally occurs in the same commit because otherwise the respective test class would produce errors during compilation.
2) Occurs in a following commit: Cases in which a test class is removed in a subsequent commit are rare. However, this is normal considering that the tests get invalidated if the production class they address is deleted. As an example, `HashCodeBuilder` was removed in commit c89cc3761f7ae49da9a6376923bd4913c7658563, while its corresponding test class was discarded in the following commit, 3aaf286c8ad87695b9f24e923ef5729d02dbce2e, several hours later.
3) Does not occur, but a different type of change is made in the test code: A production class deletion can also determine other types of test code changes. For example, a number of test cases from multiple test classes might be removed (2a2) or method-level changes (*e.g.,* statement deletions) can occur. The deletion of class `ExtendedMessageFormatBaseline` in commit 0b83f28c839deefe76a23131d0763900948527e7 for instance determines the removal of two test cases from `TextTestSuite` during the same commit.
4) Does not occur: For the situations in which no test classes were deleted, it was established that the removed production classes were either: (1) mock classes; (2) created and deleted after a few commits, therefore no tests were ever developed for them; or (3) removed and immediately reimplemented afterwards, so the corresponding test class is still valid. For example, the deletion of the `ClassUtils` class from commit ce1598398f55d046b02c26cf8b f15e47cc48639f is not correlated with any class removals in the test code.
CommonsMath and Gson: As for *CommonsLang*, a production class and its associated test class are removed simultaneously. The developers are careful not to leave a test class in the system after its corresponding production class was removed, thus avoiding compilation errors. Other types of test code changes were also identified, such as the removal of test cases or statement-level deletions and updates. These kinds of changes were made in test classes that do not address a specific production class, but rather cover a number of production classes.
PMD and JFreeChart: This is the only production change for which co-evolution happens similarly for these two projects as for the ones with a higher testing effort. If a corresponding test class does exist, it is removed in the same commit as the production class it covers. Few cases in which a test class is deleted in a following commit have been encountered. However, more cases in which other types of changes (*e.g.,* test case deletions or statement-level changes) were made in the test code have been identified compared to *CommonsLang*, *CommonsMath* or *Gson*. This is normal considering the fact that *PMD* and *JFreeChart* have more classes that contain integration tests than the other 3 systems studied.

5.1.3 Method addition

1) Occurs in the same commit: Most of the times (association rules 3a2-3a4), when new methods are added to production classes, additional test cases that address them are created

in the same commit. This type of situation occurred most frequently in the case of *CommonsLang*.
2) Occurs in a following commit: There are cases in which the corresponding test methods were created in a following commit. However, they rarely occur for *CommonsLang*, as test cases are generally added in the same commit as the production methods they cover. As an example, the addition of the `removeFinalModifier(Field,boolean)` method to class `FieldUtils` during commit edd0e3b9e294789421872f4d0b59901e1c401608 triggered the creation of the associated test case `testRemoveFinalModifierWithAccess()` in class `FieldUtilsTest` two commits after (in commit f241d34ad4a3e06abd3112928102c99c89b bf0e3).
3) Does not occur, but a different type of change is made in the test code: It was also determined that other types of test code changes can occur when production methods are added. For example, there are cases (association rules 3a6-3a8) in which a STATEMENT_INSERT type of change (corresponding to a method call) was made in an already existing test method. For instance, the creation of the `isAccessible(Class<?>)` method in production class `ConstructorUtils` (commit 832a250c0d9eba74c2f1d23a127c68ca28bab18b) causes the insertion of a statement (containing a method call to the respective production method) in one of the test cases from class `ConstructorUtilsTest` during the same commit. Also, an entire test class might be created if none of the existing test classes cover the production class in which the new methods were added.
4) Does not occur: It was difficult to identify cases in which newly created production methods where not addressed by test cases. For *CommonsLang*, this situation rarely happens, and when it does the added functionalities in the production code correspond to auxiliary methods or are part of a mock class, therefore the fact that they were not covered does not represent a serious issue. For example, when the `doIt()` method was added to mock class `Bar` (commit 7b86e4e054ab4507709795b6f63410ff03aeac92), no changes were observed in the test code.
CommonsMath and Gson: The general impression is that the co-evolution take place similarly as for the *CommonsLang* project, even though more cases in which an added production method was not addressed by at least one test case have been identified for these two projects. As for entire class additions, the corresponding test methods are usually developed in the same commit. The addition of production methods also determines other types of test code changes, especially statement-level inserts and updates (corresponding to calls to the respective production methods).
PMD and JFreeChart: For the two systems that have been labelled as poorly tested there are numerous cases in which the addition of a new production method does not trigger any changes in the test classes. For situations when corresponding test cases were created it has been found that they do not necessarily cover a single production method, but rather several production methods from more than one class. Therefore, for these projects, the JUnit guidelines that state that a test case should address a single functionality from the production code are disregarded. Furthermore, more situations in which other kinds of test code changes were made in the already existing test cases have been identified compared to the 3 systems discussed above.

5.1.4 Field addition

1) adding test cases: It has been established that the new test case does not specifically target the additional field from the production class. In general, it addresses one of the production methods that uses that respective field, thereby implicitly covering the field as well. For example, when the `fieldSeparatorAtStart` field was inserted in the `ToStringStyle` production class (commit 78c146100c9ba7f0454769e42e60ef7b523b572c) the `testAppendSuper()` test case was added to the `SimpleToStringStyleTest` class in the same commit. A deeper inspection revealed that the test case addresses the `appendSuper()` production method that uses the respective field.
2) updating existing test cases: Statement changes are the types of changes that have been the most commonly (association rules 5a1-5a3) observed when fields are added in the production code; this suggests that the developers update the tests classes accordingly when this type of change is made in production classes. Statement inserts, containing assignments or method calls, are the fine-grained changes that occurred the most. The addition of the production field `reflectionRegistry` in class `ToStringBuilder` (commit b0be90e86c346b320f52ad7b7065f110a7e1d272) triggers the immediate insertion of a statement containing an assignment to the respective field in the `ToStringBuilderTest` class.
3) no changes are made in the tests: For *CommonsLang*, case in which the addition of a field to a production class determines no changes in the tests were rarely (association rules 5a1-5a5) encountered. This shows that the developers have put a lot of effort into testing the project, thus resulting in a thoroughly tested system. The production fields that were not addressed by tests are either constants or auxiliary fields, so there was no pressing need to test them. For instance the `debug` field that is added in the production class `MethodUtils` during commit 1a6b33077c93f5d901788c6cc0df8eca89cc45c3 has remained untested.
CommonsMath and Gson: The addition of a field in a production class determines two types of changes in the test classes. Either one or more tests are modified so that the respective field is covered by the already existing test cases, or new test cases are created. However, for the second scenario, it was established that the new test cases do not explicitly address that field, but rather cover recently added production methods that use the field (*e.g.*, getters and setters). Cases in which no test changes occur as a result of the insertion of a production field have been observed more frequently than for other production changes (such as method additions). The test code changes mentioned above generally occur in the same commit as the production change that triggered them.
PMD and JFreeChart: For these two projects, the insertion of a production field is rarely backed up by changes in the test classes. When test changes are made, they correspond to statement inserts (in the existing test cases) in which the respective field is utilized. These changes usually happen in the same commit in which the production field is added.

5.1.5 Alternative condition block addition

1) adding test cases: Creating a new test case when an alternative block gets added to a condition in the production code is an action that is commonly (association rules 7a2-7a4) taken

for this project. This shows that *CommonsLang*'s developers adhere to the guidelines specified for JUnit testing, thereby obtaining a system that is properly tested. As an example, the insertion of an additional condition block in the `getReflectionArrayCycleLevel()` production method from class `ToStringBuilder` (commit b0be90e86c346b320f52ad7b7065f110a7e1d272) co-occurs with the creation of a new test case, `testReflectionArrayCycleLevel2()`, in the `ToStringBuilderTest` test class.
2) updating existing test cases: There are also situations in which changes are made to test cases that were created in previous commits. Upon further inspection it was observed that these test cases are actually integration tests. The changes that occurred most frequently (association rules 7a5-7a8) were at a statement-level (inserts and updates). For instance, when an alternative condition block is added in the `formatDate()` method from class `FastDateFormat` (commit c2003e4aa91120db3b91cdc659aded992bc68f58) a statement containing a method invocation to the respective production method is immediately inserted in the `testFormat()` test case from class `FastDateFormatTest`.
3) no changes are made in the tests: In some occasions, the developers do not make any kinds of changes in the tests when alternative blocks are added to conditional statements in the production code. Even though these situations occur rarely (association rules 7a2-7a4), the production changes still need to be addressed in order to avoid any types of problems that might arise. When alternative condition blocks were inserted in the production methods from classes RandomStringUtils (commit 11a540bcf9dcf08b238917e4fd0fb843c34f1f56) and StringEscapeUtils (commit 7eba6afba9a9e1c934985a494725f9470d34c9ec) no changes were done in the corresponding test cases.
CommonsMath: For this project, the situations observed are similar to the ones from the *CommonsLang* case. New tests are indeed developed when alternative condition blocks are added in the production code. There are also cases in which the developers updated the existing test cases instead of creating new ones when this type of production change occurs.
PMD, JFreeChart and Gson: For the other 3 systems, the introduction of an alternative condition block in a production method does not correlate with any types of changes in the test classes (especially for *PMD*). In the cases of *JFreeChart* and *Gson* there are some situations in which the respective production change triggered either the addition of new test cases or updates in the already existing ones. However, these situations rarely occur compared to the other two projects that were discussed above, *CommonsLang* and *CommonsMath*.

The examples listed above show that different things can happen in the test classes as a result of a change in the production code. They also demonstrate that in the cases when a change is made in the tests, it does not necessarily happen in the same commit as the production change that triggered it; therefore, a number of subsequent commits have to be inspected in order to ensure that all the test changes that occur due to a specific production change have been identified. Furthermore, this qualitative analysis has lead to other insightful findings; for example, if no changes are observed in the tests when a production class is created, the analysis uncovered examples of reasons why changes are not necessarily needed in those particular situations. The following section details more on the reasons why co-evolution does not happen in some specific cases.

5.2 Interpretation of the fine-grained co-evolution patterns

As explained in Chapter 3, for 5 of the changes that occur more frequently in the production code, we investigate the associated changes that are made in the test classes in greater detail. The considered changes are: (1) class addition, (2) method addition, (3) class removal, (4) field addition and (5) alternative condition block addition. For each of these types of changes, we collect examples of test changes that they trigger and study them.

5.2.1 Class addition

In terms of entire production class additions, we observe a number of interesting facts. First of all, we see that the addition of a production class causes the creation of a corresponding test class for the projects that are adequately tested (rule 1 in Table 4.1). When this does not happen, we have determined that the new production classes are either auxiliary classes or abstract classes / interfaces for which the classes that extend / implement them *are* tested. Another situation that we have witnessed is that production classes are removed and subsequently added again (therefore a test class already exists for them). For the other two projects, *PMD* and *JFreeChart*, the development of corresponding test classes was observed less frequently; the developers seem to prefer adding test classes that contain integration tests which cover multiple production classes that were recently created. For all the systems that we have analyzed, the new test class is generally developed in the same commit as the production class it addresses. Additionally, we have found other types of changes in the test code when production classes are created. The most commonly observed ones are (1) the addition of new test cases in the already existing test classes and (2) statement-level changes in some of the test methods. For the two systems that are tested less, these kinds of changes occur more frequently than the insertion of test classes.

5.2.2 Method addition

We also zoom in on the changes that are made in the test code when production methods are added. Intuitively we understand that the creation of a method in the production code should trigger the addition of at least one new test case. However, this expectation is fulfilled by only 3 of the analyzed projects, *CommonsLang*, *CommonsMath* and *Gson*; for the other two, this was rarely the case (rules 3-6). Even for the adequately tested projects, there are situations in which no changes are made in the test code when this type of change occurs in a production class. Upon further investigation we have established that the production methods that are not backed up by additional test cases are generally part of abstract or mock classes; therefore, the fact that they are not addressed does not represent a serious issue. Nevertheless, in some cases new utility methods have not been tested, thing that could prove problematic. In general, we see that the corresponding test cases are added in the same commit as the production methods they cover. There are few cases in which they are developed in a following commit. We have also found other types of test code changes, most of which are at a statement-level, corresponding to updates to the already existing test cases by inserting or modifying a number of statements.

5.2.3 Class removal

With regard to production class removals, we have determined for all the 5 projects that if an associated test class exists, it is also deleted (rule 2). However, we did find situations in which the test class is not removed in the same commit as the production class it covers. For example, in the case of *PMD*, the `TokenSetTest` class is discarded two commits after `TokenSet` is deleted. This is particularly interesting considering the fact that compilation errors arise because the production class that is being tested was already removed. Changes from other categories have also been identified in the test code. For example, statement deletions and updates have been encountered in the tests, suggesting that test cases that address more than one production class are modified accordingly. In some cases we have noticed that methods from multiple test classes are removed, indicating that the respective production class was covered by more than one test class.

5.2.4 Field addition

We have also inspected the addition of fields in production classes. We have observed several types of changes in the test code in this case, especially for the systems with a higher testing effort. In a number of cases adding a field in the production code co-occurs with the creation of a new test case. A deeper inspection revealed that the test case does not specifically address the respective field, but rather a production method that uses it. We have also noticed statements being inserted in the existing tests, with which the field from the production class is covered. In some cases, a field is added in a test class as well; it corresponds to one of the fields introduced in the production code and is used all throughout the tests. For the projects that are tested less, *PMD* and *JFreeChart*, all the test changes mentioned above occur less frequently compared to the other 3 systems. Especially in the case of *PMD*, the developers seem to completely disregard this type of production change when it comes to testing, as generally no changes can be observed in the test classes.

5.2.5 Alternative condition block addition

Finally, we have studied the insertion of alternative condition blocks. For the adequately tested projects we have seen two types of changes in the tests. First and foremost, new test cases are usually created when alternative condition blocks are added in the production code (rules 15-18). This indicates that the developers adhere to the guidelines specified for unit testing which state that a test case should be created for each independent path through the tested method. However, there are situations in which they altered existing test cases instead of adding new ones. Through code inspections we have determined that various statement-level changes are done in the tests, such as modifying the values of the parameters with which a production method is called in order to trigger a different path through the respective method. We have rarely observed cases in which no changes are made in the test code for two of the systems, *CommonsLang* and *CommonsMath*. For the other 3 projects, *Gson*, *PMD* and *JFreeChart*, our findings are significantly different. Although there are some situations in which the existing test cases are changed when alternative condition blocks are inserted in production methods, in general new test cases are not created. Most of the times the developers do not make any kinds of changes in the test code for these projects.

Chapter 6

Discussion

In this chapter we first summarize our findings with regard to the 3 research questions addressed by this thesis, while also examining the hypotheses that have been confirmed. Then we discuss threats to validity that might affect our study.

6.1 Revisiting the research questions

RQ1. What kind of fine-grained co-evolution patterns between production and test code can be identified? By using association rule mining we have observed 6 fine-grained co-evolution patterns in our 5 case study systems (see Table 4.1). These 6 patterns can be summarized as follows:

- **Patterns 1a and 1b** - simultaneous introduction of production and test class
- **Patterns 2a and 2b** - simultaneous deletion of production class and associated test class
- **Patterns 3a and 3b** - introduction / deletion of production method leads to the addition / removal of one or more test cases
- **Patterns 4a and 4b** - modification of production method leads to statement-level changes in the test cases
- **Patterns 5a and 5b** - production field changes lead to statement-level changes in the test cases
- **Patterns 6a and 6b** - conditional statement changes in the production code lead to the addition / deletion of test cases

In Chapter 3, we have formulated two hypotheses concerning this research question, namely:

- **Null hypothesis ($H1_{null}$):** There is no correlation between a change in the production code and any of the changes in the test classes.

- **Alternative hypothesis ($H1_{alt}$):** A change in the production code triggers one or more specific changes in the test classes.

Considering the fact that a number of co-evolution patterns have been obtained, it is clear that a change in a production class determines specific types of changes in the test code. This observation is further supported by the results of the qualitative analysis. For the 5 production changes that have been investigated in detail, a series of concrete examples of changes that they trigger in the tests were collected. As an example, for the addition of production methods, we have witnessed the creation of new test cases and the insertion of statements corresponding to method calls in the already existing tests. These kinds of examples have been found for all the projects considered, therefore we can safely reject **$H1_{null}$** and conclude that there is a correlation between a change in the production code and several specific changes in the test classes.

RQ2. Does the testing effort have an impact on the observed co-evolution patterns? As a first step, we have evaluated the testing effort put into each of the 5 case study projects. This has been done on the basis of 4 criteria: (1) the ratio between the number of lines of test code and the number of production code lines, (2) the number of versions that did not compile because of test failures, (3) branch coverage, and (4) the ratio between the number of changes in the test code and the total number of changes for the respective project. Based on these measurements, the systems have been classified as: extensively tested (CommonsLang), adequately covered (CommonsMath, Gson) and poorly tested (PMD, JFreeChart).

For each of the 6 patterns that we observed, we have distinguished a *positive* and a *negative* sub-pattern, namely the positive or “a” pattern in which co-evolution does occur and the negative or “b” pattern in which case the co-evolution was absent. From this classification, our main observation is that for the software projects for which we have seen high testing effort, *i.e.* CommonsLang, CommonsMath and Gson, the positive patterns are more likely to occur. Similarly, for the two systems for which we have observed a less intense testing effort (PMD and JFreeChart), the negative patterns are more common. However, there are cases in which a pattern from the *a* group can be found in a project with a lower testing effort; for example, for JFreeChart we have established that test cases are sometimes created / removed when conditional statements are modified in the production code.

Class removal is the only production change for which the same pattern has been identified for all the 5 projects; in this case, the associated test class is deleted as well, which is unsurprising considering the fact that it would cause errors during compilation if it was left in the code.

For the second research question, the following two hypothesis were considered:

- **Null hypothesis ($H2_{null}$):** The testing effort put into a project has no impact on the co-evolution patterns.

- **Alternative hypothesis ($H2_{alt}$):** Different co-evolution patterns can be observed for the projects that are thoroughly tested compared to the ones found for projects for which the testing effort is low.

We have evaluated the testing effort spent on the systems and distinguished between properly tested projects (CommonsLang, CommonsMath and Gson) and poorly tested ones (PMD and JFreeChart). We have also refined each of the co-evolution patterns observed into a *positive* and *negative* sub-pattern. Considering that in most cases the positive sub-patterns were encountered in adequately tested projects while the negative ones were found for projects with low testing effort, we are able to reject $\mathbf{H2_{null}}$. In conclusion, $\mathbf{H2_{alt}}$ holds, the testing effort put into a project does have an impact on the co-evolution between production and test code.

RQ3: How does the co-evolution between production and test code take place? An investigation into the way in which production and test code co-evolve has been carried out as part of the qualitative analysis. It uncovered 3 co-evolution scenarios: 1) the production change and the associated test code change are made simultaneously; 2) the test change is done in a subsequent commit; 3) a series of other types of test code changes are identified when the respective production change is made or no changes are observed in the test code. The frequency with which each of these scenarios occur depends on the system under study. As shown before, the testing effort put into a software project has an influence on the co-evolution between production and test code, therefore it implicitly affects these scenarios.

Another factor that determines the moment when a test change happens is the specific type of the production change that triggered it. While for some production changes the associated test changes occur either in the same or in a following commit, there are others for which the production and the test code always evolve simultaneously. In general, the addition of new source code entities (*i.e.*, classes, methods, attributes or conditional blocks) in the production code does not necessarily correlate with immediate changes in the test classes. Numerous cases have been identified in which the corresponding test changes are made in subsequent commits. For example, entire commits dedicated to testing have been found (especially for PMD) when a series of new test classes were developed and additional test cases were created in the existing classes. On the other hand, for the removal of source code entities, the second co-evolution scenario is rarely observed. This is normal considering that these changes would cause compilation errors in the test code if they were not immediately addressed. In the case of production code updates, co-evolution takes place differently depending on the specific type of the update. For example, when the signature of a production method is changed, the test cases are updated accordingly during the same commit. However, when a change is made to a conditional statement from a production method, the associated changes (*e.g.*, addition / removal of test cases) in the test classes might occur in a following commit.

Additionally, this in-depth analysis also goes into the reasons why sometimes the patterns are not upheld. For example, a test class is not added for a mock class (pattern 1b). Many other examples have been identified: test cases are not created when auxiliary methods are added in production classes (pattern 3b); no changes are made in the test code when production fields corresponding to constants are inserted (pattern 5b); when a conditional statement change occurs in a production method it might not trigger any types of test changes because the respective production method is not covered by tests (pattern 6b).

All in all, the 3 research questions addressed by this study have been answered successfully. The two analyses have provided meaningful insight into the co-evolution between production and test code: (1) 6 co-evolution patterns have been identified; (2) it was established that the testing effort put into a project has an impact on these patterns; (3) a series of co-evolution scenarios have been uncovered; and (4) reasons as to why co-evolution does not happen in some specific situations are provided.

6.2 Threats to validity

This section presents the factors that might be considered threats to the validity of the obtained results and explains how we tried to mitigate each of them. The guidelines introduced by Perry *et al.* in [31] have been followed, therefore the factors are grouped into 3 categories: construct, internal and external validity. The following threats to validity have been identified:

Construction threats

Assessing whether or not the independent and dependent variables chosen accurately model the hypotheses that were formulated.
Fine-grained production and test code changes: In order to study the co-evolution between production and test code we have decided to investigate the changes that occur in the test classes as a result of changes in the production code. A specific procedure was utilized to gather these fine-grained changes and to link the test cases in which they appear with the addressed production classes, which is discussed in Chapter 2. Construction threats might arise because of issues with the implementation of this procedure. In order to mitigate these threats, our approach has been thoroughly tested using a number of small examples to ensure that it works properly. Additionally, we have performed a manual inspection of the data obtained for each of the projects in order to be certain that the changes extracted and the coverage information inferred are correct.
Testing effort measurements: A series of metrics have been considered when assessing the testing effort put into each of the projects under study. However, this set is by no means complete; other metrics could have also been used to measure testing effort. Nonetheless, the metrics were selected based on specific criteria (as discussed in Chapter 3), therefore we believe that they provide a good indication of the amount of testing that has been done for each of the projects.

Internal threats

Establishing whether the changes in the dependent variables can safely be attributed to changes in the independent variables.
Confounding factors: Even though in Chapter 3 we tried to discuss all the factors that have an impact on the dependent variables, the list obtained might not be complete. There may be other factors that influence the results, but it is very difficult to identify and control all of them. We have taken into account as many factors as possible and tried to minimize their

influence on our findings.
Association rule generation: The support and confidence values used when generating the association rules might also represent an internal threat to validity. Different association rules would be obtained if different values for support and confidence were utilized, therefore the resulted patterns might also be different. We aim for the rules to be as reliable as possible, therefore we decided not to lower these thresholds.

External threats

Determining if the results are generalizable to settings outside the study.
Industrial systems: The biggest threat from this category is that our observations might not apply for other projects. More specifically, all 5 projects are open-source, therefore the results may not be generalizable for commercial systems. In particular, different patterns might be uncovered for industrial projects compared to the ones gathered for the 5 systems included in the analysis. As discussed in Chapter 3, the investigated projects have been chosen based on several criteria, therefore our findings should be valid for a wide range of software systems. Future replications of the study should rule out this threat to validity.
Granularity of study: Another threat may arise because of the granularity at which the systems have been studied. The results might be different if only a subset of versions were analyzed, *e.g.* official releases. However, considering that we tried to study the co-evolution as in depth as possible, the results are in concordance with our initial expectations for this empirical study. The proposed approach can easily be refined so that the systems are studied at a different level of granularity.
Development technology: The fact that all the analyzed projects have been developed in Java may also represent an external threat to validity. When creating each of these systems object-oriented principles were followed, therefore the obtained results might not be valid for projects developed using other programming paradigms, such as imperative or functional programming. However, the approach should apply for other development technologies that are based on object-oriented concepts (*e.g.,* C++ or C#). Additional experiments are needed in order to generalize the observations that were made to systems developed in these programming languages.
Project selection: Another external threat could concern the projects that were selected for the empirical study. For example, none of the chosen systems follow a different development methodology (*e.g.,* Test Driven Development). It is clear that for TDD completely different co-evolution patterns would be obtained as tests are written before production code development takes place and updated afterwards.
All in all, the proposed approach was applied to projects with different characteristics, all the versions of each system have been considered, the analyzed test suites contain a large number of test cases, and hundreds of thousands of production and test code changes have been gathered. Even though further investigation is needed in order to confirm the results, these initial findings can provide a strong basis on which new research can be conducted.

Chapter 7

Related Work

Studying the co-evolution between the production and the test code of a software system is a complex problem that can be decomposed into a number of tasks, namely: (1) data collection through repository mining, (2) fine-grained source code change extraction, and (3) linking test cases to production code. This chapter covers similar work that fellow researchers have done along these directions. It is divided into four sections: co-evolution of production and test code, source code evolution, source code change extraction and software repository mining.

7.1 Production and test code co-evolution

This section discusses publications that address the same topic as we do, the co-evolution between production and test code. Few papers have been identified from this category, as the particular topic has not been thoroughly investigated thus far.

Zaidman *et al.* have proposed a set of visualizations that aid in understanding how production code and (developer) test code co-evolve [47]. Their analysis is coarse-grained, as they only inspect whether a production / test file is added or changed, while our analysis is much more fine-grained. The authors have observed that the co-evolution does not always happen in a synchronized way, *i.e.*, sometimes there are periods of development followed by periods of testing. Three views were created by combining information extracted from versioning systems with data regarding the size of the system's components and with the test coverage reports that accompany the system. The Change History View depicts the commit behaviour of developers over time, both for production and for test code. The Growth History View presents the evolution of these two artifacts in terms of size. Finally, the Test Coverage Evolution View shows the test coverage of a system at specific moments in time. Two open-source (CheckStyle and ArgoUML) and one industrial project were used in order to determine different types of co-evolution scenarios and infer various development and test practices based on them. The results obtained were validated both internally, by inspecting the source code and the log messages created during development, and externally, by relying on the knowledge of the original developers of the systems.

Lubsen *et al.* have a similar goal, however they use association rule mining to determine co-evolution [23]. Their work is particularly close to ours, albeit they study the co-evolution at a file level, while we focus on more fine-grained changes. In particular, the authors use association rules in order to determine whether or not production and test code evolve simultaneously. Their main contribution is an approach that can be utilized to analyze the co-evolution between the production and the test code of a software system by inspecting the transactions obtained from version control. Additionally, they propose a set of co-evolution metrics including standard interest and strength that can be used to assess the extent to which the two software artifacts co-evolve. They validate the proposed approach with two software systems, one of which is open-source (Checkstyle), while the other is from industry (the SIG software system). Their results show that the distribution of programming effort can be either on both artifacts or strictly on one of the two. Furthermore, cases in which a more test-driven-like practice has been followed were also identified. By comparing the results for the two systems, it was established that the testing approaches followed when developing them are very different. For the open-source system it was determined that most of the commits contain only production code changes. This is caused by: 1) the development methodology used in which testing is generally done in phases outside of regular development; 2) a series of large commits during which the production code is beautified, therefore no changes are made in the test code. On the other hand, the system from industry employed a test-driven approach to development, therefore the co-evolution takes place differently compared to the previous case. The most notable weakness of the proposed approach is that if changes in testing practices occur over small periods of time they will not produce significant differences in the results because the approach always takes the entire history into account.

In response to observations on the lack of co-evolution, Hurdugaci and Zaidman [19] and Soetens *et al.* [39] proposed ways to stimulate developers to co-evolve their production and test code by offering specialized tool support. For example, in [19] the authors introduce TestNForce, a tool that aids developers in finding the unit tests that have to be modified and executed after a change in production code occurs. This tool was implemented as a plug-in for Microsoft Visual Studio 2010 and its main role is to create awareness and assist developers in identifying and remembering which tests address the specific part of the source code that is changed. TestNForce was created so that it can support 3 scenarios: showing covering tests, determining the test cases that need to be run after a production code change is done, and enforcing self-contained commits.

Soetens *et al.* [39] investigate whether or not fine-grained method-level changes in the base-code can be utilized to identify the subset of regression tests that have to be rerun. More specifically, their work is aimed at re-examining the existing test selection techniques in the context of developer tests. The authors created a prototype and applied it on two systems, PMD and CruiseControl, in order to evaluate the feasibility of the proposed approach. They use mutation testing and compare the selected subset of tests against the one obtained by using a "retest all" approach. The results prove that a considerable reduction in size is reached, while still killing a number of mutants that is comparable to the one reached by the entire test suite.

Previous studies have also investigated how other types of software artifacts evolve alongside one another. Fluri *et al.* [14] analyze the co-evolution between the source code of a software system and the developer comments. An approach is presented that can be utilized to link the comments to the corresponding source code entities, thereby being able to study the co-evolution between the two. The usefulness of the approach is proved by performing an empirical study on 8 software project, both open-source and from industry. The main findings are that: 1) the amounts of comments and source code grow at similar rates as the system evolves; 2) the type of a source code entity has an impact on whether or not a comment addressing it is added; 3) for 6 of the inspected projects the comments and the source code co-evolve in roughly 90% of the cases; 4) API changes do not trigger comments immediately, but they are nonetheless documented in following commits. This work is similar to ours in the sense that it studies the evolution of the production code of a software system in connection with another artifact that is part of the development process. However, there are also two main differences: 1) they try to correlate changes in the production code with the comments that are created, while we associated them with changes made in the test classes; 2) they have only performed a quantitative analysis based on which some results were inferred, while we went more in depth and also conducted a qualitative analysis in order to better understand the knowledge obtained through the quantitative analysis.

7.2 Source code evolution

Source code evolution is one of the main areas of research in the field of software engineering. We have identified several topics of interest within this area and gathered relevant publications for each of them. The following topics have been considered:

7.2.1 Evolution in open-source software systems

This topic is directly related to our work as we have used open-source systems in our analyses. Therefore, we are interested in understanding the particular characteristics of these types of systems and whether or not they are similar to the ones of commercial systems.

[30] is one of the first articles to study the evolution of open-source systems. Besides analyzing the way in which a system grows, the paper also studies how the corresponding OSS community evolves alongside it. Through a series of case studies performed on 4 open-source projects it was uncovered that different collaboration models exist which cause different evolution patterns to be observed for open-source systems and communities. The authors also propose a classification for OSS projects. They divide them into 3 categories: Exploration-Oriented, Utility-Oriented and Service-Oriented. This proves that even though these types of systems are developed following collaborative principles, the resulted projects differ in a number of ways. This observation is very important for us considering the fact that we have used 5 open-source systems in our analyses.

Until now, we have only discussed the evolution of software systems created in Java, thus following object-oriented principles. However, this evolution can be studied for project developed in other languages or using different programming paradigms (*i.e.*, imperative or functional principles). Godfre et al. [18] analyze the evolution of the source code of open-source systems developed in C. More specifically, they try to determine whether these systems evolve in the same way as commercially developed ones. The observations made are relevant for us, as we want to extend our study by including projects from industry. In order to investigate these aspects, a case study is performed on one of the most well-known open-source systems, the Linux kernel. The project's evolution was examined both at a system level and within its major subsystems, thus obtaining a series of interesting facts. The most notable one was that, contrary to initial expectations (that it would grow slower as it got more complex), the Linux kernel was evolving at super-linear rates for the last couple of years. Another insightful finding was that examining the growth patterns of the composing subsystems can provide an in depth understanding of why a system has evolved in a specific way.

In [15], a tool is introduced that can be utilized to compare the source code of different versions of a system developed in C. The presented approach uses partial abstract syntax tree matching in order to identify the changes that are made to global variables, types or functions from one version of a system to another. The authors used the tool to analyze the evolution history of a number of popular open-source programs, including Apache, OpenSSH, Vsftpd, Bind and the Linux kernel. A series of statistics were computed based on the information extracted. Some of the main findings were that: 1) function and global variable additions are more likely to occur that deletions to these source code entities; 2) function bodies are modified frequently over time, but the prototypes of the functions are rarely changed; 3) type definitions are rarely modified and when they are only small changes are made. The tool was also extended to calculate several evolution metrics, such as common coupling or cohesion. These findings are relevant to our work because we plan to extend the empirical study with projects developed in other programming languages or using different programming paradigms.

7.2.2 Software evolution visualization

Visualizing the changes that occur between two versions of a system is also an important topic in the area of software evolution. We are pondering the possibility of extending our approach by providing a visualization component. A number of articles have already addressed this topic and presented tools that were developed in order to facilitate the visual representation of software evolution data.

Telea *et al.* [40] propose Code Flows, a set of techniques that can be used to visualize fine-grained source code evolution. These rendering and layout techniques were designed to: 1) illustrate how source code entities evolve over a specific period of time; 2) track a code fragment throughout its entire history; 3) identify and highlight complex events (*e.g.,* code swaps, splits or merges) in the lifespan of a software system. They aid developers in a number of activities, such as understanding low-level source code changes or identifying small-scale code patterns. However, more information could have been added to these visualizations by extracting additional data from code repositories. For example, code quality metrics, bug information or programmer IDs represent important evolutionary data that has not been integrated into the visualization *model* proposed in this paper.

In [43] the authors introduce CVSscan, an integrated multi-view environment for visualizing the way in which a system evolves. It utilizes a line-oriented display to show the changing code, with a column for each version of the project and a horizontal direction for representing time. Additionally, CVSscan provides a series of linked displays containing various metrics and numerous options that can be used to visualize several different aspects of a system. An overview of the tool is illustrated in Figure 7.1. One of the most innovative concepts described in this paper is the *bi-level code display* that shows both the content of a source code fragment and its evolution during a specified period of commits. The researchers demonstrate the usefulness of this tool by conducting informal user studies on a number of real-world use cases. The results obtained prove that the line-based visualization introduced facilitates a quick assessment of the artifacts created during development.

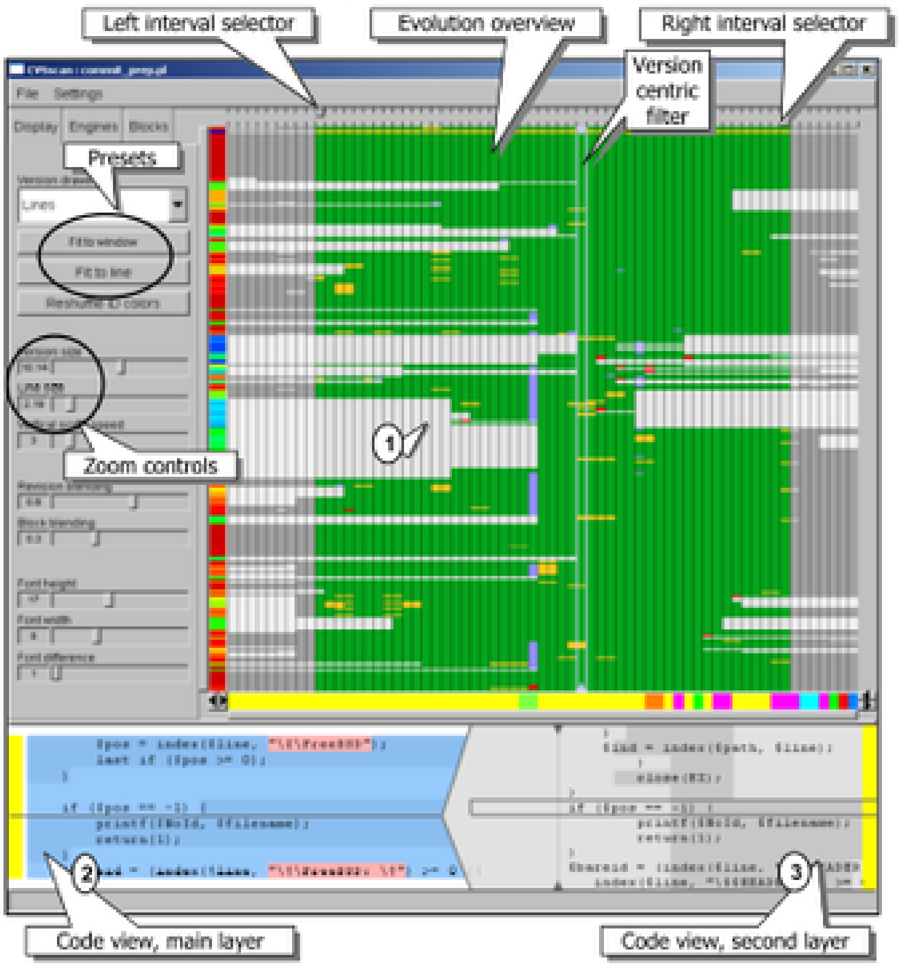


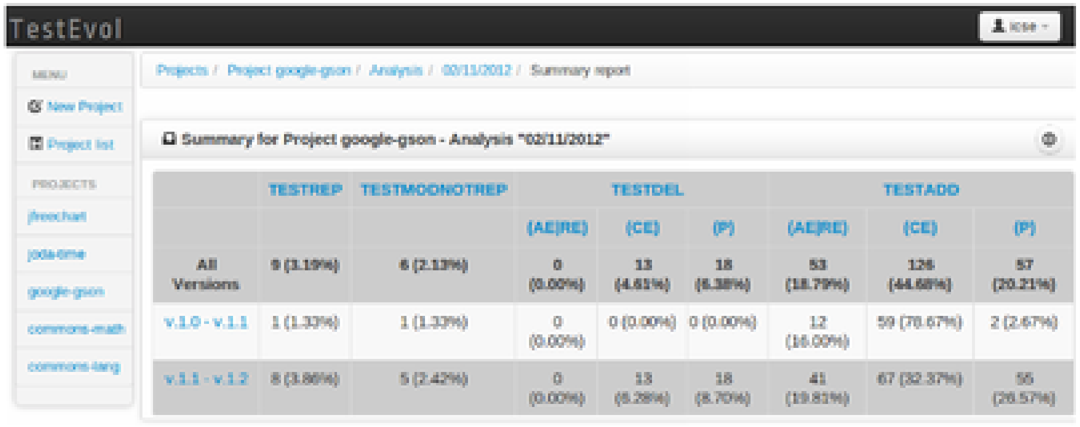
Figure 7.1: CVSscan overview [43].

7.2.3 Test suite evolution

Test suite evolution is another topic that has been investigated extensively in the context of software evolution. Evaluating how a test suite evolves is a first step towards understanding the nature of the co-evolution between production and test code.

Pinto *et al.* [33] examine how unit test suite evolution occurs. Their main finding is that test repairing is an often occurring phenomenon during evolution, indicating for example that assertions are fixed. The study also shows that test suite augmentation is another important activity during evolution aimed at making the test suite more adequate. One of the most striking observations that they make is that failing tests are more often deleted than repaired. Among these deleted test cases, tests fail predominantly (over 92% of the time) with compilation errors, whereas the remaining ones fail with assertion or runtime errors. In a controlled experiment on refactoring in connection with developer tests, Vonken and Zaidman also note that participants often deleted failing assertions instead of trying to address them [44].

In the follow-up paper [32], Pinto *et al.* present the architecture of the tool that was implemented based on the proposed approach, TestEvol, in greater detail compared to the previously discussed paper ([33]). The system was developed as a Java-based web application that runs on an Apache Tomcat server. For two versions of a software project, TestEvol generates a detailed summary report on how the test cases changed while also providing coverage information gathered for the respective versions. The structure of the summary report is depicted in Figure 7.2.



Summary for Project google-gson - Analysis "02/11/2012"

	TESTREP	TESTMODNOTREP	TESTDEL			TESTADD		
			(AE|RE)	(CE)	(P)	(AE|RE)	(CE)	(P)
All Versions	9 (3.19%)	6 (2.13%)	0 (0.00%)	13 (4.61%)	18 (6.38%)	53 (18.79%)	126 (44.68%)	57 (20.21%)
v.1.0 - v.1.1	1 (1.33%)	1 (1.33%)	0 (0.00%)	0 (0.00%)	0 (0.00%)	12 (16.00%)	59 (78.67%)	2 (2.67%)
v.1.1 - v.1.2	8 (3.86%)	5 (2.42%)	0 (0.00%)	13 (6.28%)	18 (8.70%)	41 (19.81%)	67 (32.37%)	55 (26.57%)

Figure 7.2: TestEvol summary report [32].

As we are aiming to integrate test code quality into our study, we now turn towards papers that describe methods to measure this aspect. In [2], the author proposes a model that can be utilized to assess test code quality. This model combines source code metrics related to 3 main quality aspects (completeness, effectiveness and maintainability) and aggregates them into a quality rating based on benchmarking. The model was used to examine the correlation between test code quality and issue handling performance. An empirical study was conducted using 18 open-source systems which shows that there is indeed a correlation between the quality of the test code and issue handling indicators mined from issue repositories. Additionally, 3 case studies have been done using commercial systems in which the quality ratings obtained were compared with the evaluations provided by experts. Several other papers discuss ways through which test code quality aspects can be measured. For example, [29] introduces a tool that provides feedback on the testing effort put into a project. By doing this, it aids in identifying weaknesses and determining the completeness of a specific test suite.

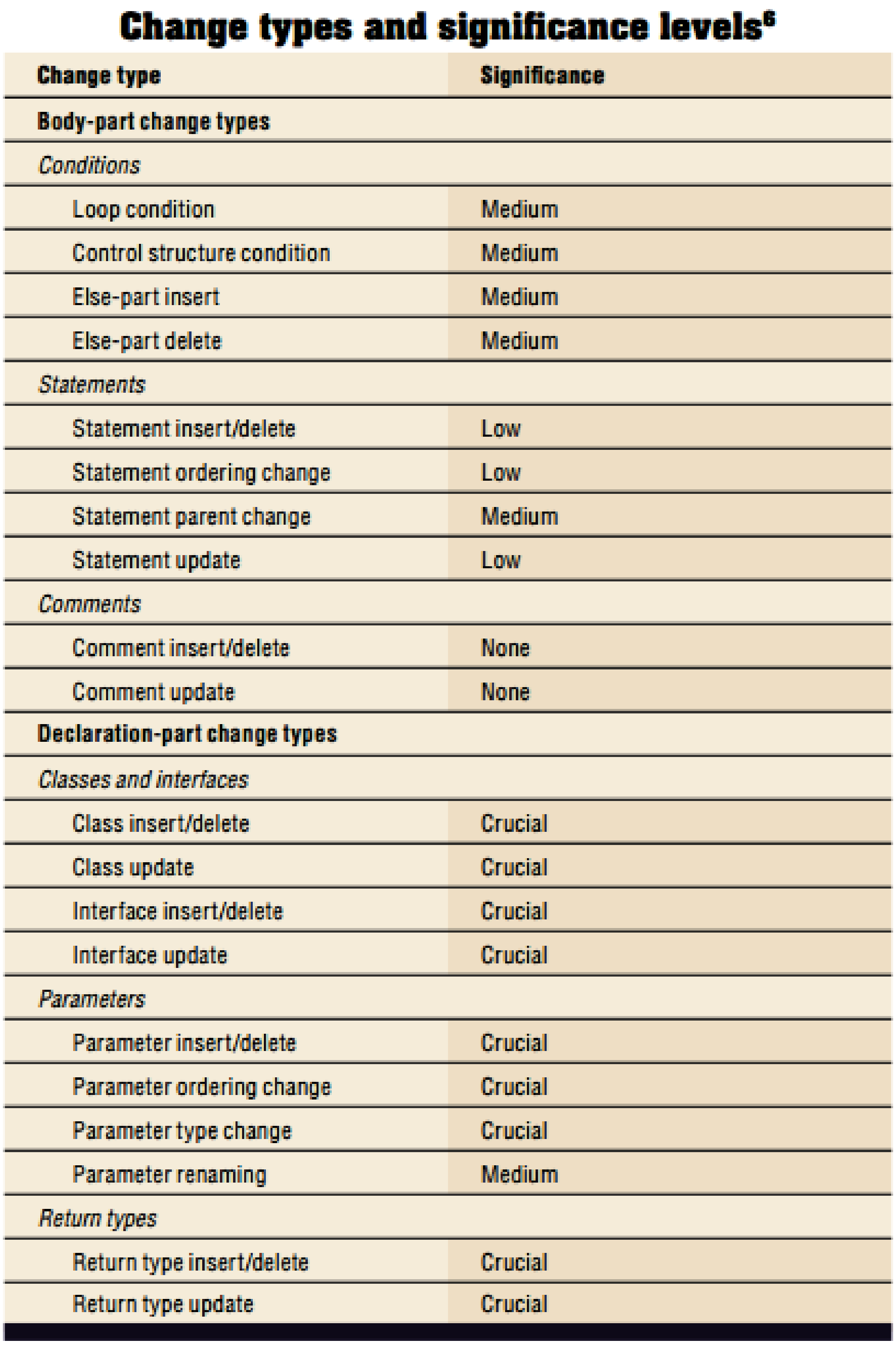
7.2.4 Other topics

There are many other topics in the area of software evolution. In the context of test-suite augmentation, Santelices *et al.* [38] present an enhanced methodology for improving existing tests as a result of evolving software. The proposed technique uses dependence analysis, partial symbolic execution, symbolic state differencing, and runtime monitoring in order to: 1) asses the adequacy of a regression test suite when changes are made in the production code; 2) facilitate the generation of new test cases that cover the untested behaviours introduced by the production changes. The authors also describe a toolset that implements the technique and perform two empirical studies with it. The results from the first study highlight that the proposed technique can successfully identify test cases with high fault-detection capabilities. The second study proves that, unlike any other approach that was developed before it, the technique can handle multiple production changes.

Impact analysis (determining the effects of changes on a software system) is also a relevant topic in the area of software evolution. Getgers *et al.* [17] discuss an adaptive approach that can be used to perform impact analysis for a source code change request. Based on a textual change request (*e.g.,* a bug report), an indexed release of the software project is utilized to estimate the impact set. The approach can also produce an improved impact set by determining the best-fit combination if additional contextual information, such as the execution trace or an initial source code entity that was verified for change, is available. In order to obtain this improved set, the approach combines techniques from information retrieval, dynamic analysis and data mining of past source code commits; therefore, this work is similar to ours, as we also use data mining to obtain the information necessary for the two analyses and dynamic analysis to link test cases to the production entities they cover. The authors conduct an empirical study on 4 open-source systems in order to validate the proposed approach. The data for this study is represented by a benchmark consisting of a set of maintenance issues, such as bug fixes or feature requests, and the corresponding source code changes that generated them; the benchmark was obtained by manually examining the systems and their change histories. The empirical study has shown that combining the 3 techniques mentioned above helps produce better results in terms of precision and recall compared to any of the 3 used independently while also improving accuracy.

7.3 Source code change extraction

Determining the changes that occur from one version of a system to another is important in the context of co-evolution. In [41], Toth *et al.* present an algorithm that is able to identify changes in source code entities (*e.g.,* classes or methods) by examining changes in source files. Additionally, they evaluated the algorithm on the WebKit system. The algorithm uses static analysis in order to determine the positions of the changed source code entities only for a limited number of revisions. For the rest of the revisions, elementary change blocks are determined by computing the differences between the positions of the entities



Change types and significance levels[6]

Change type	Significance
Body-part change types	
Conditions	
Loop condition	Medium
Control structure condition	Medium
Else-part insert	Medium
Else-part delete	Medium
Statements	
Statement insert/delete	Low
Statement ordering change	Low
Statement parent change	Medium
Statement update	Low
Comments	
Comment insert/delete	None
Comment update	None
Declaration-part change types	
Classes and interfaces	
Class insert/delete	Crucial
Class update	Crucial
Interface insert/delete	Crucial
Interface update	Crucial
Parameters	
Parameter insert/delete	Crucial
Parameter ordering change	Crucial
Parameter type change	Crucial
Parameter renaming	Medium
Return types	
Return type insert/delete	Crucial
Return type update	Crucial

Figure 7.3: Initial source code change taxonomy [16].

in the respective revision and their position in the last revision on which static analysis was performed. In this way, all the revisions of a project are addressed without having to perform expensive computations for each revision. However, there are situation in which the implications of an elementary change block cannot be identified clearly. These situations occur when the change blocks overlap the beginning or the end of a source code entity.

Gall *et al.* reported on analyzing the information obtained by mining software repositories [16]. Two tools are introduced: Evolizer, which is a platform for mining software repositories, and ChangeDistiler [13], a change extraction and analysis tool that can be used to investigate fine-grained source code changes. Of particular interest to us is ChangeDistiller, which extracts source code changes from the different versions of a Java class gathered with Evolizer. The source code of each analyzed version is represented as an abstract syntax tree (AST) and the changes between two versions are determined by computing the differences between their corresponding ASTs. A taxonomy for source code changes was also defined along with the significance level of each type of change. Figure 7.3 illustrates the initial taxonomy which was refined over time into the categorization that we used (discussed in Chapter 2). We rely on the work described in this article for our study. Other research problems have been investigated using the two tools mentioned above. For example, Romano *et al.* have utilized them to study both the feasibility of predicting change-prone Java interfaces using source code metrics [35] and the impact of antipatterns on the change-proneness of Java classes [36].

An alternative for extracting source code changes is discussed in [34]. The authors propose a different approach to collect and process data for software evolution. Instead of relying on versioning systems, this data is gathered directly from the integrated development environment used by the developers. Robbes and Lanza state that the source code changes should be regarded as first-class entities (entities that can be referenced, queried or passed along in a program) in order to obtain accurate information about the evolution of a software system. They also implemented the approach into a prototype called SpyWare. The results obtained by using this prototype show that it captures meaningful information, both at a high level and on a session-by-session basis.

7.4 Software repository mining

A first step in analyzing the co-evolution between production code and associated test cases is gathering relevant data through software repository mining. An investigation into the early approaches for repository mining is presented in [21]. The techniques were classified along 4 directions: the types of the repositories mined, the purpose for doing it, the methodology used and the evaluation method employed. An evaluation of the resulted taxonomy proved that it is both expressive and effective. The results showed that there is little variation in terms of types of repositories analyzed and evaluation methods used. However, the methodology and the purpose vary significantly. A number of threats to the validity of the discussed repository mining techniques were also presented. The most notable threat is the lack of standardization of the methodologies that can be utilized to mine software repositories and of the methods that are selected to evaluate them.

Information for a large variety of tasks can be gathered by mining software repositories. There are multiple types of repositories from which data can be collected. In this section we detail on two of these types, CVS and Git.

7.4.1 Concurrent Versions System (CVS)

Many of the studies that employ repository mining techniques extract the necessary information from CVS repositories. One of the first tools to leverage version histories is presented in [49]. Zimmermann *et al.* apply data mining techniques in order to determine related changes in the source code of a software system. Based on a change, they are able to identify other changes that were done by the same developer. Furthermore, given a set of changes, the proposed approach can: 1) suggest and predict future changes; 2) identify item coupling that is difficult to detect through program analysis; 3) prevent errors caused by incomplete changes. The approach was developed into a prototype, ROSE, that is able to predict entities that need to be changed based on an initial change by analyzing data gathered from CVS. The authors evaluate the tool on 8 open-source systems, thus proving that ROSE can successfully identify the locations where changes have to be made in more than 70% of the cases.

Ying *et al.* [46] describe an approach that uses data mining techniques to detect change patterns (groups of files that were frequently changed together in the past) in the change history of a software system. The paper brings two main contributions: 1) it proves the utility of change pattern mining; 2) it introduces a set of interestingness criteria that can be utilized to assess the usefulness of change pattern recommendations. The feasibility of the proposed approach was evaluated by applying it to two open-source systems, Eclipse and Mozilla, and demonstrating that it can recommend changes that need to be made when a specific change occurs. The biggest issue with this work is that the precision and the recall of the recommender are not very high, therefore not all the recommendations might be valid or some items might be missing.

In [20], Kagdi *et al.* propose a method for discovering traceability links between the artifacts of a software system by examining its version history. It is a heuristic-based approach that utilizes sequential-pattern mining and is applied to commits in order to uncover frequently occurring co-changing sets of software artifacts (*e.g.,* source code and documentation). This approach was evaluated on an open-source system called K Desktop Environment (KDE). The obtained links were used afterwards to predict similar types of changes in new commits.

The data mined from source code repositories can also help improve static analysis tools. Williams *et al.* [45] introduce a method that utilizes the change history of a software system to improve the search for bugs within the respective system. They developed a static source code checker that identifies commonly fixed bugs and uses data extracted from the source code repository in order to refine this search. The feasibility and performance of the approach were demonstrated by performing case studies on two systems, Apache Web Server and Wine. The results showed that the bugs that are catalogued in bug databases and the ones identified by studying code repositories differ in terms of type and level of abstraction. This is not surprising considering the fact that the bugs from the databases are reported by the users of a software system while the others are recorded by developers upon fixing them. The results have also proven that the approach is significantly better than a similar one that does not use data mined from source code repositories, therefore the usefulness of this kind of information was demonstrated once again.

In [48] Zimmerman *et al.* utilize the data extracted from CVS archives to identify *fix-inducing changes* (changes in production code that cause problems within the respective software system). They propose a methodology that can be used to automatically locate these types of changes by linking a version archive to a bug database (*i.e,* Bugzilla). This work is related to ours for the following reasons: 1) it is one of the first studies to investigate the impact of changes that occur in the production code of a software project; 2) it tries to relate these changes to defects that appear within the system, a thing that we are also considering as a future work direction. In order to detect these problematic changes, a 3 step procedure is followed: 1) it starts from a bug report that describes a fixed problem; 2) the corresponding change is extracted from the CVS, thus obtaining the location of the fix; 3) the earlier change that was made at that location and introduced the bug is retrieved. By investigating this research problem, a series of insightful facts have been uncovered, such as: 1) the specific properties (*e.g.,* the day in which they were made or the group of developers that introduced them) of the changes that raised issues; 2) an assessment of the error-proneness of the respective software system; 3) functionalities for filtering out the fix-inducing changes when performing other types of analyses on the evolution of a software system as they might lead to erroneous observations; 4) support for avoiding these kinds of changes when suggesting related changes to developers through recommenders.

7.4.2 Git

For our analyses, we have used data extracted from Git repositories. Bird *et al.* [5] discuss the promises and perils of mining Git. They start by providing statistical data which shows that Git usage has increased drastically over the last couple of years and it continues to grow. Afterwards, the promises and perils are explained in great detail. The first promise identified is that Git allows access to more information (*e.g.,* work in progress or work that does not get integrated into the stable code base) in comparison to CVS because any developer of a Git project can make this information publicly accessible. Another promise is that Git facilitates the recovery of a richer project history by providing access to a wide variety of data including commit, branch and merge information. Other important promises are that: 1) Git records authorship information for the contributors that are not part of the core developers; 2) for Git all metadata is local; 3) Git tracks content, therefore it is able to tack the history of lines as they are copied or moved; 4) Git is faster and generally uses less space than CVS. However, there are also a series of perils of using Git, the most notable ones being: 1) issues with implicit branches; 2) the absence of a mainline, therefore the analysis methods have to take into account the dependency acyclic graph (DAG); 3) the fact that Git history is revisionist (the owner of the repository can rewrite it); 4) it is not always possible to identify the branch on which a commit was made or to track the source of a merge; 5) the data that is accessible includes only the commits that were successful. All in all, the authors concluded that Git offers new and useful data that can be utilized for all kinds of analyses. The most notable improvements are accurate authorship information, the clear distinction between roles (*e.g.,* author, commiter, reviewer) and merge tracking facilities.

[7] is one of the first papers to examine data extracted from a Git-based project. The article presents the results obtained by analyzing information gathered from the Linux Git repository. The authors investigated how kernel development is done by studying changelogs and provided a series of statistical measurements. Similar to the previous article, in [8] Cobert *et al.* use data mined from Git in order to determine the way in which patches get into the mainline for the Linux kernel.

Chapter 8

Conclusions and Future Work

Testing has become an important part of the software development process. The developer tests aid in detecting possible issues in the implemented production code, while also facilitating program comprehension. Therefore, understanding how they evolve alongside the production classes they are addressing is clearly worth investigating. We propose an approach that can be used to analyze the co-evolution between production and test code. We utilize this approach to conduct an empirical study on 5 open-source systems in order to understand the nature of this co-evolution. This chapter first provides an overview of the scientific contributions made through our work. After this overview, the obtained results are summarized and a series of conclusion are drawn from them. Finally, we reflect on our work and present future research directions that are being considered.

8.1 Contributions

In this thesis we have investigated the fine-grained co-evolution of production and test code. We did this in order to: (1) gain a deeper understanding of the way in which tests evolve as a result of changes in the production classes, (2) identify possible gaps in the test base, thus signalling to the developers the parts of the production code that have not been adequately covered by tests.
In doing so, we make the following contributions:

- **We present an approach to study the fine-grained co-evolution between production and test code.** The approach consists of two steps: (1) extracting fine-grained source code changes from the production and the test code of a software system and (2) identifying links between the test cases in which changes are made and the production classes they cover. By using this approach we have been able to collect large amounts of data consisting of source code changes and links between the code entities in which these changes occur. The data has been utilized to analyze the co-evolution between these two main parts of a software system.

- **We enrich the software engineering body of knowledge with regard to production and test code co-evolution.** We have performed an empirical study on 5 open-source software projects, thereby gaining insight into the nature of this co-evolution. Additionally, we have investigated how the co-evolution between production and test code happens and whether it can be correlated with the testing effort put into a system.

- **We identify 6 co-evolution patterns based on the empirical study.** We also prove that the obtained patterns differ depending on how thoroughly the studied project is tested. Finally, we establish that for some production changes co-evolution always occurs simultaneously, while for others the corresponding test changes might be done in subsequent commits.

8.2 Conclusions

We are now in a position to answer our research questions.

For **RQ1**, *"What kind of fine-grained co-evolution patterns between production and test code can be identified?"*, we have uncovered 6 fine-grained co-evolution patterns by using an association rule mining technique. For each of these patterns, a positive and a negative sub-pattern have been identified. The positive patterns reflect co-evolution, while the negative ones point towards a lack of co-evolution.

For **RQ2**, *"Does the testing effort have an impact on the observed co-evolution patterns?"*, we first determine the testing effort put into each of the 5 projects. Afterwards, we have established that positive patterns are more likely to be encountered in thoroughly tested software systems (*i.e.,* CommonsLang, CommonsMath, Gson), while the negative ones are generally seen in projects for which the testing effort is low, such as PMD or JFreeChart. The qualitative evaluation that we have performed allowed us to gain a more in depth understanding of how the co-evolution takes place. In particular, we have found reasons why negative co-evolution patterns have been obtained. For example, we now have insight as to why sometimes new test classes are not added when production classes are created (*e.g.,* because the later is a mock class).

For **RQ3**, *"How does the co-evolution between production and test code happen?"*, we have seen that the moment when a test code change is made varies depending on the production change that triggered it. More specifically, some changes occur synchronously in a production class and associated test cases, while in other cases the corresponding test changes are made in a following commit. For example, in the case of production class additions, the associated test class is created in the same or in a subsequent commit. However, for production class deletions, the corresponding test class is always removed simultaneously. This is unsurprising considering the fact that it would cause compilation errors if it was left in the code.

8.3 Reflection

In general, no major issues were encountered while implementing the proposed approach or when the empirical study was conducted. We have successfully managed to extract fine-grained source code changes from the production and the test code of software systems.

Additionally, following a dynamic solution, we have been able to link the entities in which these changes were made, namely the production classes and the associated test cases. We have thereby obtained a large dataset that was used in the analyses that followed. An empirical study was conducted using the information gathered that allows for a deeper understanding of how production and test code co-evolve. Both when developing the approach and while conducting the study we had to take a number of decisions on how to proceed. When such situations occurred, we tried to explain the reasoning behind each of the decisions that was made. As for any scientific work, there are several ways in which the performed study can be improved, on which we will detail in the following section.

8.4 Future work

Extend the empirical study
Additional Java projects: A first direction for future work entails extending the empirical study by analyzing the co-evolution between the production and the test code of new Java projects. Commercial systems in particular might show different co-evolution patterns as more or less testing effort may have been put into them. Nevertheless, adding new open-source projects with different characteristics than the ones that were already studied is also worth doing.
New development technologies or programming paradigms: The development technology is a limiting factor for our study, as all the analyzed projects are implemented in Java. We plan to address this issue in the future by extending the study to projects developed in other object-oriented programming languages, such as C++ or C#. Even though a new implementation is needed, the proposed approach will still be valid. Besides the development technology, analyzing projects that are created following other programming paradigms, such as imperative or functional programming, is also a direction that seems worth investigating. It would be interesting to observe whether the co-evolution patterns for these projects resemble the ones obtained for the object-oriented systems.
Other development methodologies: Another area to concentrate on is studying whether specific development methodologies (*e.g.,* Test-Driven Development) show different co-evolution strategies. For TDD test cases are written before the production code is created, therefore we expect that co-evolution happens differently compared to what was observed thus far.

Add quality into the equation
Until now we have only evaluated the co-evolution between production and test code in terms of number of changes. An interesting research direction would be analyzing this co-evolution in connection with the quality of the production and the test code. More specifically, determining if the quality of the test code has an impact on the quality of the production classes that are developed. This would entail constructing a set of metrics for each of these two parts of the source code and trying to identify correlations between the values obtained for the metrics.
We also aim to improve the characterization of testing effort by making use of the recent test code quality model presented by Athanasiou *et al.* [3].

Improve existing test repair techniques
Finally, we want to use the knowledge that has been gained through this empirical study to look into test repair techniques. Of particular interest are intent-preserving repair techniques, assuring that the repaired test cases address the same production functionalities as before they were broken. This can be easily achieved considering that the second part of the proposed approach is used to link test cases to the production classes they cover. Therefore, it can be utilized to establish these links both before and after a test repair is done. By comparing the retrieved results it can be determined whether or not the intent has been preserved.

Bibliography

[1] Andrea Arcuri. On the automation of fixing software bugs. *ACM*, pages 1003–1006, 2008.

[2] Dimitrios Athanasiou. Constructing a test code quality model and empirically assessing its relation to issue handling performance. *Dissertation Masters Thesis, University of Delft, The Netherlands*, 2011.

[3] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. Test code quality and its relation to issue handling performance. *Transactions on Software Engineering*. *To appear.*

[4] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, Reading, MA, USA, 2003.

[5] Christian Bird, Peter C Rigby, Earl T Barr, David J Hamilton, Daniel M German, and Prem Devanbu. The promises and perils of mining git. *Mining Software Repositories*, pages 1–10, 2009.

[6] Tom Borthwick. Java code coverage: Cobertura vs. emma vs clover. *http://www.copperykeenclaws.com/notes-on-cobertura-vs-emma-vs-clover/*, 23 October 2010. Web. 30 April 2014.

[7] Tim Chen, Leonid I Ananiev, and Alexander V Tikhonov. Keeping kernel performance from regressions. *Proceedings of the Linux Symposium*, 1:93–102, 2007.

[8] Jonathan Corbet. How patches get into the mainline. *Linux Weekly News*, February 10, 2009.

[9] J.W. Creswell and V.L.P. Clark. *Designing and Conducting Mixed Methods Research*. SAGE Publications, 2010.

[10] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[11] Sebastian Elbaum, Dabid Gable, and Gregg Rothermel. The impact of software evolution on code coverage information. In *Proc. Int'l Conf. on Software Maintenance (ICSM)*, pages 170–179. IEEE CS, 2001.

[12] Michael Ellims, James Bridges, and Darrel C. Ince. The economics of unit testing. *Empirical Software Engineering*, 11(1):5–31, 2006.

[13] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Software Eng.*, 33(11):725–743, 2007.

[14] Beat Fluri, Michael Wrsch, Emanuel Giger, and Harald C Gall. *Analyzing the co-evolution of comments and source code*, volume 17. 2009.

[15] Jeffrey S Foster, Iulian Neamtiu, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[16] Harald Gall, Beat Fluri, and Martin Pinzger. Change analysis with evolizer and changedistiller. *IEEE Software*, 26(1):26–33, 2009.

[17] Malcom Gethers, Bogdan Dit, Huzefa Kagdi, and Denys Poshyvanyk. Integrated impact analysis for managing software changes. *IEEE*, 34(11):430–440, 2012.

[18] Michael W Godfrey and Qiang Tu. Evolution in open source software : A case study. *Software Maintenance*, pages 131–142, 2000.

[19] Victor Hurdugaci and Andy Zaidman. Aiding software developers to maintain developer tests. In *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 11–20. IEEE, 2012.

[20] H. Kagdi, J. Maletic, and B. Sharif. Mining software repositories for traceability links. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 145–154, Washington, DC, USA, 2007. IEEE Computer Society.

[21] Huzefa Kagdi, Michael L Collard, and Jonathan I Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[22] Meir Lehman. On understanding laws, evolution and conservation in the large program life cycle. *Journal of Systems and Software*, 1(3):213–221, 1980.

[23] Zeeger Lubsen, Andy Zaidman, and Martin Pinzger. Using association rules to study the co-evolution of production & test code. In *Int'l Working Conf. on Mining Software Repositories (MSR)*, pages 151–154. IEEE, 2009.

[24] A Von Mayrhauser and A M Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[25] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE)*, pages 13–22. IEEE, 2005.

[26] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[27] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[28] Leon Moonen, Arie van Deursen, Andy Zaidman, and Magiel Bruntink. The interplay between software testing and software evolution. In *Software Evolution*, pages 173–202. Springer, 2008.

[29] Nachiappan Nagappan, Laurie Williams, Jason Osborne, Mladen Vouk, and Pekka Abrahamsson. Providing test quality feedback using static source code and automatic test suite metrics. *Software Reliability Engineering*, pages 1–10, 2005.

[30] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. *Proceedings of the international workshop on Principles of software evolution. ACM*, pages 76–85, 2002.

[31] Dewayne E Perry, Adam A Porter, and Lawrence G Votta. Empirical studies of software engineering : A roadmap. *The future of Software engineering*, pages 345–355, 2000.

[32] Leandro Sales Pinto and Alessandro Orso. Testevol : A tool for analyzing test-suite evolution. *International Conference on Software Engineering*, pages 1303–1306, 2013.

[33] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Symposium on the Foundations of Software Engineering (FSE)*, page 33. ACM, 2012.

[34] Romain Robbes and Michele Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, 166:93–109, January 2007.

[35] Daniele Romano and Martin Pinzger. Using source code metrics to predict change-prone java interfaces. *Software Maintenance (ICSM)*, pages 303–312, 2011.

[36] Daniele Romano, Paulius Raila, and Martin Pinzger. Analyzing the impact of antipatterns on change-proneness using fine-grained source code changes. *Reverse Engineering (WCRE)*, pages 437–446, 2012.

[37] P. Runeson. A survey of unit testing practices. *IEEE Software*, 25(4):22–29, 2006.

[38] Raul Santelices, Pavan Kumar Chittimalli, Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. Test-suite augmentation for evolving software. *Proceedings ASE*, pages 218–227, 2008.

[39] Quinten David Soetens, Serge Demeyer, and Andy Zaidman. Change-based test selection in the presence of developer tests. In *Proc. Conf. on Software Maintenance and Reengineering (CSMR)*, pages 101–110. IEEE, 2013.

[40] Alexandru Telea and David Auber. Code flows: Visualizing structural evolution of source code. *Computer Graphics Forum*, 27(3):831–838, 2008.

[41] Zoltan Toth, Gabor Novak, Rudolf Ferenc, and István Siket. Using version control history to follow the changes of source code elements. In *Proc. Conf. on Softw. Maintenance and Reengineering (CSMR)*, pages 319–322. IEEE, 2013.

[42] Bart Van Rompaey and Serge Demeyer. Establishing traceability links between unit test cases and units under test. In *Proc. Conf. on Software Maintenance and Reengineering (CSMR)*, pages 209–218. IEEE, 2009.

[43] Lucian Voinea and Alex Telea. Cvsscan : Visualization of code evolution. *ACM symposium on Software visualization*, pages 47–56, 2005.

[44] Frens Vonken and Andy Zaidman. Refactoring with unit testing: A match made in heaven? In *Proc. of the Working Conf. on Reverse Engineering (WCRE)*, pages 29–38. IEEE Computer Society, 2012.

[45] Chadd C Williams and Jeffrey K Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Transactions*, 31(6):466–480, 2005.

[46] Annie T T Ying, Gail C Murphy, Raymond Ng, and Mark C Chu-Carroll. Predicting source code changes by mining change history. *IEEE Transactions*, 30(9):574–586, 2004.

[47] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[48] Thomas Zimmermann. When do changes induce fixes? *ACM sigsoft software engineering notes*, 30(4):24–28, 2005.

[49] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions*, 31(6):429–445, 2005.

Appendix A

Association rules

In this appendix we explain the generated association rules in greater detail. For each of the rules we discuss the correlation that can be inferred between the production change from the antecedent of the rule and the test code change from the consequent. We also refer to the support and confidence values obtained as they provide further insight into the co-evolution. Furthermore, the implications of each association rule are analyzed as well.

Added class

a. CommonsLang

Association rule 1a1

ADDED_CLASS_PRODUCTION=YES → ADDED_CLASS_TEST=YES , support: 412, confidence: 0.64375

This first association rule indicates that for *CommonsLang*, a project that has been categorized as extensively tested, the creation of a new production class leads to the addition of a corresponding test class in around 64% of the cases.

Association rule 1a2

ADDED_CLASS_PRODUCTION=YES → CLASS_DECLARATION_TEST=NONE, support: 557, confidence: 0.87031

The second rule reveals that sometimes when a production class is created additional test cases are developed in the already existing test classes in order to cover it. Even though the value of CLASS_DECLARATION_TEST is NONE, the confidence of this rule indicates that in roughly 13% of the cases a different value than NONE was registered; therefore, in these situations at least one test case is created when the new production class is added.

b. CommonsMath

Association rule 1b1

ADDED_CLASS_PRODUCTION=YES → ADDED_CLASS_TEST=SOMETHING

The fact that an association rule was not generated when the value of ADDED_CLASS_PRODUCTION is YES suggests that in the case of CommonsMath entire test classes are sometimes added when production classes are created. This observation is also supported by the ratio between added classes in the production code and added test classes which is roughly two to one.

Association rule 1b2

ADDED_CLASS_PRODUCTION=YES → CLASS_DECLARATION_TEST=NONE , support: 2595, confidence: 0.84776

The second association rule indicates that the developers rarely create additional test cases in the already existing test classes when a new production class is added. This supports the fact that even though considerable testing effort has been put into CommonsMath, the project cannot be classified as extensively tested.

c. PMD

Association rule 1c1

ADDED_CLASS_PRODUCTION=YES → ADDED_CLASS_TEST=NO , support: 4161, confidence: 0.90654

Association rule 1c2

ADDED_CLASS_PRODUCTION=YES → CLASS_DECLARATION_TEST=NONE , support: 4483, confidence: 0.97669

The two rules prove that in the case of PMD neither new test classes nor additional test cases are created in order to cover the production class that is added. This is in concordance with PMD's classification as a poorly tested project.

d. Gson

Association rule 1d1

ADDED_CLASS_PRODUCTION=YES → ADDED_CLASS_TEST=YES , support: 85, confidence: 0.77273

Association rule 1d2

ADDED_CLASS_PRODUCTION=YES → CLASS_DECLARATION_TEST=NONE , support: 100, confidence: 0.90909

Similar to the case of CommonsLang, corresponding test classes are added when new production classes are developed (rule 1d1). In some situations additional test cases are also created in the already existing test classes in order to cover newly added production classes (rule 1d2). These two rules support the initial observation that Gson is a relatively well-tested project.

e. JFreeChart

Association rule 1e1

ADDED_CLASS_PRODUCTION=YES → ADDED_CLASS_TEST=NO , support: 832, confidence: 0.80543

Association rule 1e2

ADDED_CLASS_PRODUCTION=YES → CLASS_DECLARATION_TEST=NO , support: 855, confidence: 0.87996

The situations is the same as for PMD, just that the number of test classes / cases that are added when additional classes are created in the production code is a bit higher than in the PMD case, as observed from the confidence values that were obtained.

Removed class

a. CommonsLang

Association rule 2a1

REMOVED_CLASS_PRODUCTION=YES → REMOVED_CLASS_TEST=YES , support: 568, confidence: 0.99824

This rule clearly suggests that the removal of a production class triggers the deletion of its associated test class (if such a test class exists). Situations in which this does not happen are rare, fact proven by the retrieved confidence value.

Association rule 2a2

REMOVED_CLASS_PRODUCTION=YES → CLASS_DECLARATION_TEST=NONE , support: 517, confidence: 0.90861

The association rule indicates that in some cases a number of methods from one or more test classes are removed when a production class is deleted. This shows that sometimes the developers are doing integration testing instead of unit testing.

b. CommonsMath

Association rule 2b1

REMOVED_CLASS_PRODUCTION=YES → REMOVED_CLASS_TEST=YES , support: 1554, confidence: 0.97429

When production classes are deleted the situation is similar to the CommonsLang case, the corresponding test classes are generally removed as well.

Association rule 2b2

REMOVED_CLASS_PRODUCTION=YES → CLASS_DECLARATION_TEST=NONE , support: 1333, confidence: 0.83574

There are also situations in which test cases that cover the discarded production class are deleted. This proves that a series of integration tests are part of the test code and they are removed once the production classes they address are dropped.

c. PMD

Association rule 2c1

REMOVED_CLASS_PRODUCTION=YES → REMOVED_CLASS_TEST=YES , support: 4926, confidence: 0.99838

Like for the previous two projects, a test class is removed when the production class it covers is deleted. The confidence value of 0.99838 shows that cases in which this observation does not hold almost never appear.

Association rule 2c2

REMOVED_CLASS_PRODUCTION=YES → CLASS_DECLARATION_TEST=NONE , support: 4813, confidence: 0.97548

However, situations in which test cases are discarded when a production class is deleted occur less frequently than for CommonsLang and CommonsMath. This is unsurprising considering the fact that less testing effort has been spent on this system compared to the first two projects.

d. Gson

Association rule 2d1

REMOVED_CLASS_PRODUCTION=YES → REMOVED_CLASS_TEST=YES , support: 89, confidence: 0.93684

As for the 3 systems discussed above, the deletion of a production class is correlated with the removal of its associated test class. Even though more cases in which this does not happen have been identified, the observation still holds most of the times (confidence of 0.93684).

Association rule 2d2

REMOVED_CLASS_PRODUCTION=YES → CLASS_DECLARATION_TEST=NONE , support: 83, confidence: 0.87368

There are also some situations in which test case deletions are triggered by the removal of a production class. This demonstrates that there are a number of classes that contain integration tests within the system which are updated accordingly.

e. JFreeChart

Association rule 2e1

REMOVED_CLASS_PRODUCTION=YES → REMOVED_CLASS_TEST=YES , support: 1331, confidence: 0.95419

Similar to all the other projects included in the study, if a removed production class has an associated test class, then this class is discarded as well. This happens in a vast majority of cases; a value of 0.95419 was obtained as confidence for this rule.

Association rule 2e2

REMOVED_CLASS_PRODUCTION=YES → CLASS_DECLARATION_TEST=NONE , support: 802, confidence: 0.93274

Few test cases from the other test classes are deleted when a production class is removed. This is probably because no test cases addressing the respective production class exist in these test classes in the first place.

Class declaration changes

a. CommonsLang

Association rules 3a1-3a4

CLASS_DECLARATION_PRODUCTION=LOW → CLASS_DECLARATION_TEST=NONE , support: 244, confidence: 0.95312

CLASS_DECLARATION_PRODUCTION=MED-LOW → CLASS_DECLARATION_TEST=LOW , support: 132, confidence: 0.8

CLASS_DECLARATION_PRODUCTION=MED-HIGH → CLASS_DECLARATION_TEST=SOMETHING

CLASS_DECLARATION_PRODUCTION=HIGH → CLASS_DECLARATION_TEST=HIGH , support: 85, confidence: 0.65891

The association rules indicate that to some extent there is a correlation between the number of class-level changes in the production code and similar types of changes in the corresponding test classes. When the amount of CLASS_DECLARATION changes is LOW in the production classes, the same types of changes are generally not encountered in the test code (rule 3a1). However, when the number of CLASS_DECLARATION_PRODUCTION changes grows (MED-LOW to HIGH), CLASS_DECLARATION changes are also observed in the test classes. One of the most probable scenarios would be that the developers add / remove methods from a production class while also adding / removing a number of test cases that cover them. An interesting aspect can be observed for the third rule (3a3) for which the exact amount of CLASS_DECLARATION changes from the test code was not determined. It suggests that when a MED-HIGH number of changes are performed in the production code, a series of similar types of changes are done in the test classes. However, the exact number of test changes could not be established; this situation is caused by the fact that the values for CLASS_DECLARATION changes in the tests vary from LOW to HIGH when a MED-HIGH amount of changes of the same type are made in the production code, thereby the corresponding association rule was not generated.

Association rules 3a5-3a8

CLASS_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 185, confidence: 0.72266

CLASS_DECLARATION_PRODUCTION=MED-LOW → BODY_STATEMENTS_TEST=SOMETHING

CLASS_DECLARATION_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=SOMETHING

CLASS_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=SOMETHING

These 4 association rules show that when the number of CLASS_DECLARATION changes is larger (MED-LOW to HIGH) in the production classes, BODY_STATEMENTS changes are performed in the test code. This corresponds to situations in which production methods are added / removed and the test cases are updated accordingly. Similar as for one of the previous rules from this category (3a3), the exact number of BODY_STATEMENTS changes that occurred could not be pinpointed.

b. CommonsMath

Association rules 3b1-3b4

CLASS_DECLARATION_PRODUCTION=LOW → CLASS_DECLARATION_TEST=NONE , support: 1129, confidence: 0.98174

CLASS_DECLARATION_PRODUCTION=MED-LOW → CLASS_DECLARATION_TEST=SOMETHING

CLASS_DECLARATION_PRODUCTION=MED-HIGH → CLASS_DECLARATION_TEST=SOMETHING

CLASS_DECLARATION_PRODUCTION=HIGH → CLASS_DECLARATION_TEST=SOMETHING

The 4 association rules are similar to the ones generated for CommonsLang. They suggest that there is indeed a link between CLASS_DECLARATION changes in the production code and the same types of changes in the test classes. However, these rules are not as precise as the ones for CommonsLang; the exact amount of CLASS_DECLARATION changes in the test code was not determined.

Association rules 3a5-3a8

CLASS_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 930, confidence: 0.8087

CLASS_DECLARATION_PRODUCTION=MED-LOW → BODY_STATEMENTS_TEST=NONE, support: 510, confidence: 0.74671

CLASS_DECLARATION_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=NONE, support: 159, confidence: 0.6824

CLASS_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=SOMETHING

The above rules indicate that a large number (HIGH) of CLASS_DECLARATION changes in the production classes cause BODY_STATEMENTS changes in the tests. For the CommonsLang case this occurred even when the amount of CLASS_DECLARATION production changes was smaller (MED-LOW and MED-HIGH). This is a clear indication that less effort has been put into testing CommonsMath in comparison to CommonsLang, a fact that is also supported by the observations made in Chapter 3 of this document.

c. PMD

Association rules 3c1-3c3

CLASS_DECLARATION_PRODUCTION=LOW → CLASS_DECLARATION_TEST=NONE , support: 1767, confidence: 0.99887

CLASS_DECLARATION_PRODUCTION=MED-HIGH → CLASS_DECLARATION_TEST=NONE , support: 855, confidence: 0.80813

CLASS_DECLARATION_PRODUCTION=HIGH → CLASS_DECLARATION_TEST=NONE , support: 503, confidence: 0.79715

Association rules 3c4-3c6

CLASS_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 1588, confidence: 0.89768

CLASS_DECLARATION_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=NONE , support: 924, confidence: 0.87335

CLASS_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=NONE , support: 532, confidence: 0.84311

The 6 association rules show that for PMD a CLASS_DECLARATION change in the production classes does not trigger any changes in the test code, unlike the cases of CommonsLang or CommonsMath. PMD has already been classified as a poorly tested project, the association rules just confirm the fact that no changes occur in the tests due to CLASS_DECLARATION changes in the production code.

d. Gson

Association rules 3d1-3d3

CLASS_DECLARATION_PRODUCTION=LOW → CLASS_DECLARATION_TEST=NONE , support: 159, confidence: 0.97546

CLASS_DECLARATION_PRODUCTION=MED-HIGH → CLASS_DECLARATION_TEST=LOW , support: 43, confidence: 0.65152

CLASS_DECLARATION_PRODUCTION=HIGH → CLASS_DECLARATION_TEST=SOMETHING

The 3 rules suggest a correlation between CLASS_DECLARATION changes in the production classes and CLASS_DECLARATION changes in tests when the number of production changes is high (MED-HIGH and HIGH). For the MED-HIGH values it was established that the amount of test changes is LOW, while for the HIGH values the exact number of test class changes was not pinpointed. Therefore, similar to CommonsLangs and CommonsMath, test cases are created / deleted when production methods are added / removed.

Association rules 3d4-3d6

CLASS_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 138, confidence: 0.84663

CLASS_DECLARATION_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=NONE , support: 51, confidence: 0.77273

CLASS_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=NONE , support: 27, confidence: 0.72973

The association rules indicate that, unlike the cases of CommonsLangs and CommonsMath, for Gson CLASS_DECLARATION changes in the production code do not determine BODY_STATEMENTS changes in the test classes. Considering that BODY_STATEMENTS changes in tests can be triggered by different types of changes (*e.g.*, BODY_STATEMENTS, ATTRIBUTE_DECLARATION or METHOD_DECLARATION) in the production classes and that the number of BODY_STATEMENTS_TEST changes for Gson is relatively small, it would appear that these types of test code changes are not performed when appropriate in Gson's case.

e. JFreeChart

Association rules 3e1-3e3

CLASS_DECLARATION_PRODUCTION=LOW → CLASS_DECLARATION_TEST=NONE , support: 360, confidence: 0.95443

CLASS_DECLARATION_PRODUCTION=MED-HIGH → CLASS_DECLARATION_TEST=NONE , support: 174, confidence: 0.75407

CLASS_DECLARATION_PRODUCTION=HIGH → CLASS_DECLARATION_TEST=NONE , support: 102, confidence: 0.74358

The generated association rules show that, similar to the PMD case, no changes occur in the test code when CLASS_DECLARATION changes are made in the production classes. However, considering the values for confidence, it seems that situations in which the test code is modified appear more frequently than for PMD.

Method declaration changes

a. CommonsLang

Association rules 4a1-4a4

METHOD_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST=SOMETHING

METHOD_DECLARATION_PRODUCTION=MED-LOW → BODY_STATEMENTS_TEST=SOMETHIG

METHOD_DECLARATION_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=SOMETHING

METHOD_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=MED-HIGH , support: 37, confidence: 0.61667

The 4 association rules suggest that when a method is modified in the production code (*e.g.,*, by adding a parameter or by changing the type of its return value), the test cases that cover the respective method are updated as well. With the exception of the fourth association rule (4a4), the exact amount of BODY_STATEMENTS changes was not determined; for 4a4 however, it was established that when a HIGH number of METHOD_DECLARATION changes take place in the production classes, a MED-HIGH number of BODY_STATEMENTS changes are made in the test code.

b. CommonsMath

Association rules 4b1-4b3

METHOD_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 331, confidence: 0.6141

METHOD_DECLARATION_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=SOMETHING

METHOD_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=SOMETHING

The association rules are similar to those which were generated for CommonsLang. However, a correlation between BODY_STATEMENTS changes in the tests and METHOD_DECLARATION changes in the production code was identified only when the number of production changes is high (MED-HIGH and HIGH).

c. PMD

Association rules 4c1-4c3

METHOD_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 634, confidence: 0.7254

METHOD_DECLARATION_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=NONE , support: 309, confidence: 0.88793

METHOD_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=NONE , support: 234, confidence: 0.82394

Similar to the other types of production changes, a METHOD_DECLARATION change in one of PMD's production classes rarely determines any kinds of changes in the test code. In some situations it triggers BODY_STATEMENTS changes in the test cases; however, these situations occur sporadically, fact proven by the association rules that were generated in which the value for BODY_STATEMENTS changes is NONE and the confidence is quite high.

d. Gson

Association rules 4d1-4d3

METHOD_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 60, confidence: 0.83333

METHOD_DECLARATION_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=SOMETHING

METHOD_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=SOMETHING

As in the case of CommonsMath, a large amount (MED-HIGH and HIGH) of METHOD_ _DECLARATION changes in the production classes cause BODY_STATEMENTS changes in the tests. However, when only a few (LOW) of this type of changes occur in the production code, the test classes are generally not updated to reflect them (4d1).

e. JFreeChart

Association rules 4e1-4e3

METHOD_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 134, confidence: 0.8143

METHOD_DECLARATION_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=NONE , support: 65, confidence: 0.89391

METHOD_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=NONE , support: 49, confidence: 0.87191

The situation is the same as for PMD in terms of values obtained both for the consequent (NONE) and for the confidence of the rules.

Attribute declaration changes

a. CommonsLang

Association rules 5a1-5a3

ATTRIBUTE_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST=SOMETHING

ATTRIBUTE_DECLARATION_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=SOMETHING

ATTRIBUTE_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=SOMETHING

Association rules 5a4-5a6

ATTRIBUTE_DECLARATION_PRODUCTION=LOW → CLASS_DECLARATION_TEST=NONE , support: 558, confidence: 0.7228

ATTRIBUTE_DECLARATION_PRODUCTION=MED-HIGH → CLASS_DECLARATION_TEST=SOMETHING

ATTRIBUTE_DECLARATION_PRODUCTION=HIGH → CLASS_DECLARATION_TEST=SOMETHING

Like in the case of METHOD_DECLARATION changes for CommonsLang, ATTRIBUTE_ _DECLARATION changes in the production classes trigger BODY_STATEMENTS and CLASS_DECLARATION changes in the tests. The exact number of test code changes was not established; nevertheless, the association rules suggest that when field-related changes are made in the production code, the existing test cases are updated accordingly and even new test methods are added. This is in concordance with the initial observation that CommonsLang is an extensively tested system.

b. CommonsMath

Association rules 5b1-5b3

ATTRIBUTE_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 558, confidence: 0.7228

ATTRIBUTE_DECLARATION_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=SOMETHING

ATTRIBUTE_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=SOMETHING

The 3 association rules are similar to the ones generated for METHOD_DECLARATION changes in the case of CommonsMath. They suggest that when attributes are added / removed / modified in production classes the test cases that cover them are generally updated. The precise number of BODY_STATEMENTS changes was again not determined. This is caused by the fact that each ATTRIBUTE_DECLARATION change can trigger multiple BODY_STATEMENTS changes in the test code; therefore, the number of test changes can vary significantly (from LOW to HIGH) for each production change. Because of this, the threshold value for confidence was not met.

c. PMD

Association rules 5c1-5c3

ATTRIBUTE_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 1244, confidence: 0.7374

ATTRIBUTE_DECLARATION_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=NONE , support: 528, confidence: 0.90722

ATTRIBUTE_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=NONE , support: 628, confidence: 0.834

Similar to METHOD_DECLARATION changes, for PMD an ATTRIBUTE_DECLARATION change in the production code usually does not cause any kinds of changes in the test classes.

d. Gson

Association rules 5d1-5d3

ATTRIBUTE_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 82, confidence: 0.82828

ATTRIBUTE_DECLARATION_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=SOMETHING

ATTRIBUTE_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=SOMETHING

Just as in the case of METHOD_DECLARATION changes, there is a correlation between ATTRIBUTE_DECLARATION changes in the production classes and BODY_STATEMENTS changes in the tests when the values for the first are high (MED-HIGH and HIGH). This confirms the fact that when the attributes and the methods of a production class are altered, the associated test cases are also updated.

e. JFreeChart

Association rules 5e1-5e3

ATTRIBUTE_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 148, confidence: 0.8337

ATTRIBUTE_DECLARATION_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=NONE , support: 62, confidence: 0.85361

ATTRIBUTE_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=NONE , support: 74, confidence: 0.822

The situation is the same as for PMD, little updating is done in the test cases when ATTRIBUTE_DECLARATION changes are made in the production code.

Body statement changes

a. CommonsLang

Association rules 6a1-6a4

BODY_STATEMENTS_PRODUCTION=LOW → BODY_STATEMENTS_TEST=SOMETHING

BODY_STATEMENTS_PRODUCTION=MED-LOW → BODY_STATEMENTS_TEST=SOMETHIG

BODY_STATEMENTS_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=SOMETHING

BODY_STATEMENTS_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=HIGH , support: 177, confidence: 0.61458

The 4 association rules indicate that BODY_STATEMENTS changes in the production code trigger similar types of changes in the tests. This might correspond to a scenario in which a large number of changes (including BODY_STATEMENTS, but METHOD_DECLARATION and ATTRIBUTE_DECLARATION as well) are made in the production classes and therefore the test cases are updated accordingly. The precise amount of BODY_STATEMENTS changes that occurred in the test code was only established for the fourth rule (6a4) which states that a HIGH number of BODY_STATEMENTS changes in the production classes determine numerous BODY_STATEMENTS changes in the tests.

b. CommonsMath

Association rules 6b1-6b4

BODY_STATEMENTS_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 1449, confidence: 0.82096

BODY_STATEMENTS_PRODUCTION=MED-LOW → BODY_STATEMENTS_TEST=SOMETHIG

BODY_STATEMENTS_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=SOMETHING

BODY_STATEMENTS_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=SOMETHING

Like in the case of CommonsLang, the rules show that BODY_STATEMENTS changes in the production classes trigger BODY_STATEMENTS changes in the test code. The only exception is when a small amount (LOW) of changes occur in the production code; then, no changes are observed in the test classes. However, considering the different types of production changes (*e.g.,* METHOD_DECLARATION or ATTRIBUTE_DECLARATION) that cause BODY_STATEMENTS changes in the tests, these rules might not be as straightforward as they appear.

c. PMD

Association rules 6c1-6c4

BODY_STATEMENTS_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 2050, confidence: 0.81804

BODY_STATEMENTS_PRODUCTION=MED-LOW → BODY_STATEMENTS_TEST=NONE , support: 2031, confidence: 0.95532

BODY_STATEMENTS_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=NONE , support: 1344, confidence: 0.86822

BODY_STATEMENTS_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=NONE , support: 1649, confidence: 0.81755

No changes occur in the test code when changes from the BODY_STATEMENTS category are made in the production classes. The most significant correlation for this kind of production change is with BODY_STATEMENTS changes in the tests; however, these situations occur rarely, thereby the generated association rules have NONE as the value for the consequent.

d. Gson

Association rules 6d1-6d4

BODY_STATEMENTS_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 140, confidence: 0.95238

BODY_STATEMENTS_PRODUCTION=MED-LOW → BODY_STATEMENTS_TEST=NONE , support: 140, confidence: 0.7234

BODY_STATEMENTS_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=SOMETHING

BODY_STATEMENTS_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=SOMETHING

These rules indicate that BODY_STATEMENTS changes are triggered in the test code only when a high (MED-HIGH and HIGH) number of BODY_STATEMENTS changes are made in the production classes. Considering that the ratio between the two is 1 to 5 for Gson and that BODY_STATEMENTS changes can occur in the tests due to multiple reasons, the generated association rules were to be expected.

e. JFreeChart

Association rules 6e1-6e4

BODY_STATEMENTS_PRODUCTION=LOW → BODY_STATEMENTS_TEST=NONE , support: 592, confidence: 0.93123

BODY_STATEMENTS_PRODUCTION=MED-LOW → BODY_STATEMENTS_TEST=NONE , support: 306, confidence: 0.90124

BODY_STATEMENTS_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=NONE , support: 187, confidence: 0.7198

BODY_STATEMENTS_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=SOMETHING

Surprisingly, BODY_STATEMENTS changes in the production code determine BODY_STATEMENTS changes in the test classes (6e4), similar as for the projects that are tested more thoroughly (*i.e* CommonsLang, CommonsMath and Gson). This situation resembles the Gson case, where a large (MED-HIGH and HIGH) number of BODY_STATEMENTS changes in the production classes trigger the same type of changes in the tests.

Body condition changes

a. CommonsLang

Association rules 7a1-7a4

BODY_CONDITIONS_PRODUCTION=LOW → CLASS_DECLARATION_TEST=NONE , support: 126, confidence: 0.67021

BODY_CONDITIONS_PRODUCTION=MED-LOW → CLASS_DECLARATION_TEST=SOMETHIG

BODY_CONDITIONS_PRODUCTION=MED-HIGH → CLASS_DECLARATION_TEST=SOMETHING

BODY_CONDITIONS_PRODUCTION=HIGH → CLASS_DECLARATION_TEST=SOMETHING

The 4 association rules show that when a certain amount (MED-LOW to HIGH) of BODY_ _CONDITIONS changes occur in the production code CLASS_DECLARATION changes are performed in the test classes. A logical explanation would be that test cases are added / deleted when a condition in a production method becomes more / less complex. This observation is also supported by the fact that CommonsLang has a very high value for branch coverage, as shown in Chapter 3 of this document.

Association rules 7a5-7a8

BODY_CONDITIONS_PRODUCTION=LOW → BODY_STATEMENTS_TEST=SOMETHING

BODY_CONDITIONS_PRODUCTION=MED-LOW → BODY_STATEMENTS_TEST=SOMETHIG

BODY_CONDITIONS_PRODUCTION=MED-HIGH → BODY_STATEMENTS_TEST=SOMETHING

BODY_CONDITIONS_PRODUCTION=HIGH → BODY_STATEMENTS_TEST=SOMETHING

The rules suggest that in this case BODY_STATEMENTS changes are also made in the test code. In general, when a condition is changed in a production method, a series of BODY_STATEMENTS changes are made as well. Upon further investigation, it was observed that most of the lines from the dataset that contain BODY_CONDITIONS production changes also have BODY_STATEMENTS changes for the respective production classes. This is the reason why we obtained rules that have BODY_STATEMENTS_TEST as the consequent (7a5 - 7a8).

b. CommonsMath

Association rules 7b1-7b3

BODY_CONDITIONS_PRODUCTION=LOW → CLASS_DECLARATION_TEST=SOMETHING

BODY_CONDITIONS_PRODUCTION=MED-HIGH → CLASS_DECLARATION_TEST=SOMETHING

BODY_CONDITIONS_PRODUCTION=HIGH → CLASS_DECLARATION_TEST=SOMETHING

The fact that association rules with BODY_CONDITIONS_PRODUCTION on the left side and CLASS_DECLARATION_TEST on the right were not generated indicates that there is a somewhat weak correlation between the two. This occurs because of the different values that CLASS_DECLARATION_TEST takes (NONE, LOW, MED-LOW, MED-HIGH or HIGH) for each value of BODY_CONDITIONS_PRODUCTION, which causes the confidence of the rules to be below the threshold value. This correlation suggests that test cases are added / removed when conditions are modified in the production code, a fact which is supported by the high values that were obtained for branch coverage in the case of CommonsMath.

c. PMD

Association rules 7c1-7c3

BODY_CONDITIONS_PRODUCTION=LOW → CLASS_DECLARATION_TEST=NONE , support: 1044, confidence: 0.97661

BODY_CONDITIONS_PRODUCTION=MED-HIGH → CLASS_DECLARATION_TEST=NONE , support: 357, confidence: 0.952

BODY_CONDITIONS_PRODUCTION=HIGH → CLASS_DECLARATION_TEST=NONE , support: 430, confidence: 0.92672

As in most cases for PMD, a BODY_CONDITIONS change in the production code does not determine any types of changes in the test classes. The closest connection inferred was with changes from the CLASS_DECLARATION category for the tests, implying that when a condition is changed in a production class at least one test is added / removed. However, there was not enough evidence to support this, therefore the generated association rules have CLASS_DECLARATION_TEST=NONE as the consequent.

d. Gson

Association rules 7d1-7d3

BODY_CONDITIONS_PRODUCTION=LOW $\rightarrow$ CLASS_DECLARATION_TEST=NONE , support: 72, confidence: 0.9

BODY_CONDITIONS_PRODUCTION=MED-HIGH $\rightarrow$ CLASS_DECLARATION_TEST=NONE , support: 37, confidence: 0.82222

BODY_CONDITIONS_PRODUCTION=HIGH $\rightarrow$ CLASS_DECLARATION_TEST=SOMETHING

These rules indicate that only when a large amount (HIGH) of BODY_CONDITIONS changes occur in the production code CLASS_DECLARATION changes are made in the test classes. This is reflected in the relatively low values that were obtained for branch coverage for Gson's revisions.

e. JFreeChart

Association rules 7d1-7d3

BODY_CONDITIONS_PRODUCTION=LOW $\rightarrow$ CLASS_DECLARATION_TEST=NONE , support: 141, confidence: 0.853

BODY_CONDITIONS_PRODUCTION=MED-HIGH $\rightarrow$ CLASS_DECLARATION_TEST=NONE , support: 214, confidence: 0.76324

BODY_CONDITIONS_PRODUCTION=HIGH $\rightarrow$ CLASS_DECLARATION_TEST=SOMETHING

Similar to the Gson case, BODY_CONDITIONS changes in the production classes determine CLASS_DECLARATION changes in the test code, but only when the number of BODY_CONDITIONS changes is high (HIGH). This fact is also supported by the values that were recorded for branch coverage for JFreeChart, which are low compared to the ones gathered for some of the other analyzed systems, such as CommonsLang or CommonsMath.

Studying Fine-Grained Co-Evolution Patterns of Production and Test Code

Cosmin Marsavina
Delft University of Technology
The Netherlands
c.marsavina@student.tudelft.nl

Daniele Romano
Delft University of Technology
The Netherlands
daniele.romano@tudelft.nl

Andy Zaidman
Delft University of Technology
The Netherlands
a.e.zaidman@tudelft.nl

***Abstract*—Numerous software development practices suggest updating the test code whenever the production code is changed. However, previous studies have shown that co-evolving test and production code is generally a difficult task that needs to be thoroughly investigated.**

In this paper we perform a study that, following a mixed methods approach, investigates fine-grained co-evolution patterns of production and test code. First, we mine fine-grained changes from the evolution of 5 open-source systems. Then, we use an association rule mining algorithm to generate the co-evolution patterns. Finally, we interpret the obtained patterns by performing a qualitative analysis.

The results show 6 co-evolution patterns and provide insights into their appearance along the history of the analyzed software systems. Besides providing a better understanding of how test code evolves, these findings also help identify gaps in the test code thereby assisting both researchers and developers.

I. Introduction

Lehman has taught us that a software system must evolve, or it becomes progressively less useful [1]. During this evolution, the system's source code continuously changes to cope with new requirements or possible issues that might arise. However, software is multi-dimensional, because in order to develop high-quality systems other artifacts need to be taken into account, such as requirements, tests and documentation [2]. Therefore, these artifacts should *co-evolve* gracefully alongside the production code that is being written.

One of the artifacts that is of particular importance in the software development process is the developer test, which was defined by [3] as "a codified unit or integration test written by developers". Its importance resides in the fact that it can provide immediate feedback to the developers [4] and identify bugs. Moreover, when a software system evolves (e.g., through refactoring), developers should run the persistent tests to verify whether the external behavior is preserved [5]. In this context, Moonen *et al.* have shown that even though refactorings are behavior preserving, they can invalidate tests [6]. In the same vein, Elbaum *et al.* have concluded that even minor changes in the production code can significantly affect test coverage [7].

Based on these findings, there clearly is a need for tests to evolve alongside the production code they are covering in order to obtain high-quality systems. However, creating and maintaining tests are expensive tasks. Zaidman *et al.* have shown that developing test code that co-evolves gracefully with the production classes it addresses is generally a difficult endeavour [8].

In this paper we try to identify fine-grained co-evolution patterns between production and test code. These patterns consist of changes that occur in the test code when changes are made to the production code. It is also likely that some co-evolution patterns appear more frequently in particular software systems. Hence, besides identifying these patterns, we aim at correlating them with the testing effort spent for each of the analyzed systems. This leads us to our research questions:

RQ1 What kind of fine-grained co-evolution patterns between production and test code can be identified?
RQ2 Does the testing effort have an impact on the observed co-evolution patterns?

We answer the research questions by following a mixed methods approach [9] that combines quantitative and qualitative analyses. First, we use an association rule mining algorithm to identify co-evolution patterns. Then, we refine these quantitative results through a qualitative analysis aimed at manually interpreting the patterns that have been obtained. The results show: 1) 6 co-evolution patterns mined for 5 case study systems, 2) how they occur, and 3) whether the testing effort has an impact on them.

From a research perspective, getting insight into these co-evolution patterns is particularly useful to check whether specific changes in the production code should also have consequences in the test code of a software system. This might lead to better tool support, thereby assisting developers in designing higher quality test code.

The main contributions of this paper are as follows:

1) a method to collect and relate fine-grained source code changes that co-occur in the production and the test code of a software system;
2) an empirical study to investigate the co-evolution between production and test code for 5 open-source systems; the study comprises a quantitative analysis during which a number of co-evolution patterns have been uncovered and a more in-depth qualitative analysis with anecdotal evidence on each of the patterns.

The remainder of the paper is structured as follows. Section II presents the approach adopted to collect the data for our analyses, while Section III illustrates the experimental setup. Results are described in Sections IV and V that present respectively the quantitative and qualitative analyses. In Section VI we revisit the research questions and discuss threats to validity. Related work is presented in Section VII and we conclude the paper and mention future work in Section VIII.

II. Approach

As discussed in the previous section, there is a need within the scientific community to examine and understand the co-evolution between the production and the test code of a software project. The main goal of this study is to identify a series of patterns consisting of changes that occur in the test code when the production code evolves. We expect these co-evolution patterns to vary from one project to another, because of the different working styles of the development teams or due to different priorities with regards to testing activities. Therefore, while performing our analyses, we also assess the testing effort put into each of the projects under study. Furthermore, besides uncovering the patterns, we also inspect the source code to find and understand concrete examples that help in interpreting the obtained co-evolution patterns.

In the following subsections we describe: (1) the approach adopted to extract the fine-grained changes and to link production and test code; (2) its implementation.

A. Change Extraction

In order to collect relevant data for studying the co-evolution of production and test code, we first obtain all the versions of a project. We mine Git as this facilitates the access to the repositories of a large variety of software projects. Moreover, it provides functionalities to compute high-level differences (*e.g.,* addition and deletion of classes) between one version of a project and another.

However, these differences are not detailed enough to allow for an in-depth analysis of the co-evolution between production and test code. For this reason we extract fine-grained source code changes between different versions using ChangeDistiller [10]. Table I details the change categories along with the specific changes that ChangeDistiller can detect which are related to the source code.

We have extracted these source code changes both from the production and from the test code. In order to make the dataset as comprehensive as possible, we have included additional information such as: the class in which the change occurred, the version when the change was made along with its timestamp, and the exact source code entity that was modified.

B. Linking production and test code

Once we have the fine-grained changes, we link the test cases to the production code they cover. We prefer a dynamic solution over a static analysis approach because it is more precise as pointed out by Van Rompaey and Demeyer [11]. The key idea behind our approach is to run each test case separately, thereby identifying all the entities from the production code addressed by the test, similarly to the approach used in [12]. To retrieve the covered entities (*e.g.,* Java classes) we process test coverage information gathered with Cobertura[1].

Change category	Change
ADDED_CLASS	ADDITIONAL_CLASS
REMOVED_CLASS	REMOVED_CLASS
CLASS_DECLARATION	CLASS_RENAMING, PARENT_CLASS_CHANGE, PARENT_CLASS_DELETE, PARENT_CLASS_INSERT, PARENT_INTERFACE_CHANGE, PARENT_INTERFACE_DELETE, PARENT_INTERFACE_INSERT, REMOVED_FUNCTIONALITY, ADDITIONAL_FUNCTIONALITY
METHOD_DECLARATION	RETURN_TYPE_CHANGE, RETURN_TYPE_DELETE, RETURN_TYPE_INSERT, METHOD_RENAMING, PARAMETER_DELETE, PARAMETER_INSERT, PARAMETER_ORDERING_CHANGE, PARAMETER_RENAMING, PARAMETER_TYPE_CHANGE
ATTRIBUTE_DECLARATION	ATTRIBUTE_RENAMING, ATTRIBUTE_TYPE_CHANGE, ADDING_ATTRIBUTE_MODIFIABILITY, REMOVING_ATTRIBUTE_MODIFIABILITY, ADDITIONAL_OBJECT_STATE, REMOVED_OBJECT_STATE
BODY_STATEMENTS	STATEMENT_DELETE, STATEMENT_INSERT, STATEMENT_ORDERING_CHANGE, STATEMENT_PARENT_CHANGE, STATEMENT_UPDATE
BODY_CONDITIONS	CONDITION_EXPRESSION_CHANGE, ALTERNATIVE_PART_DELETE, ALTERNATIVE_PART_INSERT

TABLE I: Categories of changes retrieved with ChangeDistiller.

C. Implementation

To implement our approach we use a process consisting of two steps that is described in Figure 1.

As a first step (see Figure 1(a)), we use the jGit API[2] to retrieve the software project's source code from the corresponding Git repository. Then, we compute the differences between two consecutive versions of the system using the same API. Based on the types of the changes retrieved between versions, one of the following two approaches is selected:

1) When entire Java classes are added or deleted, the names of the fields and the methods declared in those classes are recorded.
2) Otherwise, ChangeDistiller is utilized to extract the fine-grained changes.

A specific procedure is applied to each project version for which the test code has been modified (shown in Figure 1(b)). We first compile the production code using Maven in order to ensure that it does not contain any errors. If the compilation is successful, the test cases of that version are run separately. For each test case, we let Cobertura generate a coverage report file. This file is then parsed with the jDom API[3] to identify the methods from the production code covered by the respective

[1]http://cobertura.github.io/cobertura/ — last visited June 13th, 2014.
[2]http://eclipse.org/jgit/ — last visited June 19th, 2014
[3]http://www.jdom.org/ — Last visited June 19th, 2014.

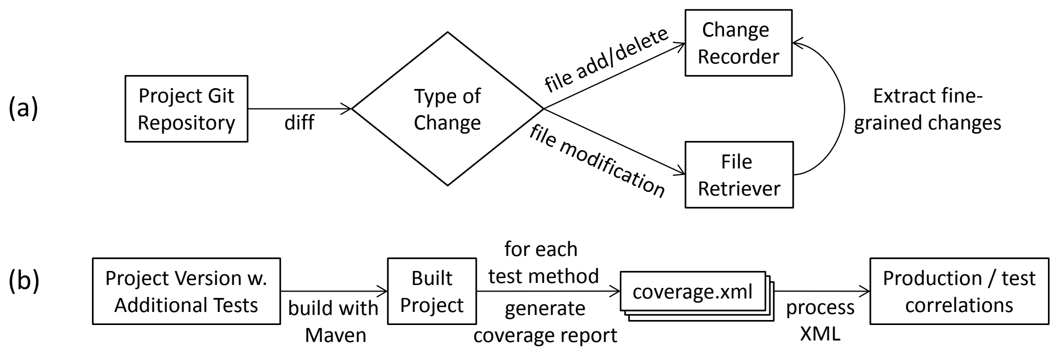


Fig. 1: Overview of the data collection process.

test method. We record these results which are used afterwards to determine the links between production classes and test cases.

III. Experimental Setup

This section describes how the empirical study has been conducted, including: project selection, the initial analysis that was performed to determine testing effort, and the process through which the quantitative and qualitative studies have been carried out.

A. Project Selection

We have chosen 5 projects on which we conduct our empirical study. We rely on the criteria set by Pinto *et al.* in [13] to select the projects, namely: (1) a large number of versions, (2) considerable size (in terms of production classes and methods), (3) an extensive JUnit test suite, and (4) be in active maintenance. For each of the systems, all their versions have been included in the analysis. An overview of the main characteristics of the 5 projects is presented in Table II; it contains the total number of versions studied and shows metrics collected for the first version of a project and the last version considered.

B. Preliminary Analysis

As a preliminary analysis, we have studied the 5 systems in order to understand how well they are tested. Four perspectives have been considered: (1) changes that occurred in the production / test code, (2) branch coverage obtained during the lifespan of the project, (3) number of versions that did not compile because of test failures, and (4) ratio between the amount of test code and production code. An overview of this preliminary analysis is shown in Table II.

The reported branch coverage has been recorded for the last version that we have considered (column Branch Coverage) and it was determined with Cobertura. We have also collected coverage information for each release of a project and observed that the overall branch coverage remains relatively stable.

The table also includes the number of versions that raised problems during compilation because of test failures (column Number of Non-Compiling Versions Test Failures). We have relied on Maven to compile the projects and recorded all the situations in which not every test from a version passed. Finally, we have calculated the ratio between the lines of test code and production code lines (globally, for all versions combined) as a measurement for quantifying the volume of testing that has been done for a system (column $\Sigma_{\forall v_i} LOC_{test}$ / $\Sigma_{\forall v_i} LOC_{prod}$).

Subsequently, we have analyzed the 5 software projects to determine which types of changes occur in their production and test code. Table III contains an overview of these changes grouped into 10 categories corresponding to the 10 major types of changes identified by ChangeDistiller. For our analyses we only consider the first 7 categories of changes, as the last 3 are not related to the source code of the system. The total number of production and test code changes per software project has been calculated. In order to get an indication of testing "effort", we have also determined the percentage of test code changes from to the total number of changes. This is depicted in the final row of Table III.

The following thresholds have been selected: 1) percentage of test changes - over 33%; 2) branch coverage - over 0.67; 3) percentage of non-building versions - less than 2.5%; 4) test code lines - over 0.25. A system is considered properly tested if at least 3 of the thresholds are met. Based on the above information which can be considered an indicator of testing effort, we classify the projects as extensively tested (*CommonsLang*), relatively well tested (*CommonsMath*, *Gson*) and rather poorly tested (*PMD*, *JFreeChart*).

C. Analyses performed

We have performed our study following a mixed methods approach [9] that combines quantitative and qualitative analyses as described in the following subsections.

1) Quantitative Analysis: We first identify frequently occurring fine-grained co-evolution patterns between production and test code. The spmf[4] tool is used to generate a series of association rules. We have configured the Apriori algorithm with support and confidence values of 50% and 60%, respectively.

The following steps have been applied to obtain the rules. For each version of a system, all the changes that occur in the production code and in the associated tests are recorded per production class using a bucket list representation. Instead of the actual values, we use discrete values to quantify the

[4]http://www.philippe-fournier-viger.com/spmf/ — Last visited June 19th, 2014

Project		First version				Final version considered					Number of Non-	$\Sigma_{\forall v_i} LOC_{test}$
	#	#	# Prod.	# Test	Release	#	# Prod.	# Test	Release	Branch	Building Versions	/
	Versions	Classes	Methods	Methods		Classes	Methods	Methods		Coverage	due to Test Failures	$\Sigma_{\forall v_i} LOC_{prod}$
PMD	7165	316	1846	340	11/2002	822	4418	1340	12/2013	0.51418	369	0.130
CommonsLang	3856	31	373	318	12/2002	177	2442	2851	02/2014	0.90678	54	0.442
CommonsMath	5174	83	758	501	12/2004	985	6548	6201	02/2014	0.80254	131	0.366
JFreeChart	519	423	5790	1297	11/2006	701	7776	2403	03/2014	0.49274	17	0.219
Gson	322	73	414	131	05/2008	142	719	1010	08/2012	0.66233	12	0.287

TABLE II: Overview of selected projects.

ChangeDistiller category	PMD		CommonsLang		CommonsMath		JFreeChart		Gson	
	Prod	Test	Prod	Test	Prod	Test	Prod	Test	Prod	Test
ADDED_CLASS	4690	599	679	410	3074	1172	929	1128	130	124
REMOVED_CLASS	4269	733	306	287	905	739	1154	185	84	35
CLASS_DECLARATION	8742	1207	2396	2179	7379	4007	1777	847	542	460
METHOD_DECLARATION	3038	169	1146	376	2730	709	641	399	286	64
ATTRIBUTE_DECLARATION	7558	307	795	198	2787	746	890	33	330	27
BODY_STATEMENTS	107831	8179	12933	15924	44098	28260	16266	17705	4947	1134
BODY_CONDITIONS	16507	58	1466	85	2365	284	774	1	500	9
COMMENTS	2285	99	709	527	2762	1015	406	224	70	10
DOCUMENTATION	2363	88	3534	513	9145	468	449	401	212	15
OTHERS	1621	136	246	101	1051	236	14	394	114	97
TOTAL	159628	10851	24495	20315	76996	36936	23485	21132	7239	1961
Total$_{Test}$/(Total$_{Test}$+Total$_{Prod}$)	6.37%		45.33%		41.10%		47.36%		21.32%	

TABLE III: Total number of changes in the production / test code per ChangeDistiller change category.

number of changes that occur from each of the categories. We did this in order to facilitate the generation of the association rules, as it would not be possible to obtain rules with the specified support and confidence if numerical values were used. In the cases of class additions and removals, only YES and NO values have been utilized, as these kinds of changes can either happen or not. For the other types of changes, one of the following 5 values is assigned: NONE, LOW, MED_LOW, MED_HIGH or HIGH. In order to assign these values, the set containing the number of occurrences of the respective change in each class in which it was made (for all the versions of a system) has been constructed; extreme values have been filtered out, to prevent the results from being skewed. The last 4 discrete values (LOW, MED_LOW, MED_HIGH and HIGH) correspond to the (0%-25%], (25%-50%], (50%-75%], [75%-100%] intervals of values from this set. After we put the data in the appropriate format (version — production class — changes in class / associated test classes = value), the association rules are computed.

2) Qualitative Analysis: To refine the results from the quantitative analysis, we perform a qualitative study in order to better understand some of the co-evolution patterns that have been obtained during the previous analysis. We concentrate on the following 5 categories of production code changes: added class, removed class, class declaration change, attribute declaration change, and body condition change. We disregard the other 2 categories because (1) a large variety of fine-grained changes are part of the METHOD_DECLARATION category, therefore it was difficult to find a substantial number of examples for each of them and (2) BODY_STATEMENT changes occur in almost every commit, thus making it hard to separate them from the other types of changes.

We carry out this qualitative analysis by studying concrete examples of test code changes that occur as a result of a particular change in the production code. From each category of production changes (see Table I), we investigate every type of change in depth. For each occurrence of the change in a production class, all the changes it has triggered in the test code are recorded and analyzed. In order to ensure that there is indeed a connection between the production and the test code changes, the links between the respective production class and the corresponding test cases are inspected, along with the actual source code and the commit message of the project version under consideration. After gathering these examples, we make a series of observations based on them regarding (1) how co-evolution happens together with (2) an interpretation of the co-evolution patterns identified during the quantitative analysis.

IV. Quantitative Analysis

A number of association rules have been generated for each category of production code changes. Table IV provides an overview of these association rules for each of the 5 systems along with the support and confidence values obtained. The second column (*i.e., Association Rule*) contains the retrieved association rule; however, the value of the consequent is missing in this column. This value can be found in the subsequent columns that are specific to each project under analysis. These columns contain the support and the confidence of the rule together with the value of the consequent (*e.g, YES*, *NO*, *SOMETHING*). For instance, for rule 1, the column *PMD* shows that no test classes are added (*i.e., NO*) when new production classes are created; the rule has a support value of 4161 and a confidence value of 0.906 for the respective project. Therefore, the complete set of association rules for a project is obtained by concatenating the second column with the specific column for that project.

In some cases an association rule has not been generated between a production and a test code change. This is caused by the fact that the changes in the production code are dispersed over a number of intervals, *i.e.* (*LOW*, *MED-LOW*, *MED-HIGH*, or *HIGH*), see for example the antecedents of rules 7 through 9 for *CommonsLang*. Because of this dispersion the threshold value for confidence might not be met, which in turn means that an association rule is not generated. However, if there was no link between the production and the test code change, we would expect that an association rule containing NONE as the value of the consequent would have been generated, thus indicating the lack of connection. The fact that this association rule with NONE is not produced suggests that there might still be a (weak) link, thing that we marked with the keyword *SOMETHING* in the consequent.

Table IV also has some empty cells. This happens for some of the rules that have MED_LOW as the value for the production change. It is caused by the fact that the respective production changes generally occur only once for a class in a commit, therefore the intervals corresponding to (0%-25%] (LOW) and (25%-50%] (MED_LOW) of the values are identical (contain only 1 values).

As an example of mined association rules, consider the following two rules that address the addition of production classes for the *CommonsLang* project:

Association rule 1.1

ADDED_CLASS_PRODUCTION=YES $\rightarrow$ ADDED_CLASS_TEST=YES

support: 412, confidence: 0.643

This first association rule indicates that for *CommonsLang*, a project that has been categorized as extensively tested, the creation of a new production class leads to the addition of a corresponding test class in around 64% of the cases.

Association rule 1.2

ADDED_CLASS_PRODUCTION=YES $\rightarrow$ CLASS_DECLARATION_TEST=NONE

support: 557, confidence: 0.871

The second rule reveals that sometimes when a production class is created additional test cases are developed in the already existing test classes in order to cover it. Even though the value of CLASS_DECLARATION_TEST is NONE, the confidence of this rule indicates that in roughly 13% of the cases a different value than NONE was registered; therefore, in these situations at least one test case is created when the new production class is added.

A. Co-Evolution Patterns

By inspecting the association rules from Table IV we have noticed a number of interesting differences between the systems under analysis. We have identified 6 co-evolution patterns (shown in Table V) that we distilled from Table IV by generalizing what has been observed for the 5 projects. These 6 co-evolution patterns are further subdivided into two categories: *positive*, marked by the *a* suffix, and *negative* as exemplified by the *b* suffix. The positive patterns reflect co-evolution, while the negative ones point towards a lack of co-evolution.

We will now discuss these co-evolution patterns per project.

1) CommonsLang: In the case of *CommonsLang*, an extensively tested project, a series of co-evolution patterns (we are referring to the numbering of Table V) have been observed, namely:

Pattern 1a The generated association rule shows that corresponding test classes are indeed created when new production classes are developed (confer rule 1 in Table IV, CONF = 0.643), suggesting that the developers actually test the production code they write.

Pattern 2a Another rule has uncovered that in most of the cases test classes are removed (rule 2, CONF = 0.998) when the production classes they cover are deleted, indicating that the programmers are careful not to leave non-compiling test classes in the system.

Pattern 3a The rules highlight that when a certain number of methods are added / removed from production classes corresponding test cases are also created / deleted (rules 4–6).

Patterns 4a and 5a They also show that test cases are updated accordingly when attribute or method related changes are made in the production code (rules 7-14).

Pattern 6a Finally, the association rules uncovered that test cases are created / removed when conditional statements are changed in the production classes (rules 16-18).

The patterns presented above indicate that thorough testing has been done for *CommonsLang*, which is in concordance with the initial observations that we made regarding this system.

2) CommonsMath: In general, the association rules generated for *CommonsMath* resemble the ones obtained for *CommonsLang*; however, from the confidence values retrieved, it can be observed that less emphasis has been put on testing for this project.

Patterns 1a and 3a For example, when new production classes / methods are developed corresponding test classes / cases are created, but their number is slightly lower than in the *CommonsLang* case (rules 1 and 4–6 respectively)

Pattern 2a Associated test classes are removed when production classes are deleted (rule 2, CONF=0.974).

Patterns 4a and 5a Test cases are altered accordingly in situations when the attributes / methods of a production class are modified (rules 9–10 and 13–14).

Pattern 6a Tests are written / dropped when conditions are changed in the production code (rules 15–18).

Even though *CommonsMath* is not tested in such detail as *CommonsLang*, the project can still be considered adequately tested as only positive co-evolution patterns occur.

3) PMD: Most of the association rules that were generated for *PMD* are negative (*i.e.,* have NONE as the value for the test related changes).

Pattern 1b We see strong indication that for *PMD* test classes are not developed when production classes are created (rule 1, CONF=0.906).

Id	Association Rule	PMD	CommonsLang	CommonsMath	JFreeChart	Gson
1	ADDED_CLASS_PRODUCTION=YES → ADDED_CLASS_TEST	NO 4161/0.906	YES 412/0.643	SOMETHING -/-	NO 832/0.805	YES 85/0.772
2	REMOVED_CLASS_PRODUCTION=YES → REMOVED_CLASS_TEST	YES 4926/0.998	YES 569/0.998	YES 1554/0.974	YES 1331/0.954	YES 89/0.936
3	CLASS_DECLARATION_PRODUCTION=LOW → CLASS_DECLARATION_TEST	NONE 1767/0.998	NONE 244/0.953	NONE 1129/0.981	NONE 360/0.954	NONE 159/0.975
4	CLASS_DECLARATION_PRODUCTION=MED_LOW → CLASS_DECLARATION_TEST	-	LOW 132/0.8	SOMETHING -/-	-	-
5	CLASS_DECLARATION_PRODUCTION=MED_HIGH → CLASS_DECLARATION_TEST	NONE 855/0.808	SOMETHING -/-	SOMETHING -/-	NONE 174/0.754	LOW 43/0.651
6	CLASS_DECLARATION_PRODUCTION=HIGH → CLASS_DECLARATION_TEST	NONE 503/0.797	HIGH 85/0.658	SOMETHING -/-	NONE 102/0.743	SOMETHING -/-
7	METHOD_DECLARATION_PRODUCTION=LOW → BODY_STATEMENTS_TEST	NONE 634/0.725	SOMETHING -/-	NONE 331/0.614	NONE 134/0.814	NONE 60/0.833
8	METHOD_DECLARATION_PRODUCTION=MED_LOW → BODY_STATEMENTS_TEST	-	SOMETHING -/-	-	-	-
9	METHOD_DECLARATION_PRODUCTION=MED_HIGH → BODY_STATEMENTS_TEST	NONE 309/0.887	SOMETHING -/-	SOMETHING -/-	NONE 65/0.893	SOMETHING -/-
10	METHOD_DECLARATION_PRODUCTION=HIGH → BODY_STATEMENTS_TEST	NONE 234/0.823	MED-HIGH 37/0.616	SOMETHING -/-	NONE 49/0.871	SOMETHING -/-
11	ATTRIBUTE_DECLARATION=LOW → BODY_STATEMENTS_TEST	NONE 1244/0.737	SOMETHING -/-	NONE 558/0.722	NONE 148/0.833	NONE 82/0.828
12	ATTRIBUTE_DECLARATION=MED_LOW → BODY_STATEMENTS_TEST	-	SOMETHING -/-	-	-	-
13	ATTRIBUTE_DECLARATION=MED_HIGH → BODY_STATEMENTS_TEST	NONE 528/0.907	SOMETHING -/-	SOMETHING -/-	NONE 62/0.853	SOMETHING -/-
14	ATTRIBUTE_DECLARATION=HIGH → BODY_STATEMENTS_TEST	NONE 628/0.834	SOMETHING -/-	SOMETHING -/-	NONE 74/0.822	SOMETHING -/-
15	BODY_CONDITIONS_PRODUCTION=LOW → CLASS_DECLARATION_TEST	NONE 1044/0.976	NONE 126/0.670	SOMETHING -/-	NONE 94/0.853	NONE 72/0.9
16	BODY_CONDITIONS_PRODUCTION=MED_LOW → CLASS_DECLARATION_TEST	-	SOMETHING -/-	-	-	-
17	BODY_CONDITIONS_PRODUCTION=MED_HIGH → CLASS_DECLARATION_TEST	NONE 357/0.952	SOMETHING -/-	SOMETHING -/-	NONE 134/0.763	NONE 37/0.822
18	BODY_CONDITIONS_PRODUCTION=HIGH → CLASS_DECLARATION_TEST	NONE 430/0.926	SOMETHING -/-	SOMETHING -/-	SOMETHING -/-	SOMETHING -/-

TABLE IV: Associations rules mined from the evolution of the analyzed projects.

Pattern	Explanation	CommonsLang	CommonsMath	PMD	Gson	JFreeChart
1a	When a new production class is added, an associated test class is also created	√	√		√	
1b	When a production class is created, no new class is added in the test code			√		√
2a	Upon the deletion of a production class, its associated test class is also removed	√	√	√	√	√
2b	When a class from the production code is removed, the test class covering it is not deleted					
3a	When a new production method is created, one or more test cases addressing it are also developed	√	√		√	
3b	Upon the addition of a method in the production code, no new test cases are created			√		√
4a	When method-related changes occur in the production code, the tests are updated accordingly	√	√		√	
4b	When modifications are made to the signature or return type of a production method, no changes occur in the test code			√		√
5a	When a field is added in the production code, the eisting test cases are updated in order to address this change	√	√		√	
5b	When a new production field is added, no modifications occur in the test code			√		√
6a	Upon modifying conditional statements in methods from the production code, new test cases are created to cover each possible path throughout the respective method	√	√		√	√
6b	When conditions are changed in production methods, no new test cases are added			√		

TABLE V: Co-evolution patterns for each system under study.

Pattern 4b and 5b In the cases when attributes or methods from the production classes are modified tests are rarely changed; only a limited amount of updating is done in the test code to ensure that the test cases still compile (rules 7, 9-10, 11, and 13-14).

Pattern 6b Also, test cases are not created / deleted when conditional statements are modified in production methods (rules 15 and 17-18).

From the patterns that were inferred, it is clear that *PMD* does not have a structured approach to co-evolving production and test code. This observation is in-line with our initial assessment that *PMD* is a poorly tested project.

4) Gson: In most cases, the rules generated for *Gson* are similar to the ones obtained for *CommonsLang* and *CommonsMath*.

Pattern 3a In contrast to the aforementioned *CommonsLang* and *CommonsMath*, for *Gson* we have found that when methods are added / removed from production classes, the number of test cases created / deleted is significantly lower in comparison to the other two projects (rules 5-6); nevertheless, a positive sub-pattern was still detected.

Pattern 6a Also contrasting *CommonsLang* and *CommonsMath*, only when numerous condition related changes are made in the production methods, test cases are created /

deleted in order to address the additional / removed branches (rule 18).

We conclude that *Gson* can be regarded as a well-tested project as most of the changes in the production code are accompanied by changes in the test classes.

5) JFreeChart: *JFreeChart* is a project that is not tested as extensively as *CommonsLang*, *CommonsMath* or *Gson*. Generally, the association rules that have been obtained in this case resemble the ones that were generated for *PMD*.

Patterns 1b and 3b Even though new production classes / methods are not backed up by additional test classes (rule 1) / cases (rules 3 and 5–6), we still see that the testing effort put into *JFreeChart* is higher compared to *PMD*'s case, because the negative association rules have a lower value for confidence.

Patterns 4b and 5b We observe that test cases are rarely updated when changes related to attributes or methods are made in the production code (rules 7, 9–10, 11, 13–14).

Pattern 6a In several cases we have noticed that test methods are created / deleted when conditional statements are altered in the production classes (rule 18).

The amount of testing that has been done while developing *JFreeChart* is on the low side, as indicated by the numerous negative association rules that were obtained.

V. Qualitative Analysis

The quantitative analysis has provided insight into the co-evolution of production and test code: 6 fine-grained co-evolution patterns have been identified for the 5 projects under analysis. We now turn towards a qualitative analysis that is aimed at 1) manually investigating how co-evolution happens and 2) interpreting the observed co-evolution patterns.

A. How Co-Evolution Happens

Examples of test code changes that occur when specific changes are made in the production code have been manually analyzed. In particular, we consider the association rules 1.1 and 1.2 from the previous section.

For *CommonsLang* we have determined that in most cases a new test class is indeed created when a production class is developed. We have come across the following 4 scenarios:

1) Occurs in the same commit: The test class is generally added during the same commit (in roughly 90% of the cases), thus suggesting that the developers actually test the new production code before committing it.

2) Occurs in a following commit: We have noticed situations in which the corresponding test class is developed during a following commit. This indicates that even though the production class was not tested at the time of its creation, the respective production code is still covered (at a later time).

3) Does not occur, but a different type of change is made in the test code: Cases in which a multitude of different types of test changes occur when a new production class is created have also been identified. This corresponds to a scenario in which the developers update the already existing tests instead of developing a separate test class to address the production class that was added. We have observed the following changes in the test classes: the creation of test cases (corresponding to association rule 1.2), the insertion of statements containing method calls, and the addition of catch blocks.

4) Does not occur: In some cases we have witnessed that a test class is not added when a production class is developed (in about 35% of the total number of cases). When such a situation occurs, the production code corresponds to either a mock class, an abstract class / an interface, or a class that is reimplemented (for which a test class does exist). Cases in which important production classes were not covered by tests have rarely been seen for *CommonsLang*.

The examples listed above show that different things can happen in the test classes as a result of a change in the production code. They have also demonstrated that in the cases when a change is made in the tests, it does not necessarily happen in the same commit as the production change that triggered it; therefore, a number of subsequent commits have to be inspected in order to ensure that all the test changes that occur due to a specific production change have been identified. Furthermore, this qualitative analysis has lead to other insightful findings; for example, if no changes are observed in the tests when a production class is created, the analysis uncovered examples of reasons why changes are not necessarily needed in those particular situations.

B. Interpretation of the fine-grained co-evolution patterns

As explained in Section III, for 5 of the changes that occur more frequently in the production code, we investigate the associated changes that are made in the test classes in greater detail. The considered changes are (1) class addition, (2) method addition, (3) class removal, (4) field addition and (5) alternative condition block addition. For each of these types of changes, we collect examples of test changes that they trigger and study them.

1) Class addition: In terms of entire production class additions, we observe a number of interesting facts. First of all, we see that the addition of a production class triggers the creation of a corresponding test class for the projects that are adequately tested (rule 1 in Table IV). When this does not happen, we have determined that the new production classes are either auxiliary classes or abstract classes / interfaces for which the classes that extend / implement them *are* tested. Another situation that we have witnessed is that production classes are removed and subsequently added again (therefore a test class already exists for them). For the other two projects, *PMD* and *JFreeChart*, the development of corresponding test classes was observed less frequently; the developers seem to prefer adding test classes that contain integration tests which cover multiple production classes that were recently created. For all the systems that we have analyzed, the new test class is generally developed in the same commit as the production class it addresses. Additionally, we have found other types of changes in the test code when production classes are created. The most commonly observed ones are (1) the addition of new test cases in the already existing test classes and (2) statement-level changes in some of the test methods. For the two systems that are tested less, these kinds of changes occur more frequently than the insertion of test classes.

2) Method addition: We also zoom in on the changes that are made in the test code when production methods are added. Intuitively we understand that the creation of a method in the production code should trigger the addition of at least one new test case. However, this expectation is fulfilled by only 3 of the analyzed projects, *CommonsLang*, *CommonsMath* and *Gson*; for the other two, this was rarely the case (rules 3–6). Even for the adequately tested projects, there are situations in which no changes are made in the test code when this type of change occurs in a production class. Upon further investigation we have established that the production methods that are not backed up by additional test cases are generally part of abstract or mock classes; therefore, the fact that they are not addressed does not represent a serious issue. Nevertheless, in some cases new utility methods have not been tested, thing that could prove problematic. In general, we see that the corresponding test cases are added in the same commit as the production methods they cover. There are few cases in which they are developed in a following commit. We have also found other types of test code changes, most of which are at a statement-level, corresponding to updates to the already existing test cases by inserting or modifying a number of statements.

3) Class removal: With regard to production class removals, we have determined for all the 5 projects that if an associated test class exists, it is also deleted (rule 2). However, we did find situations in which the test class is not removed in the same commit as the production class it covers. For example, in the case of *PMD*, the `TokenSetTest` class is discarded two commits after `TokenSet` is deleted. This is particularly interesting considering the fact that compilation errors arise because the production class that is being tested was already removed. Changes from other categories have also been identified in the test code. For example, statement deletions and updates have been encountered in the tests, suggesting that test cases that address more than one production class are modified accordingly. In some cases we have noticed that methods from multiple test classes are removed, indicating that the respective production class was covered by more than one test class.

4) Field addition: The addition of fields in production classes was also inspected. We have observed several types of changes in the test code in this case, especially for the systems with a higher testing effort. In a number of cases adding a field in the production code co-occurs with the creation of a new test case. A deeper inspection revealed that the test case does not specifically address the respective field, but rather a production method that uses it. We have also noticed statements being inserted in the existing tests, with which the field from the production class is covered. In some cases, a field is added in a test class as well; it corresponds to one of the fields introduced in the production code and is used all throughout the tests. For the projects that are tested less, *PMD* and *JFreeChart*, all the test changes mentioned above occur less frequently compared to the other 3 systems. Especially in the case of *PMD*, the developers seem to completely disregard this type of production change when it comes to testing, as generally no changes can be observed in the test classes.

5) Alternative condition block addition: Finally, we study the insertion of alternative conditional blocks. For the adequately tested projects we see two types of changes in the tests. First and foremost, new test cases are usually created when alternative conditional blocks are added in the production code (rules 15–18). This indicates that the developers adhere to the guidelines specified for unit testing which state that a test case should be created for each independent path through the tested method. However, there are situations in which they altered existing test cases instead of adding new ones. Through code inspections we have determined that various statement-level changes are done in the tests, such as modifying the values of the parameters with which a production method is called in order to trigger a different path through the respective method. We have rarely observed cases in which no changes are made in the test code for two of the systems, *CommonsLang* and *CommonsMath*. For the other 3 projects, *Gson*, *PMD* and *JFreeChart*, our findings are significantly different. Although there are some situations in which the existing test cases are changed when alternative conditional blocks are inserted in production methods, in general new test cases are not created. Most of the times the developers do not make any kinds of changes in the test code for these projects.

VI. Discussion

In this section we first summarize our findings with regards to the two research questions addressed by this paper. Then we discuss threats to validity that might affect our study.

A. Revisiting the research questions

a) RQ1. What kind of fine-grained co-evolution patterns between production and test code can be identified?: By using association rule mining we have observed 6 fine-grained co-evolution patterns in our 5 case study systems (see Table V). These 6 patterns can be summarized as follows: (1) simultaneous introduction of production and test class (patterns 1a and 1b), (2) simultaneous deletion of production class and associated test class (2a and 2b), (3) introduction / deletion of production method leads to the addition / removal of one or more test cases (3a and 3b), (4) modification of production method leads to statement-level changes in the test cases (4a and 4b), (5) production field changes lead to statement-level changes in the test cases (5a and 5b), (6) conditional statement changes in the production code lead to the addition / deletion of test cases (6a and 6b).

A more qualitative analysis revealed how the co-evolution takes place, to be more precise, whether it happens simultaneously or not. Additionally, this in-depth analysis also goes into the reasons why sometimes the patterns are not upheld, *e.g.* a test class is not added for a mock class (pattern 1b).

b) RQ2. Does the testing effort have an impact on the observed co-evolution patterns?: As a first step, we have evaluated the testing effort put into each of the 5 case study projects. This has been done on the basis of 4 criteria: (1) the ratio between the number of lines of test code and the number of production code lines, (2) the number of versions that did not compile because of test failures, (3) branch coverage, and (4) the ratio between the number of changes in the test code and the total number of changes for the respective project. Based on these measurements, the systems have been classified as: extensively tested (CommonsLang), adequately covered (CommonsMath, Gson) and poorly tested (PMD, JFreeChart).

For each of the 6 patterns that we observed, we have distinguished a *positive* and a *negative* sub-pattern, namely the positive or "a" pattern in which co-evolution does occur and the negative or "b" pattern in which case the co-evolution was absent. From this classification, our main observation is that for the software projects for which we have seen high testing effort, *i.e.* CommonsLang, CommonsMath and Gson, the positive patterns are more likely to occur. Similarly, for the two systems for which we have observed a less intense testing effort (PMD and JFreeChart), the negative patterns are more common. However, there are cases in which a pattern from the a group can be found in a project with a lower testing effort; for example, for JFreeChart we have established that test cases are sometimes created / removed when conditional statements are modified in the production code.

Class removal is the only production change for which the same pattern has been identified for all the 5 projects; in this case, the associated test class is deleted as well, which is unsurprising considering the fact that it would cause errors during compilation if it were to be left in the code.

B. Threats to validity

The following threats to validity have been identified:

Internal threats: Internal threats might be caused by issues in the code that has been developed in order to collect the information regarding production / test code changes and links between tests and corresponding production classes. In order to mitigate these threats, our approach has been thoroughly tested using a number of small examples to ensure that it works properly. Additionally, we have performed a manual inspection of the data obtained for each of the projects in order to be certain that the changes extracted and the coverage information inferred are correct.

External threats: Our observations may not be generalizable to other systems. More specifically, all 5 projects are open-source, therefore the results that we have obtained might not apply to commercial systems. In particular, different patterns might be uncovered for industrial projects compared to the ones gathered for the 5 systems included in the analysis. As discussed in Section III, the investigated projects have been chosen based on several criteria, therefore our findings should be valid for a wide range of software systems. Future replications of the study should rule out this threat to validity.

Finally, the support and confidence values that were used when generating the association rules might also represent an external threat to validity. Different association rules would have been obtained if different values for support and confidence were utilized. We aim for the rules to be as reliable as possible, therefore we decided not to lower these thresholds.

VII. Related Work

This section covers similar work from fellow researchers.

Gall *et al.* reported on analyzing the information obtained by mining software repositories [14]. Two tools are introduced: Evolizer, which is a platform for mining software repositories, and ChangeDistiler [10], a change extraction and analysis tool that can be used to investigate fine-grained source code changes. Of particular interest to us is ChangeDistiller, which extracts source code changes from the different versions of a Java class gathered with Evolizer. The source code of each analyzed version is represented as an abstract syntax tree (AST) and the changes between two versions are determined by computing the differences between their corresponding ASTs. A taxonomy for source code changes has also been defined along with the significance level of each type of change. We rely on their work in this paper.

Pinto *et al.* [13] investigate how unit test suite evolution occurs. Their main finding is that test repairing is an often occurring phenomenon during evolution, indicating for example that assertions are fixed. The study also shows that test suite augmentation is another important activity during evolution aimed at making the test suite more adequate. One of the most striking observations that they make is that failing tests are more often deleted than repaired. Among these deleted test cases, tests fail predominantly (over 92% of the time) with compilation errors, whereas the remaining ones fail with assertion or runtime errors. In a controlled experiment on refactoring in connection with developer tests, Vonken and Zaidman also note that participants usually deleted failing assertions instead of trying to address them [15].

In the context of test suite augmentation, Santelices *et. al.* [16] present an enhanced methodology for improving existing tests as a result of evolving software, that can be used to (1) asses the adequacy of a regression test suite when changes are made in the production code, and (2) facilitate the generation of new test cases that cover the untested behaviours introduced by the production changes.

Zaidman *et al.* have proposed a set of visualizations that aid in understanding how production code and (developer) test code co-evolve [8]. Their analysis is coarse-grained, as they only inspect whether a production / test file is added or changed, while our analysis is much more fine-grained. The authors have observed that the co-evolution does not always happen in a synchronized way, *i.e.*, sometimes there are periods of development followed by periods of testing. Lubsen *et al.* have a similar goal, however they use association rule mining to determine co-evolution [17]. Their work is particularly close to ours, albeit they study the co-evolution at a file level, while we focus on more fine-grained changes.

In response to observations on the lack of co-evolution, Hurdugaci and Zaidman [12] and Soetens *et al.* [18] proposed ways to stimulate developers to co-evolve their production and test code by offering specialized tool support.

VIII. Conclusions and Future Work

In this paper we have investigated the fine-grained co-evolution of production and test code. We did this in order to: (1) gain a deeper understanding of the way in which tests evolve as a result of changes in the production classes, (2) identify possible gaps in the test base, thus signalling to the developers the parts of the production code that have not been adequately addressed by tests.

In doing so, we make the following contributions:

- We present an approach to study the fine-grained co-evolution between production and test code.
- We perform an empirical study on 5 open-source software projects, thereby gaining insight into how co-evolution does (not) occur.
- We identify 6 co-evolution patterns based on this empirical study.

We are now in a position to answer our research questions. For **RQ1**, *"What kind of fine-grained co-evolution patterns between production and test code can be identified?"*, we have uncovered 6 fine-grained co-evolution patterns by using an association rule mining technique. For each of these patterns, a positive and a negative sub-pattern have been identified. The positive patterns reflect co-evolution, while the negative ones point towards a lack of co-evolution.

For **RQ2**, *"Does the testing effort have an impact on the observed co-evolution patterns?"*, we first determine the testing effort put into each of the 5 projects. Afterwards, we have established that positive patterns are more likely to be encountered in thoroughly tested software systems (*i.e.,* CommonsLang, CommonsMath, Gson), while the negative ones are generally seen in projects for which the testing effort is low, such as PMD or JFreeChart. The qualitative evaluation that we have performed allowed us to gain a more in depth understanding of how the co-evolution takes place. In particular, we have found reasons why negative co-evolution patterns are obtained. For example, we now have insight as to why sometimes new test classes are not added when production classes are created (*e.g.,* because the later is a mock class).

Future work. A first direction for future work entails extending the empirical study by analyzing the co-evolution between the production and the test code of new projects, especially from industry. Commercial systems in particular might exhibit different co-evolution patterns as more or less testing effort may have been put into them.

Another area to concentrate on is studying whether specific development methodologies (*e.g.,* Test-Driven Development) shows different co-evolution strategies.

We also aim to improve the characterization of testing effort by making use of the recent test code quality model presented by Athanasiou *et al.* [19].

Finally, we want to use the knowledge that has been obtained through this empirical study to look into test repair techniques. Of particular interest are intent-preserving techniques, assuring that the repaired test cases address the same production functionalities as before they were broken.

Acknowledgment

This work was partly funded by the NWO TestRoots project (project number 639.022.314).

References

[1] M. Lehman, "On understanding laws, evolution and conservation in the large program life cycle," *Journal of Systems and Software*, vol. 1, no. 3, pp. 213–221, 1980.

[2] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri, "Challenges in software evolution," in *Proc. Int'l Workshop on Principles of Software Evolution (IWPSE)*. IEEE, 2005, pp. 13–22.

[3] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007.

[4] P. Runeson, "A survey of unit testing practices," *IEEE Software*, vol. 25, no. 4, pp. 22–29, 2006.

[5] S. Demeyer, S. Ducasse, and O. Nierstrasz, *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[6] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink, "The interplay between software testing and software evolution," in *Software Evolution*. Springer, 2008, pp. 173–202.

[7] S. Elbaum, D. Gable, and G. Rothermel, "The impact of software evolution on code coverage information," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE CS, 2001, pp. 170–179.

[8] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer, "Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining," *Empirical Software Engineering*, vol. 16, no. 3, pp. 325–364, 2011.

[9] J. Creswell and V. Clark, *Designing and Conducting Mixed Methods Research*. SAGE Publications, 2010.

[10] B. Fluri, M. Würsch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Software Eng.*, vol. 33, no. 11, pp. 725–743, 2007.

[11] B. Van Rompaey and S. Demeyer, "Establishing traceability links between unit test cases and units under test," in *Proc. Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 2009, pp. 209–218.

[12] V. Hurdugaci and A. Zaidman, "Aiding software developers to maintain developer tests," in *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2012, pp. 11–20.

[13] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012, p. 33.

[14] H. Gall, B. Fluri, and M. Pinzger, "Change analysis with evolizer and changedistiller," *IEEE Software*, vol. 26, no. 1, pp. 26–33, 2009.

[15] F. Vonken and A. Zaidman, "Refactoring with unit testing: A match made in heaven?" in *Proc. of the Working Conf. on Reverse Engineering (WCRE)*. IEEE Computer Society, 2012, pp. 29–38.

[16] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "Test-suite augmentation for evolving software," *Proc. ASE*, pp. 218–227, 2008.

[17] Z. Lubsen, A. Zaidman, and M. Pinzger, "Using association rules to study the co-evolution of production & test code," in *Int'l Working Conf. on Mining Software Repositories (MSR)*. IEEE, 2009, pp. 151–154.

[18] Q. D. Soetens, S. Demeyer, and A. Zaidman, "Change-based test selection in the presence of developer tests," in *Proc. Conf. on Software Maintenance and Reengineering (CSMR)*. IEEE, 2013, pp. 101–110.

[19] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test code quality and its relation to issue handling performance," *Transactions on Software Engineering*, *To appear.*