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

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.

in [12]. To retrieve the covered entities (*e.g.,* Java classes) we process test coverage information gathered with Cobertura[1].

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 test method. We record these results which are used afterwards

[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.

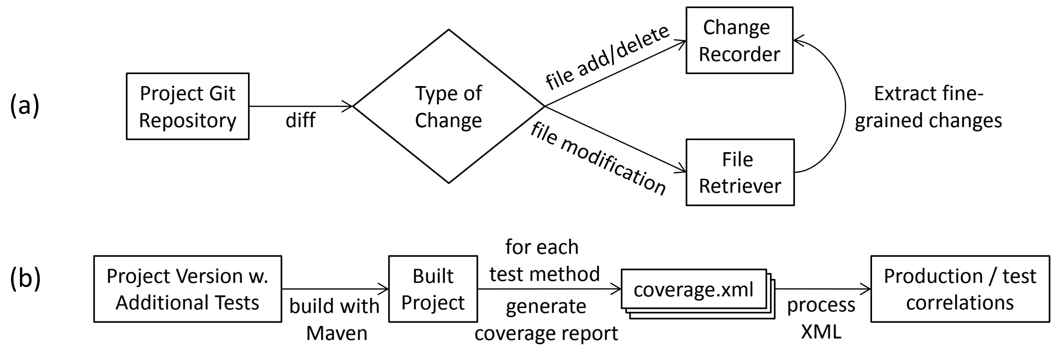

Fig. 1: Overview of the data collection process.

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-	$\Sigma_{\forall v_i} LOC_{test}$
	# Versions	# Classes	# Prod. Methods	# Test Methods	Release	# Classes	# Prod. Methods	# Test Methods	Release	Branch Coverage	Building Versions 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
$\text{Total}_{Test}/(\text{Total}_{Test}+\text{Total}_{Prod})$	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 → 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 → 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).

Pattern 4b and 5b In the cases when attributes or methods from the production classes are modified tests are rarely

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.

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*.