Delft University of Technology
Software Engineering Research Group
Technical Report Series

Studying Co-evolution of Production & Test Code Using Association Rule Mining

Zeeger Lubsen, Andy Zaidman, Martin Pinzger

Report TUD-SERG-2009-014

TU Delft

SERG

TUD-SERG-2009-014

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:
http://www.se.ewi.tudelft.nl/techreports/

For more information about the Software Engineering Research Group:
http://www.se.ewi.tudelft.nl/

Note: Long version of the short paper accepted for publication in the proceedings of the 6th International Working Conference on Mining Software Repositories (MSR 2009)

Studying Co-evolution of Production & Test Code Using Association Rule Mining*

Zeeger Lubsen
Software Improvement Group
The Netherlands
z.lubsen@sig.nl

Andy Zaidman, Martin Pinzger
Delft University of Technology
The Netherlands
{a.e.zaidman, m.pinzger}@tudelft.nl

Abstract

Unit tests are generally acknowledged as an important aid to produce high quality code, as they provide quick feedback to developers on the correctness of their code. In order to achieve high quality, well-maintained tests are needed. Ideally, tests co-evolve with the production code to test changes as soon as possible. In this paper, we explore an approach to determine whether production and test code co-evolve synchronously. Our approach is based on applying association rule mining to the change history of product and test code classes. Based on these co-evolution rules, we introduce a number of measures to assess the co-evolution of product and test code classes. Through two case studies, one with an open source and another one with an industrial software system, we show that association rule mining and our set of measures allows one to assess the co-evolution of product and test code in a software project and, moreover, to uncover the distribution of programmer effort over pure coding, pure testing, or a more test-driven-like practice.

1 Introduction

The development of high quality software systems is a complex process; maintaining an existing system is often no less challenging, an insight which Lehman formulated in his Laws of Software Evolution [11]. Runeson on the other hand notes that automated unit testing[1] can be an effective countermeasure for difficulties encountered during software maintenance [14]. Also Test-Driven Development (TDD) [3] and test-driven refactoring [13] can play an important role here.

The quality of the tests — and by consequence the added value for maintenance activities — greatly depends on the effort that the developers put into writing and maintaining tests. Typically, the quality of a test suite is expressed by code coverage: the percentage of the code that is exercised by the test suite that is executed [4]. Code coverage, however, is a shallow measure of test quality as it expresses that code is executed, but not how (well) something is tested. In this context, one should think of (1) different input values — boundary values — and (2) the number of assertions [4, 19]. Furthermore, code coverage does not provide a good indicator for the long term quality or *"test health"* of a test suite. As such, we have no insight into (1) how well test code was adapted to previous changes in the production code, (2) the current structure of the test code, and (3) how easy it will be to perform maintenance on both the production and the test code in the future.

This missing insight has motivated us to investigate the co-evolution of production and test code. In our previous work, we introduced the Change History View [19] to observe and perform a qualitative analysis of the co-evolution of production and test code by mining version control data. While change history views provide sufficient insights into the co-evolution of production of test code they require a fair amount of human effort to understand and interpret. The user might be overwhelmed by the amount of information represented by a single change history view.

In this paper, we address this shortcoming by adding a quantitative analysis approach to study the co-evolution of production and test code. In particular, we investigate whether association rule mining can be applied to study the co-evolution of test and production code and provide answers to the following research questions:

RQ1: Can association rule mining be used to find evidence of co-evolution of production and test code?

RQ2: Following RQ1, can we find measures to assess the extent to which product and test code co-evolves?

RQ3: Can different patterns of co-evolution be observed in distinct settings, for example, open source versus industrial software systems?

We address these research questions by means of two case studies. The first case study is on Checkstyle, an open source system that checks whether code adheres to a coding standard. The second case study is on an industrial software

*This work is described in more detail in the MSc thesis of Zeeger Lubsen [12].

[1] xUnit Testing Frameworks: http://www.xunit.org

system from the Software Improvement Group (SIG).[2]

The structure of this paper is as follows: in Section 2 we introduce association rule mining and explain our specific approach. Sections 3.1 and 3.2 deal with our two case studies, respectively Checkstyle and the industrial system provided by SIG. Section 4 deals with threats to validity. Section 5 relates our work to other work in the field and we present our conclusions and future work in Section 6.

2 Production and test class co-evolution

The application of data mining techniques in software engineering research has become popular [17]. This can partly be explained by the fact that software engineers are looking at studying large sets of data for which efficient analysis methods are required. Within the realm of data mining, we have chosen to use *association rule mining*, because this technique allows us to identify instances of logical coupling between classes [20], in particular between production and test classes. For this paper, production code/classes refer to Java classes and test code/classes to jUnit test classes.

The basic idea of our approach is to use association rule mining to study the co-evolution of test and production code. The change history of test and production classes, in particular commit transactions, form the input to our approach. Information about commit transactions are obtained from versioning repositories, such as, the concurrent versions systems (CVS) or Subversion (SVN). In the following, we provide background information of association rule mining and the set of metrics that we use to study co-evolution of production and test classes.

2.1 Association rule mining

Formally, an association rule is a statistical description of the co-occurrence of elements in the change history that constitute the rule in the change history. Agrawal et al. define it as [1]:

Definition 1 *Given a set of items* $I = I_1, I_2, ..., I_m$ *and a* ***database of transactions*** $D = t_1, t_2, ..., t_n$ *where* $t_i = I_{i1}, I_{i2}, ..., I_{ik}$ *and* $I_{jk} \in I$*, an* ***association rule*** *is an implication of the form* $A \Rightarrow B$ *where* $A, B \subset I$ *are sets of items called* itemsets *and* $A \cap B = \emptyset$*.*

The left-hand side of the implication is called the *antecedent*, and the right-hand side is called the *consequent* of the rule. An association rule expresses that the occurrence of A in a transaction statistically implies the presence of B in the same transaction with some probability. It is important to note that an association rule does not express a causal relation, but rather a spurious one, as the rule does not describe a proven cause-effect relation.

In our approach, we consider association rules that express a binary relation between classes, as we are looking for relations between individual production classes (PC) and test classes (TC). For example, consider the SVN transaction $\{TC_1, PC_1, PC_2\}$ committing changes to the test class TC_1, and the two production classes PC_1 and PC_2. Computing all pairs we get the following binary association rules:
$\{TC_1 \rightarrow PC_1\}$, $\{PC_1 \rightarrow TC_1\}$, $\{PC_2 \rightarrow TC_1\}$,
$\{TC_1 \rightarrow PC_2\}$, $\{PC_1 \rightarrow PC_2\}$, $\{PC_2 \rightarrow PC_1\}$,

Formally, for a transaction involving n classes we obtain $n * (n-1)$ binary association rules. We take into account *inverse* association rules, because the inverse rules can have a different probability, as we explain below.

2.2 Co-evolution rules

In order to analyze the testing practices for an entire system, we need a high-level overview of the development and testing activities of the software system. For that, we classify binary association rules according to rules that deal (1) solely with production code, (2) solely with test code, and (3) that deal with both production and test code. Table 1 shows this classification in detail.

Class	Association rule
TOTAL	The collection of all found association rules.
PROD	{*ProductionClass* ⇒ *ProductionClass*} Rules that only associate production classes.
TEST	{*TestClass* ⇒ *TestClass*} Rules that only associate test classes.
PT	Rules that associate production-test pairs, which we can subdivide into:
P2T	{*TestClass* ⇒ *ProductionClass*}. These rules express that a change in production class implies a change in test class with some probability.
T2P	{*ProductionClass* ⇒ *TestClass*}. These rules express that a change in test class implies a change in production class with some probability.
MP2T	Matching production to test rules; P2T rules where the antecedent and the consequent can be matched to belong together as unit test and class-under-test. These rules express that a change in production code implies a change in test code with some probability.
MT2P	The counterpart of MP2T.

Table 1. Classification of association rules.

While PT comprises association rules between product and test code the sub-classes refine this set by taking the direction of rules into account. The direction of rules comes into play when calculating the *interestingness* of an association rule. Furthermore, we introduce two categories containing rules that denote commit transactions in which a test class has been matched to a production class. For

[2] Software Improvement Group, Amsterdam, The Netherlands. http://www.sig.nl

Metric	Probability	Implementation	Interpretation
support($A \Rightarrow B$)	$P(A,B)n$	*count*	The fraction of commits in which the itemset $\{A, B\}$ appears in the change history. Abbreviated as: s($A \Rightarrow B$).
confidence($A \Rightarrow B$)	$P(B\|A)$	$\frac{s(A,B)}{s(A)}$	The ratio of the number of transactions that contain classes $\{A \cup B\}$ to the number of transactions that contain class A. This measure is not symmetrical.
interest($A \Rightarrow B$)	$\frac{P(A,B)}{P(A)P(B)}$	$\frac{s(A,B)n}{s(A)s(B)}$	Measures the correlation between the two classes A and B, i.e., how many times more often class A and B are contained in a commit transaction then expected if they were statistically independent. This measure is symmetrical.
conviction($A \Rightarrow B$)	$\frac{P(A)P(\neg B)}{P(A,\neg B)}$	$\frac{s(A)n - \frac{s(A)s(B)}{n}}{s(A) - s(A,B)}$	Is a measure of the implication that whenever class A is committed class B is also committed. This measure is not symmetrical.

Table 2. Metrics for individual association rules.

each commit transaction these rules are obtained by comparing the file names of product and test classes. For the comparison we rely on naming convention for test classes and use straightforward string matching. For example, a production class *Class.java* is matched with the test class *ClassTest.java*.

2.3 Co-evolution metrics

Typically, association rule mining is used to search for rules that are "interesting" or "surprising". In our case, we seek to find a *global view* on the entire change history of source files (i.e., top-level Java classes) of a software project. As such, we are mainly interested in the total number of rules that associate production and test classes and how "interesting", i.e., how strong the statistical certainty of these rules is. In the following we explore a number of standard rule significance and interest measurements to measure co-evolution between production and test classes in a software system.

The metrics presented in Table 2 allow us to reason about the significance and interest of *single* association rules. To get an overall understanding of how production and test code co-evolves in a software system we use straight forward descriptive statistics with *boxplots*. Boxplots provide a five-number summary of the distribution of significance and interest metric values. The sample minimum and maximum define the range of the values, while the median designates the central tendency of the distribution. The lower and upper quartile allow reasoning about the standard deviation and together with the median about the skewness of metric values.

These metric-values help us in interpreting the *interestingness* of the association rule classes that we have defined in Section 2.2. If a rule appears in almost all commits, its *support* is close to 100%. While this is unlikely to happen for all commits, finding outliers that exhibit a support close to 100% is interesting, e.g., as they indicate a possible bad design choice if two classes have been changed together that often. The *confidence*-metric is tightly related to the concept of co-evolution. It represents the certainty with which one can expect, for example, when the product class is changed that also the test class is changed. Confidence values higher than 0.5 give a clear indication of co-evolution between classes. The *interest* becomes higher when the rule frequently holds. As for *conviction*, high-quality rules (those that hold 100% of the time) have a value of ∞, while the less interesting rules have a value that approaches 1 (rules from completely unrelated items have a metric-value of 1) [5].

Co-evolution of production and test classes is indicated by rules in PT and its subclasses with significant support, high confidence, interest, and conviction. Separate evolution of product and test classes is indicated by rules in PROD and TEST with significant support, high confidence, interest, and conviction. If the majority of PROD, TEST, and PT rules has low support, we conclude that there is no structural co-evolution between classes.

In addition to the association rule interest measures, we introduce several metrics to measure the extent to which product classes are covered by test classes. The set of metrics is described in Table 3.

Metric	Description
PCC	Production class coverage. The average number of test classes that are changed per changed production code class. This number is calculated by $\frac{\|P2T\|}{\#productionclasses}$.
MPCC	Matching production class coverage. The percentage of production classes that co-evolve with their matched unit test class. This number is calculated by $\frac{\|mP2T\|}{\#productionclasses}$.
TCC	Test class coverage. The average number of production code classes that are changed per changed unit test class. This number is calculated by $\frac{\|T2P\|}{\#testclasses}$.
MTCC	Matching test class coverage. The percentage of test classes that co-evolve with its matched production class-under-test. This number is calculated by $\frac{\|mT2P\|}{\#testclasses}$.

Table 3. Product-test class coverage metrics.

These coverage metrics allow us to get an insight into the testing strategy. More precisely, a high ratio of PCC

and TCC indicates that many production class and test class pairs are changed together. On the other hand, high ratios of mPCC and mTCC indicate that the co-change is structural.

3 Experiments

The main goal of our experiments is to evaluate the applicability of proposed co-evolution metrics to answer the research question stated in the Section 1. For the evaluation, we performed two case-studies, one with the open source system Checkstyle[3] and another one with the industrial software analysis tool from the Software Improvement Group (SIG). For each system we compute the association rule classes and the set of co-evolution metrics. We evaluate and validate our metrics by comparing it with the results obtained by our previous experiments in which we used the Change History View technique and the feedback from developers to reason about co-evolution of production and test classes [19]. In short, the Change History View depicts the evolution of the production code and the test code throughout time (for example see, Figure 1). The X-axis represents time and the Y-axis shows the Java classes. Furthermore, we make a distinction between the creation/change of production code (red/blue dot) and the creation/change of test code (green/yellow dot). A unit test that can be associated to a production code class through naming conventions is placed on the same horizontal line. The usefulness of the Change History View has been demonstrated and validated in previous research [16, 19]. Together with the feedback from the developers (in the case of the SIG case study) it provides the basis for the evaluation and validation of our co-evolution metrics.

In the following, we first present the results from the Change History View analysis that then are compared and discussed with the co-evolution metrics. A summary of the results is given at the end of this section.

3.1 Case study 1: Checkstyle

Checkstyle is an open source coding standard checker for Java source code. Between June 2001 and March 2007, 2259 commits resulted in a total of 1160 Java classes, of which 797 refer to product code, and 363 are identified as a test class.

Change History View Figure 1 depicts the Change History View computed from Checkstyle's change log data. The view shows that initially little testing has been performed. After that, the system started to grow and tests have been added along with new production code. Around revisions 690 and 780, two phases of pure test effort can be distinguished, and after revision 850 tests for most classes existing at that point in time have been added. After these additions, we observe a significant period of pure coding with hardly any maintenance to the tests being performed. The view highlights few recurring test phases around revisions 1380 and 2100. For the larger part of the history, tests appear to receive only minor attention from developers, as only few additions and changes to production code are accompanied or closely followed by the addition or change in a related test classes. An exception to this behavior can be witnessed between commits 1350 and 1600, where for a small period of time new production code classes are accompanied by new unit tests. More striking are regular commits comprising a large number of files as indicated by blue vertical bars. Most of these commits were due to code cleanups or copyright notice changes.

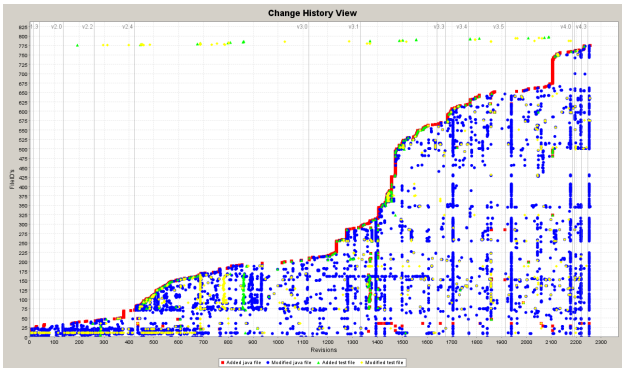

Figure 1. Change History View of Checkstyle.

Co-evolution rule mining The results of the classification of the association rules obtained from the 2259 commit transactions of Checkstyle are depicted in Table 4.

ALL(N)	58566	P2T	0.33%
PROD	98.86%	T2P	0.33%
TEST	0.48%	mP2T	0.09%
PT	0.67%	mT2P	0.09%

Table 4. Rule ratios for Checkstyle.

The ratio of PROD rules shows that $98,86\%$ of the 58566 rules express an association between two production classes. We can explain this through the fact that the developers initially hardly used unit tests, even though they adopted a more test-driven development strategy over time, e.g., between commits 1350 and 1600. The first period of development thus practically only involved production code, but the several phases of pure testing effort that were observed in the Change History View (the vertical green and

[3] http://checkstyle.sourceforge.net/

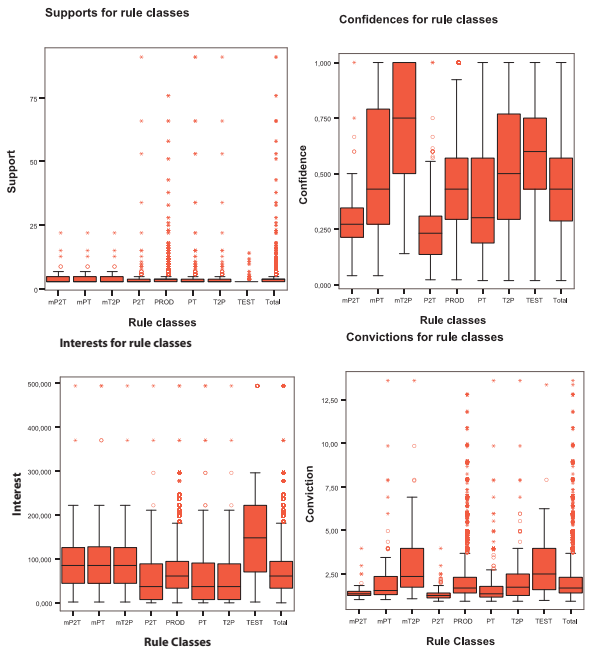

Figure 2. Checkstyle rule strengths distributions.

yellow lines in Figure 1) should have created a fair amount of TEST rules. Closer inspection however, reveals that the testing phases that we have identified involve commits with only a few tests per commit, while many other commits contain a large amount of production classes. As the change history of Checkstyle contains several recurring very large commits, there are many rules being generated from those commits.

Because of the large commits we expect many PROD rules to have a low interest and strength. The boxplots in Figure 2 show that over all association rule classes the support of rules is low, with the PROD rules having several outliers (shown as crosses). This indicates that most of the possible production and test class combinations occurred only in few commit transcations.

The ratios of TEST and PT (sub-)classes are low (see Table 4), even though the Checkstyle developers appear to have adopted a decent testing practice over time; we identified a phased testing approach in the first half of the change history (green and yellow vertical bars in the Change History View), but we also saw a more test-driven approach in the latter part (red dots being covered by green dots, e.g., between commits 1350 and 1600).

Looking at the interest values, the correlation among matching production and test classes (mP2T, mT2P and mPT) is stronger than for more unrelated classes. The correlation among TEST rules is even stronger. This observation also holds for the confidence and conviction distributions, e.g., the confidence of mT2P rules shows that 75% of those rules express a conditional probability of over 50%. Note that that number alone is not enough to conclude synchronous co-evolution between production and test classes, as we do not yet know how many tests are actively maintained.

The boxplots show significantly lower values of mP2T rules for confidence and conviction. This is because (1) mP2T and mT2P rules are not symmetric for confidence and conviction, and (2) the often changing nature of production code makes the presence of a production class in a commit so trivial that no interesting statement can be made based on its presence. The values for interest of mP2T and mT2P rules are identical, because of the symmetry of the interest metric.

In contrast to confidence and conviction, the interest-values for matching production and test classes mT2P are not evidently higher than for mP2T using the interest metric. This is because highly correlated (m)T2P rules are averaged out against the lowly correlated (m)P2T rules. From these results we can make the two following observations:

Observation 1 *High (median) values for TEST and relatively low (support) values for (m)PT rules originates from the co-change of test classes and indicate that testing is performed as a separate activity.*

Observation 2 *Interest averages the measurements for matching rules in different directions. This causes the differences to even out, and makes interest a less specific metric.*

Summarizing, we can see that most of the co-changed Checkstyle classes belong to the production code. These co-changes are mostly unintentional and caused by code cleanup activities, e.g., running Checkstyle on the Checkstyle source code. Looking at the statistics, we mainly see a large class of PROD rules, originating from some very large commits. These commits perturb our analysis somewhat. Still, going by the PT and TEST rule classes, we see some evidence both for a phased and a test-driven approach to testing. The Change History View confirms this, as there are periods where commits consists mainly out of unit tests, while there are also periods in time (e.g., between commits 1350 and 1600), where we see test-driven development taking place. To see these phases in more detail, we aim to investigate a sliding-window-based approach of our analysis to investigate the change history of a software system in more detail.

Co-evolution coverage in Checkstyle We computed the number of co-evolution coverage measures introduced before to quantify the co-evolution of product and test code of Checkstyle. The resulting coverage measures are listed in Table 5.

Coverage metric	Value
Production class coverage (PCC)	0.42
Matching production class coverage (mPCC)	0.11
Test class coverage (TCC)	0.38
Matching test class coverage (mTCC)	0.09

Table 5. Co-evolution coverage metrics for Checkstyle.

For Checkstyle, we see a low value for both PCC and TCC, indicating that for each production class that is changed (on average) only 0.42 test classes are changed (PCC). The other way around, we see that for each test class that is changed, 0.38 production classes are changed. This indicates that co-change does not happen very frequently. If we zoom in a little bit more and look at how structural the co-changes are applied, we see that for 0.11 of the production code classes, the test counterpart that matches based on naming conventions is (potentially) also changed. Vice versa, for 0.09 of the test classes, the matching production class is also (potentially) changed.

These figures should be considered low and as such do not provide any indication that co-evolution of production and test code takes place in the case of Checkstyle.

Discussion For Checkstyle we saw that actual software development and testing are mainly two separate activities, which is mainly evidenced through the rule ratios that we saw in Table 4. However, a possible complication that we came across when interpreting the results was the fact that there are some large commits of (mainly) production code, which dominate the rule ratios to a large extent, thereby perturbing the interpretation. These very large commits originate from automated code beautification operations (using Checkstyle). As such, a possible avenue for further research is to eliminate these large commits and see how this influences the results.

During our interpretation, we also observed large differences between mT2P and mP2T rules when studying the confidence and conviction rules. In particular, we saw that the statistical evidence for mT2P rules was stronger than for mP2T rules. Closer inspection revealed this to be due to commits containing a larger number of production code classes than test code classes, thereby influencing the probabilities behind confidence and conviction.

Considering the average number of production and test classes that are changed together, we can say that in general not many production and test classes are co-evolved as evidenced by the very low PCC and TCC values. This is further underlined by the low mPCC and mTCC values.

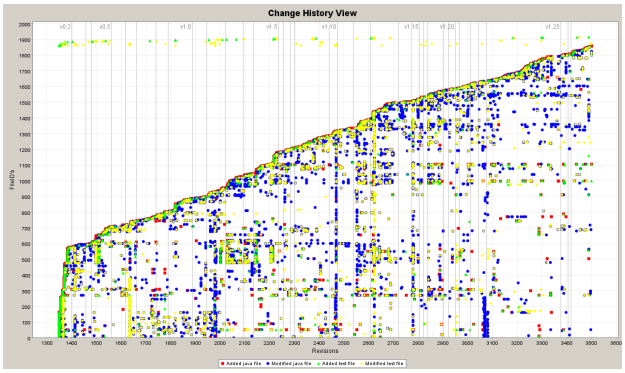

Figure 3. Change History View of the SIG software system.

3.2 Case study 2: Sofware Improvement Group

The industrial case study that we performed pertains to a software project from the *Software Improvement Group* (SIG). The SIG is a tool-based consultancy firm that is specialized in the area of quality improvement, complexity reduction and software renovation. The SIG performs static source code analysis to analyze software portfolios and to derive hard facts from software to assess the quality and complexity of a system.

For our study we investigate the development history of one of the SIG tools between April 2004 and January 2008. Over time 20 developers worked on this software project, which after about 2200 commits resulted in around 4000 classes.

Change History View The Change History View for the industrial case is shown in Figure 3 (also see [16] for more details). From the view we see that the software project shows a steady growth curve and we also observe that code and test writing efforts are overlapping for pretty much the entire change history. Red and blue dots, indicating respectively the addition and change of production classes, are frequently followed by green or yellow dots, indicating the addition and change of unit tests respectively. We investigated the code changes and log messages behind larger commits. We found out that most of these changes correspond to refactorings involving also the test classes, and code cleanups, which did not always involve the test classes.

Co-evolution rule mining The classification of the association rules obtained from the 2200 commit transaction of

the SIG tool resulted in the following ratios listed in Table 6.

ALL(N)	101896	P2T	19.37%
PROD	35.15%	T2P	19.37%
TEST	26.11%	mP2T	0.78%
PT	38.75%	mT2P	0.78%

Table 6. Rule ratios for the SIG tool.

Compared to Checkstyle, the rule classification of the SIG software system results in more evenly partitioned rule classes. In particular, the ratios of PROD (35.15%) and PT (38.75%) are very close to each other. Furthermore, there is a high ratio of TEST rules (26.11%). The high ratio of PT is in line with the observations from the Change History View (Figure 3), where we observe a strong synchronous co-evolution of production and test code for the SIG software system. Not only are the rule class ratios evenly partitioned over pure coding and test-driven development, also the support distribution presents a uniform picture (see Figure 3): PROD, TEST and PT rules show similar measurements, and so do the matching classes mPT, mP2T, and mT2P. The distributions are more uniform (resembling the normal distribution), and show less skewness than for the Checkstyle case.

Of interest to note is the surprisingly large set of TEST rules, which we attribute to commits that contain multiple pairs of production and test code. Such a big set of TEST rules can occur when some combinations of test classes occur often in the history. This can be the result of development cycles including a significant amount of testing.

Continuing on the fact that the number of TEST rules is high, we also see that the support for association rules of this rule class is low. The high confidence and conviction values for the rules must be the result from not many, but from structural co-occurrences. That is, specific combinations of test classes frequently occur together in commits, but these test classes do not occur frequently in other combinations. This indicates that developers focus on writing tests for specific parts of the system. Talking to the developers of the SIG we learned that the software system is actually a collection of analysis tools that grows and changes over time. Developers are assigned to different customers, so their work on the tools is cross-cutting throughout the entire system; this causes more combinations of classes to occur and brings down the correlation between classes, and thus the support for PROD rules. Following the same reasoning, we expect tests to focus on specific parts of the code, as the correlation among tests is high, i.e., high confidence and interest. We shared our findings with the SIG developers who confirmed our insights. The results led us to the following observation:

Observation 3 *High confidence and interest of only production classes (or only test classes) indicate that programmers focus on specific parts of the system (or the test suite).*

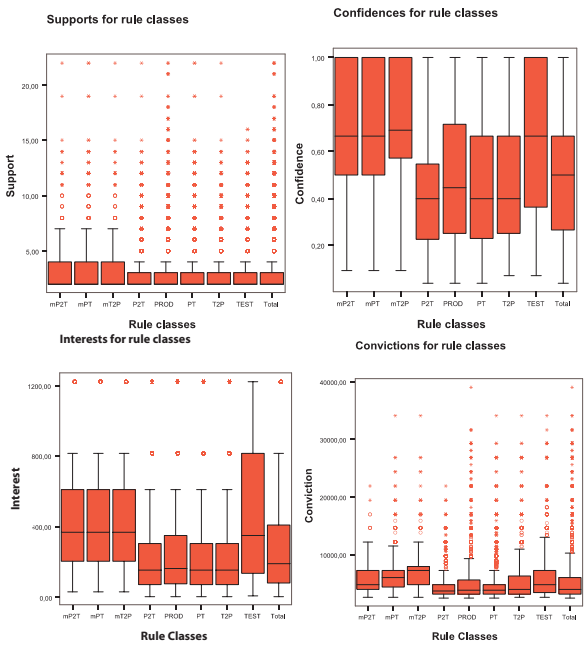

Figure 4. SIG rule strengths distributions.

Co-evolution coverage in SIG Table 7 lists the values obtained for the co-evolution rule coverage metrics.

Coverage metric	Value
Production class coverage (PCC)	7.96
Matching production class coverage (mPCC)	0.32
Test class coverage (TCC)	11.79
Matching test class coverage (mTCC)	0.48

Table 7. Co-evolution coverage metrics for SIG.

We see that for every production class that is being changed, there are also on average 7.96 test classes being changed. The other way around, we see that on average 11.79 production classes are being changed for each test class that is changed. For a more structural view, we look at the mPCC and mTCC values, which indicate how many association rules linking matching production and test code exist, we find that for 32.22% of the production code classes, the associated test class was changed together at least once. Vice versa, 47.70% of the test classes were changed together with their production-code-counterpart at least once.

These coverage metrics indicate that the SIG software development process does indeed follow a more test-driven development strategy, because we have indications that many of the test/production class pairs co-evolve.

Discussion In our industrial case study we observed that the SIG developers are following a development and testing strategy that resembles that of a test-driven development strategy. The first indication is given by the fact that the rule class ratios are fairly evenly distributed over PROD, TEST and PT. Another important indicator for test-driven development are the rule coverage ratios for the SIG software system. Here we saw that for each production class that has been changed, also a significant number of test classes has been changed (and vice versa). This phenomenon is also structural, as also matched production and test class pairs have been changed together.

3.3 Answers to research questions

Based on the results obtained from the two cases studies we can provide the answers to the research questions stated in the introduction of the paper.

RQ1 *Can association rule mining be used to find evidence of co-evolution of production and test code?* The results of our two case studies clearly showed that association rule mining is an adequate technique to investigate the co-evolution in software systems. For the SIG case study we found evidence of co-evolution by looking at the PT rule class, which contained 38.75% of all rules, indicating many co-changes of production and test classes. Furthermore, high support and confidence values for the PT class (and its subclasses) provide further evidence for this co-evolution. For the Checkstyle case study, we did not get a clear indication of intentional co-evolution. This can be attributed to two factors, namely: (1) the co-evolution is only taking place during short periods of time, while our technique is mainly aimed towards providing an overview of longer periods of time, and (2) due to a number of large commits of superficial changes to the production code. This led to an explosion of the number of PROD rules biasing the results.

RQ2 *Can we find measures to assess the extent to which product and test code co-evolve?* Through our case studies we found out that the extent of co-evolution can be measured by the PCC, TCC, and respectively the mPCC and mTCC metrics in combination with the confidence of association rules. In the case of SIG high values for these metrics clearly indicated co-evolution of product and test classes. This result is validated by the Change History View (see Figure 3) and the SIG developers. In case of Checkstyle the values for these metrics are significantly lower indicating little co-evolution. This finding is underlined by the Change History View depicted by Figure 1.

RQ3 *Can different patterns of co-evolution be observed in distinct settings, for example, open source versus industrial software systems?* Our two case studies, of which one was an open source and one was an industrial software system, have shown two different development practices, which affirm a 'yes' to this question. Our metrics indicate test-driven development in the SIG software system while this is not the case for the Checkstyle. Note, that our findings have been validated for these two systems, but, must not be generalized for other open source and industrial software systems.

4 Threats to validity

We have identified a number of threats to validity, which we have classified into threats towards the (i) internal validity, (ii) external validity and (iii) construct validity.

Internal validity The case studies are subjective in the sense that they were performed by the developers of the tools. As a countermeasure we involve external sources of information in our evaluation. More specifically, we used (i) log messages from the developers to confirm or reject our observations — in particular for the observations from the Change History View [16, 19] — and (ii) we used the insights that we have previously obtained when researching the same software projects. These insights were confirmed by the developers of the software projects [16].

Our tool-chain might contain faults which explain the results of the case studies. As a countermeasure, we thoroughly tested our tool-chain.

External validity While we have chosen two case studies that are very different from each other — in terms of problem domain, in terms of closed/open source development, etc. —, they might not be representative. For example, during our case studies we have observed a test-driven-like development process, but at this point we are not sure whether our approach is also capable of detecting other development processes. We are currently planning other case studies in order to widen the scope further.

We use a simple heuristic that matches the classname of the unit of production code to the classname of the unit test, e.g., we matched `String.java` to `StringTest.java`. Our approach is purely based upon naming conventions and might not be generizable, yet our 2 case studies adhered to it. This convention is also promoted in literature and tutorials [8, 6]. In order to analyze

case studies that do not follow such a naming convention, a call graph based approach that associates test cases with production classes can be used.

Construct validity For the evaluation we use the versioning system's log messages to confirm or reject our observations (also see [16, 19]). As no strict conventions are in place for what should be specified in such messages, there are large differences in the content and quality of log messages across projects, tasks and developers. The external evaluation, i.e., checking our conclusions with the original developers, complements the internal evaluation as an additional source of validation.

We also identify two variation factors of the development process with regard to the use of the version control system. Firstly, the individual *commit style* — short cycles, one commit per day, ... — of developers can influence the results. A countermeasure in this area is using inter-transactional association rule mining [15], which we see as future research. Secondly, developers can use branching and as we are only studying the main branch, this might interfere with our results. In the case of Checkstyle and the SIG case, however, branching is not a common practice. If a large part of a project's development effort happens in branches, it can be useful to specifically apply the approach to these branches.

Finally, a remark on the limitations of studying the testing process by analyzing the contents of a version control system. The focus of our approach is on testing activities that are performed by the developers themselves, i.e., unit testing and integration testing, as these tests are typically codified and stored alongside the production code. We acknowledge that the testing process is much more than only unit and integration testing, e.g., acceptance testing, yet, as these acceptance tests are typically not stored in the version control system, we have no means of involving these tests in our approach.

5 Related work

The idea to analyze the change history of software systems was first coined by Ball et al. in 1997 [2]. In this section we will give an overview of some of the advances in this area that are particularly close to our own research.

Fluri et al. investigate whether code comments are updated when production code changes [7]. They use code metrics and charts to study these changes. A major difference between our own approach and Fluri's approach is that they analyze the changes at the code level, while we remain at the file level.

Both Hindle et al. [10] and German [9] look into multiple dimensions of co-evolution of software artifacts. Hindle et al. study whether release patterns can be detected in software projects. That is, behavioral patterns in the revision frequency of four different artifacts: source code, test code, build files and documentation. They observe repeating patterns around releases for distinct systems, but the data shows large differences between the systems. German meanwhile combines information from many different sources, like mailing lists, version control logs, web sites, software releases, documentation and source code, the so-called *software trails* [9]. He correlates these trails to each other in order to recover information such as: the growth of the software system, the interaction between the contributors, the frequency and size of contributions, and important milestones in the development.

We found two uses of association rule mining in literature. Zimmermann et al. [20] attempt to guide the work of developers based on dependencies found in the change history. For each change a developer makes, his support tool guides the programmer along related changes in order to suggest and predict likely changes, prevent errors due to incomplete changes and identify couplings that are undetectable by program analysis. Their approach derives association rules in real time while the programmer is writing code. As such, their approach does not build a descriptive model of the data, but rather a predictive model. Xing and Stroulia use association rule mining to detect class co-evolution [18]. They apply the mining at the class level, and are able to detect several class co-evolution instances. They also intend to give advice to developers on what action to take for modification requests, based on experiences learned from past evolution activities.

6 Conclusion and future work

In this paper we have used association rule mining to study the co-evolution between production code and test code. In this context, we make the following contributions:

- An approach using association rule mining to study the co-evolution of production and test code in a system. Co-evolution rules are computed from commit transactions obtained from version control data of production and test classes.
- A set of co-evolution metrics including standard interest and strength association rule mining metrics to asses the extent to which product and test classes evolve.
- An evaluation with two case studies, one performed with the open source software project Checkstyle, and another one performed with an industrial software system provided by the Software Improvement Group. In both case studies, the findings have been evaluated and validated with the findings of our previous research and the original developers/maintainers of the software systems under study.

The two case studies that we performed have shown a greatly differing testing approach. In the case of Checkstyle, we saw a very mixed picture at first, since we observed that most of the commits are dominated by changes to production code. This is (1) due to the development style, where testing is mainly done in phases outside of regular development (this is true during the early development of Checkstyle), but also (2) due to a small number of large commits of production code that perturbs the rule classification (these large commits are due to code beautification). Our industrial case study, on the other hand, has shown a test-driven development approach to testing, evidenced by a large number of commits that contained both additions/changes to production and test code.

The analysis techniques that we have explored in this work prove to be useful for (retrospective) assessment of the unit test suite. A weak point of our approach, however, is the fact that changes to the testing practices over small periods of time will not yield noticeable differences in the results, as our technique summarizes the entire history.

Future work. We have identified a number of ideas to build upon this research.

- The use of an *inter-transactional association rule mining* algorithm, which allows to widen our analysis from a single commit to a window of commits that were made in a short amount of time [15].
- The automatic identification and removal of very large commits that are often the result of an automated code-beautification operation. Removing these commits will sharpen the results from our analysis.
- Traversing the change history with a sliding window, so that time-intervals can be studied more in depth, details become more clear and trends can be identified.

Acknowledgments. We would like to thank the Software Improvement Group for their support during this research and Bas Cornelissen and Bart Van Rompaey for proofreading this paper. Funding for this research came from the NWO Jacquard Reconstructor project and from the Centre for Dependable ICT Systems (CeDICT).

References

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 207–216. ACM, 1993.

[2] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk. In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, 1997.

[3] K. Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2002.

[4] R. V. Binder. *Testing Object-Oriented Systems; Models, Patterns, and Tools*. Addison-Wesley, 2000.

[5] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proc. of the International Conference on Management of Data (SIGMOD)*, pages 255–264. ACM, 1997.

[6] M. Fewster and D. Graham. *Software Test Automation: effective use of test execution tools*. Addison-Wesley, 1999.

[7] B. Fluri, M. Würsch, and H. C. Gall. Do code and comments co-evolve? On the relation between source code and comment changes. In *Proc. of the Working Conference on Reverse Engineering (WCRE)*, pages 70–79. IEEE CS, 2007.

[8] E. Gamma and K. Beck. Test infected: programmers love writing tests. *The Java Report*, 3(7):37–50, 1998.

[9] D. M. German. Using software trails to reconstruct the evolution of software: Research articles. *J. Softw. Maint. Evol.*, 16(6):367–384, 2004.

[10] A. Hindle, M. W. Godfrey, and R. C. Holt. Release pattern discovery via partitioning: Methodology and case study. In *Proceedings of the International Workshop on Mining Software Repositories (MSR)*, pages 19–26. IEEE CS, 2007.

[11] M. M. Lehman. Program evolution: Processes of software change. *Information Processing and Management*, 20(1):19–36, 1985.

[12] Z. Lubsen. Studying co-evolution of production and test code using association rule mining. Master's thesis, Software Engineering Research Group, Delft University of Technology, 2008.

[13] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink. *Software Evolution*, chapter The interplay between software testing and software evolution, pages 173–202. Springer, 2008.

[14] P. Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006.

[15] A. K. H. Tung, H. Lu, J. Han, and L. Feng. Breaking the barrier of transactions: Mining inter-transaction association rules. In *Proc. of the Int'l Conference on Knowledge Discovery and Data Mining (KDD)*, pages 297–300. ACM, 1999.

[16] B. Van Rompaey, A. Zaidman, A. van Deursen, and S. Demeyer. Comparing the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Technical report, TUD-SERG-2008-034, Delft University of Technology, Software Engineering Research Group, 2008.

[17] T. Xie, J. Pei, and A. E. Hassan. Mining software engineering data. In *International Conference on Software Engineering (ICSE) Companion*, pages 172–173, 2007.

[18] Z. Xing and E. Stroulia. Understanding the evolution and co-evolution of classes in object-oriented systems. *International Journal of Software Engineering and Knowledge Engineering*, 16(1):23–52, 2006.

[19] A. Zaidman, B. van Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production and test code. In *Proceedings of the 1st International Conference on Software Testing, Verification and Validation (ICST)*, pages 220–229. IEEE CS, 2008.

[20] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 563–572. IEEE CS, 2004.

TUD-SERG-2009-014
ISSN 1872-5392