Using Association Rules to Study the Co-evolution of Production & Test Code*

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 based on association rule mining to determine whether production and test code co-evolve synchronously. Through two case studies, one with an open source and another one with an industrial software system, we show that our association rule mining approach 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. Runeson notes that automated unit testing[1] can be an effective countermeasure for difficulties encountered during software maintenance [5]. Also Test-Driven Development (TDD) and test-driven refactoring [4] 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. Traditional test suite quality measures are typically not good at indicating the long term quality or *"test health"* of a test suite [7]. 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 [7] 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.

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

2 Production and test class co-evolution

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 [8], 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.

*This work is described in more detail in the MSc thesis of Zeeger Lubsen [3].

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

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

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 is obtained from versioning repositories, such as, the concurrent versions systems (CVS) or Subversion (SVN).

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 [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.

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:
$\{TC_1 \rightarrow PC_1\}$, $\{PC_1 \rightarrow TC_1\}$, $\{PC_2 \rightarrow TC_1\}$,
$\{TC_1 \rightarrow PC_2\}$, $\{PC_1 \rightarrow PC_2\}$, $\{PC_2 \rightarrow PC_1\}$,

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.

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

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.

containing rules that denote commit transactions in which a test class has been matched to a production class. For 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.

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.

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) [2].

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.

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 Preliminary results

We tried out our approach using two software systems: Checkstyle, an open source coding standard checker, and an industrial software system provided to us by the Software Improvement Group (SIG).

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.

Checkstyle. For Checkstyle we saw that actual software development and testing are mainly two separate activities, which is mainly evidenced through the rule ratios. 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).

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.

SIG software system. 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.

4 Conclusion and future work

In this paper we have made the following contributions:

- An approach using association rule mining to study the co-evolution of production and test code in a system using transactions obtained from version control.
- A set of co-evolution metrics including standard interest and strength association rule mining metrics to assess 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 [6].
- 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] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] A. Zaidman, B. van Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production and test code. In *Proc. Int'l Conf. on Software Testing, Verification and Validation (ICST)*, pages 220–229. IEEE, 2008.

[8] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Int'l Conf. on Software Engineering (ICSE)*, pages 563–572. IEEE, 2004.