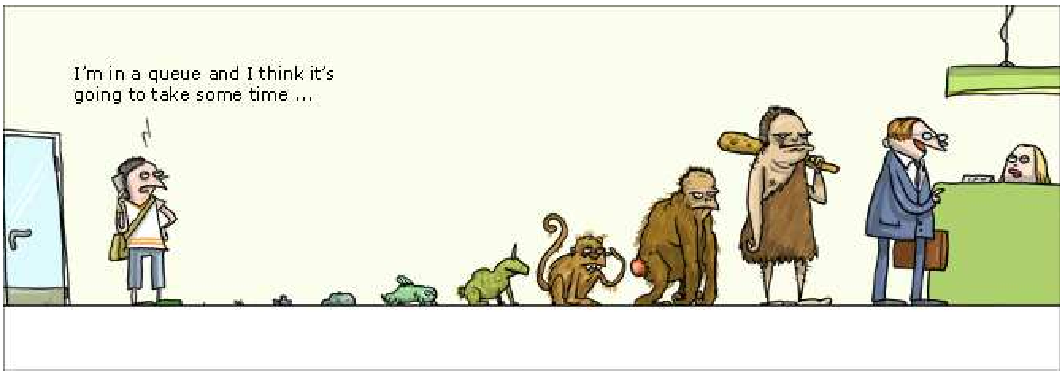
Studying Co-evolution of Production and Test Code Using Association Rule Mining

Master's Thesis



Zeeger A. Lubsen

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

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Zeeger A. Lubsen
born in Amsterdam, the Netherlands

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

Software Improvement Group
A.J. Ernststraat 595-H
1082 LD Amsterdam
the Netherlands
`www.sig.nl`

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

Author: Zeeger A. Lubsen
Student id: 1054503
Email: `z.a.lubsen@student.tudelft.nl`

Abstract

Unit testing is generally accepted as an aid to produce high quality code, and can provide quick feedback to developers on the quality of the software. To have a high quality and well maintained test suite requires the production and test code to synchronously co-evolve, as added or changed production code should be tested as soon as possible. Traditionally the quality of a test suite is measured using code coverage, but this measurement does not provide insight in how tests are used by developers. In this thesis we explore a new approach to analyse how tests in a system are used based on association rules mined from the system's change history. The approach is based on the reasoning that an association rule between two entities, possibly of a different type, is a measure for the co-use of the entities. Case studies show that analysing all the resulting rules allows us to uncover the distribution of programmer effort over pure coding, pure testing, or a more test-driven practice. Another application of our approach is that we can express the number of tests that are truly co-evolving with their associated production class.

Preface

“History will be kind to me for I intend to write it.”
Sir Winston Churchill

The topic of this graduation thesis is how code and its tests change over time. While programmers write their software they are often not aware what great consequences their actions can have. Pick a random book on software engineering and you will encounter at least one example of software terribly gone wrong because of trivial errors. Thoroughly testing your software can prevent these painful situations, and will repay the invested effort in time. The quote by Winston Churchill is applicable to software developers: you can write your own software history, or become an example in a textbook. Just take your time to do it right.

Eventually it is all about about how, and with who, your spend your time. It is not as much about how much time you spend in university, or about the time you have in front of you after graduation, but whether you did it in a way you feel good about.

Personally, I had a great time doing this thesis project. You learn from both the ups and the downs, and I am happy I was able to do the project in a great environment with many interesting people. All the people I have shared a room with since last August, Reinier, Rinse, Gerard, Leo (Who needs Apple?), Peter, Mitchell, Frank, Tim and Johnny, thanks for the pleasant times and the much needed distraction from work. All the other people at SIG, and especially Joost Visser and Ilja Heitlager for the great feedback and insights on my work.

I’d like to thank both my supervisors, Andy Zaidman from TU Delft, and Michel Kroon from SIG, for the time you both took to keep me going in the right direction. Andy, thank you for reviewing my writings so remarkably fast, everytime again, and giving so much space to find my own way in this project.

All my friends and family, who showed interest in how I was doing, you were a great support. And last but not least, and certainly the most important, Arina and my parents, thank you for all the patience, understanding and support you gave all these years. I’m lucky to spend my time with you.

Zeeger A. Lubsen
Amsterdam, the Netherlands
June 24, 2008

Contents

Preface iii

Contents v

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Problem Statement . . . 2
- 1.2 Proposed Solution and Approach . . . 3
- 1.3 Software Improvement Group . . . 4
- 1.4 Thesis Structure . . . 4

2 Background and Related Work 5
- 2.1 Software Evolution . . . 5
- 2.2 Software Repository Mining and Data Mining . . . 6

3 SIGAR: Association Rule Mining Implementation 9
- 3.1 Toolchain Introduction . . . 9
- 3.2 Toolchain Structure and Implementation . . . 10

4 Association Rules Analysis 17
- 4.1 Association Rules Interpretation . . . 17
- 4.2 Evaluation: JPacman Test-Case . . . 27

5 Case Studies 37
- 5.1 Systems Desciptions . . . 38
- 5.2 Test Process Analysis . . . 42
- 5.3 Test Suite Quality Evaluation . . . 52
- 5.4 Evaluation . . . 55

6 Conclusions and Future Work **57**
6.1 Conclusions 57
6.2 Contributions 58
6.3 Future work 58

Bibliography **61**

List of Figures

3.1 An excerpt of an extracted change history. 11
3.2 SIGAR toolchain structure. 12

4.1 Measurement of logical coupling based on association rules. 26
4.2 ChangeHistoryView of JPacman. 29
4.3 Boxplots of support and confidence for JPacman. 31
4.4 Histograms of support for JPacman. 32
4.5 Histograms of confidence for JPacman. 33
4.6 Class and tests occurrences for JPacman. 35

5.1 Checkstyle ChangeHistoryView 39
5.2 System A.I ChangeHistoryView 41
5.3 System B.I ChangeHistoryView 42
5.4 System B.II ChangeHistoryView 43
5.5 System C.1 ChangeHistoryView 44
5.6 Checkstyle rule strengths distributions 46
5.7 A.I rule strengths distributions 48
5.8 A.II rule strengths distributions 49
5.9 B.I rule strengths distributions 51
5.10 B.II rule strengths distributions 52
5.11 C.I rule strengths distributions 53
5.12 C.II rule strengths distributions 54
5.13 Example of the number of occurrences per class type. 55

List of Tables

3.1 Classification of association rules. 15

4.1 Individual association rule metrics. 21
4.2 Classification of production and test classes. 27
4.3 Rule ratios for JPacman. 30
4.4 Summary of rule distributions of metrics for JPacman. 34

5.1 Characteristics of the case studies. 39
5.2 Rule ratios for the case studies. 44
5.3 Ratios of rules, entities and revisions for the cases. 45
5.4 Rule coverage ratios of classes for the case studies. 56

Chapter 1

Introduction

The development of high quality software systems is a complex process, and maintaining an existing system over time no less. After the initial release of the system, the eroding effects of software evolution cause systems to become harder to maintain and even obsolete, as formulated by Lehman's Laws of Software Evolution [23]. Successful and increasingly more adopted methods to counter the effects of software evolution are automated unit testing (see xUnit Testing Frameworks[1]) and the practice of Test-Driven development [5]. Unit testing is becoming an essential aspect for the development of reliable and high quality systems, and can ease the ongoing maintenance of the system after its initial release [30].

But unit tests are executed in a simulated environment, and the quality of the tests greatly depends on the effort that the developer who wrote the test put into it. The behaviour of code units must be checked for different input values, and possibly many exceptional cases [8]. Tests are only as good as how the tester writes them. This leaves the desire to be able to assess the quality of the test suite of a system.

In popular fashion, the quality of a test suite is typically expressed by code coverage: the percentage of the code that is exercised by the set of tests that is executed [8]. But code coverage (coverage for short) is a somewhat shallow measure of test quality. Code coverage expresses that some code is executed, not how or what is tested. One should think of different input values and the number of assertions checked by the test. Simply running the code only guarantees that the code compiles and does not crash in trivial situations.

So we are left with open questions regarding the quality of the test suite. A tester wants to know if his testing effort is any good, or a project manager wants to know if his testing team is meeting the required standards. These questions involve the *testing effort* and the *long term quality* of the unit tests. Being able to answer these questions aids in at least two ways for different stakeholders [33]:

- Assessment of the testing process, for example to estimate future maintenance, and in first-contact situations with an existing system.

- Monitoring of the testing process, to compare the current process to the intended process, and for the identification of trends.

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

Combining these observations of the importance of high quality test suites and test-driven development, we argue that the production and test code in a system should co-evolve synchronously. New functionality added to a system should be unit tested as soon as possible, and the preservation of behaviour should be checked after changes have been made.

1.1 Problem Statement

The situation that is presented in this introduction is a continuation of the work done by Zaidman et al. [33]. In their work they present three lightweight visualisation techniques to study the co-evolution of production and test code, focussed around the main research question: *'How does testing happen in open-source software systems?'* The proposed approach has the downside that there are no explicit measurements available to support the observations from the visualisations. Interpretation of the presented high-level views of the system's history is left to the viewer.

Data mining is a collective name for techniques that attempt to find hidden information in large amounts of data [11]. These techniques allow to draw more sophisticated information from databases than normal query languages and visualisations of the data offer. The main source of historical data of software systems are Version Control Systems (VCS) [4]. Each change in a system is committed to a VCS in a *transaction*, or *commit*. A VCS contains the history of transactions of a software system (from now on: the change history). To use data mining techniques on the change history is an intuitive and proven method to study the evolution of a software system [6, 32, 34].

The employment of data mining techniques appear to be an interesting candidate to extend the previous work on the co-evolution of production and test code. To build on the approach by Zaidman et al. the central research question of this thesis is:

Central Research Question: How can data mining techniques be applied to (retrospectively) study the unit testing process in software systems?

More specific, the purpose of this research is to solve the following supplemental, and subsequent, research questions (RQ):

RQ1: Can data mining techniques be used to find evidence of intentional synchronous co-evolution of production and test code?

RQ2: Can the nature of the test code (unit test vs. integration test) be distilled from the change history?

RQ3: Can a quality measurement for co-evolution of production and test code be defined based on a presented technique?

RQ4: Can different patterns of co-evolution be observed in distinct settings, for example different cultures like open-source software versus industrial systems?

1.2 Proposed Solution and Approach

To solve the raised research questions we propose to mine association rules from the change history contained in the VCS of a software system. Association rule mining attempts to find common usage of items in data, and is often applied to assist in activities like marketing, advertising, floor placement in stores, and inventory control [11]. An association rule is a statistical implication between a set of items that occur together in a database for a specified minimum number of times. Association rules are typically derived from transactional data, that is, records containing a collection of items and a timestamp.

Association rules are frequently applied in everyday situations. A common example is the suggestions of other products that are presented to customers who are viewing a product in an online store. Websites like Amazon.com often have a section of the webpage suggesting a number of products ("people who bought this product also bought..."), depending on the article that is currently viewed by the customer. These suggestions are derived from previous purchases of the current article in combination with other articles. In this example association rules describe the individual suggestions.

An association rule expresses a statistical implication between items based on their common usage, as recorded in the change history. Depending on how many times the combination of items is found in the change history, the association rule has a certain *strength*. By mining association rules from a software system's VCS, we expect that the resulting association rules can express the characteristics of the co-evolution of the production and test code in the system. For example, if a production class and its associated unit test are always changed and committed together in the change history, we expect that the mining process will result in an association rule that connects these two entities based on their historical co-change. The strength of the association between the two entities can be used as a measure for the co-evolution of the two entities. This principle can be generalised for multiple rules.

In this research project we perform an explorative study on how association rules can be used to study co-evolution of production and test code. The proposed solution consists of two parts:

- Design and implementation of an association rule mining tool to obtain co-evolution information of a system, and exploration of the interpretation of the association rules.
- A number of case studies to validate and explore the applicability of the approach, as well as to observe differences in testing processes in detail.

The separate parts are now explained in more detail.

1.2.1 Tool design and implementation

First we will develop a tool that can mine association rules from a system's VCS. Next we explore measurements that express the co-evolution of production code and test code in a system, based on the derived association rules. We implement these measurements in the tool to study the co-evolution of production and test code, and the underlying testing process of a number of software systems.

1.2.2 Case Studies

We perform several case studies to evaluate the association rule mining approach, and to observe different development and testing approaches in OSS vs. commercial software settings. The main observations of the cases will be derived from four sources: the visualisation technique to study the change history (ChangeHistoryView) by Zaidman et al., the results from the association rule mining tool, log messages from the commits in the VCS and information from the developers of the systems evaluated in the cases.

Each case will be assessed by the results from the association rule metrics, and we use the visualisations to validate whether the computational results match the visual observations. The log messages are used to internally validate the observations, and the information from the system developers serves as external reference of the results.

1.3 Software Improvement Group

The Software Improvement Group[2] is specialised in the area of quality improvement, complexity reduction and software renovation in software engineering. The SIG performs static source code analysis to analyse huge software portfolios and to derive hard facts from software to assess the quality and complexity of a system. The analysis of systems is used in activities like risk assessment, software monitoring, documentation generation and renovation management. These kind of activities are useful for large legacy systems as well as newly developed systems. By gaining more insight into the process that drives the testing and development processes of a system, the SIG can give more informed and accurate assessments to its customers regarding the quality and 'health' of the testing strategy and the current testset. With test health, we mean the long term quality of, and the amount of effort being put into maintaining the test suite.

1.4 Thesis Structure

This thesis is structured as follows. First we discuss related work in chapter 2 to describe the context of this thesis. In chapters 3 and 4 we introduce and explain the proposed technique to study co-evolution of production and test code. The case studies that are performed to explore the proposed technique, and the results follow in chapter 5. Finally conclusions are drawn and future work is proposed in chapter 6.

[2]http://www.sig.nl

Chapter 2

Background and Related Work

2.1 Software Evolution

Software Evolution is the process of continual fixing, adaptation, and enhancements to maintain stakeholder satisfaction [23]. Systems must adapt to the changing environment they operate in by adding features or correcting bugs [24], which causes the structural composition of the system to decay [12, 28], unless preventive or corrective measures are taken. This observation is formulated in Lehman's Laws of Software Evolution [23]. While these empirical finding have been disputed [19], all arguments for and against illustrate the diversity and complexity of evolving software.

A recent research trend in software evolution research is to visualise the evolution history [31], and research is based on empirical findings. Active topics in software evolution research are how to identify entities or parts of a system which are bottlenecks in the maintenance of systems, how to refactor these problematic entities or how to avoid entities from becoming complex and problematic. Examples of approaches to identify bottlenecks are Gîrba's Yesterday's Weather [18] and the Evolution Radar by Marco D'Ambros [10].

The visual approach to study software evolution originates from the ability of visualisations to communicate trends [29]. A XY-chart can quickly show growth or other changes over time. An increasingly popular type of visualisations in this area are polymetric views, like Lanza's Evolution Matrix [21, 22]. The visualisations typically plot entities against a time measurement, and use size and colour to incorporate different metrics into the view.

2.1.1 Software Co-evolution

Software co-evolution is the multi-dimensional cousin of software evolution research. Software itself and its underlying development process are multidimensional. The development of high quality software requires other artefacts besides source code, like specifications, constraints, documentation, tests, etc. [26]. This is what makes it multidimensional. All these artefacts are inherently intertwined with the source code of a system, and thus influences its evolution. Software co-evolution studies how these relations between artefacts exhibit themselves and change over time.

One of the more important concepts in software evolution is *logical coupling*. Two entities are logically coupled when they change at the same time in the history [15, 14]. The more times entities are changed together the stronger the logical coupling is. Logical coupling is a measure for the number of co-changes of entities in a given period.

The work by Zaidman et al. [33] is preceding the work in this thesis, as already mentioned in chapter 1. The visualisations proposed by Zaidman allow the analyst to study how production and test code grow and change over time, and whether changes in production code is followed by changes in test code, or vice versa. They seek for evidence of intentional synchronous co-evolution. This can provide insight in the test process of a development cycle, and can visualise the testing efforts for a system. It is based on the understanding that ideally additions to a system should be tested as soon as possible, and the preservation of behaviour should be checked when changes are applied. This is contrasted to a more phased approach where (longer) periods of dedicated code writing are alternated by (shorter) periods of increased testing effort.

A similar idea is explored by Fluri et al. [13]. Instead of production and test code, Fluri studies whether code comments are updated when production code changes. They use code metrics and charts to study these changes. A main difference in Fluri's approach is that they analyse the changes on the code level, while Zaidman remains on the file level.

Both Zaidman and Fluri explore two dimensions of software evolution. Even more dimensions are combined by Hindle et al. [20] and German [17]. Hindle studies whether release patterns can be detected in software projects. That is, behavioural patterns in the revision frequency of four different artefact classes: source code, test code, build files and documentation. They do observe repeating patterns around releases for distinct systems, but the data shows large differences between the systems.

Daniel German combines information from many different sources, like mailing lists, version control logs, web sites, software releases, documentation and source code. He calls this information that is left behind by developers software trails. He extracts useful facts from the trails and correlates them to each other in order to recover the evolution of the software system. The approach reveals interesting facts about the history of a system: its growth, the interaction between its contributors, the frequency and size of contributions, and important milestones in the development.

2.2 Software Repository Mining and Data Mining

Software evolution research relies on historical data to reason over past development activities. These development activities are often contained in software repositories [27]. The source code of a software system is most often contained in Version Control Systems (VCS). A VCS provides functionality to allow developers to work collaborative on a system, to keep backups of code and to revert changes.

Each time a developer makes a change to source code, he submits the changes to the VCS, and with this action, he creates a new revision of the software. The set of changed files that are added to a VCS are called a commit (or transaction, we will use both terms interchangeably in the rest of this work). A VCS not only records the changes to the files in the commit (structural information), but also who made the change, when the change was made, and what files were added, changed or deleted in the commit (meta-data). The complete set of transactions that transforms each revision in the VCS to the next is called the *change history* of a system. The meta-data that is recorded in the change history is used extensively in modern software evolution research.

The idea to analyse the change history was first coined by Ball et al. [4]. They explored the idea by studying who changed what files, and that one can measure the connection strength of two entities based on the probability that two classes are modified together in the change history. This last idea was further explored by Gall et al. [14], and is now known as the before mentioned logical coupling.

Data mining is the use of algorithms to extract or find useful information, hidden dependencies and patterns in data [11]. Data mining techniques prove to be very useful in software evolution research when applied to the change history. The particular technique of association rule mining that is used in this work is discussed in more detail in the next sub section, and followed by a discussion of more general related work that employs data mining to study software evolution.

2.2.1 Association Rules

Association rules mining [1, 2] is a data mining technique that produces rules that show the relationships between items from transactional data in a database, as introduced in chapter 1 of this thesis (think of market baskets). It is very important to note that association rules detect common usage of items. The uncovered relationships are not inherent in the data, as with functional dependencies, and they do not represent any sort of causality or logical relation [11]. Zimmermann states that a rule has a probabilistic interpretation based on the amount of evidence in the transactions they are derived from [34].

2.2.2 Data Mining to Study Software Evolution

We found two uses of association rule mining in literature. The first is the work by Zimmermann et al. [34]. He attempts 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. The tool works with a ‘Programmers who changed these functions also changed...’ metaphor, similar to suggestions encountered in online webstores (again, see the example in chapter 1). Zimmermann’s approach derives association rules at function and variable levels, in real time while the programmer is writing code. This is different from the typical application of retrospectively mining association rules to build a descriptive model of the data. The intent of this approach is to build a predictive model.

The second work that utilises association rules is by Xing and Stroulia. They use an association rule mining algorithm to detect class co-evolution [32]. They apply the rule mining at 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 learned experiences from past evolution activities. Their approach focusses on the design-level, in contrast to Zimmermann's more low-level approach. Both of these approaches differ from our approach in that both parse the source code, and we remain on the file level. At the file level less information is available, but as we will argument further on in this thesis, that is not a large problem, and a good trade-off between information and performance is made.

Another frequently employed data mining technique in software evolution research is clustering. Clustering attempts to groups items by using a distance measurement [11]. In their initial paper on mining VCSs, Thomas Ball et al. illustrate the potential of repository mining by clustering the change history. Their clustering algorithm places classes closer together in a pane when they are changed together more often [4]. A clustering algorithm is used to determine the layout of the visualisation. Very similar are the evolution storyboards by Dirk Beyer [7], and work by Stephen Eick et al. [12]. All these approaches use the logical coupling of items as the distance metric for the clustering.

Chapter 3

SIGAR: Association Rule Mining Implementation

As described in the introduction in chapter 1, the main idea is to mine association rules from the change history of a VCS to obtain a model of the co-change of different code entities. Now the tool to mine association rules is proposed, which we dub SIG Association Rules (SIGAR). The technical implementation of the tool used to mine the association rules and the design considerations are described.

3.1 Toolchain Introduction

First the general characteristics of the approach are introduced. The main motivations to use association rules to study co-evolution of production and test code are:

- Association rules are based on the common usage of items, they describe the logical coupling (see chapter 2) between the items in the rule. This implies that they can express the actual way that developers use production classes and unit tests. It seems intuitive to mine association rules from change histories.

- Analysing systems on the file level (described next) is a 'lightweight' approach, in that no static or dynamic source code analyses is needed. These other approaches provide a deeper level of granularity, but are more costly in terms of required time and computations. For example, statical analysis of all 3000 revisions of a system of 2000 classes is an enormous task. Comparing classes which each other class for every revision yields a quadratical number of comparisons, times the number of transactions.

For the scope of this research project we have set some limitations to the approach. Currently, the analysis is limited to systems written in Java, and the extraction of the change history is restricted to Subversion (SVN)[1] repositories. The reason that only Java systems can

[1] http://subversion.tigris.org/

be analysed is that Java uses conventions that imply that classes (except inner and anonymous classes) are all written in a separate file. VCS's only version flat text files, and in the case of Java, classes and files are practically mapped one-to-one. The file level relations that are extracted using this approach are also valid on the class level.

The approach is restricted to SVN repositories because SVN actually tracks changes to the repository by storing transactions. This means that the log data from SVN can be easily transformed to the data format used in the tool. Concurrent Versions System (CVS)[2] repositories can be converted to a SVN repository using a script[3] so that systems that are versioned in CVS can also be analysed by this approach. CVS and its successor SVN are two of the most popular and widely used VCS's.

3.2 Toolchain Structure and Implementation

In this section the different components, the data-flow and in- and output of the tool are described. The tool has a number of design goals:

- Generation of association rules from the change history of software systems.
- Ability to analyse large systems with long histories.
- Ability to configure the tool to accept a range of different systems.

The tool itself is also written in Java. It consists of several separate modules that can be chained together by a configuration file. The configuration of the modules and the chain is based on the Spring Framework[4] and the Chain of Responsibility and Command design patterns [16]. In the configuration file the order of the modules can be specified, and settings for each module can be configured, for example from which file to read the input data.

The shared data that is needed for the entire computation travels along the chain in a *context* object. The initial configuration settings and paths of the input and output files are stored in this object. The modules that compose the tool are described in the following subsections. We discuss their workings, the design considerations and the encountered implementation problems. An illustration of the data-flow and structure of the toolchain can be found in figure 3.2

3.2.1 Change History Extraction

The input data for the entire association rule mining process is gathered by this module. It operates on a SVN repository and extracts the log data for a project in the repository. It stores for each commit made to the repository the revision number, the author, the timestamp and all the files that were added, modified or deleted in the commit. The change history is stored in a XML format, of which an example can be found in figure 3.1. This format is what the actual toolchain operates on.

[2]http://http://www.nongnu.org/cvs/
[3]http://cvs2svn.tigris.org/
[4]Spring Framework: http://www.springframework.org/

```
<ProjectHistory>
  ...
  <commit revision="88">
    <revision>
      88
    </revision>
    <author>
      arie
    </author>
    <date>
      Mon Jul 18 21:04:19 CEST 2005
    </date>
    <message>
      <![CDATA[moving the observer interface to the model.]]>
    </message>
    <M>
      /trunk/src/java/pacman/model/Engine.java
    </M>
    <D>
      /trunk/src/java/pacman/controller/PacmanObserver.java
    </D>
    <A>
      /trunk/src/java/pacman/model/Observer.java
    </A>
  </commit>
  ...
</ProjectHistory>
```

Figure 3.1: An excerpt of an extracted change history.

The figure shows an excerpt of an extracted change history, in this case from the JPacman project. Each commit element contains the revision number, author, timestamp and the files added (A), modified (M) or deleted (D) in the commit.

This module is a proverbial exception to the rule. The module is run separate from the rest of the toolchain, because it needs to be executed on locations other than where the analyses are performed. Examples are the VCS's from industrial partners for the case studies, whose repositories are not accessible from outside their offices. As stated in the introduction of this chapter, only logs from SVN and CVS (through conversion) repositories can be extracted.

3.2.2 Change History View Generation

This module builds a data file for the ChangeHistoryView visualisation from the change history. The data underlying the visualisation is stored in an XML file containing all the points in the plot and can be used to generate the ChangeHistoryView on-screen. The generation of this data file is incorporated in the toolchain to speed up the computation. For this work the computations are typically performed on a fast server that has no monitor. The

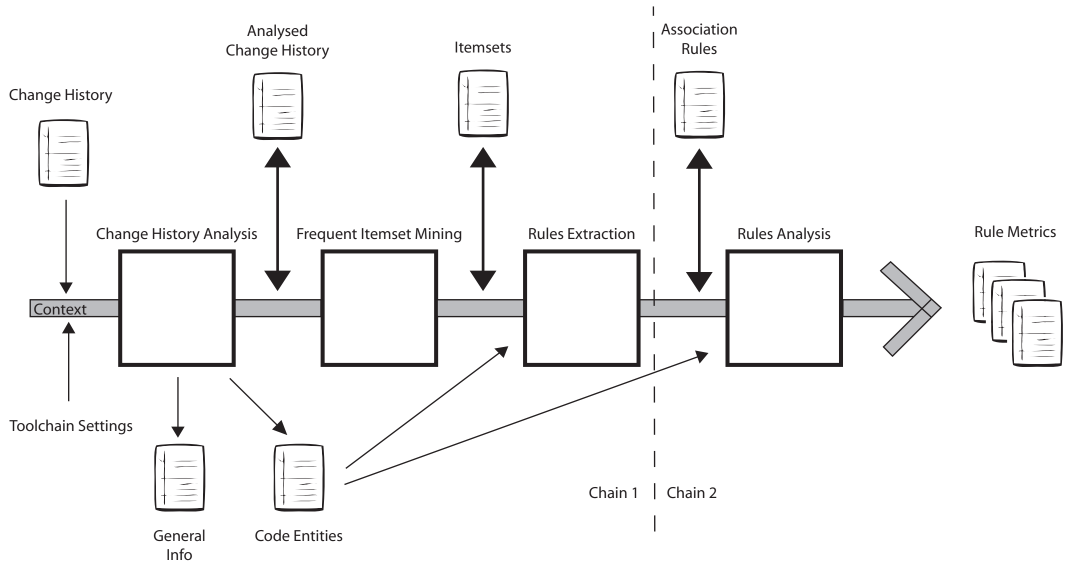


Figure 3.2: SIGAR toolchain structure.

The structure of the SIGAR toolchain. All input and intermediate data are stored in a context object that travels along the chain. Chainable modules with storage and retrieval of intermediate results allow for flexibility of the analysis tool. Two typically used chains are illustrated in the figure. Chain 2 consists of only analyses of the rules, but utilises the results (the list of code entities and the mined rules) from chain 1. The calculation of new metrics can be performed without running the entire chain again, and thus skipping performance intensive modules like change history analyses and frequent itemset mining.

actual rendering of the view is a quick operation, and can be performed independent from the generation of the data at any time desired, and on slow workstations without annoying waiting times.

3.2.3 Change History Analyses

The first step in the mining of association rules consists of a pre-processing task, and the extraction of general information from the change history. Simple queries are run against the change history file, and extract global information like the total number of distinct files (distinguished between production and test code) and the number of revisions in the history.

The first part of the pre-processing consists of filtering the input data of everything but actual code and test files. The input data contains log information of all files in the repository, like maven project files or configuration files. These are not of interest to the mining of association rules, so they are left out of the process. Files are filtered on their extension (e.g. only `.java` files are kept in the history). Each code file encountered is stored as a *code entity*. A code entity is a tuple consisting of an integer identifier, a filename and a type.

The frequent itemset mining algorithm (described in the next module) expects its input to be a sequence of transactions that contain only integer values. The original input consists of transactions containing strings (see figure 3.1), so this implies that each file needs to be assigned a unique numeric identifier. The analyser traverses the change history to swap each code file in the history with its identifier, assigning a new identifier to files that it has not yet encountered and storing the identifier/file pair as a code entity. The code entities are stored in a bi-directional hashmap. This allows lookup of the code entities by both filename and identifier, as both ways are required in the entire chain. Since a change history often contains several thousands of unique files a hashmap provides good performance of insertion and retrieval of code entities. Each transaction in the change history contains several files that all need to be looked up in the list of encountered code entities, so the data structure must be efficient to keep performance acceptable. Each code entity that is encountered in the change history is also tagged as being a production code, test code or undefined. This tagging is done by matching the filename (and path) to a regular expression that is configured in the context object. Production and test code files often have their own place in the directory structure of the system. This is often a path similar to `/Project/src/java/...` and `/Project/src/test/...`, but in the case studies we encountered several exotic variations. The regular expressions are used to describe the pattern that distinguishes different files. Test code mostly follows a naming convention that includes the word `Test` in the filenames. This convention is utilised in the regular expression for tagging of test code. Files that are not recognised as production or test code are tagged as undefined.

The analyser produces three results: (1) general information on the change history, (2) a filtered change history containing only integer identifiers in the transactions, and (3) a collection of tagged code entities. While finding frequent itemsets is the most essential module of the entire association rule mining process, the change history analyses and building of the code entities is the most costly part. Note that this is in relation to the performance of the frequent itemset mining module with the computational considerations described in that module (see next module). Not pre-processing this data slows down the other modules significantly. It generally cuts the running time of the entire chain in half. An illustration: for a system containing n files, and each file is changed on average m times, the total number of lookups in the list of code entities is $n \times m$. As larger systems contain thousands of files during their lifetime, and the longer the change history is (more transactions), the number of lookups can grow very large.

3.2.4 Frequent Itemset Mining

The frequent itemset mining module is responsible for the most essential part of the toolchain. It finds all sets of items in a transactional database that occur at least a given number of times. The number of times that an itemset appears in the database is called the *support*. When the support of an itemset is at least equal to the given minimum support, the itemset is said to be a *frequent itemset*. Support is described in more detail in section 4.1.

The frequent itemsets are mined using an implementation of the Apriori algorithm[5] [2]. Apriori is one of the earliest algorithms for mining itemsets, and is still the major technique used by commercial products to detect frequent itemsets [11].

Apriori attempts to find frequent itemsets by making several passes over the transactions and counting the support of itemsets. In the first pass, all itemsets of size one are counted. In each following pass the size the itemsets is increased by one by joining the found itemsets. Thus in pass n, itemsets of size n are counted. This approach generates many possible itemsets. To limit the number of possibilities Apriori makes use of the *frequent itemset property*: "Any subset of a frequent itemset must be frequent" [11]. This principle states that for any itemset I found that is not frequent, there can be no larger itemset containing I that is frequent. Apriori can thus discard any itemsets that are not frequent, as these will not generate frequent itemsets of a larger size.

The performance of the algorithm is dependent on the cardinality of the largest frequent itemset. The number of database scans is one more than the cardinality of the largest frequent itemset. This potentially large number of database scans is a weakness of the Apriori approach [11]. We believe that in general the nature of change history data is sparse and narrow (i.e., not often recurring items and a low number of items per transaction). However, analysis of the ChangeHistoryView for some systems reveals that there are often many files changed at the same time. A recent study by Alali et al. [3] shows that the number of items in a typical commit is small (under 5 files) for 75% of the commits, but that there are very large extremes (up to thousands). Very large commits most often occur when the code is automatically changed by using code checkers (e.g. Checkstyle or PMD) or features from the IDE (e.g., 'organise imports' in Eclipse). When large commits occur a number of times, the Apriori algorithm will find very large itemsets, and thus make many passes over the database. The potential number of large itemsets is $2^m - 1$ [11], where m is the size of the largest transaction in the database. Simple tests show that even for a small project (JPacman, discussed later in this chapter) the algorithm performs a large number of passes over the change history, and the number of generated itemsets explodes exponentially. To control the running time of Apriori, and the huge number of generated itemsets, we decided to let the algorithm only generate itemsets of size 2. Because we are primarily interested in association rules that link single production classes to single unit tests, we believe that this is a defendable decision. It also greatly simplifies the remainder of the analyses as the resulting association rules are easier to interpret.

The Apriori algorithm was chosen because it is a widely used algorithm with a proven track record. We currently only perform a few passes over the database, and the Apriori algorithm is quite fast in the earlier passes [11]. With a reference implementation in Java available, incorporation in the tool required little work. We do not believe that alternatives to the Apriori algorithm yield significant performance gains for this particular setting, and that an evaluation is not within the scope of this project.

[5]Credits for the implementation go to Bart Goethals and Michael Holler. Their implementations of Apriori in C++ and Java were used as a reference.

Rule	**Classification**
$\{ProductionClass => ProductionClass\}$	Pure production rule
$\{ProductionClass => TestClass\}$	Production to test rule, Production-test pair
$\{TestClass => ProductionClass\}$	Test to production rule, Production-test pair
$\{TestClass => TestClass\}$	Pure test rule
Containing an Undefined class	Undefined rule

Table 3.1: Classification of association rules.

Summary of classifications of association rules, based on the types of the code entities that occur in a rule. Note that rules that associate production classes to test classes and vice versa are assigned two classifications. These rule receive both a general classification (Production-test pair), and a directional classification for when the direction of the association between the pair is important.

3.2.5 Association Rules Extraction

After frequent itemset are found, the generation of rules is trivial [11]. Each itemset of size two or more can be mapped to two or more rules (itemset $\{A,B\}$ produces rules $\{A => B, B => A\}$). For each found rule, rule specific metrics are calculated. These are described in section 4.1.1. With the collection of code entities in hand, the rule extractor tags each rule with a classification based on the type of the code entities that occur in the rule. The different classifications of rules are described in table 3.1. The classifying of the rules is required to give meaning to the measurements over multiple rules.

3.2.6 Association Rules Analyses

Now actual association rules are mined from the change history, measurements can be calculated over them. Measurements can be applicable on all rules or only on rules of a certain type. Each measurement is implemented as a visitor design pattern. The rules analyser itself is a walker that traverses over all the rules, and lets the visitors perform their calculation on each rule. The analyser can be configured with what visitors to traverse the rules in a similar way as the entire toolchain is configured. New measurements can be added at a later time by implementing a new visitor. Benefits of this implementation are flexibility and performance, as all measurements are calculated in one pass over the rules (instead of having each measurement take a traversal on its own). The measurements themselves are discussed in the next section.

3.2.7 Reading and Writing Intermediate Results

In addition to the modules that perform computations on the input data, the toolchain provides functionality to write and read intermediate data to and from files. The intermediate data includes the analysed change history, the labelled code entities, the mined itemsets and the mined typed association rules. All this data can be stored, and be fed back into a different chain. The use and order of the different modules, and the data flow are depicted in figure 3.2. The input/output mechanism has two benefits: inspection of the intermediate data, and the construction of short chains that rely on pre-calculated data.

Chapter 4

Association Rules Analysis

Now we explore and discuss measurements to understand the unit test suite of a software system and the underlying testing process, based on the generated association rules (section 4.1). The discussed interpretations of the association rules are divided into two groups: metrics that are derived from multiple rules (rule based), and metrics that use rules to give data on code entities (entity based). We illustrate the presented measurements with a running example in section 4.2.

4.1 Association Rules Interpretation

An association rule is a statistical description of the co-occurrence of the elements that constitute the rule in the change history. Agrawal [1] presents a formal description:

Definition 4.1 *Given a set of items $I = I_1, I_2, \ldots, I_m$ and a **database of transactions** $D = t_1, t_2, \ldots, t_n$ where $t_i = I_{i1}, I_{i2}, \ldots, I_{ik}$ 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 $A \cap B = \emptyset$.*

For an association rule, 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 association rules are not causal, but spurious, e.g., the co-occurrence of X and Y is caused by one (or a chain of) unknown external event(s). An association rule only describes that there is a relation between the two items, but there is no proven cause-effect relation. Applying the definition to a version control log, the database of transactions *D* is the change history (containing *n* transactions), and the itemsets are sets of production or test classes. As described in chapter 3, we only consider itemsets of size 2.

In many applications where association rules are used, the search is for rules that are interesting or surprising (i.e., for marketing purposes one seeks for striking combinations of items or interesting correlations between products), In this case we seek to find a global view of the entire change history. We are not primarily interested in specific production/test code class pairs that follow from the rules, but more in the total number of rules that associate production and test code and how strong the statistical certainty of these rules is. We seek to express the global co-evolution of production and test code classes, and not specific pairs. The interpretation of the rules we seek is thus different than in most applications of association rules.

We explore measurements in two directions, which follow from the direction of the research questions:

Test suite quality: Metrics that describe the quality of the test suite. Question to be answered are if the test suite is up to date with the production code, and if it consists of actual unit tests or mainly high level integration tests.

Test effort indication: Metrics that provide understanding of the testing process. These metrics should give evidence of intentional synchronous co-evolution, or a different testing strategy (or the lack thereof).

This section explores the different measurements that can be performed on the mined association rules, and how these can be interpreted to answer the different research questions we have put forward. While we advance through the remainder of the chapter, we present lemmas and hypothesises and raise questions on the metrics to build understanding of how to interpret the metrics.

4.1.1 Individual Rule Metrics

First we introduce metrics that describe one single association rule. These metrics help us to determine the significance and strength of the statistical model that a rule represents. They give argument and weight to the metrics that are described in the next sections. A summary of the metrics can be found in table 4.1.

Support

The support for a rule $\{A \Rightarrow B\}$ is the absolute number of times that the itemset A, B appears in a transaction in the change history. This metric expresses the statistical significance of a rule, or why someone should care about a rule. The more times the items in a rule appear together in the change history, the stronger the statistical basis of the rule is. The support of an itemset (and thus of a rule that is derived from an itemset) is counted by the frequent itemset mining algorithm. An itemset is frequent when its support is equal or larger than the configured minimum support.

While support is counted as an integer value, it can also be expressed as a percentage, by dividing the absolute number by n (the number of transactions in the change history). For example, when the support of an itemset is 10%, the combination of items occurs in 10% of the change history. This is often more intuitive to understand for an analyst. Here, we call this relative support of the frequency of an itemset. The frequency has the property that it is an approximation of the statistical probability of the occurrence of the itemset in the change history ($P(A,B)$). An example: when an itemset $\{A,B\}$ has a support of 2 in 10 transactions, the probability of A and B occurring together in a transaction $P(A,B)$ is 0,2, or 20%. The relative support is a normalisation to the length of the change history.

Support is often used in conjunction with one of the metrics defined below, where support shows the relevance of the rule, and the other metric shows the 'interestingness' of the rule. In a typical analysis of association rules, one looks for specific rules with high support. As mentioned before, our intent is different in that we seek for a global view of the change history. For this purpose we require as many associations between classes in the change history, and can later determine whether they contribute interesting information to the analysis, while with a high minimum support these rule would not be generated. This situation is called the *rare item problem*: classes that occur very infrequently in the change history are pruned although they would still produce interesting and potentially valuable rules. The rare item problem is important for transaction data which usually have a very uneven distribution of support for the individual items (few items are used all the time and most item are rarely used) [25]. The rare item problem can be circumvented by mining with a very low minimum support, but can cause an explosion of the number of found itemsets.

The SIGAR tool is typically configured to mine rules with a minimum support of 2, thus a combination of classes must occur at least twice in the entire history. This is a low number, but we need as much data on the change history as possible, and we expect that there is a significant number of classes that is not changed that often (possibly two to five times) in the history. We will need to verify this second assumption using case study data.

Confidence

The confidence for a rule $\{A \Rightarrow B\}$ is the ratio of the number of transactions that contain $A \cup B$ to the number of transactions that contain A. Confidence expresses the conditional probability $P(B|A)$. Confidence is also called the *strength* of a rule, and is, together with support, the most common measurement for association rules. Since confidence expresses a probability, it takes on values between 0 and 1.

The most common way to express an association rule is by looking at both support and confidence. This is called the suppport-confidence framework. The combination of relevance and strength of the rule is often enough to derive the desired information, and the metrics are easy to grasp.

But a problem with confidence is that it does not take into account possible negative correlations between the items [11]. A rule with a confidence of 0.8 might seem interesting, but the *a priori* probability of B might be 0.9. The occurrence of A thus actually lowers the probability of B. Confidence has no ability to express this situation. Also, according to Brin, confidence assigns high values to rules simply because the consequent is popular [9].

The rule mining algorithm that derives rules from found frequent itemsets takes a minimum confidence value as a parameter. Minimum confidence is used, like the minimum support for the itemset mining, to limit the number of rules that is found, and to set a a lower bound for the 'interestingness' of the derived rules. Because of the exploratory nature of this work, we set the minimum confidence to zero. This causes all rules to be generated. In this way we can learn whether rules with low confidence contribute to increased understanding of the change history, and we can always cut the generated rules at a minimum confidence boundary at a later time.

Lift

The lift (originally called interest) for a rule $\{A \Rightarrow B\}$ is a measure of the relationship between A and B using correlation. Its calculation is derived from the calculation of the correlation between two probabilities [11], and is essentially a measure of departure from independence [9], based on co-occurrence of the antecedent and consequent. Lift measures how many times more often (hence *lift*) the antecedent and the consequent occur together than expected if they where statistically independent.

Lift assigns one to associations where the items are completely independent. Associations that get found by the algorithm have a real value larger or equal to one as the items are more correlated. Negative correlations are between 0 and 1, positive correlations are larger than 1. This measurement is symmetric, which means that the interest of $A \Rightarrow B$ is equal to $B \Rightarrow A$.

When we relate this to our context, this means that when a pair of entities has a large lift value, the entities in question appear to be correlated. Correlated entities are more likely to be changed because of changes in its correlated counterpart than not correlated entities. Low lift values imply that the entities are close to being independent, thus that co-occurrences of the entities are more likely to be coincidence.

Lift does not suffer from the rare item problem, but is susceptible to noise in small databases [9]. This could cause the lift metric not to be very suitable to smaller change histories.

Conviction

The conviction for a rule $\{A \Rightarrow B\}$ is a measure of the implication that the rule expresses. Lift only measures the correlation between items, but conviction also measures the implication of the items. It is based on the statistical notion of correlation and logical implication [9]. The benefit that conviction has over lift in measuring correlation between items, is that conviction is not symmetrical, and thus truly measures the implication of an association rule.

The conviction of two items is a real value between 1 and infinity. Totally independent items will have a conviction of 1, and rules that always hold have infinite conviction. Similar to confidence, conviction always assigns the same value to rules that holds 100% of the time. Unlike confidence, conviction factors in both $P(A)$ and $P(B)$. When two items are likely to occur in a transaction, but are completely unrelated to each other, confidence will assign a high value. Conviction, on the other hand, assigns a lower value because the items are likely to occur by themselves. Co-change of the items is then very likely, but not because they are related.

The potential benefit of conviction over confidence and lift is that it measures the direction of the association. This means that we potentially can measure whether there is a difference between the probability of classes being changed because of testing, or tests being changed because of coding.

Strength typically means the confidence of a rule, but from now on we use it as a general expression to indicate the probability of a rule (i.e., confidence, lift or conviction). In this

Metric	**Probability**	**Interpretation**	**Implementation**
$support(A \Rightarrow B)$	$P(A,B)n$	Statistical significance	Counted by Apriori
$frequency(A \Rightarrow B)$	$P(A,B)$	Statistical significance	Normalised support
$confidence(A \Rightarrow B)$	$P(B\|A)$	Conditional probability	$\frac{s(A,B)}{s(A)}$
$interest(A \Rightarrow B)$	$\frac{P(A,B)}{P(A)P(B)}$	Correlation between items	$\frac{s(A,B)n}{s(A)s(B)}$
$conviction(A \Rightarrow B)$	$\frac{P(A)P(\neg B)}{P(A,\neg B)}$	Logical implication	$\frac{s(A)n-\frac{s(A)s(B)}{n}}{s(A)-s(A,B)}$

Table 4.1: Individual association rule metrics.

Summary of individual association rule metrics. Here n is the total number of transactions, and $s(A)$ is shorthand notation for $support(A)$.

discussion we encountered the first pieces of the exploration-puzzle, which we formulate in the following lemmas.

Lemma 4.1 *The support of an association rule is equal to the logical coupling between two code entities, and determines the statistical relevance of the association rule.*

Lemma 4.2 *The confidence, lift and conviction of an association rule each give a probabilistic describtion of the occurrences of the entities in a rule. Larger values of these metrics correspond to stronger rules.*

4.1.2 Rule Classification Based Metrics

An individual rule does not provide much information on the logical coupling in a system, but only on one single pair of classes. To be able to analyse logical coupling on a larger scale, the rules have to be aggregated. We define metrics based on the aggregated rules through the reasoning that follows.

Logical coupling between entities

The support of a single rule is the number of time the entities in the rule co-change (or the amount of logical coupling, by lemma 4.1). The strength of the rule is expressed as a probability by a second metric (see previous section). Logical coupling is a measurement for the co-change of entities. In terms of change history data, co-change implies co-usage of the entities that appear in the rule: they are changed together because an addition or change in the code intersects both entities. The reason why the change intersects both entities is not known from the rule, but the types of the entities can indicate how the co-change can be interpreted. A programmer may change two production classes in the same commit, because he changes the way the classes interact. A production and a test class may be changed together because newly added functionality is tested by the test class.

Lemma 4.3 *Logical coupling expresses co-change of classes. Co-changing classes imply that the classes are used together by a programmer. With lemma 4.1, an association rule implies co-usage between the items in the rule.*

When rules are grouped together, the group describes the logical coupling among the entities within that group. The interpretation of logical coupling among entities of different types is described as follows:

Logical coupling between production classes: This describes logical coupling in the traditional sense [10, 15, 14, 34]. In good OOP practice (abstraction, separation of concerns), changes to classes should be local, and not cross-cutting between classes. This implies that two classes should not be changed together often. Strong logical couplings between production classes is considered to be harmful, as it points to dependencies between classes that should not be there. Many logical couplings of average strength between many classes is probably the result of pure production code programming effort, as many production classes are committed together.

Logical coupling between test classes: This is the test class equivalent of logical coupling among production classes. Unit tests should only test one production class, so moderate to high logical couplings between test classes makes little sense. Again, many logical couplings between many test classes could be the result from pure testing effort by the programmers.

Logical coupling between production and test classes: This describes the logical coupling between production and test classes. In contrast to logical coupling between only production classes or only test classes, logical coupling between a production and a related test class are considered to be positive. The notion that production and test classes should change together is the driving assumption of this research.

Logical coupling between undefined classes: Classes that cannot be resolved as a production or test class can appear in the extracted rules. These logical couplings cannot be directly related to programmer effort, as information on the nature of the classes is unknown.

The different types of logical coupling are illustrated in figure 4.1. We summarise this in the following lemma:

Lemma 4.4 *Co-usage between the same or different types of entities is an indication of the distribution of programmer effort.*

Classification of rules

The classification of the rules is used to group the rules and to relate them to a type of logical coupling. The classifications of rules, as listed in table 3.1, allows us to separate the total collection of rules in several groups. Some groups can be divided into subgroups that are increasingly more specific on the rules that belong to the group. Each group can explain different types of logical coupling in a system, and reveal specific information.

Lemma 4.5 *Multiple association rules of some classification (a rule class) describe the logical coupling between entities of the types that determine the classification.*

The main thought behind the grouping of association rules based on their classification is that the composition of the total number of rules indicates what type of logical couplings contribute to the couplings in the entire system, and through lemma 4.4 how programmer effort is composed.

The different rule classification groups, and the interpretation of their ratios and strengths are described in the following overview. The interpretations are hypothetical, and the case studies must show their validity.

All association rules (ALL): The collection of all found association rules.

This group can be used as a reference for other groups. The strength of this group can be a combination of the strengths of the subgroups, but its strength can also be dominated by a single classification.

Pure production rules (PROD): Rules that associate only production classes.

PROD rules will be generated when production classes are often changed together. The ratio of PROD can indicate how much effort is put into changing production code. When there is no (structural) testing performed, transactions will have few test classes in them, and the ratio of PROD will dominate ALL. With phased testing, PROD will also be the dominant group, but TEST could be more present, and have a relative high strength, as the co-use among test classes is expected to be high in the testing phases.

Pure test rules (TEST): Rules that associate only test classes.

Analogous to PROD, TEST rules can indicate test writing effort. The higher the ratio for TEST is, the more dedicated test writing effort can be expected to have occurred in the history of the system. Comparison of the ratios and strengths of PROD and TEST could reveal how much pure testing is performed related to pure coding.

Production-test pairs (P&T): All rules that associate both a production class and a test class. This group describes logical coupling between production and test classes. Within this group we distinguish four subgroups:

The ratio of P&T to ALL, and compared to PROD and TEST tells whether production and test code is often changed together, or that production and test code is more often written in separate stints.

Production to test rules (P2T): Rules that have a production class as antecedent and a test class as consequent. These rules express that a change in production code implies a change in test code with some probability.

Test to production rules (T2P): Rules that have a test class as antecedent and a production class as consequent. These rules express that a change in test code implies a change in production code with some probability. These rules are symmetric to P2T rules, and the union of P2T and T2P equals P&T.

The union of P2T and T2P yields P&T, and both are always symmetric to each other. For each rule $\{A \Rightarrow B\}$ in P2T, its inverse $\{B \Rightarrow A\}$ is in T2P. P2T and T2P rules provide a more detailed view of P&T, as the direction of the association can come into play. For example, when the strength T2P is much stronger than P2T, there are more transactions that contain only production code or both production and test code, than there are transactions that only contain test code.

Matching production to test rules (mP2T): P2T rules where the antecedent and consequent can be matched to belong together as unit test and class-under-test on naming conventions (e.g., $\{Class.java \Rightarrow ClassTest.java\}$, or vice versa).

Matching test to production rules (mT2P): The symmetric counterpart of mP2T.

mP2T and mT2P are subsets of P2T and T2P respectively. These groups are even more specialised than P2T and T2P, as they give weight to actual classes and their tests that, ideally, should co-evolve.

Undefined rules (UNDEF): Rules that cannot be resolved to a classification.

UNDEF rules are most likely the result of entities that cannot be recognised during change history analysis. This could mean , for example, that files are placed at strange locations in the systems file hierarchy, or that the regular expressions used for matching files is not complete. If the cause of the non-identification of the entities is known, UNDEF rules can be of value. Otherwise, there should be as as few UNDEF rules as possible.

Using lemmas 4.4 and 4.5, we state the following theorem:

Theorem 4.1 *The ratios between different rule classes, and the distribution of strengths of each rule class related to other rule classes is a measure for the distribution of programmer effort among different types of code classes.*

We will now describe the strenght measures for classes of association rules in more detail.

Statistical analyses of rule classes

All individual rules are derived from the change history with a statistical certainty, expressed by the different metrics described in section 4.1.1. By aggregating the metrics for all rules and for the different classifications, we can build understanding of how, and how strong, the different rules contribute to the complete picture.

For each class of association rules, the distribution of the values of the different metrics over the rules can show us how strong the statistical model of the rules is. For example, when the majority (say 60%) of all rules has support lower then 3 or 4, the statistical relevance of the complete picture is not very strong. On the other hand, when the confidence of the production-test pairs is generally more towards 1.0 than for the pure production rules, the evidence for co-change among production and test classes is stronger than among production classes only.

We compute the following statistics for each metric for each class, which are the basics of standard descriptive statistics:

- Minimum
- Maximum
- (Arithmetic) Mean
- Standard deviation
- Variance
- Skewness
- Kurtosis

Together these values describe the distribution of the individual metrics of the rules. This distribution can be visualised by using histograms or boxplots. The minimum and maximum define the range of the values, the mean designates the central tendency of the distribution. The more the mean is toward the maximum for the support distribution, the more relevant the rules are for that class, and its ratio is of more importance. The standard deviation (and variance, which is the squared standard deviation) quantifies the spread of the values around the mean.

Skewness is a measure of the asymmetry of the distribution, e.g., whether more weight of the distribution is to the right (negative skewness) or to the left (positive skewness). A symmetrical distribution (like the normal distribution) has a skewness of zero.

Kurtosis is a measure of the 'peakedness' of the distribution. Higher kurtosis means more of the variance is due to infrequent extreme deviations, as opposed to frequent modestly-sized deviations. The normal distribution has a kurtosis of zero.

4.1.3 Entity Based Metrics

With rule based metrics, we look at the logical coupling in a system from a high level view. A test suite consists of many individual tests, which should stand on their own when it concerns unit tests. Integration test have a more cross-cutting nature. To get more information on the amount of effort that is put into maintaining the test suite, we have to step down to the entity level. Here we discuss several measurements that are centered around the entities that occur in the change history.

Test code classification

Given the classification of rules, we can classify the classes that appear in rules, based on in how many rules the class appears as antecedent. That number of rules is equal to the distinct number of other classes a class is associated with. These classifications are described in

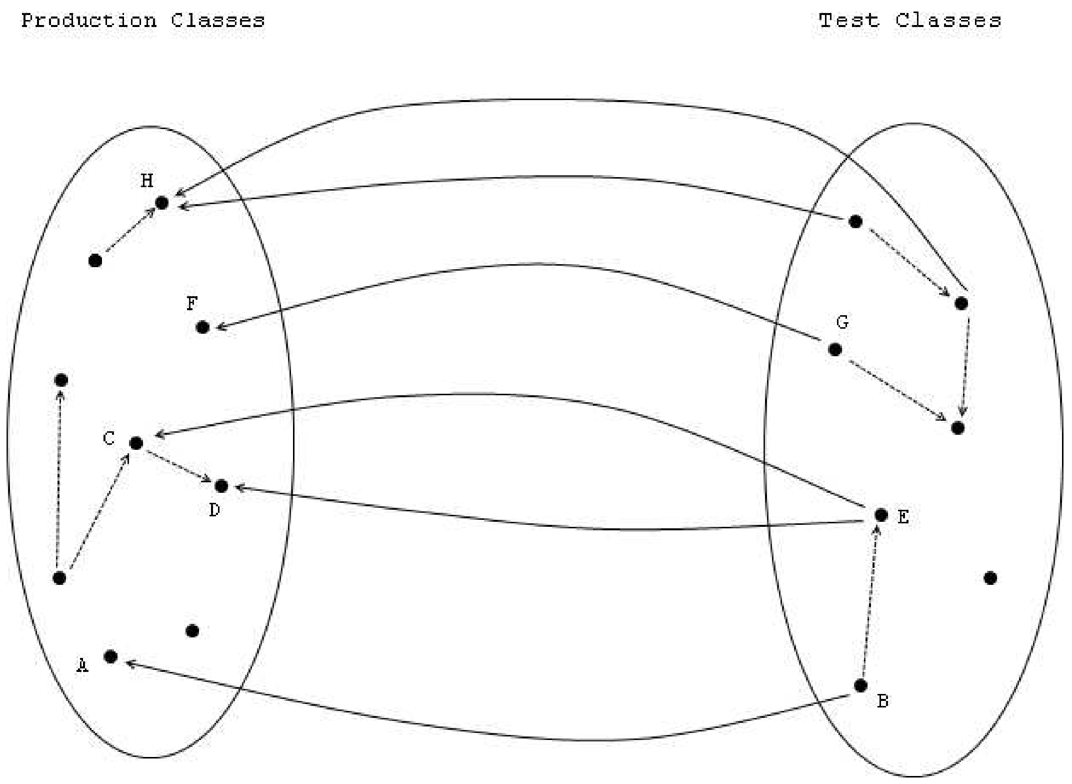


Figure 4.1: Measurement of logical coupling based on association rules.

Interpretation of association rules to understand logical coupling. Rules that connect production or test classes within their own set (dotted lines) measure the logical coupling among classes within that set (pure production or pure test rule classification). Solid lines represent association rules that associate production and test classes (Production-test pair classification). These rule give measure to the co-evolution between the two types of classes. The number of rules that associate a specific class to other classes defines its classification. For example, B is a pure unit test of class A, and so is G of F. E is an integration test of classes C and D. Production class H is associated to several test classes, and becomes classified as a multiple tested class.

table 4.2. Using this classification, we can potentially see what tests are actually used as a unit test or as an integration test.

Rule coverage of classes

Where the quality of a test suite is typically expressed by measuring structural code coverage, we explore an analogous way to logically express the number of classes that are

Rule type	**Cardinality**	**Classification**
$\{TestClass => ProductionClass\}$	$(1:1)$	Unit test
	$(1:n)$	Integration test
	$(1:0)$	Orphan test
$\{ProductionClass => TestClass\}$	$(1:0)$	Untested class
	$(1:1)$	Unit tested class
	$(1:n)$	Multiple tested class
$\{ProductionClass => ProductionClass\}$	$(1:n), n \geq 1$	Logically coupled class
$\{TestClass => TestClass\}$	$(1:n), n \geq 1$	Logically coupled test

Table 4.2: Classification of production and test classes.

Classification of classes, based on the number of rules of a certain type for each specific class. Cardinalities of a class are always of the form $(1:x)$, meaning that the antecedent of the rule is associated with x different tests or classes (consequents) by x association rules. Undefined rules are not applicable.

associated with test classes, and vice versa.

We define the following logical coverage metrics:

Production class mapping ratio: The ratio of production classes that are associated with one or more test classes by a rule. This number is calculated by $\frac{|P2T|}{\#productionclasses}$.

Production class matching coverage: The percentage of production classes that gets associated to a test matching on naming conventions. This number is calculated by $\frac{|mP2T|}{\#productionclasses}$. This is the percentage of production classes that potentially synchronously co-evolve with their unit test, based on the co-usage of the class and its unit test.

Test class mapping ratio: The ratio of test classes that are associated with one or more production classes by a rule. This number is calculated by $\frac{|T2P|}{\#testclasses}$.

Test class matching coverage: The percentage of test classes that gets associated to a production class matching on naming conventions. This is completely analogous expressed by $\frac{|mT2P|}{\#testclasses}$. This is interpreted as the percentage of tests that are potentially synchronously co-evolving with its class-under-test, based on the co-usage of the class and its unit test.

4.2 Evaluation: JPacman Test-Case

We use the educational game JPacman as a ‘guinea pig’ to illustrate and evaluate the technique and metrics presented in this chapter. We generate a ChangeHistoryView of the change history, and walk through the different measurement, comparing the observations with the knowledge we have of JPacman (log messages) and the interpretation of the ChangeHistoryView.

JPacman is a small game, written for educational purposes by a single developer. It was developed using a test-driven development model. The SIGAR tool is run on the change history of JPacman. The tool extracts the following information: in the history of development, 46 classes where found in 246 revisions. Of the total number of classes, 25 where classified as a production class, 20 as a test class, and one class could not be defined. This is an almost one-to-one ratio, and supports the observation from the ChangeHistoryView (figure 4.2) that most classes have an associated unit test. Closer inspection of the list of entities reveals that the unidentified file was a completely separated file called `Aap.java`, that was added for no apparent reason. The tool extracted a total of 1334 association rules from the change history. The rules where mined with a minimum support of 2, and a minimum confidence of $0,0$.

4.2.1 Test Process Understanding

In table 4.3 the ratios for the rules different are listed. The first thing we notice is that almost half of the rules associate a production to a test class. Compared to the ratios of pure production rules (36,88%) and pure test rules (15,74%) this is a large share of the total number of rules. This is the first indication that production and test code are developed simultaneously and not in distinct phases, as can also be observed in the ChangeHistoryView.

When looking at the statistics for the different types of rules, listed in table 4.4, we can observe the distributions the four basic metrics for the different rule classifications. This overwhelming amount of numbers can be conveniently composed into boxplots and histograms. The boxplots for support and confidence for all rule classes are depicted in figure 4.3. The histograms in figures 4.4 (support) and 4.5 (confidence) provide an alternative view on the distributions. For the sake of this test-case, only the plots for support and confidence are considered. For the case studies in the following chapter we will only discuss the boxplots, as they provide enough information, but the other data is included here as an example.

We can make some quick observations from the data of JPacman:

- In addition to the ratios in table 4.3, we computed the ratios for matching rules. From the third row of data we can compute that 3,48% (twice 1,74%) of the P&T rules actually match on naming conventions.

- The means of all metrics for mP2T and mT2P are higher than for P2T and T2P. This confirms that classes and tests that belong together are more likely to change together, and actually have changed more often together (support) than other pairs of classes and tests. This could mean that co-change between seemingly unrelated classes and tests is coincidental, and not structural, as it most likely is for matching pairs.

- The interest (correlation measure) in table 4.4 has minimum values that are below 1, which means that some rules are negatively correlated. This only occurs in P&T

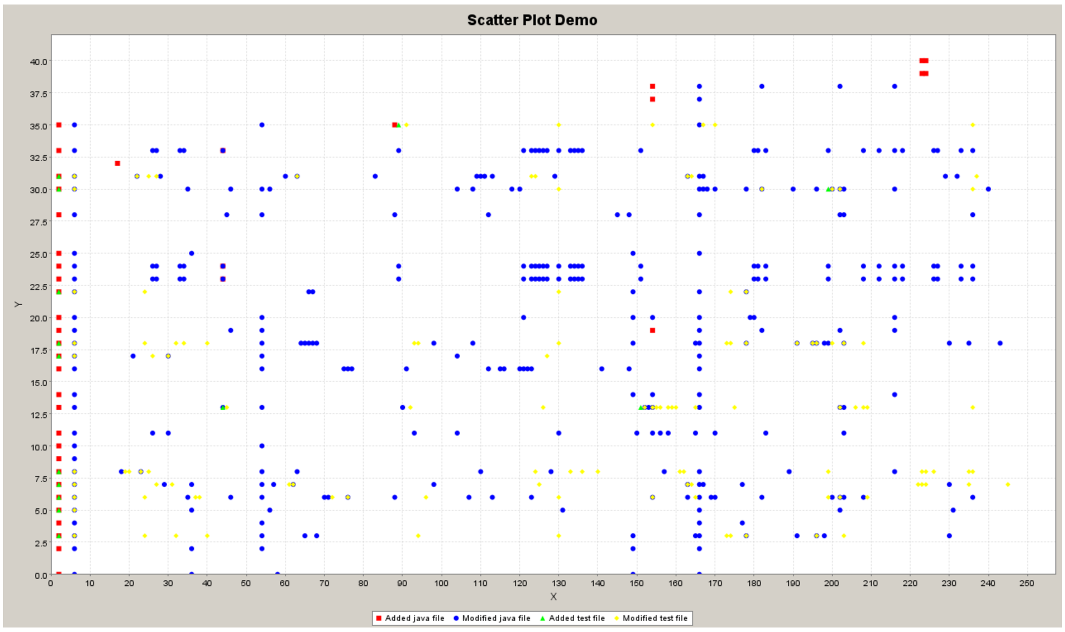


Figure 4.2: ChangeHistoryView of JPacman.

In the ChangeHistoryView we can see that the initial commit contains a large number of classes. From the log messages, we can see that the project existed before it was put under version control in this repository. No real development on the project begins until about revision 20. From that point on we can observe coinciding production and test classes among more individual changes to classes and tests. This observation matches the test-driven development model of the author. There are some tests that get changed often in the history, and inspection of the log data suggests that these are mostly high-level, organisational tests (`TestAll.java`) (note that in the view, these tests are mistakenly shown as production classes). Commits that touch many files together are typically the result of automated, and mostly cosmetic, changes to the code base.

rules, but not in the matching variants, so we can see that the negatively correlated entities are production and test classes that most likely do not belong together.

- The support of PROD rules is somewhat spread out, but has generally higher values than TEST rules. Thus logical coupling (and co-usage) among production classes is higher than among test classes. Thus there is more dedicated code writing effort than pure testing effort

- Support for P&T, P2T and T2P rules hit rock-bottom. Only a few outliers have a support over 2. The interesting observation here is that practically all the outliers are

Type	**N**	**Percentage**
ALL	1334	100%
P&T	632	47,38%
P2T	316	23,69%
T2P	316	23,69%
mP2T	11	0,83%
mT2P	11	0,83%
PROD	492	36,88%
TEST	210	15,74%
UNDEF	0	0%
Production classes	25	54,35%
Test classes	20	43,38%
Undefined classes	1	2,17%

Table 4.3: Rule ratios for JPacman.

matching rules. Thus the production and test code that we expect to occur together actually does occur more often than not trivial combinations. This is evidence of intentional co-evolution among production and test classes.

- The observation of high support for matching rules is also visible in the confidence distributions. The weight of the matching rules is significantly higher than for other P&T rules. PROD rules span the entire spectrum. There are no matching rules that occur always together (confidence of 1.0), but some P&T rules do. So there could be some integration tests that get changed every time the class under test is changed.
- The observation that both support and confidence are higher for matching rules than for P&T rules, add more weight to the distribution of the confidence of mP&T rules. The matching combinations generally occur more often, so the co-change more structural in nature.
- Support and confidence are generally higher for TEST rules than for P&T (and subclasses) rules. This indicates that, beside the intentional co-change among matching rules, there is more dedicated test writing effort than continuous 'random' testing of not directly related production and test classes.
- The scale of the histograms makes the bars for matching rules nearly invisible. The boxplots provide a clearer view on the distributions. The histograms however are better for spotting interesting peaks and outliers.

4.2.2 Test Suite Quality

Figure 4.6 shows the number of P2T or T2P association rules in which a production or a test class occurs as antecedent. We can see that the majority of entities is associated often

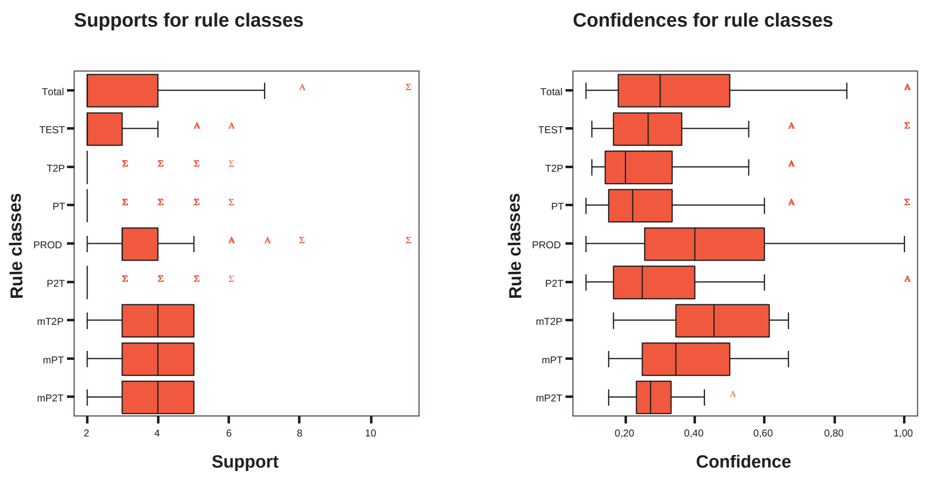


Figure 4.3: Boxplots of support and confidence for JPacman.

(15 to 23 times), while some classes are never encountered as antecedent in a P&T rule. Based on the presented classification of code entities (table 4.2), most of the tests would be classified as integration or orphan tests. Based on the number of mT2P rules (11 rules), we know that we should encounter at least eleven true unit tests. So for the JPacman test-case this analysis is not providing insightful results.

The scatterplot in figure 4.6 shows the number of times each production or test class occurs in a P2T or T2P rule respectively. We can see that test classes are more often associated to a production class than vice versa. Since there are more production classes than test classes, this is plausible. But it means that most classes are associated with almost every other class. We can see that the majority of entities is associated often (15 to 23 times), while some classes are never encountered as antecedent in a P&T rule. Production classes are associated with 14 to 16 tests, on a total of 20 test classes. The other way around, tests get associated with about all but three production classes. Based on the number of mT2P rules (11 rules), we know that we should encounter at least eleven true unit tests. The consequence of this is that the data of the JPacman test-case is not applicable to the classification scheme of table 4.2.

Based on the observation that 11 tests are used as true unit tests, the other 9 tests are potentially integration tests. This could explain why the collective strengths of TEST rules are higher than P&T rules. Integration testing could be more of a ‘separated’ activity than true unit testing, which appear to be a continuous effort.

We compute the different rule coverage measures for classes. The production class mapping coverage is $\frac{316}{25} = 12,64$. On average, each production class is associated with about 12 test classes.

The production class matching coverage is $\frac{11}{25} = 0,44$, which indicates that based on the usage of tests, only eleven classes can be seen as a (possibly) co-evolving unit test and

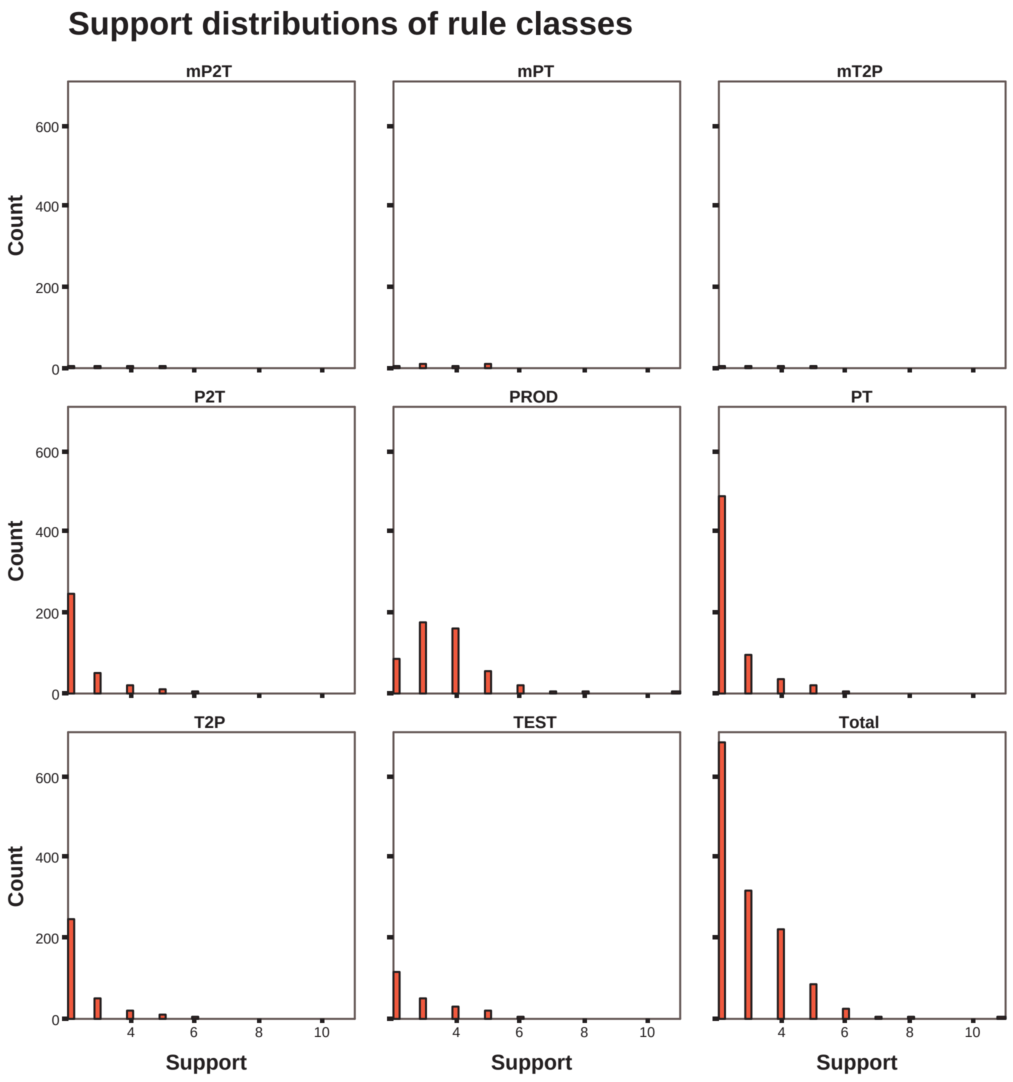


Figure 4.4: Histograms of support for JPacman.

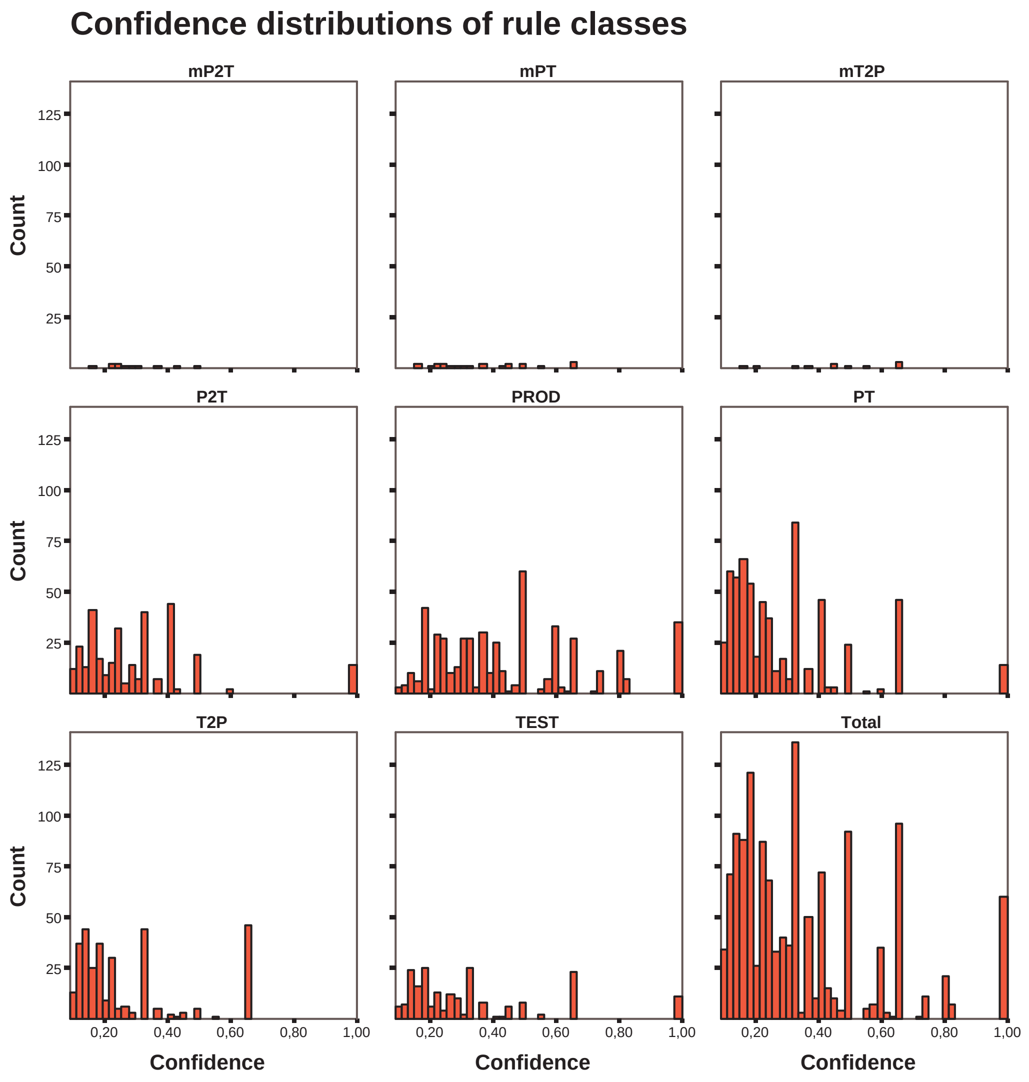


Figure 4.5: Histograms of confidence for JPacman.

	All	P&T	P2T	T2P	mP2T	mT2P	PROD	TEST	UNDEF
% to total	100	47,38	23,69	23,69	0,83	0,83	36,88	15,74	0
% to P&T					1,74	1,74			
Support									
min	2	2	2	2	2	2	2	2	NaN
max	11	6	6	6	5	5	11	6	NaN
mean	2,86	2,34	2,34	2,34	3,73	3,73	3,55	2,82	NaN
std dev	1,11	0,72	0,72	0,72	1,19	1,19	1,19	1,06	NaN
variance	1,24	0,52	0,52	0,52	1,42	1,42	1,40	1,12	NaN
skewness	1,64	2,38	2,39	2,39	-0,23	-0,23	1,52	1,15	NaN
kurtosis	4,80	5,58	5,64	5,64	-1,51	-1,51	6,21	0,38	NaN
Confidence									
min	0,09	0,09	0,09	0,11	0,15	0,17	0,09	0,11	NaN
max	1,00	1,00	1,00	0,67	0,50	0,67	1,00	1,00	NaN
mean	0,36	0,29	0,30	0,28	0,30	0,46	0,45	0,33	NaN
std dev	0,23	0,19	0,19	0,18	0,10	0,18	0,24	0,23	NaN
variance	0,05	0,03	0,04	0,03	0,01	0,03	0,06	0,05	NaN
skewness	1,23	1,71	2,15	1,25	0,87	-0,33	0,78	1,51	NaN
kurtosis	0,95	3,23	5,70	0,23	0,50	-0,98	-0,17	1,68	NaN
Lift									
min	0,73	0,73	0,73	0,73	2,11	2,11	1,55	1,13	NaN
max	50,67	50,67	50,67	50,67	12,67	12,67	38,00	50,67	NaN
mean	6,73	5,52	5,52	5,52	6,16	6,16	8,68	5,81	NaN
std dev	6,23	5,85	5,86	5,86	4,11	4,11	6,31	6,01	NaN
variance	38,76	34,27	34,33	34,33	16,90	16,90	39,76	36,11	NaN
skewness	2,88	3,90	3,91	3,91	0,81	0,81	1,72	4,55	NaN
kurtosis	12,73	22,60	22,79	22,79	-1,10	-1,10	3,31	29,19	NaN
Conviction									
min	0,95	0,95	0,97	0,95	1,16	1,11	1,08	1,01	NaN
max	5,80	2,96	2,27	2,96	1,92	2,84	5,80	2,92	NaN
mean	1,55	1,37	1,30	1,43	1,36	1,86	1,86	1,44	NaN
std dev	0,72	0,45	0,24	0,58	0,24	0,66	0,95	0,52	NaN
variance	0,52	0,21	0,06	0,34	0,06	0,44	0,91	0,27	NaN
skewness	2,81	2,33	1,14	1,79	1,61	0,62	2,18	1,88	NaN
kurtosis	9,55	4,74	1,34	1,59	2,10	-1,17	4,64	2,23	NaN
# infinite	60	14	14	0	0	0	35	11	0

Table 4.4: Summary of rule distributions of metrics for JPacman.

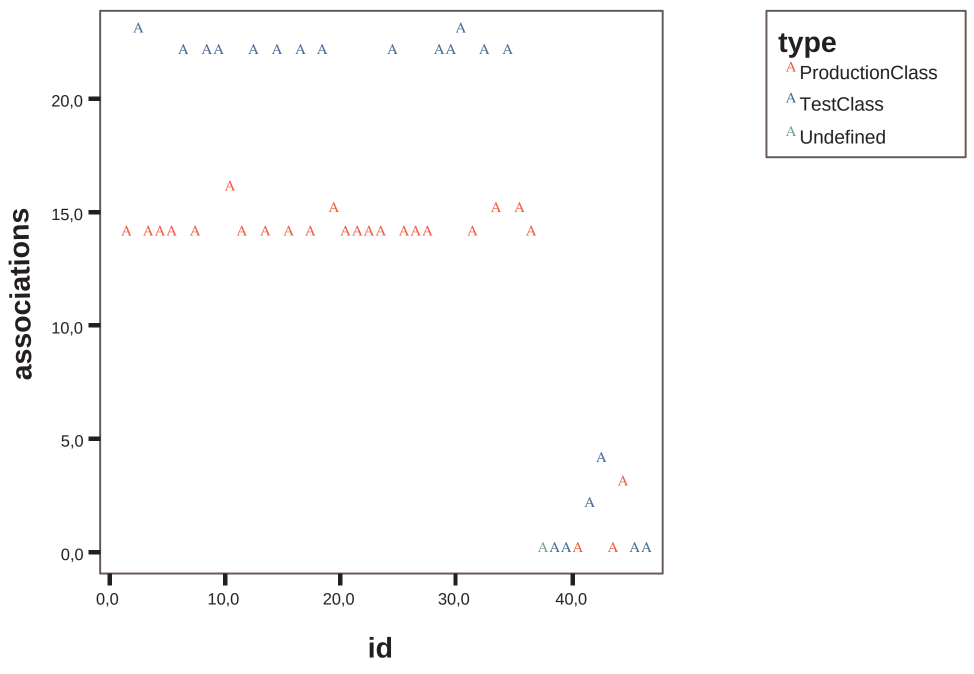


Figure 4.6: Class and tests occurrences for JPacman.

these test 44% of the production classes. When we look at the data for mP2T, the means are not particularly strong, especially compared to mT2P. The test class matching coverage is $\frac{11}{20} = 0,55$. We could say that 44% of the system is unit tested by 55% of the test classes, based on the co-usage of the classes.

Chapter 5

Case Studies

In the previous chapter we proposed several measurements based on association rules. Using a number of case studies on systems with different characteristics, we attempt to explore and interpret the applicability and usefulness of the different measurements. We use the proposed measurements to answer the research questions posed in the first chapter for the case studies. Using the insight into the history of the case studies that the measurements provide, the applicability and usefulness of the metrics can be evaluated.

We discuss the following different categories of questions with the case studies data:

Analysing the development and testing effort: Can we find evidence of intentional co-evolution of production and test code, or a phased testing approach? How is the coding effort of programmers distributed over writing production code and testing?

Test suite evaluation: Are tests found in the system used as a unit or a high-level integration test? Are tests up-to-date with respect to changes in the production classes?

Basic rule metrics interpretation: How should we interpret and value the meaning of the individual association rule metrics support, confidence, lift and conviction?

We have data on the case studies from a number of sources, and each source serves its own purpose.

System characteristics: Basic information of the structure of the system, like the number of production classes, tests and revisions. This information defines the system.

ChangeHistoryView: The ChangeHistoryView of a case is used to get a high level view of the system. The view is also used to select systems for the case studies.

Log data: The commit messages of the change history can provide information on why things happened in the change history. Why was a change made, or did all test automatically pass after a refactoring? Log messages are used as internal validation of the development and testing process.

System developers feedback: We had each developer of the systems explain their development and testing practices to understand what we see in the ChangeHistoryView and the data. This is used as external validation of the development and testing process.

Association Rules: The association rules that are derived from the change history are the basis for the explored metrics.

Important observations that we come across in this chapter will be noted in separate statements. In section 5.4 we will evaluate the observations to deduce general statements on our approach.

ChangeHistoryView Inconsistency

During the analysis of the ChangeHistoryViews of the systems, some inconsistencies where found in the visualisation of the change history. It appears that some test classes not only get plotted on top of the class-under-test, but also as a separate production class. Because of this, too many files are listed on the vertical axis of the view. For example, the view of the JPacman test-case (figure 4.2) lists some 40 classes. From the numbers extracted by the change history analyser of the tool, we learn that there were 25 production classes in the history of the application, out of a total of 46 files. From the number of matching rules extracted by the tool we know there are at least 11 unit tests in the system. That leaves 9 possible integration tests. Thus we expect a total of 34 rows in the view. Despite these inconsistencies, we believe that the global observations on co-evolving entities and phases of increased production or test code writing effort are still valid. All production classes and associated tests are plotted correctly and will allow us to see the patterns we are looking for.

5.1 Systems Desciptions

We have selected a number of systems for the case studies. All systems had to comply to the requirements that follow from the tool implementation: it must be written in Java and automated unit tests must be included in the SVN or CVS version control system. We obtained about 20 candidate systems from three different industrial companies and made a selection based on the ChangeHistoryView and upfront knowledge of characteristics and origins of the system. We include one open source software system. Here we will discuss the selected systems. The ChangeHistoryView for the case studies is also presented to get a feel for the systems.

5.1.1 OSS: Checkstyle

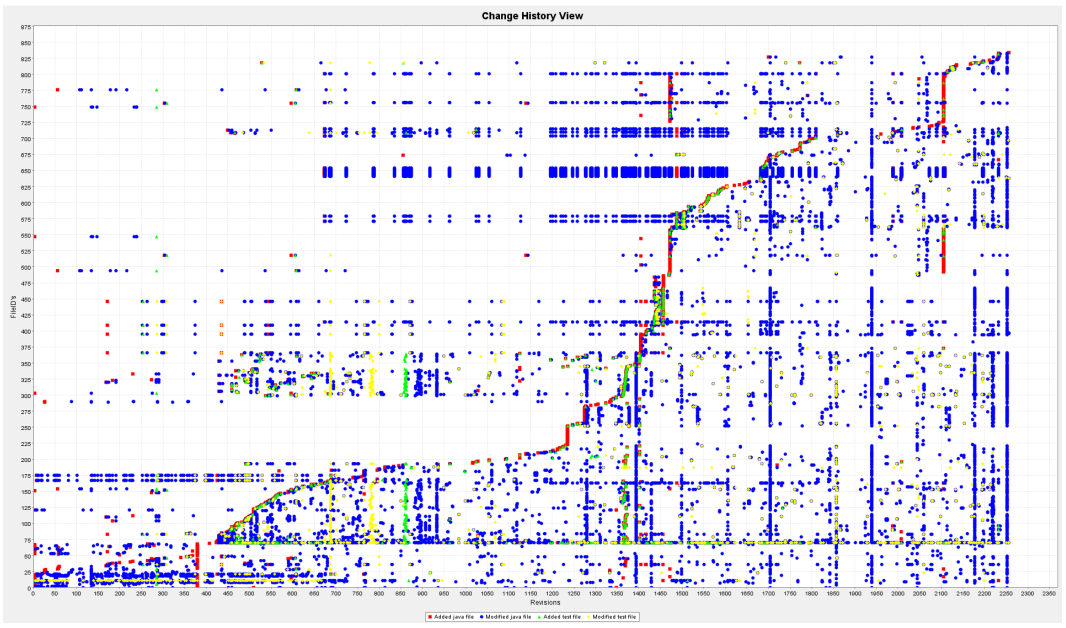
Checkstyle is a open-source coding standard checker for Java source code. Between June 2001 and March 2007, 2259 commits resulted in a total of 1160 files, of which 797 where Java classes, and 363 were identified as a test class.

The ChangeHistoryView for Checkstyle is shown in figure 5.1. From the view, we can see that initially little testing is performed. There is only one test in the system up to about revision 250. After that, the system starts to grow and tests are added with new code. Around revisions 690 and 780, to phases of pure test effort can be distinguished, and after

System	# Java	# Test	# Undefined	Revisions	First commit	Last commit
Checkstyle	462	519	0	2259	June 2001	March 2007
System A.I	2480	1675	0	2838	April 2004	January 2008
System A.II	375	170	8	1244	July 2003	January 2008
System B.I	737	368	0	8853	August 2006	May 2008
System B.II	2151	362	0	10395	December 2004	May 2008
System C.I	1038	383	72	4403	August 2006	May 2008
System C.II	122	81	5	1076	August 2006	March 2008

Table 5.1: Characteristics of the case studies.

Figure 5.1: Checkstyle ChangeHistoryView



revision 850 many tests are added. After these additions, there is a significant period of pure coding with hardly any maintenance to the tests being performed. We see some recurring test phases around revisions 1380 and 2100. For the larger part of the history, tests appear to receive a fair amount of attention from developers, as many additions and changes to production code are accompanied or closely followed by the addition or change in a related test file. Regular commits touching many files can be seen (blue vertical bars) and these are, with some exceptions, because of code cleanups or copyright notice changes.

5.1.2 Company A Systems

Company A is a IT consultancy service provider. For its business, it develops several tools for static source code analysis. Originally, different analysis tool were developed separately, but over time different projects were merged into one system (system A.I), and common functionality was abstracted into a utility library (system A.II). Our change history begins with the merging of different projects into one repository. We will mainly focus on system A.I, as both projects share the same development model, the histories show similar practices, and the two systems are developed in tandem. System A.I is significantly larger than A.II (see table 5.1), and changes more frequently. System A.II is also considered to evaluate what impact the size and length of the change history have on the analysis with our approach. We expect both systems to show similar results.

SCRUM was recently introduced as the main development methodology. Before that adoption, the process has always been centered around agile practices.

Company A employs a strong test-driven development model. Developers are expected to test all code and keep the test up to date. Code coverage measurements are used to monitor and control the test suite. Unit testing has been in use since the beginning of development of the initial projects, which was around the year 2000.

The ChangeHistoryView for A.I shows a steady growth curve, but because of the merges of different projects and reorganisations of the repository many outliers clutter the view. The total overview of the view (figure 5.2) shows code writing and testing effort overlapping for the entire change history. Close inspection shows additions and changes to the code base being accompanied by added and changed tests most of the time. Large commits correspond to refactorings (with changing tests) and code cleanups (less often changing tests). System A.II yields very similar observations.

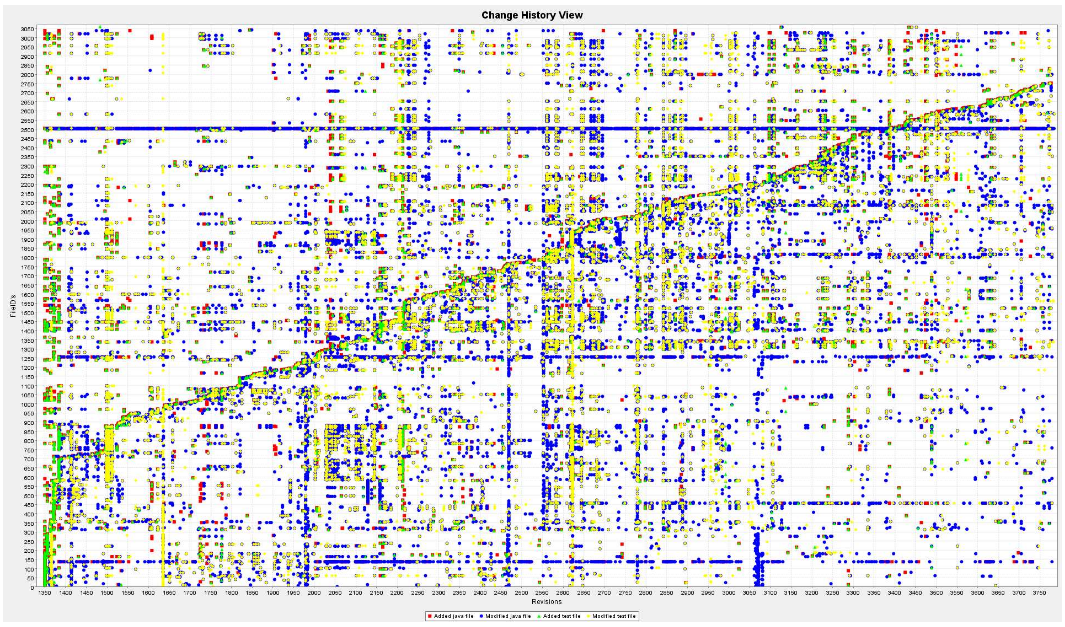
5.1.3 Company B Systems

Company B is a large international transportation company that uses many information systems to support its activities. Systems are both developed in-house, as well as by an external contractor. For some projects coding is outsourced.

We selected two company B systems, which both have been developed by the same external developer, and have around the same length of the change history. They, however, have a striking difference: the use of unit tests. The first system (B.I), makes heavy use of unit tests, while the second system (B.II) only shows the sporadic use of tests, and hardly any evidence of intentional co-evolution of production and test classes can be found. The two systems show a completely different picture, and we are interested in the effect this has on our approach. The ChangeHistoryView for B.I is shown in figure 5.3, and for B.II in figure 5.4.

System B.I appears to be thoroughly tested, and many tests are added together with new production code being introduced. There are two large spots of test activity noticeable: between 2000 and 2500, and after the addition of many files (IDs 650–700) after revision 2500. The first bulk of changed tests is the result of cleaning up code, and appears to have little to do with actual structural testing. The second hot-spot involves an overhaul in the

Figure 5.2: System A.I ChangeHistoryView



software, and the changes to tests that follow the large change actually are adaptations to the new production software. Thus here the testing is partially done after the code has been committed to the repository.

System B.II tells a different story. Testing occurs sporadic during the long history of the system. There are two minor, and one major concentrations in the view: at the early beginning, and around revision 6000. The large concentration of test activity after revision 6000 around file-ID's 2400 and 2500 occurs both because of the refactoring of tests (the tests are added after the actual code is written), and cleaning up the test code.

5.1.4 Company C Systems

Company C develops both hard- and software for mission-critical systems for a wide range of large financial, service oriented and industrial customers. One of its main selling points is commitment to quality, and to this effect the company very actively uses unit tests in their development model. The systems we consider were developed by small teams of under 10 developers, and using a blend of RUP, DSDM and eXtreme Programming, picking elements as they see fit. The use of unit testing is required for each project, and is controlled using code coverage measurement and test code is peer-reviewed.

Similar to Company A, we evaluate two systems: one large system (C.I), and a small system with a 'supporting' role (B.II). The systems are web-based applications used to manage contracts for an international financial service provider. Again, we primarily discuss the larger system.

Figure 5.3: System B.I ChangeHistoryView

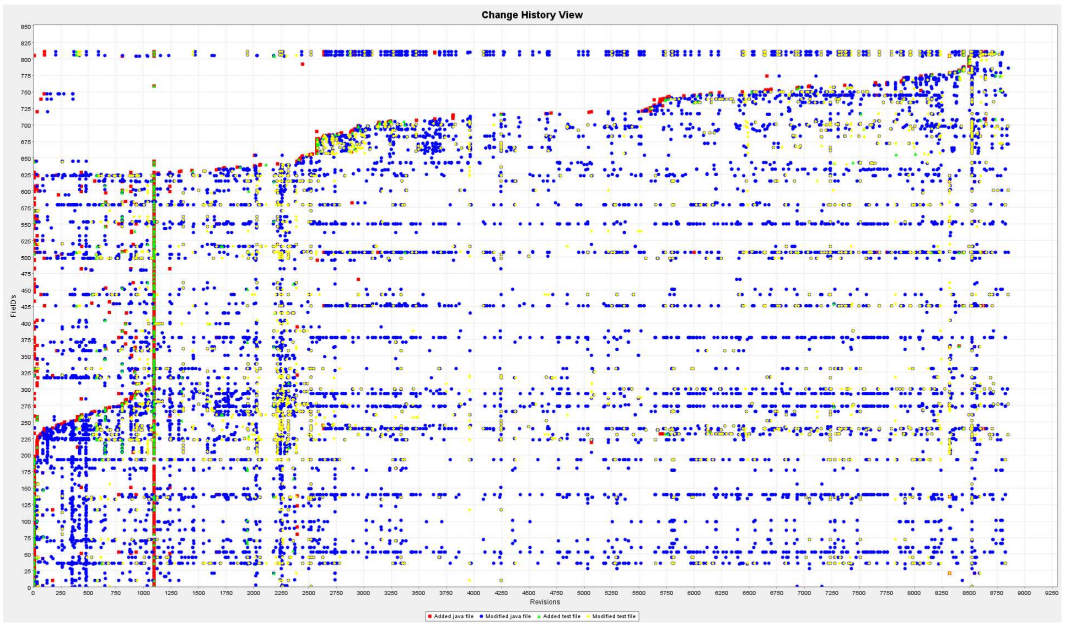
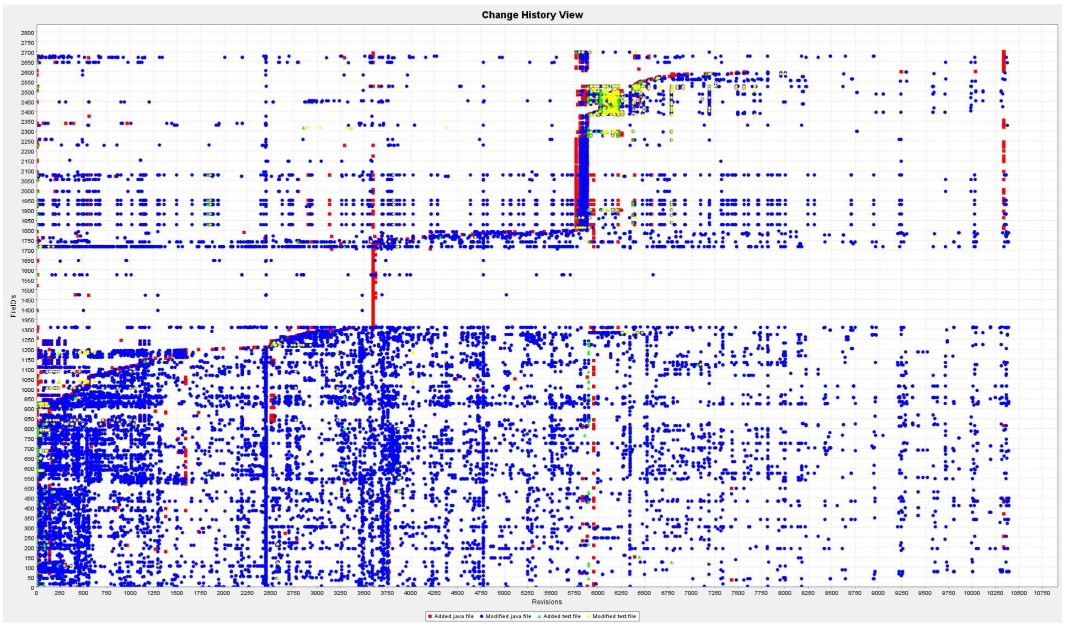


Figure 5.5 show the ChangeHistoryView for the history of system C.I. The view reveals a smooth development curve of the system. There are not many outliers, which indicates that the code in the repository was very stable, e.g., not many files have been moved to other locations. The overlap of tests and production classes looks very consistent. There is a steady line of changes to test code trailing the edge of the growth curve, indicating that testing effort accompanies additions to the system. The general observation is a disciplined and consistent test-driven development practice. Large commits are not very evident, and most occurrences involve actual work on the system, and not many cosmetic changes as with the other considered systems. A possible cause can be that very clear and stable coding standards are set or the requirements for the system do not change often, and thus not many clean up work or large overhauls have to be done. The smooth growth curve supports this last possibility (except for the ‘bumps’ around revision 415 and 3540, where files for complete modules are added in one commit). System C.II shows an identical change history.

5.2 Test Process Analysis

Our first attempt at applying the mined association rules to the case studies is to study the ratios of the rules and the distributions of their strength metrics. We interpret the data of the systems to deduce how the coding effort of programmers is distributed among writing code and testing. In table 5.2 the total number of rules (ALL rule class) and the percentages that each of the different rules classes contributes to the total are listed. We will discuss the ratios and strengths of the rules for the cases in the following subsections. The observations

Figure 5.4: System B.II ChangeHistoryView



we make assume the cases (sections) are read in the order that they are discussed.

5.2.1 Checkstyle

The first observation when looking at the rule ratios for Checkstyle is the huge amount of PROD rules. 98,86% of the 58566 rules express an association between two production code entities. While initially the developers hardly used unit tests, they adopted a more test-driven development model over time. The first period of development thus practically only involved production code, but the several phases of pure testing effort that were observed in the ChangeHistoryView (figure 5.1) could have created a fair amount of TEST rules. Revisiting the ChangeHistoryView in close detail reveals that the testing phases involve commits with only a few tests per commit, while many other commits contain a larger amount of production files. As the change history of Checkstyle contains several recurring very large commits, the generation of many rules from those single commits is very large ($2^m - 1$, with m the size of the commit).

We reason that many of the PROD rules should have low strengths because of the incidental nature of the combinations. We check this by turning to the distributions of the rule strength metrics. The distributions of the metrics are shown as boxplots in figure 5.6. The average support of rules is generally low, with the PROD rules having many extreme outliers (shown as crosses). This shows that many of the combinations of files only occur a few times in the change history. In table 5.3, we compute the ratio of the number of rules that get generated per revision, and the number of rules that get generated per code entity. Looking up the ratios for Checkstyle, we find that each entity results in almost 60 rules. This

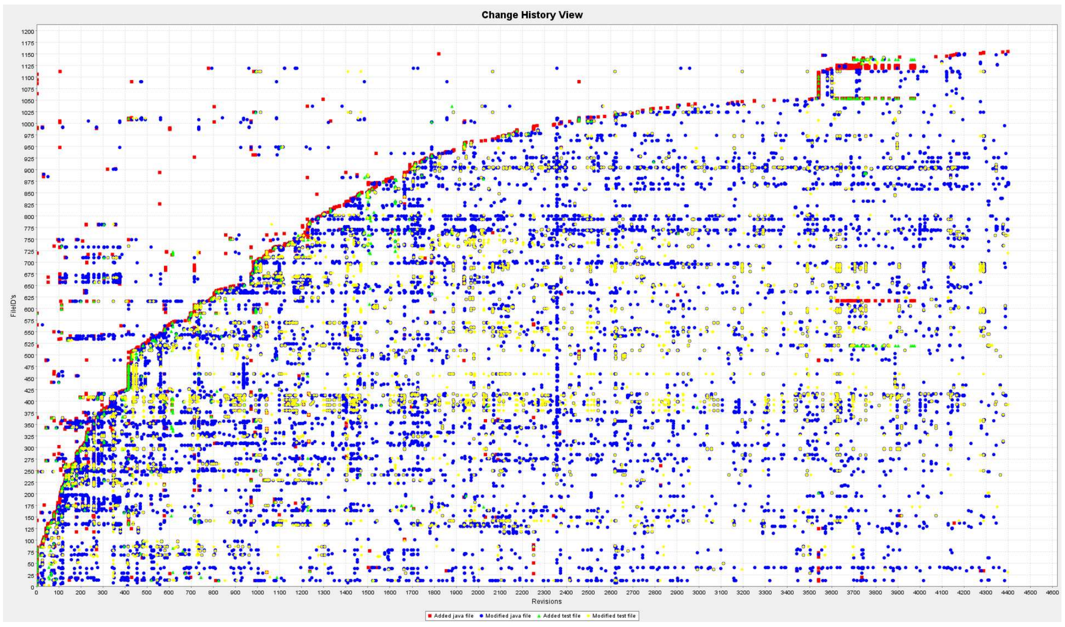


Figure 5.5: System C.1 ChangeHistoryView

Rule Class	**Checkstyle**	**A.I**	**A.II**	**B.I**	**B.II**	**C.I**	**C.II**
ALL (N)	58566	101896	14590	8820	219248	27308	498
PROD	98,86%	35,15%	49,64%	39,00%	99,12%	51,84%	40,96%
TEST	0,48%	26,11%	9,95%	24,81%	0,20%	9,99%	16,87%
P&T	0,67%	38,75%	40,25%	36,19%	0,69%	32,44%	32,13%
P2T	0,33%	19,37%	20,12%	18,10%	0,34%	16,22%	16,06%
T2P	0,33%	19,37%	20,12%	18,10%	0,34%	16,22%	16,06%
mP2T	0,09%	0,78%	0,74%	0,83%	0,01%	0,78%	4,82%
mT2P	0,09%	0,78%	0,74%	0,83%	0,01%	0,78%	4,82%
UNDEF	0,00%	0,00%	0,16%	0,00%	0,00%	5,73%	10,04%

Table 5.2: Rule ratios for the case studies.

supports the idea that the huge ratio for PROD is there because there are commits with many production files in them. We can also see this in the ChangeHistoryView of Checkstyle, by noticing the large blue vertical lines in the view.

Observation 5.1 *Commits touching large numbers of files generate an exponential number of rules. These rules can dominate the ratios of the rule classes.*

The ratios of TEST and P%T (sub-)classes are minimal at best. But the Checkstyle developers appear to have adopted a decent testing practice over time. We can recognise

System	Rules/revision	Rules/entity
Checkstyle	25,93	59,70
A.I	35,90	24,52
A.II	11,73	26,38
B.I	1,00	7,98
B.II	21,09	87,25
C.I	6,20	18,29
C.II	0,46	2,39

Table 5.3: Ratios of rules, entities and revisions for the cases.

several phases of testing in the first half of the change history, and a more test-driven approach in the latter part. Looking at lift, the correlation among matching production and test classes 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. For example, 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 tests and code, as we do not yet know how many tests are actively maintained.

An interesting contrast is the significantly weaker distribution of mP2T rules for confidence and conviction. As these two metrics are not symmetric for a rule, 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 lift of matching rules are identical, because of the symmetry of the lift metric. The measured strength between matching production and test classes is not as evidently higher using lift than it is with confidence and conviction, since the highly correlated (m)T2P rules are averaged out against the lowly correlated (m)P2T rules.

Observation 5.2 *Strong distributions for TEST, related to (m)P&T rule strength distributions, originate from co-usage of test classes and indicate that testing is performed as a separate activity.*

Observation 5.3 *Lift averages the measurements for matching rules in different directions. This causes the differences to even out, and makes lift a less specific metric.*

Summarising, we can see that Checkstyle has a very high co-change of production classes. These co-occurrences are mostly unintentional and caused by code cleanup activities. Apart from that, there is reasonable evidence for both a phased and a test-driven testing practice.

5.2.2 System A.I and A.II

Recalling observations of the ChangeHistoryViews from systems A.I (figure 5.2) and A.II, we expect these two systems to show similar results. The ratios of the rule classes are much

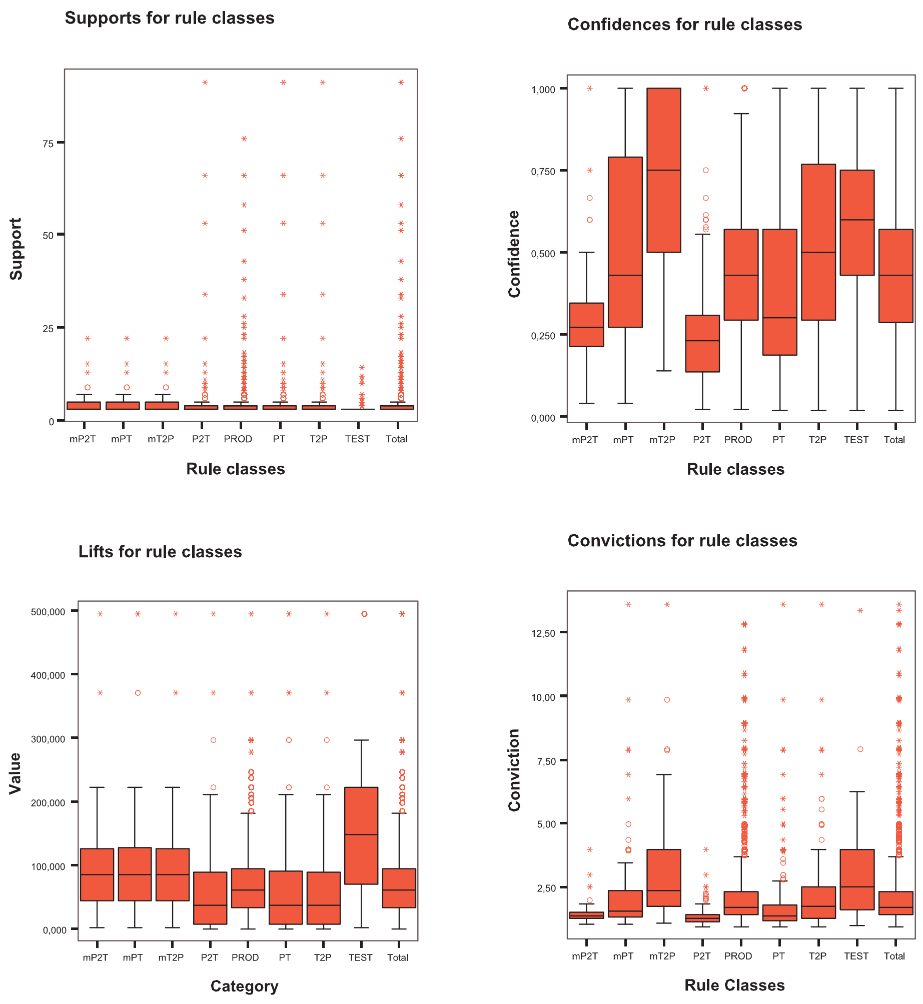


Figure 5.6: Checkstyle rule strengths distributions

more even partitioned than for Checkstyle. For system A.I, the ratios of PROD and P&T are almost one to one. System A.II has a slightly larger difference, but nearly 10% of the total number of rules is of class TEST. Both systems A.I and A.II show remarkable uniform strength distributions for all rule classes in addition to the evenly partitioned ratios.

In line with the observations from the ChangeHistoryView, the association rules reveal a strongly synchronous co-evolution of production and test code for company A's systems. Not only are the rule class ratios evenly partitioned over pure coding and test-driven development, the strength distribution present a uniform picture. PROD, TEST and P&T rules show equal measurements, and so do the matching classes mP&T, mP2T, and mT2P. The distributions are more uniform (resembling the normal distribution), and show less skewness than for the Checkstyle case. We have four remarks on the data for these two systems.

System A.I has surprisingly strong TEST rules, and they represent twice as much of the total compared to system A.II. This could be caused by commits that contain multiple pairs of production and test code. Two pairs generate P&T, PROD and TEST rules. The strong TEST rules can occur when the combinations of test classes occur often in the history. The strength of these rules is possibly a by-product of the normal development cycle combined with more dedicated test effort.

In each of the up to now discussed systems the strengths of mT2P rules are high. For A.II, however, T2P is quite low even though mT2P is strong. This means that the other part of the T2P rules (i.e., the non-mT2P rules) are weak to such an extent that they bring down the average of T2P. Thus the non-mT2P rules are significantly more independent than mT2P rules, and occurrences of non-mT2P rules are incidental in nature, versus structural co-use of truly related production and test code.

System A.I shows values for conviction that reach up in the thousands, where for the other cases the values concentrate around 2. We are not sure whether this is because of miscalculations for this specific case, or that it is a property of the system. Lift values for A.I are not noteworthy exceptional, thus we assume an error, even though none was found.

System A.II contains a number of undefined rules. Support for these rules is very low, but the confidence and lift are very high. We inspect the nature of these rules, and find that they are generated from a number of test stubs and helper objects. These classes belong to test code, but are not classified as such because they do not adhere to the regular naming conventions of tests. This explains the found data: the stubs do not occur frequently in the change history, because they have a very static function. Since they are part of a very isolated and not frequently changed part of the test suite (only belonging to some specific tests), the metrics are very strong.

We summarise the following observation:

Observation 5.4 *For systems with synchronous co-evolution of production and test code, the ratios of the rule classes do represent the distribution of programmer effort. For these systems the strength distributions of the rule classes are similar. The strength of typical rules is not low.*

5.2.3 System B.I and B.II

With systems A.I, A.II and Checkstyle we have been able to distinguish different patterns of co-evolving production and test code. Systems B.I and B.II have been selected because the systems have very different test patterns, as analysed with the ChangeHistoryView. System B.I is well tested, and shows the characteristics of a synchronous co-evolution. System B.I only has two main periods of testing, and the test effort is located to very specific parts of the system.

The ratios of the rule classes for both systems look almost identical to what we have already encountered. With minor variations, B.I is similar to A.I, and B.II has a huge PROD ratio, very similar to Checkstyle. So are the testing practices also similar? Judging from

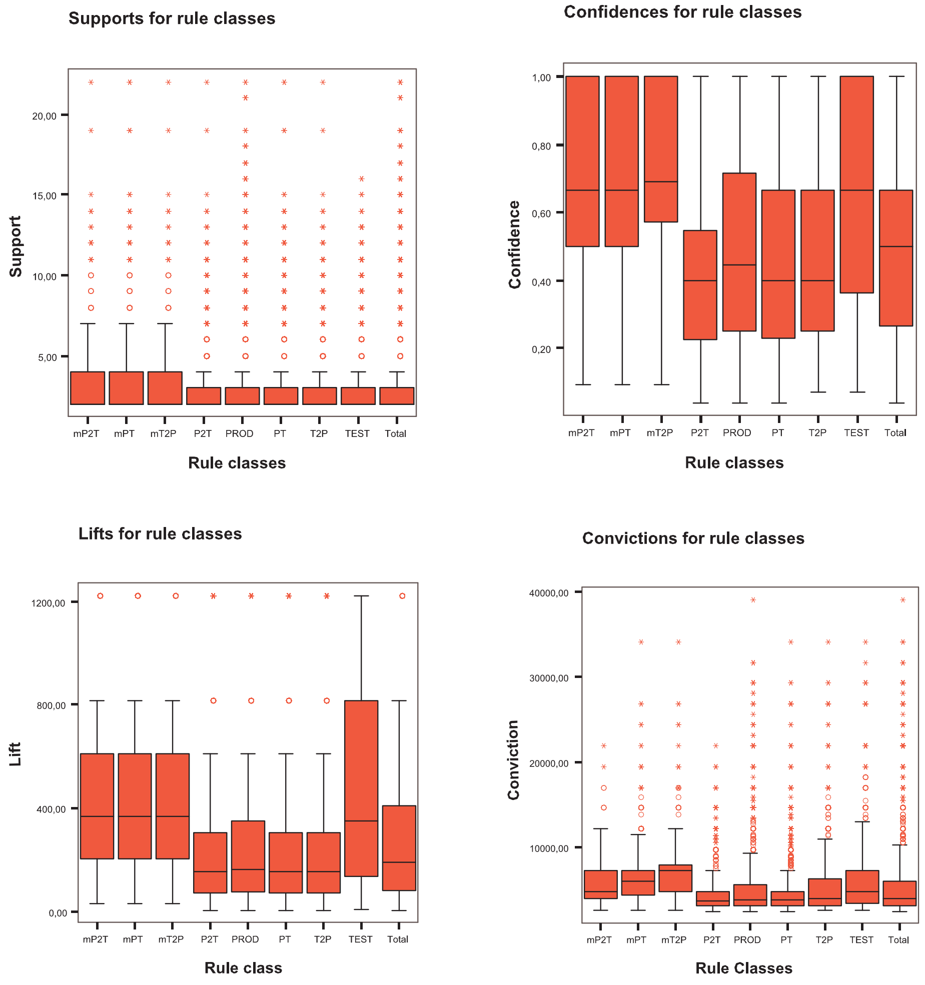


Figure 5.7: A.I rule strengths distributions

the ChangeHistoryViews, B.I is much more sparingly tested than A.I, and Checkstyle is significantly better tested than B.II.

The strength distributions of systems A.I and B.I show similar relations among the rule classes. There is however a striking difference. A.I has very strong TEST rules, and B.I. has strong PROD rules (compared to the other rule classes). For both systems, the supports for these rule classes are not spectacular, so the strong confidence, lift and conviction values for the rules must be the result from not many, but from structural co-occurrences. That is, the entities that occur in combinations of test classes in TEST (for A.I) and production classes in PROD (for B.I) do not occur that often in other combinations. We believe this tells us that the programmers of the systems are working with a strong focus on specific parts on the system or test suite. For example, B.I is a system build from scratch, and appears

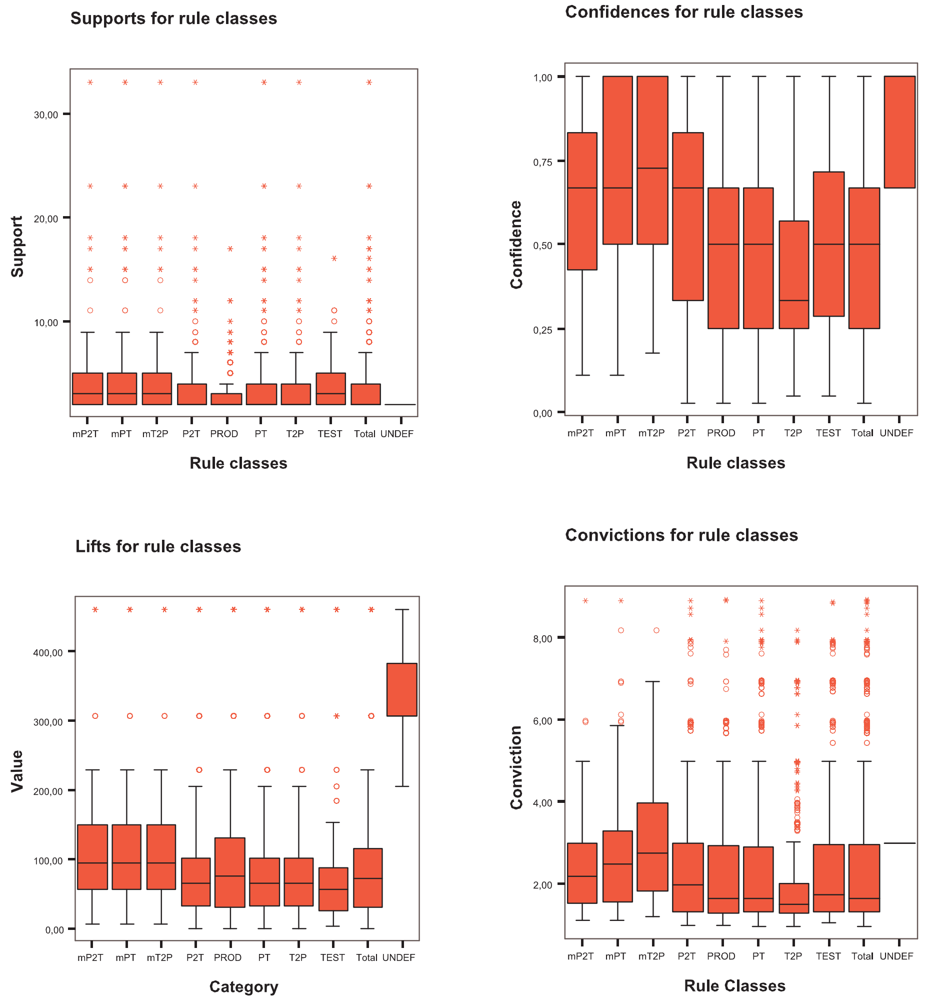


Figure 5.8: A.II rule strengths distributions

to be build incrementally. The rounded growth curve in the ChangeHistoryView supports this idea. Inspection of the commit messages that often modules are build by a specific programmer. That programmer only changes the classes in that specific part of the system, and moves on to another part when finished. Therefor the combinations of classes that are checked in are limited to the specific parts of the system, and the correlation between those classes is high.

For system A.I there is a similar explanation for the strong TEST rules. Developers focus on writing tests for specific parts of the system. The development of system A.I is different than for B.I, because it is a collection of analysis tools that grows and changes over time when analyses for customers require this. Developers are assigned to different customers, so their work on the tools is cross-cutting through the entire system. The causes more combinations of classes to occur, and brings down the correlation between classes, and thus the distribution of strengths of PROD rules. Following from this reasoning, we expect tests to focus on specific parts of the code, as the correlation among tests is high. Company A developers confirm that

Observation 5.5 *High correlations between only production classes (or only test classes) indicates that programmers focus on specific parts of the system (or the test suite).*

When we compare Checkstyle and B.II, because of their similar huge number of PROD rules, we expect to see a difference in testing related associations. Checkstyle adopted a testing strategy over time, but B.II only shows two minor periods of very localised testing in the ChangeHistoryView. The number of rules generated per entity is large for both systems. Both systems show strong rules for TEST and mT2P, indicating many co-usage of tests, and some co-changing tests and production classes. Checkstyle has a slightly higher ratio of TEST rules, so there is more dedicated testing than in B.II. This is in line with the observations from the ChangeHistoryViews. B.II has exceptionally strong PROD rules, which is interpreted as dedicated and localised production code writing effort. Despite these differences, we cannot make a definite distinction between the testing practices of Checkstyle and B.II.

Observation 5.6 *It is hard to determine differences between hardly tested and reasonably, but phases tested systems based on ratios of rules classes and distributions of rule strengths alone.*

5.2.4 System C.I and C.II

Company A systems show similar results to the systems from company A. Just like with the discussion of the ChangeHistoryViews for C.I and C.II, there are only a few noteworthy observations for these systems that have not already been encountered in the other cases. Even the ratios of the number of rules per entity do not differ much. These systems show an impeccable test practice. We will only make a few remarks.

First, system C.I has significantly strong matching rules (mP&T, mP2T and mT2P). For confidence, lift and conviction the interquartile range (IQR) of the boxplots of matching rules is often higher than the IQR of the other rule classes. This means that the upper 75% of the matching rules is stronger, or more correlated, than 75& of the other rule classes. This is strong evidence that production classes and its test code enjoy concurrent attention from developers.

System C.II shows similar distributions, but has stronger co-usage among test classes. We can trace the strong correlations of test classes to larger commits with many tests and classes. The large differences for the boxplots of conviction, related to C.I and A systems, are mainly due to scale differences of the vertical axis.

Observation 5.7 *Cases A.I, A.II, C.I and C.II all reinforce observation 5.4.*

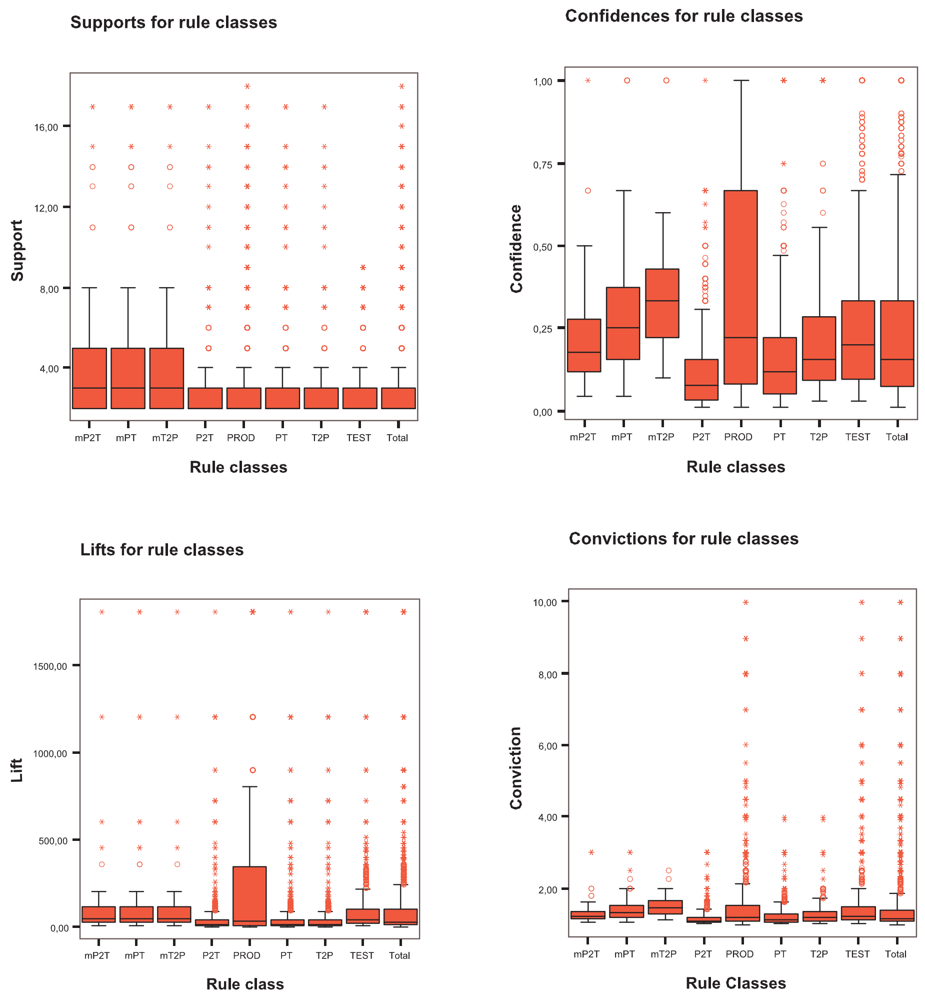


Figure 5.9: B.I rule strengths distributions

Both C.I and C.II have UNDEF rules, but these are not very strong. Inspection of the rules learns us that, again, these rules are generated because of stub and helper classes in the test suite. For system C.II we can see that the support of these rules is quite high, but the strength measures are obviously lower than for other rule class. This means that the combinations of classes with the stubs occur quite often, but with many different classes per stub. It is plausible that a number of different tests use the same stub or helper class. Note that this is a different practice than we encountered in system A.II, where UNDEF rules occurred very few times, but had high strengths.

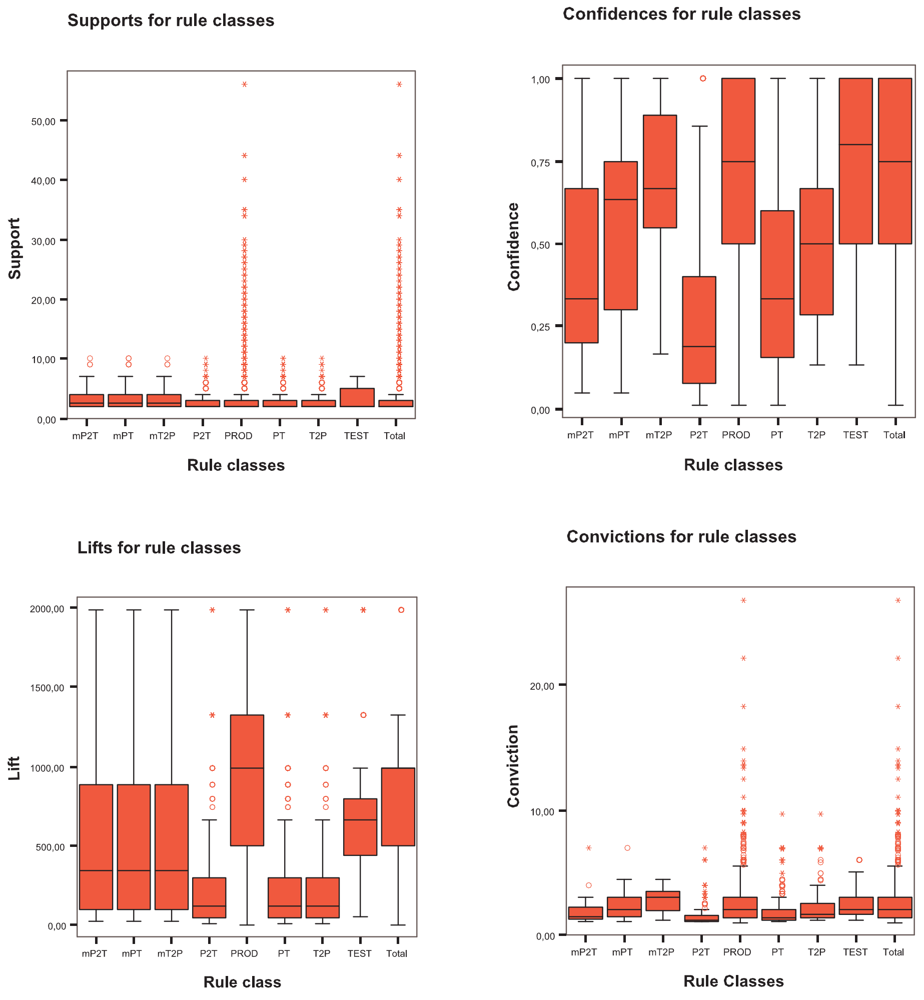


Figure 5.10: B.II rule strengths distributions

5.3 Test Suite Quality Evaluation

The second interpretation of association rules we explore is by looking from a code class point of view. We formulated two variants: code entity classifications based on the number of associations, and the ‘logical coverage’ of associations rules to code entities. We now discuss the results from these attempts.

5.3.1 Code entity classification

The classification of code entities based on the number of P2T or T2P rules they occur in is not proving to be a useful metric. For many of the cases, the majority of the classes is

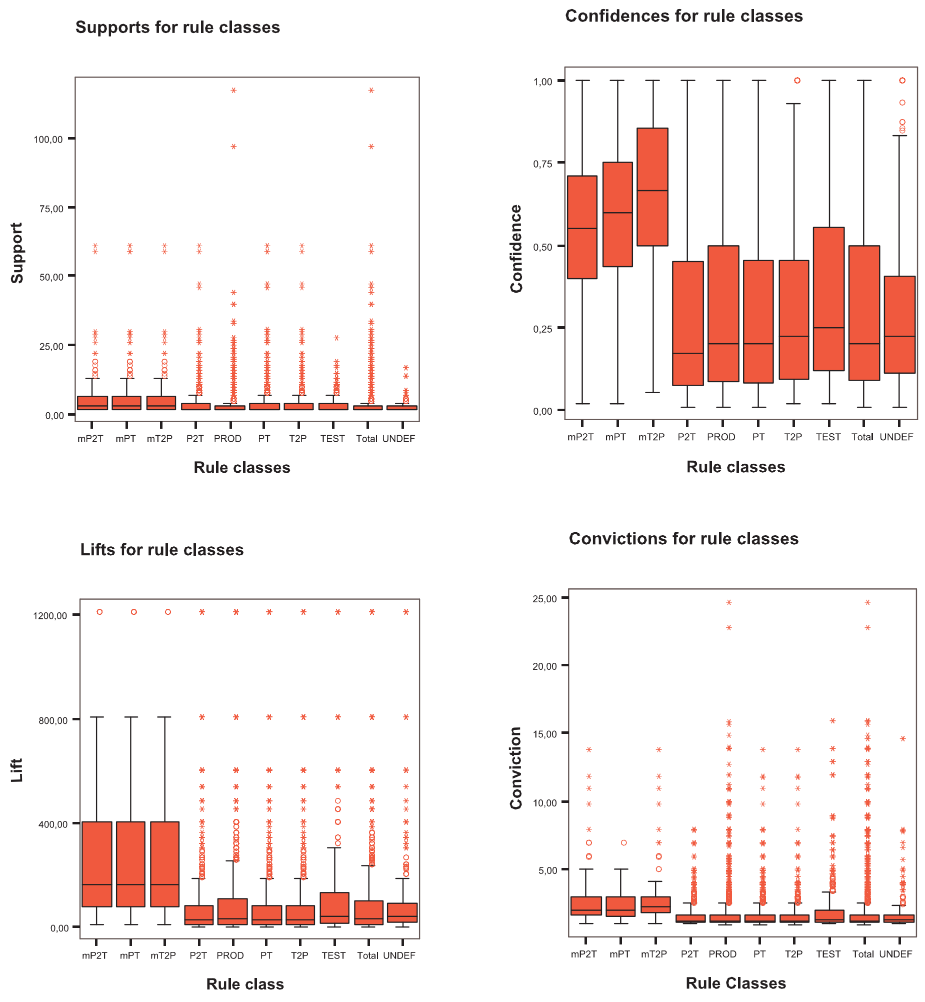


Figure 5.11: C.I rule strengths distributions

not being associated in an interesting way. An example boxplot of the number of distinct rules associating a production or a test class, is shown in figure 5.13. Where the number of occurrences of a class as antecedent in a P&T rules was very high for the JPacman test-case (in chapter 4), associating every class to every test, the number of occurrences of a typical class as antecedent for larger projects is either completely none, or growing very large. The number of classes that get associated with a sensible number of other classes that resembles the actual structural coupling is low (around 10% of the total number of classes). One would expect a few number of occurrences for tests: once, twice, four times. A test that is structurally testing more than a few classes is just not a good practice, so the numbers obtained here are completely not representative. We conclude that it is not feasible to make

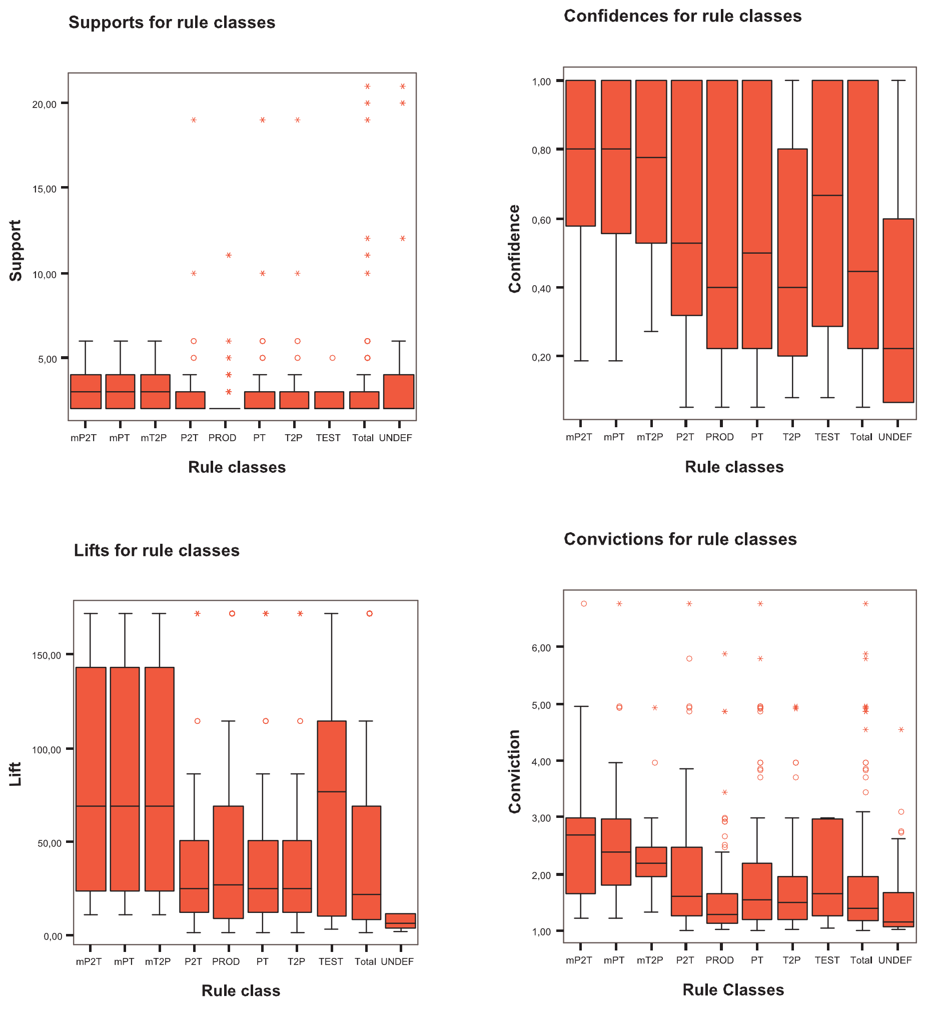


Figure 5.12: C.II rule strengths distributions

any useful analysis based on this interpretation of association rules.

Observation 5.8 *Classification of classes and tests cannot be done based on their usage by programmers.*

5.3.2 Rule coverage

Another take on test quality is done by use of the rule coverage of classes. In section 4.1.3 we define four types of coverage measures based on the ratio of the number of (matching) rules and the number of classes and tests in a system. We compute the four ratios for all the cases, these can be found in table 5.4.

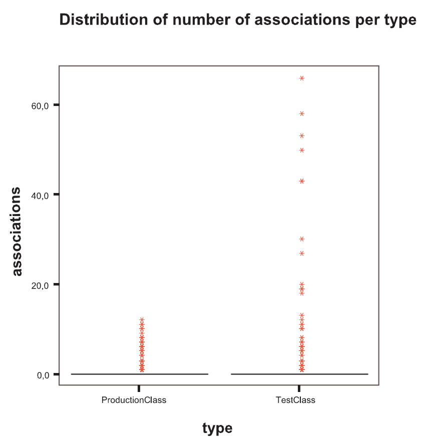


Figure 5.13: Example of the number of occurrences per class type.

We already made the observation that strong rules for matching production and test class pairs originate from a disciplined test-driven testing strategy. The production and test matching coverage percentages can actually give a more relevant interpretation to the strengths, because together they express (1) how many production and test class pairs are co-used, and (2) how structural the co-usage is.

We immediately see a clear separation: the system that showed test-driven testing practices, and Checkstyle and B.II. The first category of systems show large (20%, 30%, 50%) parts of the test suite that potentially synchronous co-evolve. We can check the mT2P rule strengths to see to what extent they actually co-evolve. In the previous section we already discussed these strengths and saw that the relations between test and production code is high for these systems.

For the other two cases, the matching coverage ratios give slightly more insight in a potential difference between the testing practice. Both have an equal part of their test suite that potentially co-evolves (nearly 10%), but only about 1,5% of the production classes in B.II have their tests updated when changes are made to the class, against 11% in Checkstyle. So Checkstyle has a 7 times larger part of production code that has maintained tests than B.II.

5.4 Evaluation

In this chapter we evaluated our proposed approach to study the co-evolution of production and test code in a system using several case studies. We have observed that based on assocation rules we can express how developers divide their efforts over production code writing and testing activities, and to what extent parts of the system or the test suite are co-evolving.

While we can also detect the absence of synchronous co-evolution in a system, it proves to be hard to determine whether the system is poorly tested. A large ratio of strong TEST

Case	**Production class mapping ratio**	**Production class matching coverage**	**Test class mapping ratio**	**Test class matching coverage**
Checkstyle	0,42	11,04%	0,38	9,83%
A.I	7,96	32,22%	11,79	47,70%
A.II	7,83	28,80%	17,27	63,53%
B.I	2,17	9,91%	4,34	19,84%
B.II	0,35	1,49%	2,08	8,84%
C.I	4,27	20,42%	11,57	55,35%
C.II	0,66	19,67%	0,99	29,63%

Table 5.4: Rule coverage ratios of classes for the case studies.

rules would indicate a phased testing approach, but the two cases we considered where this would possibly occur, huge amounts of PROD rules, generated by large commits, cluttered the view. Valid detection of structural phased testing requires futher investigation.

Chapter 6

Conclusions and Future Work

This chapter gives an overview of the project's results and contributions.

6.1 Conclusions

The central research question of this thesis was "How can data mining techniques be applied to (retrospectively) study the unit testing process in software systems?". We formulated four supplemental research questions, and we will provide answers to these questions here.

RQ1: We applied association rule mining to the change history of software systems, and presented an approach to analyse the co-usage between production and test code based on the mined association rules. Using several case studies we can conclude that association rule mining can be successfully applied to find and express synchronous co-evolution of production and test code.

RQ2: Our attempt to classify test classes based on their co-change with production classes leads us to conclude that this does not lead to satisfactory results. The occurrences of test classes in the change history is generally to wide spread to make any sensible statements.

RQ3: Using the association rules we can determine what percentage of the test suite is actually used as a unit test, and the distribution of rule strengths can be used to express how structural the production and test code is synchronous co-evolving. We believe that this can be applied to measure (long term) unit test quality

RQ4: While we have discussed five industrial case studies that show similar testing practices, we have also observed one industrial case and one open source software case that show radically different patterns. In our complete body of industrial systems (not discussed in this thesis), we observed different patterns between, and even inside, several companies.

To return to the central research question, we found that the application of association rules to the change history of a software system can very well be used to study the unit testing process of the system. We believe that there is a lot of potential in this type of analysis, as it can truly express the way in which classes and tests are used by developers.

In the introduction of chapter 1 we identified to potential uses for this type of analysis. The analyses techniques that we explored in this work prove to be useful for both (retrospective) assessment of the unit test suite and the way testing is employed. We did not investigate whether monitoring of the testing process is a possibility. We expect that changes to the test practice over small periods of time will not yield noticeable differences in the results, as our technique summarises the entire history. Changes to longer histories will not influence the complete picture enough to make useful judgements.

6.2 Contributions

In this thesis we have made the following contributions.

- We have developed a tool to mine association rules at file level from the change history of a system, based on the log data of the system's version control system. This is a more lightweight approach to find co-usage among classes than using static code analyses at class level. For our purpose, analysis at file level is fast and provides enough precision compared to code level.
- We have presented an approach to study the co-evolution of production and test code in a system based on the co-usage of classes and unit tests.
- We have collected a large number of change histories of industrial systems to study testing processes in real world situations.

6.3 Future work

We have identified some ideas to build upon this explorative research.

Refine and operationalise the measurements presented in this work: We would like to refine the interpretation of the measurements that we presented. To make the results easier to understand for more people, we would like to be able to express the data in terms like "Programmers spent *x* amount of time to testing" or "*y*% of the system exhibits a synchronous co-evolution of production and test code". Using this simpler interpretation, we would like to build a benchmark of systems and be able to formule 'quality-standards'. This allows for better comparison among systems.

Use of more detailed data: By adding more information from other sources the level of detail of the rule classes can be enhanced to get more detailed data on what is changing in the system. We could add more VCS log data, link to information from bug tracking systems, or perform static source code analysis. By statically analysing the actual classes we can determine in more detail what class is a unit test or an integration test, or analyse if more complex classes are tested better or worse than an average class.

Compare unit testing practice differences in distinct development cultures: We have studied the change histories of a large number of industrial systems and reported on six of them. The previous study by Zaidman et al. [33], and our OSS case study show that open source systems have less disciplined and more phased testing practices compared to the majority of the industrial cases we have seen. We would like to study how unit testing is employed in different cultures of software development.

Bibliography

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, DC, USA, May 26-28 1993. ACM Press.

[2] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of 20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499, Santiago de Chile, Chile, September 12-15 1994. Morgan Kaufmann.

[3] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. What's a typical commit? a characterization of open source software repositories. In *Proceedings of the 16th International Conference on Program Comprehension*, pages 182–191. IEEE Computer Society Press, June 2008.

[4] Thomas Ball, Jung-Min Kim, Adam A. Porter, and Harvey P. Siy. If your version control system could talk. In *ICSE '97 Workshop on Process Modelling and Empirical Studies of Software Engineering*, May 1997.

[5] Kent Beck. *Test-Driven Development: By Example*. Number 0321146530. Addison-Wesley Longman Publishing Co., Boston, MA, USA, 2002.

[6] Dirk Beyer and Ahmed E. Hassan. Animated visualization of software history using evolution storyboards. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, volume 00, pages 199–210. IEEE Computer Society Press, october 2006.

[7] Dirk Beyer and Ahmed E. Hassan. Evolution storyboards: Visualization of software structure dynamics. In *Proceedings of the 14th International Conference on Program Comprehension (ICPC 2006)*, pages 248–251. IEEE Computer Society Press, 2006.

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

[9] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Shalom Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, volume 26,2 of *SIGMOD Record*, pages 255–264. ACM Press, May 1997.

[10] Marco D'Ambros, Michele Lanza, and Mircea Lungu. The evolution radar: Visualizing integrated logical coupling information. In *Third International Workshop on Mining Software Repositories (MSR)*, pages 26–32. ACM Press, May 2006.

[11] Margaret H. Dunham. *Data Mining: Introductory and Advanced Topics*. Number 0-13-088892-3. Prentice Hall, August 2002.

[12] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[13] Beat Fluri, Michael Würsch, and Harald C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE)*, pages 70–79. IEEE Computer Society Press, 2007.

[14] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, pages 190–197. IEEE Computer Society Press, November 1998.

[15] Harald Gall, Mehdi Jazayeri, René R. Klösch, and Georg Trausmuth. Software evolution observations based on product release history. In *Proceedings of the International Conference on Software Maintenance*, page 160. IEEE Computer Society Press, 1997.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

[18] Tudor Gîrba, Stéphane Ducasse, and Michele Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 40–49. IEEE Computer Society Press, 2004.

[19] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *Proceedings of International Conference on Software Maintenance (ICSM 2000)*, pages 131–142. IEEE Computer Society Press, 2000.

[20] Abram Hindle, Michael W. Godfrey, and Richard C. Holt. Release pattern discovery via partitioning: Methodology and case study. In *Fourth International Workshop on Mining Software Repositories (MSR)*, pages 19–26. IEEE Computer Society Press, 2007.

[21] Michele Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the 4th International Workshop on the Principles of Software Evolution (IWPSE 2001)*, pages 37–42. ACM Press, 2001.

[22] Michele Lanza and Stéphane Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(6):782–795, September 2003.

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

[24] Bennet P. Lientz and E. Burton Swanson. *Software Maintenance Management*. Addison-Wesley, 1980.

[25] Bing Liu, Wynne Hsu, and Yiming Ma. Mining association rules with multiple minimum supports. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 337–341. ACM Press, 1999.

[26] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *IWPSE 2005: Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society Press.

[27] John Mylopoulos and Thomas Rose. Software repositories, tutorial 4. In Li-Yan Yuan, editor, *18th International Conference on Very Large Data Bases*, page 455. Morgan Kaufmann, 1992.

[28] David Lorge Parnas. Software aging. In *Proceedings of the 16th International Conference On Software Engineering (ICSM)*, pages 279–287. IEEE Computer Society Press, May 1994.

[29] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 67–75. ACM, 2005.

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

[31] Margaret-Anne D. Storey, Davor Čubranić, and Daniel M. German. On the use of visualization to support awareness of human activities in software development: A survey and framework. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 193–202, New York, NY, USA, 2005. ACM.

[32] Zhenchang Xing and Eleni 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.

[33] Andy Zaidman, Bart van Rompaey, Serge Demeyer, and Arie 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), page Accepted for Publication. IEEE Computer Society Press, 2008.

[34] Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, pages 563–572. IEEE Computer Society Press, 2004.