Constructing a Test Code Quality Model and Empirically Assessing its Relation to Issue Handling Performance

Master's Thesis

Dimitrios Athanasiou

Constructing a Test Code Quality Model and Empirically Assessing its Relation to Issue Handling Performance

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Dimitrios Athanasiou
born in Thessaloniki, Greece

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

Software Improvement Group
Amstelplein 1 - 1096 HA
Amsterdam, the Netherlands
www.sig.eu

In the context of this thesis, participation of the author in the PASED summer school in Montreál, Canada, was partly funded by Universiteitsfonds Delft.

Constructing a Test Code Quality Model and Empirically Assessing its Relation to Issue Handling Performance

Author: Dimitrios Athanasiou
Student id: 4032020
Email: dmitri.athanasiou@gmail.com

Abstract

Automated testing is a basic principle of agile development. Its benefits include early defect detection, defect cause localization and removal of fear to apply changes in the code. Therefore, maintaining high quality test code is essential. This study introduces a model that assesses test code quality by combining source code metrics that reflect three main aspects of test code quality: completeness, effectiveness and maintainability. The model is inspired by the SIG Software Quality model which aggregates source code metrics into quality ratings based on benchmarking. To validate the model we assess the relation between test code quality, as measured by the model, and issue handling performance. An experiment is conducted in which the test code quality model is applied on 18 open source systems. The correlation is tested between the ratings of test code quality and issue handling indicators, which are obtained by mining issue repositories. The results indicate a significant positive correlation between test code quality and issue handling performance. Furthermore, three case studies are performed on commercial systems and the model's outcome is compared to experts' evaluations.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. A. Zaidman, Faculty EEMCS, TU Delft
Company supervisor:	Dr. Ir. J. Visser, Software Improvement Group B.V.
Company co-supervisor:	Dr. A. Nugroho, Software Improvement Group B.V.
Committee Member:	Dr. Ir. P. Wiggers, Faculty EEMCS, TU Delft

To my parents, whose endless support brings my dreams closer...

Preface

When I started working on this thesis I could not imagine the impact it would have on my way of thinking. My daily presence in a challenging environment expanded my horizon significantly. What made the process even more worthy was the fact that I enjoyed every minute of it. I owe this to several people and the least I can do is to express my gratitude through the following lines.

First of all, I would like to thank Dr. Andy Zaidman and Dr. Ir. Joost Visser. They have both been ideal mentors, always available to discuss with me all my concerns and providing invaluable feedback on my work. The third person who I would like to thank especially is Dr. Ariadi Nugroho. Ariadi worked close with me and he provided me with answers whenever I needed them, particularly about research methodology and statistics. All three of them spent a considerable amount of time proofreading my thesis and helping me to convey my work in the best way.

This thesis project was performed at the Software Improvement Group (SIG). I would like to thank all the people there for doing everything they could to make me feel welcome. Special thanks go to Miguel, Zé Pedro, Xander, Tiago and the rest of the research team, for the innumerable discussions we had about my work and software engineering in general.

Part of the work in this thesis would not be possible without the contributions of the experts that were interviewed while performing a series of case studies. Moreover, during the project I had the opportunity to participate in the PASED summer school in Montreál, Canada. I would like to thank SIG and Universiteitsfonds Delft, for their sponsorship made this possible.

I would also like to thank my firends Nassos, Vitto, Momchil and Stanislava for patiently listening to my mumbling about correlations, metrics and coding, sometimes even against their will. Last but not least, I would like to thank my parents, Pantelis and Vaso, for without their endless support all this would have never happened.

Dimitrios Athanasiou
Delft, the Netherlands
September 1, 2011

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter defines and motivates the problem in question of this study, formulates the research questions and outlines the approach followed to provide the corresponding answers.

1.1 Problem Statement

Software testing is well established as an essential part of the software development process and as a quality assurance technique widely used in industry [14]. Developer testing (a developer test is "a codified unit or integration test written by developers" [95]) in particular, has risen to be an efficient method to detect defects in a software system early in the development process. In the form of unit testing, its popularity has been increasing as more programming languages are supported by testing frameworks (e.g. JUnit for Java, NUnit for C#, etc.). A significant research effort is put on various aspects of unit testing. Literature suggests that 30 to 50% of a project's effort is consumed by testing [28].

One of the main goals of testing is the successful detection of defects. Developer testing adds to this the ability to point out *where* the defect occurs [59]. The extent to which detection of the cause of defects is possible depends on the quality of the test suite. In addition, Kent Beck in his book "Test-Driven Development" [10] explains how developer testing can be used to increase confidence in applying changes to the code without causing parts of the system to break. This extends the benefits of testing to include faster implementation of new features. Consequently, it is reasonable to expect that there is a relation between the quality of the test code of a software system and the developer team's performance in fixing defects and implementing new features.

In order to assess the aforementioned relation, we first need to be able to assess test code quality. The assessment of test code quality is an open challenge in the field [14]. Monitoring the quality of a system's test code can provide valuable feedback to the developers' effort to maintain high quality assurance standards. Several test adequacy criteria have been suggested for this purpose. Zhu et al. summarize a rich set of test adequacy criteria and compare them [98]. The applicability of some of these criteria is limited since, for instance, some of them are computationally too expensive. A combination of criteria that provides a

model for measuring test code quality is desirable and the target of exploration within the scope of this thesis.

After developing a way of measuring test code quality, it is necessary to quantify a team's performance in fixing defects and implementing new features. Defects and feature requests, among others, are considered as *issues*. The process of analysing the issues, assessing their validity, managing when and by whom they should be resolved is called *issue handling*. To facilitate this process, special systems have been developed in order to enable developers to track the issues of the system they build. These systems are called *Issue Tracking Systems (ITSs)* and they contain lists with the issues and the information that forms the issues' life-cycle as we will see in Section 2.2. The use of ITSs in order to track issues and coordinate the effort of the developers has become a standard in a modern software development environment.

In order to measure the performance of the issue handling process of software development teams, ITSs can be mined. We expect that defect resolution time for a software system is reflected in its associated ITS as previous work suggests [51, 88, 30, 45, 4]. In addition, further indicators of issue handling performance, such as throughput and productivity, can be derived by studying ITSs data as shown in [16].

Intuitively, issues in software systems that are accompanied by high quality test suites are resolved more quickly compared to systems of lower quality test suites. The reasoning behind this is that when a high quality test suite is in place, it is easier to apply changes without worrying about affecting parts of the code that are already correct. Moreover, teams working on systems with high test code quality are able to resolve more issues than teams working on systems with lower test code quality. The purpose of this thesis is to investigate and assess the existence of such relations in order to provide empirical evidence of the value of testing and of the proposed test code quality model.

1.2 Research Questions

This study attempts to define a test code quality model. The existing test adequacy criteria are explored towards the goal of selecting among them a combination that provides a feasible, applicable quality indicating model for assessing developer tests' effectiveness. This aims at achieving the main goal of the study, namely the construction of a model that reflects the quality of the main aspects of test code. Furthermore, the study attempts to provide validation of the usefulness of the proposed test code quality model as an indicator of issue handling performance.

Our research is driven by the following research questions:

- **RQ1** : How can we evaluate the quality of test code?
- **RQ2** : How effective is the developed test code quality model as an indicator of issue handling performance?
- **RQ3** : How useful is the test code quality model?

In order to answer these questions, it is necessary to refine them by breaking them down to subsidiary questions. To answer RQ1 we need to answer the following questions:

- **RQ1.1** : What makes a codified test suite effective?
- **RQ1.2** : Can we define a set of feasible and applicable metrics that reflects the quality of codified testing?

To answer RQ2, hypotheses are formulated about the benefits of automated testing in the issue handling process. In particular, we expect that compared to systems of lower test code quality, in systems of higher test code quality:

- defect resolution times are shorter because of the ability of the test suite to trace the root of defects.
- throughput[1] and productivity[2] of resolved issues are higher because automated tests remove the fear of change and at the same time, test code serves as documentation that facilitates the comprehension of the code to be modified.

Therefore, we can formulate the following research questions:

- **RQ2.1** : Is there a correlation between the test code quality ratings and the defect resolution time?
- **RQ2.2** : Is there a correlation between the test code quality ratings and the throughput of issue handling?
- **RQ2.3** : Is there a correlation between the test code quality ratings and the productivity of issue handling?

Finally, in order to assess the usefulness of applying the test code quality model, its strengths and its limitations, the study of the results of applying the model to particular cases and the comparison of the model's ratings with the opinion of an expert would answer the question:

- **RQ3.1** : How is the test code quality model aligned to experts' assessment of particular systems?

The answers to these questions will provide a deeper understanding of codified testing and its importance in software engineering. Results can reveal an insight about unit test economics and add empirical evidence to the understanding of the impact of test quality to software engineering.

1.3 Approach

The first step towards addressing the research questions regarding the construction of the test code quality model (RQ1.1, RQ1.2) is the exploration of the test adequacy criteria proposed in the literature. Next, we use the GQM approach [91] to define metrics that measure

[1] number of resolved issues in a time period for the whole project
[2] number of resolved issues in a time period per developer

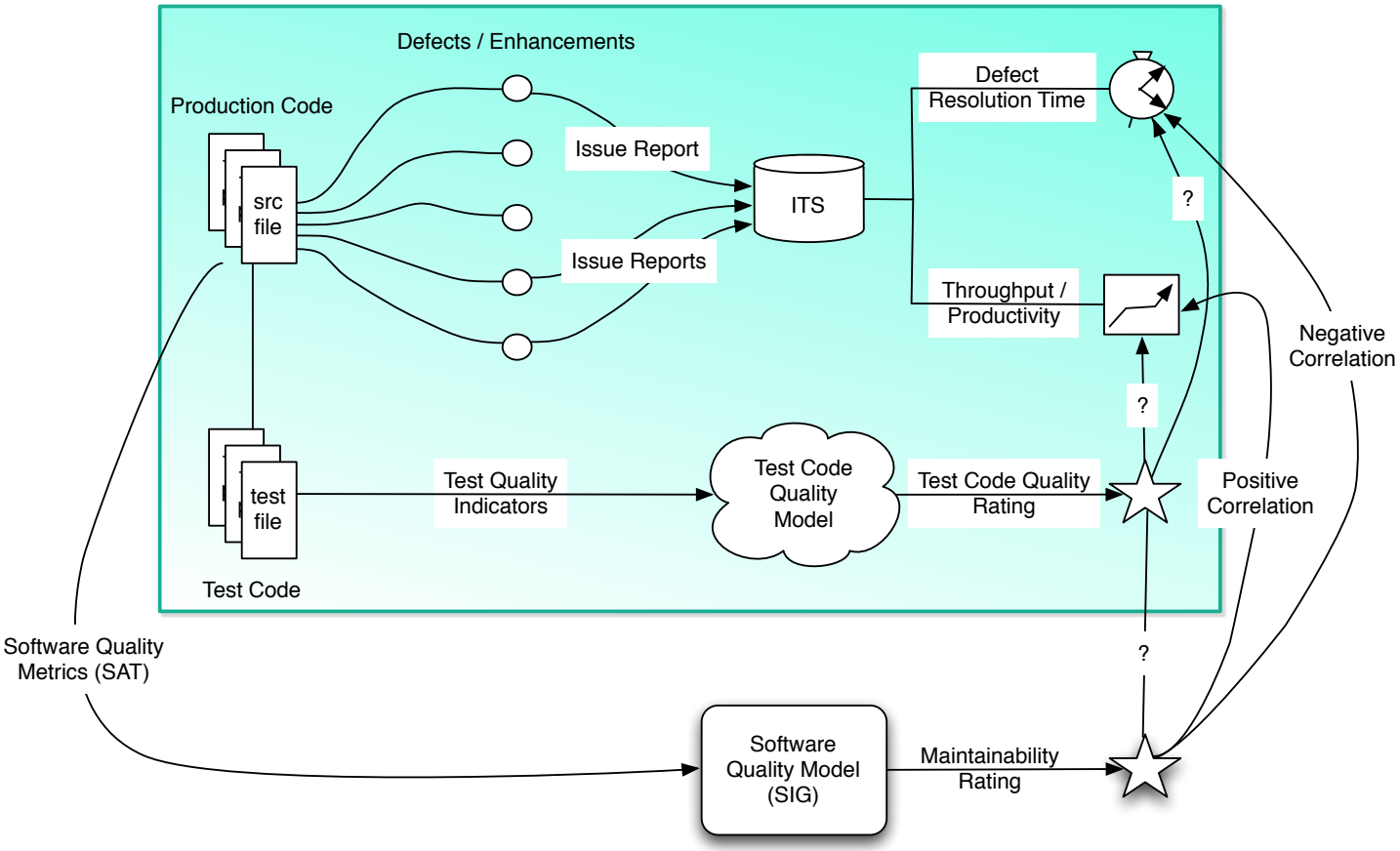

Figure 1.1: Research Context

test code quality. We then build the model, inspired from SIG's software maintainability model [34, 72, 8]. SIG has developed a software assessment model that aggregates software quality metrics in order to rate software's maintainability according to the maintainability characteristics as they are defined in the ISO 9126 [1] software quality standard.

The approach to provide answers to the rest of the research questions is empirical in nature. An empirical study is simply "just a test that compares what we believe to what we observe" [75]. In the context of this study, we seek to study the relation between test code quality and issue handling performance (RQ2.1, RQ2.2, RQ2.3). As explained earlier, the benefits of automated testing, such as the ability to localise the root cause of a defect and the removal of the fear of change, lead to the expectation that higher test code quality results in better issue handling performance. In this study, we conduct an experiment in order to test the aforementioned expectation against empirical data from several open source projects.

An overview of the research context of the experiment in this study is shown in Figure 1.1. The coloured frame focuses on the objective of this study while what is out of the frame is related work performed by Luijten et al. [51] and Bijlsma [16] that forms the basis for

this study.

Production code is the actual source code that implements the system. Test code contains test cases, which represent possible execution scenarios, exercises these cases on the production code and asserts whether the behaviour is as expected. When tests fail, which means a defect occurred, they should detect where the cause of the defect lies. During the life-cycle of the system various defects are detected either automatically or manually. In addition, new features are requested. Some of these defects and feature requests are reported as issues to the ITS.

Luijten et al. [51] studied and established the correlation between software maintainability and defect resolution time. Software maintainability was determined based on SIG's software quality model. SIG's SAT (Software Assessment Tool) implements and applies software metrics on a software system in order to provide a rating for the system's maintainability. Defect resolution time was acquired by applying data mining to the ITS of a system. A negative correlation was found, a fact which suggests that defects are resolved more efficiently in systems with higher maintainability.

Bijlsma [16] extended the study of Luijten et al. by studying the correlation between software maintainability and enhancement resolution time. Furthermore, he proposes three more issue handling performance indicators, namely throughput, productivity and efficiency. In his study, an experiment is conducted demonstrating significant correlation between software maintainability and these three indicators.

This study focuses on the impact of test code quality to the issue handling performance. A Test Quality Model is proposed as a set of test effectiveness indicators that can be applied on the test code of a software system in order to obtain a rating of its quality. The primary goal of the study is to discover the relation between the quality of the test code and defect resolution time, throughput and productivity. An opportunity to address a secondary research question exists: *Is there a relation between the maintainability rating and the test quality rating?*

Last, but not least, we address RQ3.1 using the case study methodology [80]. In particular, we apply the test code quality model on certain industry systems. Finally, we interview experts that have studied the system's technical quality in order to explore whether their opinion is aligned to the information that is derived from the model's results.

1.4 Outline

The thesis is structured as follows. Chapter 2 discusses background information as well as related work. In Chapter 3 we illustrate the approach we followed in order to develop a test code quality model. Chapter 4 discusses the design of the study which is split into an experiment and a series of case studies. Chapter 5 presents and discusses the results of the experiment with which we attempt to assess the relation between test code quality and issue handling performance. Chapter 6 presents the results of three case studies which illustrate the usefulness of the application of the test code quality model. Finally, Chapter 7 summarises the findings of the study, lists the contributions of the study and discusses topics for future research.

Chapter 2

Background and Related Work

Providing answers to the study's research questions requires knowledge foundations on the topics involved, namely: test code quality, issue handling and the SIG software maintainability model. This chapter summarizes the existing literature related to these topics. In addition, related work that is not covered by these topics is discussed in the final section of the chapter.

2.1 Test Code Quality

What makes a good test? How can we measure the quality of a test suite? Which are the indicators of the test effectiveness? Answers to these questions have been sought by software and reliability engineering researchers for decades. However, defining test effectiveness remains an open challenge in the field [14]. Zhu et al. [98] provide a deep look into existing test adequacy criteria up to 1997. Based on that work and complementing it with more recent research work, a taxonomy of the test adequacy criteria will be constructed in order to provide a clear picture of the state of the art in test effectiveness assessment.

The main role of test adequacy criteria is to assist software testers to monitor the quality of the software in a better way by ensuring that sufficient testing is performed. In addition, redundant and unnecessary tests are avoided, thus contributing in controlling the cost of testing [98, 89].

Various classifications can be applied to test adequacy criteria. The classification schemes proposed by Zhu et al. [98] are adopted in this study. First, we can distinguish *program-based* criteria, which assess testing of the production code, and *specification-based* criteria, which assess testing of the specifications of a software project. Specifications testing is not in the scope of this study because it depends on specification languages while we aim at assessing the quality of test code. Thus, we focus on program-based test adequacy criteria. Furthermore, according to the testing approach, criteria can be classified into categories for:

- **structural testing** : test criteria that focus on measuring the coverage of the test suite upon the structural elements of the program.

- **fault-based testing** : test criteria that focus on measuring the defect detecting ability of a test suite.
- **error-based testing** : test criteria that focus on measuring to what extent the error-prone points of a program (as derived from the current knowledge level) are tested.

After presenting the criteria that fall into the above categories, other techniques (e.g. assertions, test smells, etc.) that can be used to measure test code quality will be discussed.

2.1.1 Structural Testing Adequacy Criteria

Criteria that fall in this class can be further split between control-flow criteria and data-flow criteria. They are mostly based on analysis of the flow graph model of program structure. Control-flow criteria are concerned with increasing the coverage of the elements of the graph as much as possible. Different criteria assess coverage in a different scope: statement coverage, branch coverage or path coverage. Data-flow criteria are concerned with analysing whether paths associating definitions of variables to their uses are tested. Finally, we will see efforts that combine both of the aforementioned criteria.

Control-flow Test Adequacy Criteria

A synopsis of the control-flow adequacy criteria is shown in Table 2.1.

Hetzel [36] formalizes basic criteria that concern coverage. The simplest criterion is called *statement coverage* and it requires every statement to be exercised by the test code. Nevertheless, this is frequently impossible because of the existence of dead code. A stronger criterion is *branch coverage*. The requirement for branch coverage is that the tests must exercise all control transfers. Branch coverage subsumes statement coverage since if all control transfers are exercised, all statements are exercised as well. In order to achieve complete coverage of all possible combinations of control transfers, we have to assess if the tests exercise every possible execution path in the flow graph of the system. This is the *path coverage* criterion. However, infinite test code has to be written to achieve path coverage and therefore, it is not finitely applicable.

In fact, neither the statement coverage nor branch coverage criteria can be applied finitely. As it was already mentioned, there may be dead statements or dead branches, meaning code that cannot be executed. However, these criteria can be restated so that they become feasible by only requiring the coverage of feasible code. Still, it is possible that they will not be finitely applicable. For example, it is not always possible to decide if a piece of code is dead or not. In order to work around such problems, criteria with specified restrictions on the selection of execution paths to be considered under testing have been defined. The *elementary path coverage* criterion requires that only elementary paths, paths where no node occurs more than once, have to be covered. Similarly, the *simple path coverage* criterion requires that only simple paths, paths where no edge occurs more than once, have to be covered [98].

Some more flexible criteria towards the same direction exist. Gourlay's [31] *length-n path coverage* criterion requires that all subpaths of length at most n should be covered.

Note that the length of a path is measured by counting the nodes of the flow graph which are covered by the path. Any node in the flow graph contains a single statement. Paige [73, 74] proposed the *level-i path coverage* criterion. According to this criterion, at level 0 all the elementary paths from the start to the end of the program's flow graph are checked against whether they are covered. As the level i increases, the remaining, uncovered, elementary subpaths whose entry and exit nodes belong in the paths covered at level $i-1$ are tested.

In order to provide finitely applicable approaches that handle loops, corresponding test adequacy criteria were sought. Bently et al. [13] proposed a set of *loop count* criteria. Howden's [37] *loop count-K* criterion requires that the execution of every loop takes place $0, 1, 2, ..., K$ times. Moreover, there is the *cycle combination* criterion [98] according to which all execution paths that do not contain a cycle more than once should be exercised.

The well known McCabe cyclomatic complexity metric [56] was used to derive the *cyclomatic-number* criterion [87]. McCabe's complexity metric counts the linearly independent paths and consequently, the cyclomatic-number criterion requires that the test suite covers all of these independent execution paths. McCabe's cyclomatic complexity is also used as a rule of thumb to indicate a lower bound for the number of test cases necessary to cover a piece of code.

A set of criteria that are based on textual analysis follow. In particular, Myers [63] proposed a series of three criteria that focus on condition exploration in conditional or loop structures. *Decision coverage* criterion requires the existence of at least one test that satisfies a given condition and at least one test that falsifies it. *Condition coverage* criterion requires that test cases exist for both truth values of each atomic predicate of a condition, where atomic predicates are the lowest level conditions that form a higher level condition with the use of logical operators. Finally, *multiple condition coverage* criterion, the strongest among the three, demands that all possible combinations of the atomic predicates of every condition are tested.

Control-flow adequacy criteria are concluded by presenting the *linear code sequence and jump (LCSAJ) coverage* criterion proposed by Woodward et al. [93]. According to this criterion, test cases are modified execution orders of the production code in the sense that a code block is executed and then a jump is performed. Criteria that take advantage of textual analysis of the code are easy to calculate. Nevertheless, they depend on language details which limits their application.

Based on the aforementioned criteria, metrics can be derived in order to control the quality of testing code. Though the criteria are defined as requirements, the fulfilment of the requirements can be measured (e.g. in percentage), thus providing an indicator of test code quality. All of the aforementioned criteria are applicable only after redefining them to require the coverage of feasible elements, except from the path coverage criteria which is not applicable.

Data-flow Test Adequacy Criteria

A synopsis of the data-flow adequacy criteria is shown in Table 2.2.

Before the criteria are presented, some terminology should be discussed. Data-flow adequacy criteria focus on the analysis of the associations between definitions of variables and

Table 2.1: Control-flow Test Adequacy Criteria

Criterion	Proposed by	Description
Statement Coverage	Hetzel [36]	All statements in the program are exercised by testing.
Branch Coverage	Hetzel [36]	All control transfers are exercised.
Path Coverage	Hetzel [36]	All combinations of branches are exercised.
Simple Path Coverage	Zhu et al. [98]	All simple paths are exercised.
Elementary Path Coverage	Zhu et al. [98]	All elementary paths are exercised.
Length-n Path Coverage	Gourlay [31]	All subpaths of length $\leq n$ are exercised.
Level-i Path Coverage	Paige [73, 74]	At level 0 all elementary paths from the start to the end are exercised. Level increases up to i and in each level subpaths whose entry and start nodes were in level $i-1$ path set, but not other nodes or edges, are exercised.
Loop Count	Bently et al. [13]	A set of criteria dealing with loops.
Loop Count-K	Howden [37]	Every loop is executed $0,1,2,...,K$ times.
Cycle Combination	Zhu et al. [98]	All execution paths that do not contain a cycle more than once are exercised.
Cyclomatic-Number	McCabe et al. [87]	Test cases exercise at least as many paths as the cyclomatic complexity of the corresponding piece of code.
Decision Coverage	Myers [63]	For every condition there is at least one test case that satisfies it and one that falsifies it.
Condition Coverage	Myers [63]	For every atomic predicate of every condition there is at least one test case that satisfies it and one that falsifies it.
Multiple Condition Coverage	Myers [63]	Test suite covers all combinations of the truth values of atomic predicates of all conditions.
Linear Code Sequence And Jump (LCSAJ)	Woodward et al. [93]	Measuring different levels of coverage by applying tests that execute a block and then a jump.

the uses of these variables. A definition is a statement in the code that sets a new value in a variable. Uses can be either computational, when a variable is used in a mathematical computation of some kind, or predicate, when the content of the variable is used in a condition that is checked to define the control-flow of the program.

Frankl and Weyuker [29] redefined a set of criteria initially proposed by Rapps and Weyuker [78] in order to make them applicable. The *all definitions* criterion requires the test suite to exercise at least one subpath connecting each definition of a variable to one of its uses. A particular definition of a variable may be used more than once. The *all uses* criterion, also proposed by Herman [35] as the *reach-coverage* criterion, demands that subpaths between each definition of a variable to all of its uses are exercised.

A set of four criteria proposed by Rapps and Weyuker [78] follow. The *all-c-uses/some-p-uses* criterion requires the coverage of the subpaths that connect a definition to all its computational uses and at least to one of its predicate uses. The *all-p-uses/some-c-uses* criterion is the opposite; all predicate uses should be reached by the tested subpaths plus at least one computational use. The *all-predicate-uses* and *all-computation-uses* criteria are similar but they ignore completely the computational and the predicate uses respectively.

Due to the fact that between a definition of a variable and a use there may be more than one path, a stronger criterion would be to exercise all possible paths from each definition to each of its uses. Such a criterion, though, lacks applicability. Frankl and Weyuker [29] and Clarke et al. [23] attempted to limit the number of paths by introducing the *all definition-use-paths* criterion so that it only requires paths that are cycle-free or contain only simple cycles to be exercised. Yet, there is still a probability that such a path does not exist, maintaining the applicability problem of such a criterion.

Ntafos [66, 67] studied the interactions of different variables. He defined as $k-dr$ *interactions* the chains of k alternating definitions and uses of variables. In particular, each definition reaches its paired use at the same node where the next definition in the chain occurs. He proposed the *required k-tuples* criterion according to which for all $j-dr$ interactions, where $j = 2,3,...,k$, there is at least one path that contains a subpath which is an interaction path for the corresponding $j-dr$ interaction.

Further work performed by Laski and Korel [48] focuses on studying how the variables used for computation at a node are affected by definitions that reach that node from other nodes. They defined the notion of the context of a node v as a set of the definitions of variables at different nodes that may possibly reach the uses of the variables in v. Hence, this excludes definitions that will be definitely overwritten before they reach the node. In addition, the set of ordered contexts contains all the possible definitions sequences by which that node can be reached. The *ordered-context coverage* criterion requires that there is at least one path exercised for each ordered context path, where an ordered context path is an execution path between ordered context elements so that after a node is considered as the definition of a variable, there will be no other node that contains a definition of that same variable. The *context coverage* criterion is a bit weaker since it ignores the ordering between the nodes.

Finally, the *Dependence-Coverage* criteria that combine control-flow and data-flow testing were proposed by Podgurski and Clarke [76, 77]. These criteria are derived by replacing the definition-use association relation with various dependence relations, such as semantic

dependence.

As well as the control-flow adequacy criteria, data-flow criteria can be converted into metrics that indicate the quality of a test suite by quantifying the extent of the fulfilment of the tests' requirements.

2.1.2 Fault-based Testing Adequacy Criteria

Error seeding and *mutation analysis* are the main approaches to fault-based test adequacy criteria. Other techniques include variations of mutation testing and perturbation testing. An overview of how these techniques can be applied to acquire test effectiveness indicators is presented in this section. A synopsis of the techniques is shown in Table 2.3.

Error seeding, proposed by Mills [60], is the technique of planting artificial errors in a software system. The system is then tested against these artificial errors and the ones that have been successfully detected are counted. The ratio of the detected errors to the total number of the planted errors is an indicator of the ability of a system's test suite to detect defects. The planting of the errors can be done either manually or automatically, by random modifications of the source code. However, the approach has certain downsides [57]. When the planting of the errors is done manually, the process is labour intensive. When the planting is automatic, the artificial errors are usually easier to detect than true errors, thus introducing bias in the measurement of the effectiveness of the tests. Another negative issue is the fact that the errors are not reproducible, making comparisons between different applications of the measurement on the same program unreliable.

DeMillo et al. [26] and Hamlet [33] introduced *mutation analysis*, a systematic way of performing error seeding. In mutation analysis modifications are applied to the source code according to rules that describe simple changes, such as swapping the comparison operators in conditions or removing statements. These rules are called mutation operators and finding an effective set of those has been the goal of several research works [20, 46, 53, 69]. Mutation operators are applied one at a time to the original program, creating the so-called mutants. The mutants are tested against the test suite. The mutants that cause a test to fail, are considered killed. Some of the mutants have the same output as the original program. In that case, they are considered equivalent to the original program. The number of killed mutants divided by the total number of mutants minus the number of equivalent mutants is called *Mutation adequacy score*. This technique is also known as the *strong* or *traditional* mutation testing.

The mutation adequacy score is an indicator of the effectiveness of the test suite and it is supported by literature [54, 52] that it should be used in combination with test coverage criteria. Test coverage criteria count how thoroughly the source code is covered, but not how effectively it is covered. Mutation analysis compensates for that part.

The applicability of mutation analysis to real software projects is based on two assumptions, the *competent programmer* assumption and the *coupling effect* assumption [26]. According to the competent programmer assumption, an experienced programmer introduces only small faults when working on an application. Coupling effect assumes that complex faults are a result of a series of smaller, simpler faults. These assumptions lead to the hypotheses that every test case that detects a simple fault, will also detect a complex fault

Table 2.2: Data-flow Test Adequacy Criteria

Criterion	Proposed by	Description
All Definitions	Frankl and Weyuker [29]	At least one subpath connecting each definition of a variable x to one if its uses is exercised.
All Uses/Reach-Coverage	Frankl and Weyuker [29]/Herman [35]	At least one subpath connecting each definition of a variable x to each of its uses is exercised.
All-c-uses/some-p-uses	Rapps and Weyuker [78]	At least one subpath connecting each definition of a variable x to each of its computational uses and to at least one of its predicate uses is exercised.
All-p-uses/some-c-uses	Rapps and Weyuker [78]	At least one subpath connecting each definition of a variable x to each of its predicate uses and to at least one of its computational uses is exercised.
All-predicate-uses	Rapps and Weyuker [78]	At least one subpath connecting each definition of a variable x to each of its predicate uses is exercised.
All-computation-uses	Rapps and Weyuker [78]	At least one subpath connecting each definition of a variable x to each of its computational uses is exercised.
All Definition-Use-Paths	Frankl and Weyuker [29], Clarke et al. [23]	For all definitions and all of their uses, all possible paths that connect a definition to each of its uses are exercised as long as the paths are cycle-free or contain only simple cycles.
Required k-tuples	Ntafos [66, 67]	At least one path is exercised, for all $j = 2,3,...,k$, that includes a subpath which is a $j-dr$ interaction.
Ordered-Context Coverage	Laski-Korel [48]	At least one path is exercised for all ordered contexts of all nodes of the flow graph of a program so that a subpath that is an ordered context path is contained.
Context-Coverage	Laski-Korel [48]	At least one path is exercised that covers all definition context paths for all nodes of the flow graph of a program.

Table 2.3: Fault-based Test Adequacy Criteria

Criterion	Proposed by	Description
Error Seeding	Mills [60]	Planting artificial errors in the program and testing it to calculate the percentage of detected errors by the test suite as a quality indicator.
Mutation Adequacy Score	DeMillo et al. [26] and Hamlet [33]	After systematically creating mutated versions of the original program (mutants), the tests are run and mutation score is calculated as the number of mutants that caused a test to fail divided by the total number of non-equivalent mutants.
Neighborhood Adequacy	Budd and Angluin [21]	Local correctness of a program in respect to its neighbourhood Φ requires that all programs in Φ are either equivalent to p or they cause at least one of the tests to fail.
Perturbation Testing	Zeil [96]	Possible alternative expressions in the program that produce the same output on the tests indicate weakness of the tests to distinguish between the correct expression and its alternatives.

which is composed of that simple fault. Since experienced programmers tend to introduce simple faults, simulation of these faults should be sufficient by applying a single mutation operator. Consequently, combinations of multiple mutation operators can be avoided and hence, the exploding increase of the number of the mutants and the corresponding computing cost are also avoided [32].

A different fault-based testing criterion was proposed by Budd and Angluin [21]. In their work, neighbourhood Φ of a program p is defined as the set of all the programs that depend on p, meaning that they were derived by small modifications of p. It should be noted that Φ includes p itself. In order for p to be locally correct in respect with Φ, the *neighbourhood adequacy* criterion requires that for any program q in Φ, either q is equivalent to p or there exists a test which fails when it is run on q.

The advantages of mutation analysis include the high possibility of automating the process and the ability to spot the cause of a defect by observing the mutation operator that has been applied in combination with the specific location in the code where the mutation took place. In addition, mutation analysis can be used for benchmarking testing strategies [32].

However, mutation analysis is a computationally expensive method. There have been continuous efforts to reduce the cost of mutation testing by introducing variations. These efforts can be split into those that reduce the execution cost of creating or testing the mutants and those that reduce the number of mutants. Table 2.4 shows an overview of the most important mutation analysis cost reduction techniques as discussed by Zhu et al. [98] and

Table 2.4: Mutation Analysis Cost Reduction Approaches

Technique	Proposed by	Description
Strong or Traditional Mutation	DeMillo et al. [26]	Full cost approach.
Weak Mutation	Howden [39]	Testing performed only on the component that contains the change that derived the mutant.
Firm Mutation	Woodward et al. [92]	Parametrized approach, providing an intermediate solution, faster than strong mutation and more effective than weak mutation.
Mutant Schema Generation	Untch [84]	A meta-program representing all the mutations is created reducing the cost to one-time compilation.
Bytecode Translation	Ma et al. [70]	Mutations occur at the bytecode level.
Mutant Sampling	Acree [2] and Budd [19]	After all mutants are created, a random percentage is selected to participate in the testing.
Mutant Clustering	Hussain [42]	Mutants are clustered according to the type of the test case that kills them. Only some of the mutants of each cluster are used for testing.
Selective Mutation	Mathur [55] and Offutt et al. [71]	A small set of mutation operators is sought and applied trying to minimize the loss of test effectiveness.
Higher Ordered Mutation	Jia et al. [43]	Combining more than one mutation operator to create higher ordered mutants (HOMs) that are harder to kill than the first order mutants (FOMs) from which they were derived. Cost is reduced by testing these HOMs instead of the corresponding FOMs.

Jia et al. [44] and points to the corresponding literature.

Perturbation testing, proposed by Zeil [96], is an approach similar to mutation testing. It focuses on finding the possible alternative expressions (predicates or computations) that may produce the same output on the tests. The existence of these alternative expressions, called perturbations, can be interpreted as points in the code where faults cannot be detected, given the current test suite. Therefore, new tests have to be added in order to distinguish between the correct expression and its possible perturbations. The set of the perturbations is called error space of the program. The effectiveness of the tests is measured by its ability to limit the error space of the program [98]. However, perturbation testing has limited applicability due to the assumption that the error space of the program is a vector space.

2.1.3 Error-based Testing Adequacy Criteria

As it was previously mentioned, error-based test adequacy criteria focus on testing programs on error-prone points. In order to identify error-prone points, a domain analysis of a program's input space is necessary [98].

The input space can be divided in subdomains so that in each subdomain the behaviour of the program is the same. Consequently, subdomains in programs coincide with the possible execution paths. For different sets of inputs that lead to the execution of a certain path, exactly the same computations will be applied. Thus, subdomains are separated according to the set of the control-flow conditions that are required to activate each of the execution paths. After partitioning the domain into subdomains, test adequacy criteria propose in which way test cases should be chosen.

For example, the input space of an integer x includes all the possible integers. Suppose the variable x is used in an *if* statement where the condition demands x to be greater than zero and at most 100. The input space can then be split in three subdomains: (1) the integers that are less or equal than 0, (2) the integers that are greater than zero and less or equal than 100 and (3) the integers that are greater than 100. The error-prone points are the borders between the subdomains, in this case the values around 0 (e.g. -1, 0, 1) and around 100 (e.g. 99, 100, 101).

In case only a single test case for each subdomain is required, the criterion is equivalent to path coverage. However, points that are closer to the boundaries between the subdomains are known to be critical. Therefore, error-based adequacy criteria fall into two categories: those that focus on the boundaries between the subdomains and those that concern the computations within the subdomains. Boundary errors occur because of faults in the transfer control predicates. Computation errors occur because of faults in the implementation of the computation statements. Table 2.5 summarizes the criteria. A brief presentation of the criteria follows.

White and Cohen [90] proposed the $N \times 1$ *domain testing strategy*. This requires N test cases, where N is the number of the input variables of the program, on each border of each subdomain. An additional test case close to the border is also required. In particular, the additional test case should be *out* of the subdomain if the border belongs to the sub domain or *in* otherwise.

Clarke et al. [22] proposed a stricter criterion. The $N \times N$ criterion is similar to the $N \times 1$ domain testing strategy, except that it requires N linearly independent test cases close to the borders instead of just one.

Taking into consideration that the vertices of the subdomains are the intersections of their borders, Clarke et al. [22] proposed the $V \times V$ *domain adequacy* criterion. The requirement in this case is that there is a test case for every vertex of the subdomain and for each vertex, there is also a test case for a point close to the vertex. Again, the additional point should be *out* of the subdomain if its corresponding vertex is in the subdomain or *in* otherwise.

The aforementioned criteria are effective for domains whose subdomains' borders are linear functions. When this is not the case, the $N+2$ *domain adequacy* criterion proposed by Afifi et al. [3] is suitable. In particular, the criterion requires that there are $N+2$ test cases

Table 2.5: Error-based Test Adequacy Criteria

Criterion	Proposed by	Description
$N \times 1$ domain testing strategy	White and Cohen [90]	Requires N test cases on each border of each subdomain and another test case close to the border.
$N \times N$ domain adequacy	Clarke et al. [22]	Requires N test cases on each border of each subdomain and N test cases close to the border.
$V \times V$ domain adequacy	Clarke et al. [22]	Requires a test case for each intersection of the borders of the subdomains paired with a test case close to the intersection.
$N + 2$ domain adequacy	Afifi et al. [3]	Requires N test cases on each border of each subdomain and two test cases close to the border, one in each side of the border. Effective for non-linear domains.
Functional adequacy	Howden [38, 40]	Required test cases derived from the analysis of the functions of each subdomain.

for each border of each subdomain, where N is the number of the input variables of the program. From these, N test cases should be on the border and from the additional two test cases, one should be close to the border and outside of the subdomain and the other should be close to the border and inside the subdomain. The criterion has more requirements that are analytically explained in [3] .

Howden's *functional adequacy* criterion [38, 40] focuses on the possible computation errors. The correctness of the computation is checked by requiring a number of appropriate test cases, according to the analysis of the functions of each subdomain, supposing that each function is a multinomial.

Error-based testing should be performed by using criteria focused on boundary errors and computation errors at the same time. Unfortunately, its application is limited when the complexity of the input space is high or when the input space is non-numerical [98].

2.1.4 Assertions and Test Code Smells

In the previous subsections well-defined criteria were presented and discussed. There is further research work performed on techniques that indicate the quality of test code.

Empirical research on the use of assertions in the code reports that software systems with extended use of assertions have less faults compared to systems that make less use of assertions. In particular, Kudrjavets et al. [47] defined *assertion density* as the number of assertions per thousand lines of code and showed that there is a negative correlation between assertion density and fault density. Voas [86] researched how assertions can increase the test effectiveness by increasing the error propagation between the components of object oriented systems, so that the errors are detected more easily. Assertions are also the key points of

test cases at which something is *actually* tested. According to some practices, each test case should contain a single assertion. In that case, assertions and test cases are associated. Therefore, it is reasonable to expect assertion density of testing code to be an indicator of the effectiveness of the tests.

Test code has the same requirements for maintenance as production code. It is important to ensure that it is clear to read and understand, so that its modification is possible. Moreover, integrating the execution of the tests in the development process requires that the tests are run efficiently. Thus, the need for test code refactoring is obvious. In order to detect possible points of low quality in the test code that require refactoring, van Deursen et al. [27] introduced *test smells*. Test smells adapt the notion of code smells in testing. Further research followed, either towards defining more test smells and appropriate refactorings [59] or towards automated detection of the test smells [85, 79].

2.2 Issue Handling

2.2.1 Issue Tracking Systems and the Life-Cycle of an Issue

ITSs are software systems used to track defects as well as enhancements or other types of issues, such as patches or tasks. ITSs are commonly used in the development process [41]. They enable developers to organise the issues of their projects. In the context of this study, we focus on defects and enhancements, although the process is relatively similar for the other types.

When defects are discovered or new features are requested, they are reported to the ITS. Naturally, not all issues which arise follow this procedure. Some of them are handled informally and no information exists in order to assess the properties of their resolution. Those that are reported follow a specific *life-cycle*. Even though there is a variety of implementations of ITSs (e.g. BugZilla[1], Jira[2], SourceForge[3], Google Code[4]), essentially they all adapt the same process.

Figure 2.1 shows the life-cycle of an issue report. Initially, the report is formed and submitted as an *unconfirmed* issue. After it is checked whether the issue has already been reported or if the report is not valid, the issue status is changed to *new*. The next step is to assign the issue to an appropriate developer, an action which results in the issue state *assigned*. Next, the developer will examine the issue in order to resolve it. The possible resolutions are:

- **Invalid** : The issue report is not valid (e.g. not described well enough to be reproduced, etc.).
- **Duplicate** : The issue has already been reported.
- **Fixed** : The issue is fixed.

[1] http://www.bugzilla.org/
[2] http://www.atlassian.com/software/jira/
[3] http://sourceforge.net/
[4] http://code.google.com/

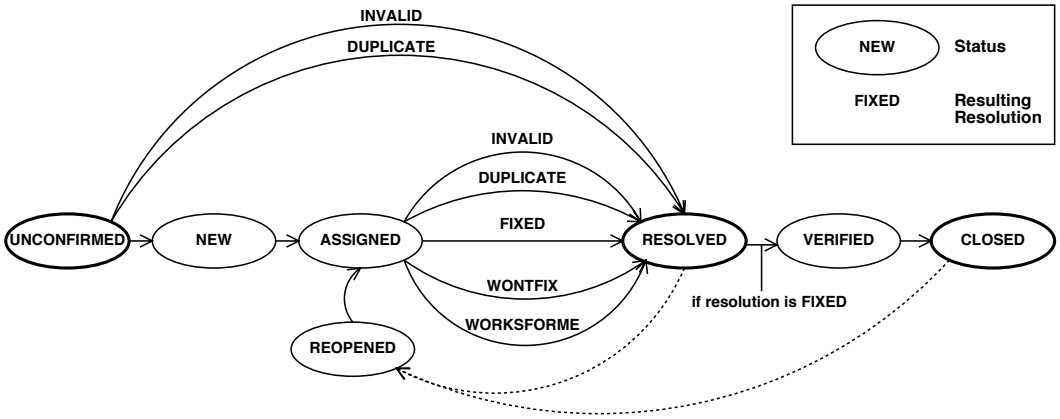

Figure 2.1: Issue Report Life-cycle (adapted from [97])

- **Won't fix** : The issue will not be fixed (e.g. what the reporter thought of as a defect is actually a desired feature, etc.).
- **Works for me** : The issue could not be reproduced in the environment of the developer.

The issue is marked as *resolved* and then it is *closed*, unless it was a fixed issue. In that case, the correctness of the fix is checked and if it is confirmed the issue is marked as *verified* and then it is deployed, resulting the status change to *closed*. It is possible that the issue will emerge again in the future. If this occurs, the issue's state is set to *reopened* and a new assignment of the issue to a developer has to take place for the resolution process to start over.

2.2.2 Defect Resolution Time

Defect resolution time is of great interest in empirical software engineering. Quality-driven software development is motivated by the expectation of the decrease of the effort needed to maintain the system, both in the context of adding new features or resolving defects. Defect resolution time is an indicator of the time that is necessary to resolve a defect and thus, an indirect indicator of the effort that was invested. As it has been discussed previously, high quality testing is translated into better detection of the cause of defects and consequently, it is expected to result in the reduction of the necessary time to resolve a defect. However, before this claim can be evaluated, a representative measurement of the defect resolution time has to be defined. Such a measurement can be derived from the data stored in ITSs.

As shown in 2.2.1, issue resolution is tracked by logging a series of possible actions. Whenever there is an action in the ITS that changes the status of an issue, the date and time are recorded among other kind of data. An arguably straightforward measurement of the defect resolution time would be to measure the interval between the moment when the

defect was assigned to a developer and the moment it was marked as resolved. Complicated situations where the issue is reopened, reassigned and marked as resolved again can be dealt with by aggregating the intervals between each assignment and its corresponding resolution.

In fact, this practice has been followed in most of the empirical studies that involved defect resolution time. In particular, Luijten [49] mined defect resolution time from several projects and showed that there exists negative correlation with the software's maintainability. Giger et al. [30] worked on a prediction model of the fix time of bugs, acquiring the fix time from ITSs in the same way as described above. In addition, Nugroho [68] investigated the correlation between the fixing effort of defects related to modelled behaviours of functionalities and defects related to non-modelled behaviours of functionalities. Fixing effort was again measured similarly.

Ahsan et al. [4] also proposed a bug fix effort estimation model. They obtained the defect resolution time as described above, but they further normalized the time by taking into account the total number of assignments of a developer to defects at a given time. For instance, a developer that worked 20 days for a month and worked on the resolution of 4 defects for 10 days each in this particular month, could not have worked 40 days in total. Thus, they normalize the fix times by multiplying them with the multiplication factor which is obtained by dividing the total actual working days of a month with the sum of all the assigned working days for all the assigned bugs.

Different approaches towards measuring the defect resolution time follow. Weiss et al. [88] predict the defect fixing time based on the exact duration of the fix as it was reported by developers in Jira, one of the few ITSs that allow the specification of the time spent on fixing a defect. Unfortunately, this has a restricted application either because of the fact that many projects use a different ITS or because even if they use Jira, few developers fill this information in (e.g. In JBoss, which was used in [88], only 786 out of the $11,185$ reported issues contained effort data).

Finally, Kim et al. [45] obtained the defect-fix time by calculating the difference between the commit to the Version Control System (VCS) that solved the defect and the commit that introduced it. Specifically, they spot the commit that solved the defect by mining logs in the VCS for keywords such as "fixed" or "bug" or references to the identification number of the defect report. They spot the commit that introduced the defect by applying the fix-inducing change identification algorithms proposed by Sliwerski et al. [82]. This approach is based on linking the VCS to the ITS. Bird et al. [18] investigated the bias in such approaches concluding that they pose a serious problem for the validity of the results.

There are many threats to the validity of such a measurement. ITSs are operated by humans and hence, the information that can be acquired is prone to inaccuracies. For instance, defects that have been practically resolved, remain open for a long time. Furthermore, even though a defect seems that it was being fixed for a certain time interval, it does not mean that a developer was working on that continuously for the whole duration of the interval. Even worse, there is no information on whether more than one developer was working on the defect, increasing the actual fixing effort. Nevertheless, data retrieved from an ITS are as accurate as possible for usage in empirical analyses and even though not completely accurate, useful results can be derived in order to comprehend and improve software engineering.

2.2.3 Throughput and Productivity

Bijlsma [16] introduced additional indicators of issue handling performance. Among the indicators, measurements of throughput and productivity were defined[5].

Throughput is measuring the total productivity of a team working on a system in terms of issue resolution. It is defined as follows:

$$throughput = \frac{\#\ resolved\ issues\ per\ month}{KLOC}$$

The number of resolved issues is normalised per month so that fluctuations of productivity because of events such as vacation periods, etc. have less impact. Moreover, in order to enable comparison between systems of different size, the number of resolved issues per month is divided by the volume of the system in lines of code.

Throughput measures how productive the whole team that works on a system is. However, many other parameters could be affecting that productivity. One of the parameters is the number of developers within the team. This is solved by calculating productivity, the number of resolved issues per developer. Productivity is defined as follows:

$$productivity = \frac{\#\ resolved\ issues\ per\ month}{\#\ developers}$$

When the indicator is used in the context of open source systems, as intended in this study, the challenge in calculating productivity is to obtain the number of developers of the team. In [16] this is performed by mining the VCS of the system and applying data mining to obtain the number of different users that committed code at least once. However, Mockus et al. [61] in their study of open source software development, formulate their concern that in open source teams, the Pareto principle applies: 80% of the work is performed by 20% of the members of the team. This 20% comprises the core team of the system. This suggests the investigation of the difference of the productivity indicator when the number of developers includes the whole team or just the core team.

2.3 The SIG Quality Model

SIG has developed a model for assessing the maintainability of software. A description of the model is provided in [8]. A summary follows in this section.

The quality model has a layered structure for measuring technical quality of software as it is suggested by the quality characteristics of ISO/IEC 9126 [34]. Figure 2.2 presents the structure of the model. The ISO/IEC 9126 defines analysability, changeability, stability and testability as the main sub-characteristics of software maintainability. The SIG quality model defines source code metrics and maps these metrics to the sub-characteristics in order to make the ISO/IEC 9126 standard operational. The metrics are calculated based on source

[5] In the initial work these names correspond to project productivity, developer productivity and enhancement ratio. They were later renamed to throughput, productivity and efficiency respectively in unpublished research work.

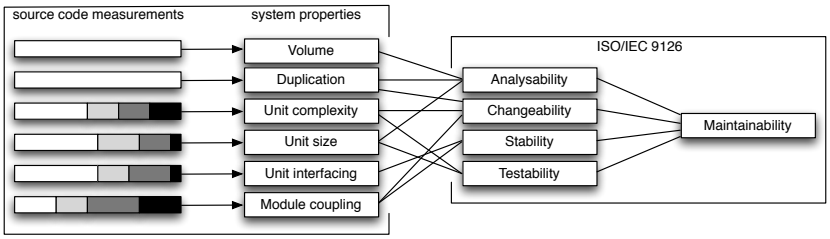

Figure 2.2: The SIG Quality Model maps source code measurements onto ISO/IEC 9126 quality characteristics (image taken from [50]).

code analysis and they are converted to ratings that reflect system level properties. Afterwards, the system level properties are mapped to the ISO/IEC 9126 sub-characteristics, which are in turn aggregated into an overall maintainability rating. The ratings are presented using a star rating system, ranging from 1 star to 5 stars. More stars always mean better quality.

2.3.1 Source code metrics

The source code metrics are the following:

Estimated rebuild value The size of the system's production code is measured in lines of code. Based on the lines of code, the rebuild value of the system is estimated in man-years using language productivity numbers found in the Programming Languages Table of the Software Productivity Research[6]. The system property *volume* is based on this metric.

Percentage of redundant code This metric is calculated as the number of lines of code that are considered to be redundant. Duplication is detected when a code fragment of more than 6 lines of code is repeated at least once in the system. The system property *duplication* is based on this metric.

Lines of code per unit Unit is the smallest piece of invokable code in a language (e.g. a method for Java). The metric is calculated by counting the lines of code in a unit. The system property *unit size* is based on this metric.

Cyclomatic complexity per unit For each unit the metric is equal to McCabe's cyclomatic complexity [56]. The system property *unit complexity* is based on this metric.

Number of parameters per unit The metric represents the number of parameters that are declared in the interface of a unit. The system property *unit interfacing* is based on this metric.

[6]http://www.spr.com/programming-languages-table.html

Number of incoming calls per module For a module (e.g. a class or a file) the metric is calculated by counting the number of invocations of the module (e.g. calling one of the module's units). The system property *module coupling* is based on this metric.

2.3.2 Converting the source code metrics to system properties ratings

After the measurements are obtained from the source code, they are converted in ratings that correspond to system level properties. This is performed through the use of benchmarking. SIG possesses and curates a database with hundreds of systems that were built using various technologies [6]. The model is calibrated based on the benchmark so that the metrics can be converted into star ratings that reflect the system's performance in comparison with the benchmark.

In particular, the levels of quality are defined so that they correspond to a $\langle 5, 30, 30, 30, 5 \rangle$ percentage-wise distribution of the systems in the benchmark. This means that a system that is rated 5 stars in a property performs similarly with the best 5% of the systems in the benchmark. At the same time, a 2 star rating means the system performs better than the 5% worst systems and worse than the 65% best systems in the benchmark.

Based on the aforementioned calibration technique, thresholds are calculated that serve as the values that separate the different quality levels. There are two types of system properties: the ones that are calculated on the system level (volume and duplication) and the ones that are calculated on the unit or module level (the rest). The calibration is performed differently for each of these types of properties.

Volume and duplication are based on measurements that correspond to the entire system, a number of man-years for Volume or a percentage of redundant lines of code for duplication. The calibration technique will be illustrated using an example. Repeating the example that is shown in [8], to acquire the rating of the property duplication, thresholds are calculated based on the percentage of redundant lines of code. The mapping of thresholds to the star ratings is shown in Table 2.6.

Table 2.6: Thresholds for Duplication

Rating	Duplication
⋆⋆⋆⋆⋆	3%
⋆⋆⋆⋆	5%
⋆⋆⋆	10%
⋆⋆	20%
⋆	-

The thresholds are interpreted as the maximum allowed value that a system may have in order to receive the corresponding rating. A system with 8% redundant code will be rated as a 3-star system with regard to duplication, since the percentage of redundant lines of code is greater than the thresholds for 5- and 4-star systems and lower than the threshold for a 3-star system.

At this point, it is important to observe that a system with 10.1% duplicated code and another with 19% would both score 2 stars in the duplication property. This would reduce a lot of the flexibility of the model to compare different systems to each other. Therefore, linear interpolation is used in order to convert the metric to a rating in a continuous scale in the interval $[0.5, 5.5]$.

The system properties that are calculated in a more fine-grained level than volume and duplication are aggregated using the *quality profiles* technique. The metrics upon which these properties are based are calculated for each unit/module. Borrowing again an example from [8], the technique will be explained for unit complexity. First, the cyclomatic complexity metric is calculated for every unit in the benchmarking set. This enables us to study the distribution of the metric and detect the points where the behaviour of the metric changes. This method is illustrated in detail by Alves et al. [6]. Based on the distribution of the metric, thresholds are derived in order to classify units in four risk categories: low, moderate, high and very high. For cyclomatic complexity this results in the thresholds that can be seen in Table 2.7. It should be noted that the calibration of the SIG model is a continuous process so that the state-of-the-art of industrial software development is reflected. Therefore, the values of the thresholds that are presented in this study may not be up-to-date.

Table 2.7: Risk categories for cyclomatic complexity [6]

McCabe cyclomatic complexity	risk category
1-6	low
7-8	moderate
9-14	high
> 14	very high

After assessing the risk level for each unit in a system, the percentage of the volume of the system that falls in each category is calculated. For instance, a unit of 10 lines of code that has a cyclomatic complexity of 3 is in the low risk category, thus 10 lines of code of the system are in the low risk category. From the benchmark, thresholds are derived that enable mapping the 5 quality levels to the percentages of code that are allowed to be in each risk category.

As with the properties that are calculated on the system level, the thresholds are derived so that the quality levels represent the $\langle 5, 30, 30, 30, 5 \rangle$ percentage-wise distribution of the systems in the benchmark. The quality profiles for unit complexity are shown in Table 2.8. The low risk category is missing because it is the complement of the sum of the other three adding up to 100%.

The thresholds are the maximum percentages of code that a system may have in each risk category in order to receive the corresponding rating. It is important to mention that the percentages are accumulated from the higher risk categories to the lower ones. For example, given the thresholds in Table 2.8, a system with 3% of code in the very high risk category and 14% of code in the high risk category is not eligible for a 2-star rating because the value that is checked against the threshold for the high risk category is 17%.

Table 2.8: Profile thresholds for cyclomatic complexity (taken from [8])

	maximum relative volume		
rating	moderate	high	very high
⋆⋆⋆⋆⋆	25%	0%	0%
⋆⋆⋆⋆	30%	5%	0%
⋆⋆⋆	40%	10%	0%
⋆⋆	50%	15%	5%
⋆	-	-	-

The interpolation method that was previously discussed is applied here as well. Linear interpolation according to the percentage in each of the risk categories (moderate, high and very high) is applied. As a final rating, the minimum of the three ratings is chosen.

2.3.3 Mapping the system properties to the ISO/IEC 9126 sub-characteristics of maintainability

Finally, the last step for creating the model was to map the system properties to the sub-characteristics of the ISO/IEC 9126. This was performed by selecting the most important properties that affect each sub-characteristic. The selection was based on expert opinion. A survey was later performed [25] in order to validate the selection. The mapping can be seen in Figure 2.3. For each sub-characteristic a $\times$ is placed in the columns that correspond to the system properties that are considered the most influential factors.

The rating for each sub-characteristic is calculated by averaging the ratings of the system properties that are mapped to it. Finally, the overall maintainability rating is the average of the four sub-characteristics. The layered structure of the model enables answering questions at different levels of granularity. For example, starting from the overall maintainability, one can go to the sub-characteristics level to identify which main aspects of the system do not satisfy the desired quality. One additional step down to the properties level, the system level properties indicate what in particular has to be improved in the source code in order to improve the system's technical quality.

2.4 Related Work

In this section, previous work that aimed at a systematic way of assessing test code quality is discussed.

To our knowledge the efforts of assessing test code quality are limited to individual metrics and criteria such as the ones that were presented in Section 2.1. In this study we aim at constructing a test code quality model in which a set of source code metrics are combined. There is one more research group with similar work. Nagappan et al. proposed and assessed a suite of test code related metrics that resulted to a series of studies that are summarised in a PhD thesis [64].

The Software Testing and Reliability Early Warning (STREW) static metric suite is composed of nine metrics which are separated in three categories: test quantification, com-

ISO 9126 maintainability \ properties	Volume	Duplication	Unit size	Unit complexity	Unit interfacing	Module coupling
Analysability	×	×	×			
Changeability		×		×		×
Stability					×	×
Testability			×	×		

Figure 2.3: Mapping of system properties to ISO/IEC 9126 sub-characteristics (taken from [8])

plexity and object-orientation (O-O) metrics, and size adjustment. The group of the test quantification metrics contains four metrics: (1) number of assertions per line of production code, (2) number of test cases per line of production code, (3) the ratio of number of assertions to the number of test cases and (4) the ratio of testing lines of code to production lines of code divided by the ratio of the number of test classes to the number of production classes. These four metrics are intended to cross-check each other to compensate for different programming styles. For example, some programmers tend to have one assertion per test case but some others have a lot of different assertions in single test cases. The group of the complexity and O-O metrics examines the relative ratio of test to source code for control flow complexity and for a subset of the CK metrics. Finally, a relative size adjustment metric is used [65].

In order to validate STREW as a method to assess test code quality and software quality a controlled experiment was performed [65]. Students developed an open source Eclipse[7] plug-in in Java that automated the collection of the STREW metrics. The student groups were composed by four or five junior or senior undergraduates. The duration of the experiment was six weeks. Students were required to achieve at least 80% code coverage and to perform a set of given acceptance tests.

The STREW metrics of 22 projects were the independent variables. The dependent variables were the results of 45 black-box tests that were applied on the projects. Multiple regression analysis was performed assessing the STREW metrics as predictors of the number of failed black-box tests per KLOC (1 KLOC is 1000 lines of code). The result of the study revealed a strong, significant correlation ($\rho = 0.512$ and p-value $\ll 0.01$). On that basis they concluded that a STREW-based multiple regression model is a practical approach to assess software reliability and quality.

Several limitations in their study are discussed in [65]. The generalisation of the results beyond students is difficult. Furthermore, the developed Eclipse plug-ins were small rela-

[7] http://www.eclipse.org/

tive to industry applications. The set of black-box tests is relatively small and doubtful as to whether it is representative of problems that would have been found by the customers. In addition, there is no effort to control confounding factors such as the quality of the production code or the variance of the experience between the students.

As mentioned previously, in our study we also aim at selecting a set of metrics in order to assess test code quality. The STREW metrics provide the basis for selecting suitable metrics. However, our goal is to go a step further and develop a model which is based on the metrics. In addition, we identify some weak points of the STREW metrics. Coverage-related metrics are completely left out of the suite. The complexity and O-O metrics are always taking into consideration the ratio of these metrics between test and production code (e.g. ratio of cyclomatic complexity of test code to cyclomatic complexity of production code). These ratios are hard to interpret. Last but not least, it is difficult to compare systems based on the raw metrics' values and that provides our motivation to build upon the SIG quality model and the benchmarking technique that is used there.

Chapter 3

Constructing a Test Code Quality Model

In this chapter *RQ1* is addressed: *How can we evaluate the quality of test code?* First, we investigate how test code quality can be measured and what information is needed in order to assess the various aspects of test code. In particular, the main aspects of test code quality are identified and the motivation for their importance is provided. Afterwards, metrics that are related to each of the identified aspects are being presented. By mapping the metrics to the main aspects of test code quality, a test code quality model is created and presented. The model combines the metrics and aggregates them in a way that extracts useful information for the technical quality of test code. Finally, the benchmarking technique is applied in order to calibrate the model and convert its metrics into quality ratings.

3.1 Assessing Test Code Quality: A GQM Approach

The Goal, Question, Metric approach (GQM), developed by Basili [9], is an approach for selecting metrics to quantify a concept. Goals are capturing the concepts to be measured. Questions capture a goal's different aspects to be assessed. Finally, metrics are associated with each question providing an answer in a measurable way.

Using the GQM template [91], we define our main goal in this section as follows:

Analyze test code
for the purpose of developing an assessment method
with respect to test code quality
from the perspective of the researcher, developer

In order to achieve this goal, we refine it into several subsidiary goals.

3.1.1 Goals

To answer the question "how can we evaluate the quality of a system's test code" we can generate the following questions:

Q1 *How completely is the system tested?*

Context: The test code is exercising parts of the production code in order to check that functionality works as expected. The more parts of the production code are tested, the more complete the test suite. Between systems where all other aspects of test code quality are the same, a system that is tested more completely, is better tested.

Interpretation: The more parts of code are exercised by a test suite, the higher the confidence that defects can be detected throughout the entire system.

Q2 *How effectively is the system tested?*

Context: Test code is executing parts of the production code. At the same time, it can test whether the system behaves as expected. This can happen at different granularity levels (unit testing to integration testing). Between systems where all other aspects of test code quality are the same, a system that is tested more effectively, is better tested.

Interpretation: Exercising the production code is essential but not sufficient. Higher effectiveness improves the ability of the test suite to detect a defect and trace its cause.

Q3 *How maintainable is the system's test code?*

Context: Test code is above all code. Nowadays, systems contain an increasing amount of test code. In many cases there is more test code than production code [95]. Software metrics for maintainability can be applied on test code as well. A system with maintainable test code is more likely to sustain its quality as the system evolves.

Interpretation: The higher the maintainability of the test code, the more easily it can be adapted to the changes of the production code.

These three questions form the measurable sub-goals that we need to achieve in order to accomplish the main goal, which is measuring the quality of test code.

3.1.2 Questions

To answer **Q1** we can consider different ways to measure how completely a system is tested. As shown in Section 2.1, there are various code coverage criteria. In fact, An and Zhu [7] tried to address this issue by proposing a way to integrate different coverage metrics in one overall metric. However, their approach is complicated . For example, it requires an arbitrary definition of weights that reflect the criticality of the modules of the system and the importance of each of the coverage metrics. In order to increase simplicity, applicability and understandability of the model, we will answer **Q1** by refining it in the following sub-questions:

Q1.1 *How much of the code is covered by the tests?*

Refines: **Q1**.

Context: A covered part of code is code that is executed when the tests are run.

Metrics involved: *Code coverage*.

Interpretation: The higher the percentage of the code that is covered, the higher the completeness of the test code.

Q1.2 *How many of the decision points in the code are tested?*

Refines: **Q1**.

Context: A decision point (e.g. IF statements, FOR loops) is covered when during tests there is at least one test that targets each of the possible decisions that can be taken. The number of decision points can be measured using McCabe's cyclomatic complexity [56]. The number of tests can be approximated by the number of assertions (or test cases, depending on the programming style). This measurement is similar to the *cyclomatic-number* criterion [87] as seen in Section 2.1.

Metrics involved: *Assertions-McCabe ratio*.

Interpretation: The higher the Assertions-McCabe ratio, the higher the completeness of the test code.

To answer **Q2**, we have to consider what makes test code effective. When test code covers a part of production code, it can be considered effective when it enables the developers to detect defects and after they detect them, it enables them to locate the cause of these defects in order to facilitate the fixing process. Consequently, the following sub-questions refine **Q2**:

Q2.1 *How able is the test code to detect defects in the production code that it covers?*

Refines: **Q2**.

Context: Defects are detected in a covered part of code by providing tests that assert that the behaviour and the outcome of the code were as expected. Testing value is delivered through the assert statements.

Metrics involved: *Assertion density*.

Interpretation: The higher the assertion density, the higher the effectiveness of the test code.

Q2.2 *How able is the test code to locate the cause of a defect after it detected it?*

Refines: **Q2**.

Context: When a test fails, the developer starts tracking the cause of the defect in the part of the code that is covered by that test. Figure 3.1 illustrates a test *testFor()* that exercises *foo()* directly and *bar()* indirectly, since *foo()* is calling *bar()*. Imagine a change in the code introduces a defect in method *bar()*. After running *testFoo()* the test fails. The developer will first think that the problem is in *foo()* when the problem is actually in *bar()*, for which there is no unit test.

Metrics involved: *Directness*.

Interpretation: The higher the directness, the higher the effectiveness of the test code.

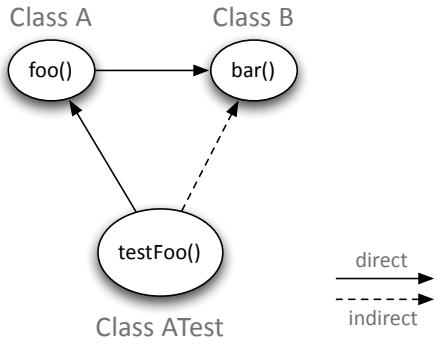

Figure 3.1: Direct and indirect code coverage

To answer **Q3**, we base our approach on the already established maintainability model that was developed by SIG [34, 8].

3.1.3 Metrics

The metrics that were selected as indicators of test code quality are defined and described as follows.

Code Coverage

Code coverage is the most used metric for test code quality assessment. Most of the times it is the only metric that is used in order to monitor testing. As mentioned previously, there are various ways to calculate code coverage. There is a variety of dynamic code coverage estimation tools (e.g. Clover[1] and Cobertura[2] for Java, Testwell CTC++[3] for C++, NCover[4] for C#, etc.). Any of these tools can be used to obtain a code coverage estimation.

[1] http://www.atlassian.com/software/clover/
[2] http://cobertura.sourceforge.net/
[3] http://www.testwell.fi/ctcdesc.html
[4] http://www.ncover.com/

The aforementioned tools use a dynamic analysis approach in order to estimate code coverage. Dynamic analysis has two main disadvantages. First, the analyser must be able to compile the source code. This is an important drawback both in the context of this study and in the intended context of application. In this study an experiment is performed where the model is applied to a number of open source projects. Compiling the source code of open source systems can be very hard due to missing libraries or because a special version of a compiler is necessary. Furthermore, application in the context of industrial systems' evaluation by an independent third party would be difficult because a working installation of the assessed system is rarely available [5]. Second, dynamic analysis requires execution of the test suite, a task that is possibly time consuming.

Alves and Visser [5] developed a code coverage estimation tool that is based only on the static analysis of the source code. In summary, the tool is based on slicing the static call graphs of Java source code and tracking the calls from methods in the test code to methods in the production code. A production code method that is called directly or indirectly (the method is called by another production code method, which in turn is called directly or indirectly by some test code method) is considered covered. The final coverage percentage is calculated by measuring the percentage of covered lines of code, where it is assumed that in a covered method all of its lines are covered. Validation of the technique was performed in [5] by assessing the correlation of the static estimation of code coverage with the dynamic coverage (using Clover), revealing significant strong correlation at the system level. This approach is used to obtain a code coverage metric in the proposed test code quality model.

Assertions-McCabe Ratio

The *Assertions-McCabe ratio* metric indicates the ratio between the number of the actual points of testing in the test code and of the decision points in the production code. The metric is defined as follows:

$$Assertions\text{-}McCabe\ Ratio = \frac{\#assertions}{cyclomatic\ complexity}$$

where *#assertions* is the number of assertion statements in the test code and *cyclomatic complexity* is the aggregated McCabe's cyclomatic complexity metric [56] for the whole production code.

Assertion Density

Assertion density aims at measuring the ability of the test code to detect defects in the parts of the production code that it covers. This could be measured as the actual testing value that is delivered given a certain testing effort. The actual points where testing is delivered are the assertion statements. At the same time, an indicator for the testing effort is the lines of test code. Combining these we define *assertion density* as follows:

$$Assertion\ Density = \frac{\#assertions}{LOC_{test}}$$

where *#assertions* is the number of assertion statements in the test code and LOC_{test} is lines of test code.

Directness

As explained in 3.1, when effective test code detects a defect it should also be providing the developers with the location of that defect in order to facilitate the fixing process. Towards that purpose, adequate unit testing is desired. When each unit is tested individually by the test code, a broken test that corresponds to a unit immediately makes the developers aware that a defect exists in the functionality of that particular unit. *Directness* measures the extent to which the production code is covered directly, i.e. the percentage of code that is being called directly by the test code.

In order to measure *directness*, the static code coverage estimation tool of Alves and Visser [5] was modified so that it can provide the percentage of the code that is directly called from within the test code.

Maintainability

As a measurement of the maintainability of test code, various metrics are used and combined in a model which is based on the SIG quality model (see Section 2.3). The SIG quality model is an operational implementation of the maintainability characteristic of the software quality model that is defined in the ISO/IEC 9126 [1]. The SIG quality model was designed to take into consideration the maintainability of production code. For convenience, the SIG quality model is presented again in Figure 3.2.

ISO 9126 maintainability \ properties	Volume	Duplication	Unit size	Unit complexity	Unit interfacing	Module coupling
Analysability	×	×	×			
Changeability		×		×		×
Stability					×	×
Testability			×	×		

Figure 3.2: The SIG quality model

However, there are certain differences between production and test code in the context of maintenance. In order to better assess the maintainability of test code, the SIG quality model was modified into the test code maintainability model which is presented in Figure 3.3. In the rest of this section, we discuss the design decisions that were considered while modifying the maintainability model. The relevance of each one of the sub-characteristics

and the system properties of the model to test code quality is evaluated. Furthermore, *test code smells* [59] are considered during the process of adjusting the maintainability model so that the metrics of the model capture some of the essence of the main code smells.

Test Code Maintainability	properties: duplication	unit size	unit complexity	unit dependency
analysability		×	×	
changeability	×		×	×
stability	×			×

Figure 3.3: The test code maintainability model as adjusted from the SIG quality model [8]

As explained in detail in Section 2.3, the model has 4 sub-characteristics: analysability, changeability, stability and testability. Within the context of test code, each of the sub-characteristics has to be re-evaluated in terms of its meaningfulness.

Analysability is "the capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified" [1]. Test code is also analysed when necessary both for verifying that it performs the desired functionality and for comprehending what should be modified when the tests have to be adjusted to changes in the system.

Changeability is "the capability of the software product to enable a specified modification to be implemented" [1]. Changes in the test code are often necessary when changes in the requirements lead to changes in the system [62, 58].

Stability is "the capability of the software product to avoid unexpected effects from modifications in the software" [1]. Tests can start failing because of modifications in utility test code or because of changes in parts of the production code on which the tests depend.

Testability is "the capability of the software product to enable modified software to be validated" [1]. This would mean that it should be easy to verify that test code is correctly implemented.

Analysability, changeability and stability are clearly aspects of test code maintainability. On the other hand, testability, although applicable, implies that a new step of verification exists for testing the test code. This could create an infinite recursion. Therefore, for practical reasons it was chosen to omit testability as a sub-characteristic of the maintainability model for test code.

After the sub-characteristics of the model have been defined, the system properties have to be re-evaluated and mapped to the sub-characteristics. The system properties used in the SIG quality model are volume, duplication, unit size, unit interfacing, unit complexity and module coupling.

Volume in production code influences the analysability because the effort that a maintainer has to spend to comprehend a system increases as the volume of the system increases. There is an important difference between the maintenance of test code and production code: maintenance of test code is performed locally, on the piece of test code that is currently under a maintenance task. This is happening because of the very low coupling that exists among test code. In practice, most of the times, in test code written using xUnit frameworks a test is self-contained in a method or function. Understanding the test might require analysing the production code that is being tested, but this is covered by assessing the analysability of the production code. Therefore, the volume of the test code does not directly influence its analysability. In addition, the volume of test code is depending on the volume of the production code. This means that bigger systems require more test code to be tested adequately. For these reasons, we chose not to use volume of test code as an indicator of its maintainability.

Test code duplication occurs when *copy-paste* is used as a way to reuse logic. This results in many copies of the same code, a fact that may significantly increase the test maintenance cost. Test code duplication is identified as a *code smell* by Meszaros in [59]. Duplication affects changeability, since it increases the effort that is required when changes need to be applied to all code clones. It also affects stability, since the existence of unmanaged code clones can lead to partially applying a change to the clones, thus introducing logical errors in the test code.

As unit size increases, it becomes harder to analyse. Unit size could be a warning for the *Obscure Test* code smell [59]. An obscure test is hard to understand. The consequences are that such a test is harder to maintain and it does not serve as documentation. One of the causes of the smell is that a test is eager to test too much functionality. Unit size can be an indication of this.

Unit interfacing seems to be irrelevant in the context of test code. Most of the test code units have no parameters at all. Utility type methods or functions exist, but are the minority of the test code.

Unit complexity on the other hand, is something that should be kept as low as possible. As mentioned above, in order to avoid writing tests for the test code, the verification of the test code should be so simple that tests are not necessary. This is also underlined in the description of the *Conditional Test Logic* code smell in [59]. High unit complexity is therefore affecting both the analysability and the changeability of the test code.

Module coupling measures the coupling between modules in the production code. In the context of test code, the coupling is minimal as it was previously discussed. Nevertheless, there is a different kind of coupling that is interesting to measure. That is the coupling between the test code and the production code that is tested.

Of course, in integration tests it is expected that the coupling is high, since a lot of parts of the system are involved in the tests. In unit testing, ideally every test unit tests one production unit in isolation. In many cases, additional units of the production code must be called in order to bring the system in an appropriate state for testing something in particular. In object oriented programming for instance, collaborative objects need to be instantiated in order to test a method that interacts with them. A solution to avoid this coupling is the use of test doubles, such as stubs and mock testing (see [59] for more details).

In order to measure the dependence of a test code unit to production code, a new metric is proposed: the number of unique outgoing calls from a test code unit to production code units. This metric is mapped to a new system property which is named *unit dependency*. Unit dependency affects the changeability and the stability of the test code. Changeability is affected because changes in a highly coupled test are harder to be applied since all the dependencies to the production code have to be considered. At the same time, stability is affected because changes in the production code can propagate more easily to the test code and cause tests to brake (*fragile test* code smell [59]), increasing the test code's maintenance effort.

3.2 The Test Code Quality Model

After selecting the metrics, the test code quality model is presented.

Based on the GQM approach, the sub-characteristics of the model are derived from the questions **Q1**, **Q2** and **Q3**. Thus, the three sub-characteristics of test code quality are *completeness*, *effectiveness* and *maintainability*. The metrics' mapping to the sub-characteristics is done as follows: code coverage and assertions-McCabe density are mapped to completeness, assertion density and directness are mapped to effectiveness and finally, the adjusted SIG quality model combines duplication, unit size, unit complexity and unit dependency into a maintainability rating. Table 3.4 depicts the test code quality model and the mappings of the metrics to the sub-characteristics.

Test Code Quality	properties: Code Coverage	Assertions-McCabe Ratio	Assertion Density	Directness	SIG Quality Model (adjusted)
Completeness	×	×			
Effectiveness			×	×	
Maintainability					×

Figure 3.4: The test code quality model and the mapping of the system properties to its sub-characteristics

The aggregation of the properties per sub-characteristic is performed by obtaining the mean. For maintainability, this is done separately in the adjusted maintainability model (see Section 3.1.3).

The aggregation of the sub-characteristics into a final, overall rating for test code quality is done differently. The overall assessment of test code quality requires that all three of the sub-characteristics are of high quality. For example, a test suite that has high completeness but low effectiveness is not delivering high quality testing. Another example would be a test suite of high maintainability but low completeness and effectiveness. Therefore, the three sub-characteristics are not substituting each other. In order for test code to be of high quality, all three of them have to be of high quality. For this reason, a conjunctive aggregation function has to be used [12].

There are several conjunctive aggregation functions from which to choose. However, another criterion for choosing such a function is the ability to communicate it. Based on these criteria, the geometric mean was selected. The formula that is used to calculate the overall test code quality rating is as follows:

$$TestCodeQuality = \sqrt[3]{Completeness \cdot Effectiveness \cdot Maintainability}$$

3.3 Calibration

The metrics on which the test code quality model is based were calibrated in order to derive thresholds for risk categories and quality ratings. Calibration was done using benchmarking as shown in Section 2.3. The set of systems in the benchmark includes 86 commercial and open source Java systems that contained at least one JUnit test file. From the 86 systems, 14 are open source and the rest are commercial.

Table 3.1: The open source systems in the benchmark. Volume of production and test code is provided in lines of code (pLOC and tLOC respectively).

System	Version	Snapshot Date	pLOC	tLOC
Apache Commons Beanutils	1.8.3	2010-03-28	11375	21032
Apache Commons DBCP	1.3	2010-02-14	8301	6440
Apache Commons FileUpload	1.2.1	2008-02-16	1967	1685
Apache Commons IO	1.4	2008-01-21	5284	9324
Apache Commons Lang	2.5	2010-04-07	19794	32920
Apache Commons Logging	1.1.1	2007-11-22	2680	2746
Apache Log4j	1.2.16	2010-03-31	30542	3019
Crawljax	2.1	2011-05-01	7476	3524
Easymock	3.0	2009-05-09	4243	8887
Hibernate core	3.3.2.ga	2009-06-24	104112	67785
HSQLDB	1.8.0.8	2007-08-30	64842	8770
iBatis	3.0.0.b5	2009-10-12	30179	17502
Overture IDE	0.3.0	2010-08-31	138815	4105
Spring Framework	2.5.6	2008-10-31	118833	129521

Table 3.1 provides some general information on the open source systems in the benchmark. We observe that the systems production Java code volume ranges from $\sim$ 2 KLOC to $\sim$ 140 KLOC. For the commercial systems the range is entirely different: from $\sim$ 1.5 KLOC to $\sim$ 1 MLOC with several systems having more code than the largest open source system. For test code, the range for open source systems is from $\sim$ 1.7 KLOC to $\sim$ 130 KLOC lines of code. For the commercial systems test code ranges from 20 LOC to $\sim$ 455 KLOC. For both production code and test code we observe that commercial systems have a significantly wider range with several systems being in a different order of magnitude than the largest open source system. Further information about the commercial systems cannot be published due to confidentiality agreements.

Table 3.2 summarizes descriptive statistics for the metrics. For the system level metrics, we observe that they cover a large range starting from values that are close to zero. In combination with Figure 3.5, where histograms and box-plots are used to illustrate the distributions of the system level metrics, we see that there are systems whose metrics have very small values. This is the reason that the 2-star rating threshold for some of the metrics is low, as shown in Table 3.3. For example, 0.6% code coverage is enough for a system to be rated 2-stars, even though intuitively such levels of code coverage are very low. This reflects the quality of the systems in the benchmark, indicating that at least 5% of the systems are tested inadequately.

Table 3.2: Metrics and their summary statistics

Metric	**Scope**	**Min**	**Q1**	**Median**	**Mean**	**Q3**	**Max**	**STDV**
Code Coverage	System	0.1%	29.8%	45.9%	44.1%	60.8%	91.8%	22.3%
Assertions-McCabe Ratio	System	0.001	0.086	0.270	0.372	0.511	1.965	0.371
Assertion Density	System	0.0%	5.9%	8.4%	9.1%	12.0%	36.4%	5.8%
Directness	System	0.06%	8.0%	21.0%	23.6%	36.6%	71.0%	18.6%
Duplication	System	0.0%	9.6%	12.2%	13.3%	18.6%	23.9%	5.7%
Unit Size	Unit	1	8	15	23.3	27	631	30.8
Unit Complexity	Unit	1	1	1	1.9	2	131	3.07
Unit Dependency	Unit	1	1	2	3.05	4	157	3.70

Table 3.3: Thresholds for system level metrics

Metric	⋆⋆⋆⋆⋆	⋆⋆⋆⋆	⋆⋆⋆	⋆⋆	⋆
Code Coverage	73.6%	55.2%	40.5%	0.6%	-
Assertions-McCabe Ratio	1.025	0.427	0.187	0.007	-
Assertion Density	18.9%	10%	7.2%	1.5%	-
Directness	57.4%	28.5%	12.3%	0.29%	-
Duplication	05.5%	10.3%	16.4%	21.6%	-

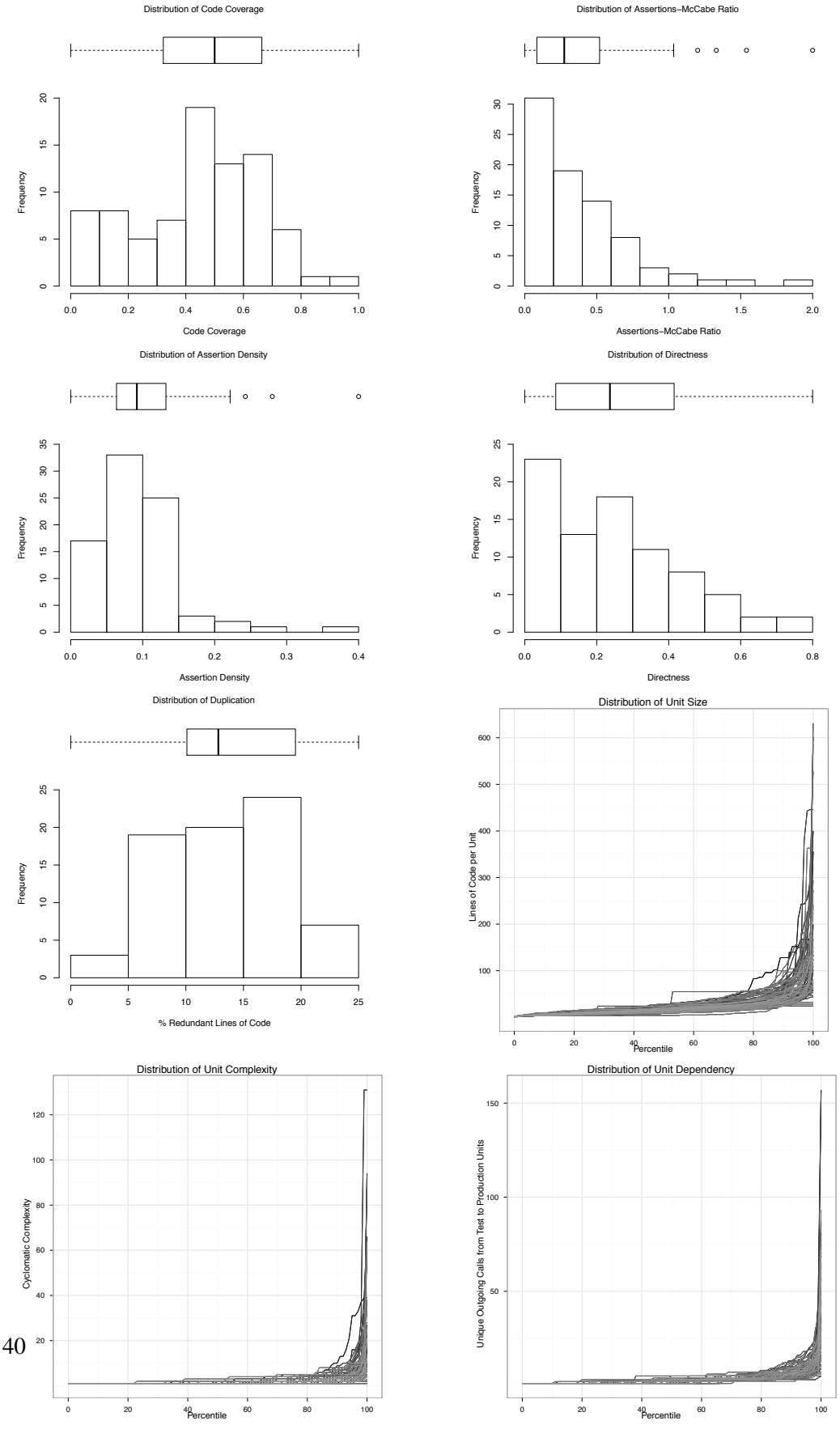

Figure 3.5: The distributions of the metrics

Table 3.4: Thresholds for unit level metrics

Metric	**Low Risk**	**Moderate Risk**	**High Risk**	**Very High Risk**
Unit Size	24	31	48	> 48
Unit Complexity	1	2	4	> 4
Unit Dependency	3	4	6	> 6

Table 3.5: Profile thresholds for Unit Size, Unit Complexity and Unit Dependency

(a) Profile thresholds for Unit Size

	maximum relative volume		
rating	moderate	high	very high
⋆⋆⋆⋆⋆	12.3%	6.1%	0.8%
⋆⋆⋆⋆	27.6%	16.1%	7.0%
⋆⋆⋆	35.4%	25.0%	14.0%
⋆⋆	54.0%	43.0%	24.2%
⋆	-	-	-

(b) Profile thresholds for Unit Complexity

	maximum relative volume		
rating	moderate	high	very high
⋆⋆⋆⋆⋆	11.2%	1.3%	0.3%
⋆⋆⋆⋆	21.6%	8.1%	2.5%
⋆⋆⋆	39.7%	22.4%	9.9%
⋆⋆	62.3%	38.4%	22.1%
⋆	-	-	-

(c) Profile thresholds for Unit Dependency

	maximum relative volume		
rating	moderate	high	very high
⋆⋆⋆⋆⋆	10.0%	4.3%	1.2%
⋆⋆⋆⋆	19.3%	13.9%	7.8%
⋆⋆⋆	33.5%	24.1%	14.6%
⋆⋆	52.1%	38.9%	24.1%
⋆	-	-	-

Code coverage ranges up to $\sim 92\%$, with a large group of systems ranging between 40% and 70%. The median is only $\sim 46\%$, which means that only half of the systems in the benchmark have test code that covers at least 46% of the system. Assertions related metrics as well as directness appear to be skewed with most of the systems having a very low value in both of the metrics.

The unit level metrics resemble a power-law-like distribution. The summary statistics in Table 3.2 as well as the quantile plots in Figure 3.5 show that most of the values of these metrics are low but there is a long tail of much higher values towards the right. This

confirms the behaviour of software metrics that has been seen in previous studies [24, 6]. We can also observe that there is higher variation in unit size and unit complexity than unit dependency.

An interesting observation about the behaviour of cyclomatic complexity in test code can be seen in Table 3.4. The thresholds are 1 for low risk, 2 for moderate and 4 for high risk, with units that have complexity higher than 4 being already in very high risk. The thresholds lie considerably lower than the corresponding ones for production code (see Table 2.7, Section 2.3), confirming the lower leniency against complexity in test code.

The thresholds for the system level metrics are shown in Table 3.3. For the unit level metrics, Table 3.4 summarizes the thresholds for the risk categories and Tables 3.5 (a), (b) and (c) present the thresholds for their quality profiles. The interpretation of the model's thresholds is the same as in the SIG quality model and it is explained in Section 2.3. Furthermore, linear interpolation is used in the same way, in order to be able to compare systems within the same quality level.

Chapter 4

Design of Study

In this chapter, the design of the study is discussed. In order to answer *RQ2*, an experiment is conducted. In the first part of the chapter, the experiment's goal, subjects and materials, hypotheses formulation, measured variables, and methods used are described in order to provide an insight into the approach followed in the study. Furthermore, *RQ3* is addressed by performing a series of case studies. The design of the case studies is analysed in the second part of the chapter.

4.1 Design of the Experiment

Based on the GQM template [91], the goal of the experiment is formulated as follows:

Analyse the quality of codified testing
for the purpose of investigating its relation
with respect to issue handling performance indicators
from the perspective of the researcher
in the context of a set of open source Java projects

As stated in *RQ2* in chapter 1, the main goal of the study is to assess the relation between test code quality and issue handling performance. In order to answer that question, subsidiary questions were formed. The questions are repeated below:

- **RQ2.1** : Is there a correlation between the test code quality ratings and the defect resolution time?
- **RQ2.2** : Is there a correlation between the test code quality ratings and the throughput of issue handling?
- **RQ2.3** : Is there a correlation between the test code quality ratings and the productivity of issue handling?

The design of the experiment with which we attempt to answer these questions is discussed in the following sections.

4.2 Hypotheses Formulation

In *RQ2.1*, *RQ2.2* and *RQ2.3*, we aim at investigating the relation between test code quality and defect resolution time, throughput and productivity. We use the test code quality model that was presented in Chapter 3 as a measurement of test code quality. We extract data from ITSs of several open source Java projects in order to retrieve measurements for the three issue handling indicators.

As seen in Section 1.2, we assume that systems of higher test code quality will have faster defect resolution times, and higher throughput and productivity. In order to investigate whether these assumptions hold, we assess whether there are correlations between the test code quality rating of systems and the three issue handling indicators.

We translate the three questions into three hypotheses:

Hypothesis 1

- **Null hypothesis ($\mathbf{H1}_{null}$)** : There is no significant correlation between test code quality and defect resolution time.
- **Alternative hypothesis ($\mathbf{H1}_{alt}$)** : Higher test code quality decreases defect resolution time.

Hypothesis 2

- **Null hypothesis ($\mathbf{H2}_{null}$)** : There is no significant correlation between test code quality and throughput.
- **Alternative hypothesis ($\mathbf{H2}_{alt}$)** : Higher test code quality increases throughput.

Hypothesis 3

- **Null hypothesis ($\mathbf{H3}_{null}$)** : There is no significant correlation between test code quality and productivity.
- **Alternative hypothesis ($\mathbf{H3}_{alt}$)** : Higher test code quality increases productivity.

All three hypotheses are formulated as one-tailed hypotheses because we have a specific expectation about the direction of the cause-effect relationship between the independent and the dependent variables: higher test code quality increases issue handling performance.

4.3 Measured Variables

The measured variables are summarised in Table 4.1.

The independent variable in all three hypotheses is the test code quality. In order to measure the quality of test code, we apply the test code quality model that was presented in Chapter 3. The outcome of the model is a rating that reflects the quality of the test code of the system. The ratings are in ordinal scale and the values are in the range of $[0.5, 5.5]$.

Table 4.1: Measured Variables

Hypothesis	Independent Variable	Dependent Variable
H1	Test code quality	Defect resolution speed rating
H2	Test code quality	Throughput
H3	Test code quality	Productivity

The dependent variables are defect resolution speed rating, throughput and productivity for hypotheses 1, 2 and 3 respectively. Starting from the last two, throughput and productivity are measured as shown in Section 2.2. Possible approaches in deriving the resolution time of a defect based on its life-cycle were also discussed in the aforementioned section. However, the approach that is used in this study was introduced and analysed in previous studies that explore the relation of the maintainability of production code with defect resolution [17, 50]. The summary of the approach follows.

Defect Resolution Speed Rating

The dependent variable of *Hypothesis 1* is the resolution time of defects in a system, which is measured by calculating a rating that reflects the defect resolution speed.

As discussed in Section 2.2 of Chapter 2, to measure the resolution time of a defect, the time during which the defect was in an open state in the ITS is measured. This means that all the intervals between the moment that the status of the defect was *new* and the moment it was marked as *resolved* or *closed* are taken into consideration. Furthermore, if a defect was closed and later reopened, the intervals during which the defect's status was *resolved* or *closed* are excluded.

In order to acquire a measurement of the defect resolution speed of a system's snapshot during a particular period of time, all the defects that were resolved during that period are mapped to the snapshot. The individual resolution times of the defects need to be aggregated in a measurement that represents the defect resolution speed. The distribution of defect resolution times resemble a power-law-like distribution as illustrated by Bijlsma et al. [17]. In their study, Bijlsma et al. observed that the resolution times of most of the defects were at most four weeks, but at the same time there were defects with resolution times of more than six months. Thus, aggregating the resolution times by taking the mean or the median would not be representative of the actual central tendency of the distribution.

The technique of benchmarking that was used for the construction of the SIG quality model and the test code quality model is also used in this case. This way defect resolution times can be converted into a rating that reflects resolution speed. The thresholds for the risk categories and the quality profiles that are used in this study are the ones that were acquired by the calibration that was performed in [17]. The thresholds of the risk categories are shown in Table 4.2 and the thresholds for the quality ratings are shown in Table 4.3.

As with the test code quality model, the thresholds for the risk categories are applied on the measurement of a defect's resolution time in order to classify it in a risk category. Afterwards, the percentage of defects in each risk category is calculated. Finally, the thresholds

Table 4.2: Thresholds for risk categories of defect resolution time

Category	Thresholds	
Low	0 - 28 days	(4 weeks)
Moderate	28 - 70 days	(10 weeks)
High	70 - 182 days	(6 months)
Very high	182 days or more	

Table 4.3: Thresholds for quality ratings of defect resolution time

Rating	Moderate	High	Very High
*****	8.3%	1.0%	0.0%
****	14%	11%	2.2%
***	35%	19%	12%
**	77%	23%	34%

for the quality ratings are used in order to derive a quality rating for the defect resolution speed. Interpolation is used once again in order to provide a quality rating in the range of $[0.5, 5.5]$ and to enable comparisons between systems of the same quality level. This rating is used in order to measure the dependent variable for *Hypothesis 1*. It should be noted that higher rating means faster defect resolution times.

4.3.1 Confounding Factors

In this experiment we aim at assessing the relation of test code quality to issue handling and we expect test code quality to have a positive impact. Of course, test code quality is not the only parameter that influences the performance of issue handling. There are other factors that possibly affect issue handling. The observations of the experiment can be misleading if these co-factors are not controlled. Identification and control of all the co-factors is practically impossible. However, several co-factors and confounding factors were identified and they are discussed below.

- **Production code maintainability** : While issues are being resolved, the maintainer analyses and modifies both the test code and the production code. Therefore, issue handling is affected by the maintainability of the production code.
- **Team size** : The number of developers working on a project can have a positive or negative effect on the issue handling efficiency.
- **Maintainer's experience** : The experience of the person or persons who work on an issue is critical for their performance on resolving it.
- **Issue granularity** : The issues that are reported in an ITS can be of different granularity. For example, an issue might be a bug that is caused by a mistake in a single

statement and another issue might require the restructuring of a whole module in the system. Therefore, the effort that is necessary to resolve an issue may vary significantly from issue to issue.

- **System's popularity** : High popularity of a project may lead to a larger active community that reports many issues. The issues could create pressure to the developers, making them resolve more of them.

In order to control these factors we have to be able to measure them. The maintainability of the production code is measured by applying the SIG quality model to the subject systems. Team size is measured by obtaining the number of developers that were actively committing code in a system during a period of time.

Measuring the experience of the maintainer, the granularity of issues and the system's popularity is difficult. The maintainers of open source systems are many times anonymous and there is no reliable data in order to assess their experience at the time of their contributions to the projects. As far as the granularity of issues is concerned, most ITSs offer a field of severity for each issue. However, this field is often misused making it an unreliable measurement for the granularity of the issues [15]. For the project's popularity, the potential of obtaining the number of downloads was explored but the lack of data for all subject systems was discouraging. Thus, these three factors are not controlled and will be discussed in Chapter 5 as threats to validity for the outcome of the experiment.

4.4 Data Collection and Preprocessing

In order to investigate the relation between test code quality and issue handling two kinds of data are necessary: source code and issue tracking data of systems. For this reason, open source systems were selected as the experiment's subjects. Open source systems provide their source code publicly and in some of them, the ITS data is also publicly available. Therefore, they are suitable candidates as subjects for the experiment of this study.

In order to compare the technical quality of a system with the performance in issue handling it is necessary to map the quality of the source code of a specific snapshot of the system to the issue handling that occurred in that period. For this reason we perform snapshot-based analysis. Snapshots of systems were downloaded and analysed in order to acquire quality ratings. We consider each snapshot's technical quality influential on the issue handling that occurred between that snapshot and the next one.

Certain criteria were defined and applied during the search for appropriate systems. The criteria are summarised as follows:

- The system has a publicly available source code repository and ITS.
- Systems have to be developed in Java due to limitations of the tooling infrastructure.
- More than 1000 issues were registered in the ITS within the period of the selected snapshots.

Further selection was applied in order to omit irrelevant data from the dataset. All the issues that were marked as *duplicates*, *wontfix* or *invalid* were discarded. Furthermore, issues of type *task* and *patch* were omitted. Finally, the issues that were resolved (resolution field set to *fixed* and status field set to *resolved* or *closed* were selected for participation in the experiment.

Some additional selection criteria were applied in order to mitigate the fact that snapshots of the same system are not independent. For this reason, snapshots were taken so that (1) there is a certain period of time and (2) there is a certain percentage of code churn (added and modified lines of code) between two consecutive snapshots.

4.5 Data Measurement and Analysis

This section discusses the tools that were used in order to obtain the experiment's data as well as the methods that were used in order to analyse the data.

Source code was downloaded from the repositories of the subject open source systems. The necessary metrics were calculated by using SIG's Software Analysis Tool (SAT). The test code quality model ratings were calculated by processing the metrics with the use of R scripts.

In order to obtain the issue handling performance indicators the ITSs of the subject systems were mined. In particular, the tool that was created by Luijten and was used for the studies of Luijten [51] and Bijlsma [16] was updated and used for this study. The tool supports four ITSs, namely Bugzilla, Issuezilla, Jira and Sourceforge. The data is extracted from the ITSs into XML form. In addition, the VCS history log is being extracted. Afterwards, the issue tracking data, the VCS log and the metrics of the source code are read and mapped to a unified repository database object model. Finally, the data is persisted in a database from which it can be used for further analyses. The repository database object model is shown in Figure 4.1.

Correlation tests were performed in order to test the formulated hypotheses. The ratings for test code quality and for defect resolution time are *ordinal* measurements. Therefore, a *non-parametric* correlation test is suitable. In particular, Spearman's rank-correlation coefficient [83] was chosen for the correlation tests of the study. Based on the expectation that higher test code quality decreases the defect resolution time, the hypotheses are directional and thus one-tailed tests are performed.

In addition, in order to assess the influence of the confounding factors to the issue handling indicators we performed multiple regression analyses. The multiple regression analysis indicates the amount of variance in the dependent variable that is uniquely accounted for by each of the independent variables that are combined in a linear model. We want to select the independent variables that have significant influence on the dependent variable. For that reason we use stepwise selection. In particular, we apply the backward elimination model, according to which one starts assessing the significance of all the independent variables and iteratively removes the least significant until all the remaining variables are significant [68].

An overview of the experiment's procedure is shown in Table 4.2.

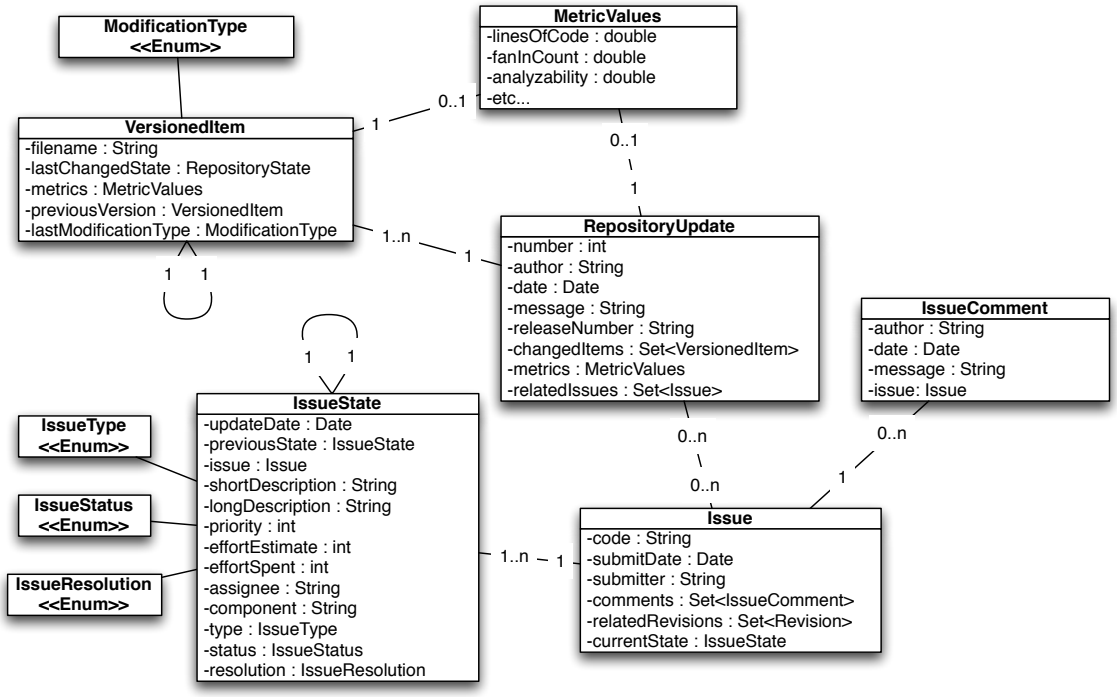

Figure 4.1: Repository database object model (taken from [49])

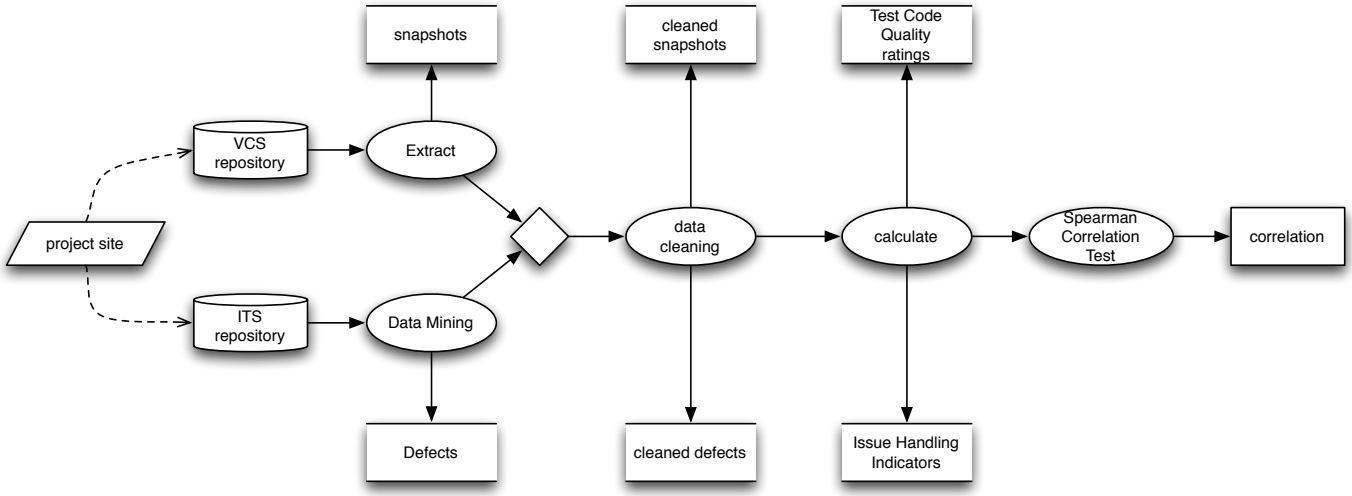

Figure 4.2: Procedure Overview

4.6 Design of the Case Studies

After developing the test code quality mode, it is important to assess the usefulness of its application. Ideally, the model provides an overview of the strengths and weaknesses

of the test code of a system and enables the monitoring of the test code quality during development. *RQ3* is addressing this matter. In particular, we chose to approach *RQ3* by studying the alignment of the model's results to experts' evaluations. For this reason, three case studies are performed. The design of the case studies is discussed in the section.

Each case study is performed in three steps. First, the system's source code is analysed and the test code quality model is applied. Afterwards, the results of the model are interpreted in order to summarise the knowledge that was derived from the model about the quality of the system's test code. Finally, an expert on the system's technical aspects is interviewed and the alignment of the expert's evaluation with the model's results is assessed.

Three systems were selected as subjects of the case studies based on the availability of the source code and the expert. All of the systems are commercial and thus, they are kept anonymous, as well as the experts that were interviewed.

The interview with the expert for each system follows the *focused interview* paradigm [94]. The interviews aim at asking questions to the expert in order to obtain a solid evaluation of the test code quality of the systems under study. The first questions were system specific and their goal was to acquire some general information about the system. Afterwards, the interviewee was asked to answer to a structured stream of questions during approximately one hour. The questions are presented below in the order they were asked:

- How completely is the system tested?
- How effectively is the system tested?
- How maintainable is the system's test code?
- To which extent is the test code targeting unitary or integration testing?
- How would you rate the aspects of the test code quality of the system?

The last question was asking the experts to provide ratings for the sub-characteristics of the test code quality model as well as the overall test code quality. The ratings were explicitly asked to be provided in the range from 0.5 to 5.5 with the unit of measurement being 0.5. Therefore, the next higher rating of 4.0 is 4.5.

It is important to note the order in which the parts of the interview took place. In order to avoid introducing bias in the experts' evaluations, the experts had no knowledge about the test code quality model while they were answering the questions. After the questions were answered the model's results were presented to the interviewee together with the logic behind the model. Finally, there was an open discussion on the results and the reasons behind the discrepancies between the model's ratings and the expert's evaluation.

The interviews were not recorded and the interviewer kept track of them by taking personal notes.

Chapter 5

The Relation between Test Code Quality and Issue Handling Performance

In order to address *RQ2* an experiment was conducted aiming at assessing the relation between test code quality and issue handling performance. This chapter presents the results of the experiment. Furthermore, the interpretation of the results and the threats to their validity are discussed.

5.1 Data

In Section 4.4, criteria were defined to guide the search of suitable systems as well as selection and cleaning criteria for preprocessing the data. The search for suitable systems led to 18 open source systems. Among the selected systems there are very well-known and researched ones (notably ArgoUML and Checkstyle), and also systems that were in the list of the top-100 most popular projects in SourceForge, even though they are not so well known.

The total number of snapshots that were collected is 75. All systems have non-trivial size, with Stripes Framework being the smallest (17 KLOC in the most recent snapshot) and ArgoUML being the largest (163 KLOC). The total number of collected issues for the 18 systems is almost $160,000$, where about $110,000$ are defects and $49,000$ are enhancements. An overview of the dataset is shown in Table 5.1.

5.1.1 Data Selection and Cleaning

Data were selected and cleaned based on the criteria that were previously defined (see Section 4.4). The selection criteria included the existence of a sufficient time interval and a percentage of code churn between two consecutive snapshots. The actual thresholds for these criteria were defined by experimenting in order to find the balance between the austerity for snapshot independence and the preservation of sufficient amount of data. This resulted in selecting snapshots so that between two consecutive snapshots (1) the time interval is at least one year and (2) code churn is at least 30%.

Project	Snap-shots	Earliest Snapshot Date	Latest Snapshot Date	KLOC (latest)	Devel-opers (max)	Issues	Defects	Enhanc-ements	pCode Main-tain-ability (latest)	tCode Quality (latest)
Apache Ant	7	18/07/2000	13/03/2008	100	17	25,474	17,647	7,827	3.201	2.693
Apache Ivy	3	17/12/2006	26/09/2009	37	6	3,688	2,064	1,624	3.022	3.330
Apache Lucene	4	02/08/2004	06/11/2009	82	19	36,874	32,287	4,587	2.795	2.917
Apache Tomcat	3	21/10/2006	07/03/2009	159	13	2,160	1,851	309	2.531	1.595
ArgoUML	7	12/03/2003	19/01/2010	163	19	10,462	8,093	2,369	2.915	2.733
Checkstyle	6	05/02/2002	18/04/2009	47	6	5,109	2,676	2,433	3.677	2.413
Hibernate code	3	18/04/2005	15/08/2008	105	14	10,547	6,850	3,697	2.934	2.484
HSQLDB	6	06/10/2002	09/09/2009	69	8	6,347	5,029	1,318	2.390	2.039
iBatis	4	16/05/2005	12/10/2009	30	4	2,620	1,544	1,076	2.999	2.868
JabRef	4	28/11/2004	02/09/2009	83	17	4,500	3,151	1,349	2.574	2.727
jMol	2	06/06/2006	10/12/2007	92	9	1,213	923	290	2.208	1.814
log4j	4	17/05/2002	05/09/2007	21	6	3,592	2,949	643	3.966	2.365
OmegaT	3	20/06/2006	12/02/2010	112	6	3,433	1,530	1,903	3.278	2.448
PMD	5	14/07/2004	09/02/2009	35	15	6,457	3,863	2,594	3.865	2.975
Spring framework	4	13/05/2005	16/12/2009	145	23	28,492	13,870	14,622	3.758	3.152
Stripes framework	3	29/09/2006	28/10/2009	17	6	2,364	1,208	1,156	3.704	3.123
Subclipse	4	11/04/2006	11/08/2009	93	10	3,370	2,502	868	2.348	2.449
TripleA	3	21/07/2007	06/03/2010	99	4	2,808	2,261	547	2.493	2.449
18	75			1489		159,510	110,298	49,212		

Table 5.1: General information about the data set and the number of snapshots and issues per system before selection and cleaning. The issues' numbers are the total reported issues during the period that is covered by the snapshots.

In addition, data cleaning was necessary in order to remove inconsistencies in the way the ITSs were used. Manual inspection of a sample of the data revealed that there are occasions where large numbers of issues are closed within a short period of time. This happens because the developers decide to clean the ITS by removing issues whose actual status changed but it was not updated accordingly in the system. For instance, an issue was resolved but the corresponding entry in the ITS was not updated to *resolved*. We remove such issues automatically by identifying groups of 50 or more of them that were closed on the same day and with the same comment.

A final step of data cleaning occurred by removing snapshots with less than 5 resolved defects for the hypothesis related to defect resolution speed, and with less than 5 resolved issues for the hypotheses related to throughput and productivity. This cleaning step leaves us with two different datasets: one for defect resolution speed and another for throughput and productivity.

The reduction was significant and is shown in Table 5.2.

5.2 Descriptive Statistics

Before the results are presented, descriptive statistics of the measured variables are presented. Test code and ITSs of 75 snapshots belonging to 18 open source systems were analysed. Figure 5.1 shows the distributions of the properties of the test code quality model.

It is interesting to observe that for code coverage, assertions-McCabe ratio, assertion

Project	Data for Defect Resolution Speed				Data for Throughput & Productivity			
	Snapshots	Issues	Defects	Enhancements	Snapshots	Issues	Defects	Enhancements
Apache Ant	6	2,275	1,680	595	5	1,944	1,467	477
Apache Ivy	2	331	228	103	2	467	309	158
Apache Lucene	3	2,222	1,547	675	4	4,092	3,274	818
Apache Tomcat	2	295	268	27	2	275	244	31
ArgoUML	7	758	635	123	6	621	508	113
Checkstyle	6	251	248	3	4	203	200	3
Hibernate code	2	270	166	104	3	999	620	379
HSQLDB	4	295	295	0	3	356	354	2
iBatis	3	266	150	116	3	266	150	116
JabRef	4	480	480	0	4	480	480	0
jMol	2	64	63	1	2	64	63	1
log4j	4	384	323	61	2	101	86	15
OmegaT	3	353	192	161	3	353	192	161
PMD	4	176	153	23	3	98	77	21
Spring framework	3	5,942	2,947	2,995	1	3,829	1,923	1,906
Stripes framework	3	340	197	143	2	317	179	138
Subclipse	3	156	95	61	3	190	113	77
TripleA	2	204	204	0	2	187	187	0
18	63	15,062	9,871	5,191	54	14,842	10,426	4,416

Table 5.2: Snapshots and issues per system after selection and cleaning

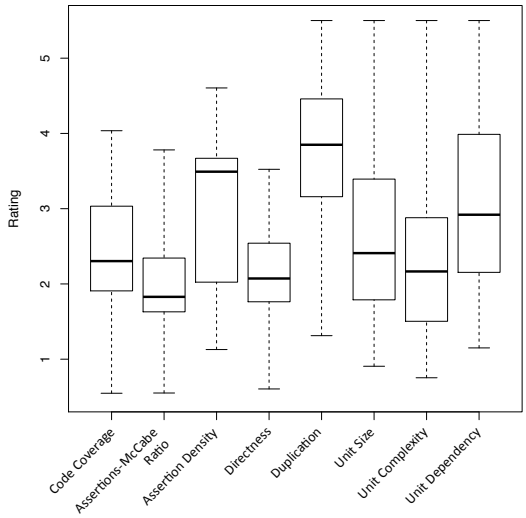

Figure 5.1: Boxplots of the ratings of the test code quality model's properties

density and directness the vast majority of the systems is rated below 4.0. This does not apply for the properties that are related to the test code maintainability where we can see the ranges of the ratings to be wider. The subject systems seem to perform well in duplication. Half of the snapshots were rated approximately 4.0 and above. On the other hand, the systems do not perform well in assertions-McCabe ratio and directness, where more than 75% of the snapshots received a rating that was less than 3.0. In particular, the median in these two properties is approximately 2.0, when within the systems of the benchmark the

median is by definition greater than 2.5. Finally, it is interesting to observe that there are a few snapshots that are rated very low in code coverage, revealing that some of the snapshots have almost zero code coverage.

Figure 5.2 shows the distributions of the ratings of the model's sub-characteristics and the overall test code quality. Overall, test code quality was rated from ~ 1.5 to ~ 3.5, which means that the spread between the test code quality of the snapshots was rather small. This is a potential limitation for our study because we cannot generalise our findings for systems that would receive a higher rating than 3.5. We observe that none of the snapshots was rated higher than 4.0 for completeness and effectiveness. In contrast, there are snapshots of very high quality with regard to maintainability.

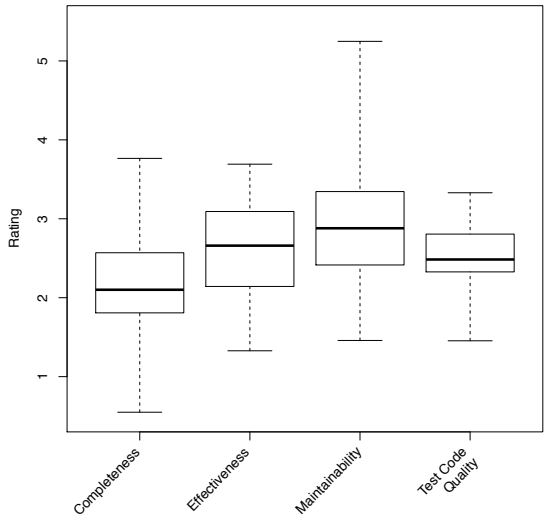

Figure 5.2: Boxplots of the ratings of the test code quality model's sub-characteristics and overall test code quality

Next, descriptive statistics about the dependent variables are presented. Figure 5.3 show the distribution of the ratings for defect resolution speed and Table 5.3 summarises statistics for throughput and productivity.

The ratings for defect resolution speed cover the whole range of the model's scale. However, at least 75% of the snapshots is rated less than 3.0.

Throughput has a median of 0.13 and a mean of 0.39. We observe that the maximum value (5.53) is in a different order of magnitude. Further investigation reveals that the highest values in throughput belong to snapshots of different systems (i.e. Apache Ant 1.1, Apache Lucene 1.4.1 and Spring Framework 3.0). Manual inspection of a sample of their issues did not reveal any peculiarity that would justify considering these snapshots as outliers.

The median for productivity is 0.99, meaning that half of the developers are resolving at least one issue per month. In the fourth quartile we observe that, similarly to throughput, productivity is in a different order of magnitude. Again, no justification for considering the snapshots as outliers could be found.

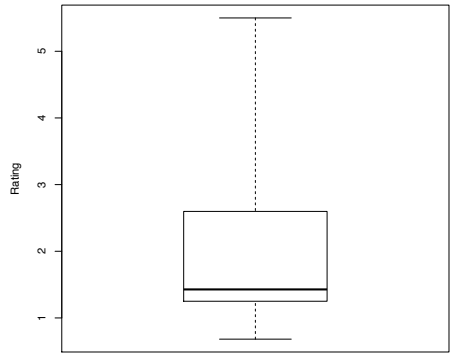

Figure 5.3: Boxplot of the defect resolution speed ratings

Table 5.3: Descriptive statistics for the dependent variables throughput and productivity

Metric	Min	Q1	Median	Mean	Q3	Max	STDV
Throughput	0.02	0.06	0.13	0.39	0.42	5.53	0.81
Productivity	0.12	0.50	0.99	1.77	1.68	14.54	2.72

5.3 Results of the Experiment

As discussed in Chapter 4, the relation between test code quality and issue handling performance is assessed by testing three hypotheses. In particular, we test whether there is correlation between test code quality and three issue handling performance indicators, namely defect resolution speed, throughput and productivity. For each of these tests, a Spearman correlation test was performed. In addition, correlation results between test code quality and production code maintainability are presented in order to investigate the relation between them as mentioned in Chapter 1. Table 5.4 shows the results of the correlation tests.

	ρ	*p*-value	N
Defect Resolution Speed	0.06	0.330	63
Throughput	0.50	**0.000**	54
Productivity	0.51	**0.000**	54
pCode Maintainability	0.42	**0.000**	75

Table 5.4: Summary of correlations with the test code quality rating of the systems

All the correlations are significant in the 99% confidence level except from the correlation between test code quality and defect resolution speed. No significant correlation was found in that case. Therefore we cannot reject the hypothesis $\mathbf{H1}_{null}$: there is no significant correlation between test code quality and defect resolution speed. Throughput and productivity have significant strong correlations with test code quality. The correlation coefficient is 0.50 and 0.51 for throughput and productivity respectively in the 99% confidence level. This enables us to reject $\mathbf{H2}_{null}$ and $\mathbf{H3}_{null}$ and maintain the alternative hypotheses: there are significant positive correlations between test code quality and throughput, and test code

quality and productivity. With regard to the correlation between test code quality and production code maintainability, significant correlation of moderate strength was revealed in the 99% confidence level.

Even though only the correlations between the overall test code quality rating and each of the issue handling performance indicators are required to test the formulated hypotheses, we present the correlations between all the underlying levels of the test code quality model and the issue handling indicators in order to acquire an indication of which aspects of test code are particularly influential on issue handling performance.

5.3.1 Hypothesis 1 : The relation between test code quality and defect resolution speed

Table 5.5 presents the correlation between the test code quality model's ratings and the defect resolution speed rating for the subject snapshots. None of the correlations is significant except from the correlation between code coverage and defect resolution speed. However, code coverage is weakly correlated with defect resolution speed ($\rho = 0.28$).

	Defect Resolution Speed (N=63)	
	ρ	p-value
Code Coverage	0.28	**0.013**
Assertions-McCabe Ratio	0.01	0.480
Assertion Density	0.02	0.427
Directness	0.08	0.260
Duplication	−0.45	0.999
Unit Size	−0.11	0.800
Unit Complexity	−0.09	0.747
Unit Dependency	−0.17	0.905
Completeness	0.12	0.182
Effectiveness	0.07	0.282
Maintainability	−0.29	0.989
Test Code Quality	0.06	0.330

Table 5.5: Correlation results for defect resolution speed.

5.3.2 Hypothesis 2 : The relation between test code quality and throughput

Table 5.6 presents the correlation between the test code quality model's ratings and throughput for the subject snapshots. On the level of the model's properties, we can differentiate the results between the properties that relate to completeness and effectiveness, and the properties that relate to maintainability. Code coverage, assertions-McCabe ratio, assertion density and directness are all significantly correlated with throughput. The higher correlation is between throughput and the assertions-McCabe property ($\rho = 0.48$ and p-value $\ll 0.01$).

	Throughput (N=54)	
	ρ	*p*-value
Code Coverage	0.28	**0.021**
Assertions-McCabe Ratio	0.48	**0.000**
Assertion Density	0.29	**0.017**
Directness	0.31	**0.012**
Duplication	0.10	0.246
Unit Size	0.06	0.330
Unit Complexity	0.06	0.321
Unit Dependency	0.10	0.236
Completeness	0.42	**0.001**
Effectiveness	0.41	**0.001**
Maintainability	0.10	0.244
Test Code Quality	0.50	**0.000**

Table 5.6: Correlation results for throughput

On the sub-characteristics level, completeness and effectiveness have similar, significant correlations with throughput. Maintainability is not significantly correlated with throughput. Finally, overall test code quality is significantly correlated with throughput in the 99% confidence level. In fact, it has the highest correlation coefficient ($\rho = 0.50$).

5.3.3 Hypothesis 3 : The relation between test code quality and productivity

Table 5.7 presents the correlation between the test code quality model's ratings and productivity for the subject snapshots. We observe that the correlations behave similarly to those with throughput. Code coverage, assertions-McCabe ratio, assertion density and directness are significantly correlated with productivity. The completeness related properties appear to have a stronger correlation with productivity than the effectiveness related ones. The properties that are related to test code maintainability are not significantly correlated to productivity.

On the sub-characteristics level of the model, completeness's correlation with productivity is the highest ($\rho = 0.56$ and p-value $\ll 0.01$). Effectiveness is also significantly correlated with productivity. Once again, maintainability lacks correlation with productivity. The overall rating of test code quality has a significant correlation with productivity ($\rho = 0.51$ and p-value $\ll 0.01$).

5.4 Interpretation of the Results

Contrary to our expectations, test code quality was not found to be significantly correlated with defect resolution speed in the conducted experiment. None of the model's properties was correlated with defect resolution speed except for code coverage, which was weakly

	Productivity (N=54)	
	ρ	*p*-value
Code Coverage	0.49	**0.000**
Assertions-McCabe Ratio	0.53	**0.000**
Assertion Density	0.33	**0.007**
Directness	0.36	**0.004**
Duplication	−0.13	0.827
Unit Size	−0.09	0.747
Unit Complexity	−0.08	0.719
Unit Dependency	−0.20	0.927
Completeness	0.56	**0.000**
Effectiveness	0.49	**0.000**
Maintainability	−0.24	0.957
Test Code Quality	0.51	**0.000**

Table 5.7: Correlation results for productivity

correlated. Of course, absence of significant correlation in the experiment we performed does not mean that there is no correlation. Further replications of the experiment have to be conducted in order to be able to draw a reliable conclusion. However, a close examination of the process of software development might provide a hint in order to explain this result.

During development changes are applied to the production code in order to implement new features, fix defects or refactor the current code. One of the main ideas behind automated tests is that after a change the tests are executed, in order to make sure that the change did not cause any test to fail. In the scenario where the execution of the tests results in a failed test, the developer will realise that his change introduced some problem in the system. Thus, the developer will re-work on the source code, in order to make sure that his change does not make any test to fail. This is how automated tests prevent defects to appear in the system.

At the same time, we observe that a defect that is reported in an ITS is probably a defect that was not detected by the test code. Therefore, the resolution speed of defects that lie in ITSs should not be expected to be influenced by the test code of the system. This is one possible reason why no significant correlation was found between test code quality and defect resolution speed. Another reason would be the limited reliability of the ITS data. As it is discussed in Section 5.6, issues may be have been resolved earlier than the moment they were marked as closed, or not marked as closed at all [15].

On the other hand, throughput and productivity confirm our expectation that they are related to the quality of test code. Figures 5.4 and 5.5 compare throughput and productivity with the different levels of test code quality. In particular, the snapshots were grouped in star ratings according to their test code quality rating. Snapshots with ratings between 0.5 and 1.5 are one star, 1.5 and 2.5 two star, and so on. Unfortunately, our dataset has no snapshots with test code quality that is above 3 stars (> 3.5). The extreme values depicted

in Figures 5.4 and 5.5 as circles are not considered outliers due to the lack of evidence after manual inspection as discussed in Section 5.2.

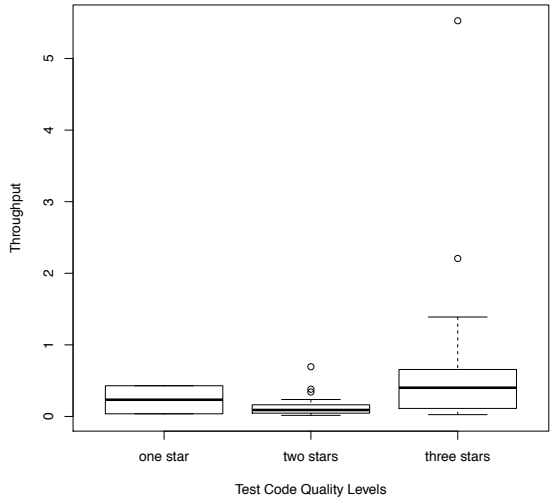

Figure 5.4: Comparison between throughput and different test code quality levels

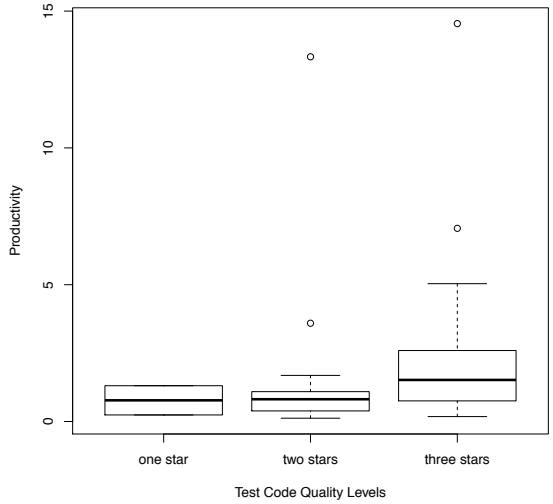

Figure 5.5: Comparison between productivity and different test code quality levels

For throughput we observe that there is a significant increase between 3-star snapshots and 1- and 2- star snapshots. However, the difference between 1 and 2 star snapshots is very small, with the 2 star snapshots having a median that is lower than the median for 1-star snapshots. This can be explained by examining the thresholds of most of the properties as shown in Section 3.3. There we saw that the thresholds of several properties that separate 1 and 2-star systems are very low, thus significantly decreasing the difference between 1 and 2 star systems.

The same observations apply for productivity. Even though the median of 2-star snapshots is this time a bit higher than the median of 1-star snapshots, the difference is small.

Productivity significantly improves for 3-star systems.

As far as the influence of each of the sub-characteristics is concerned, we observe that completeness and effectiveness are both significantly correlated to throughput and productivity. On the other hand, maintainability is not significantly correlated with either one. Completeness and effectiveness have a direct relation to the benefits of automated testing during development. Completeness reflects the amount of the system that is searched for defects, while effectiveness reflects the ability of the system to detect defects and locate their causes. Maintainability's role is different. Maintainability has nothing to do with the testing capability of the system. It rather focuses on the effort that is necessary in order to maintain the test code so that it remains as complete and as effective as it is. Thus, the lack of correlation between maintainability and issue handling performance indicators is not a surprise.

In addition, it is interesting to note that assertions-McCabe ratio has the highest correlation among the properties of the model both for throughput and productivity. This is an important finding that implies that assertions per decision point are potentially a better indicator of test code quality than simply measuring the percentage of lines that are covered.

A final remark concerns the correlation between test code quality and production code maintainability. In Section 1.3 we noted the opportunity to explore the relation between test code quality and production code maintainability. The correlation that was found is significant positive correlation of medium strength ($\rho = 0.42$ and p-value $\ll 0.01$). However, it is hard to draw any conclusion. The reason behind this finding could be either of the following three: (1) better production code makes it easier to develop better test code, (2) better test code facilitates writing better production code or (3) the skill of the development team is reflected both in test code and production code quality.

5.5 Controlling the confounding factors

In Section 4.1 we identified several factors that could be influencing issue handling performance. In this section we are assessing the influence of the confounding factors which we can measure on issue handling. There are two factors which we measure, namely production code maintainability and team size. Production code maintainability is measured by applying the SIG quality model on the source code of the systems. Team size is measured by counting the users that committed code at least once in the VCS of the systems.

In the case of the correlation between test code quality and defect resolution speed, no significant correlation was found. However, it could be that the correlation could not be observed because of the effect of confounding factors. In the cases of throughput and productivity, confounding factors might be the reason correlation was found. In order to establish a clearer view on the relations between test code quality and issue handling indicators, we will use the method of multiple regression analysis.

In particular, we apply stepwise multiple regression analysis. The linear models under analysis are the following:

Def. Res. Speed Rating = tCode Quality Rating + pCode Maintainability Rating + Team Size + c

Throughput = tCode Quality Rating + pCode Maintainability Rating + Team Size + c

Productivity = tCode Quality Rating + pCode Maintainability Rating + Team Size + c

The results of applying the multiple regression analysis for defect resolution speed did not qualify any of the independent variables as significant predictors of defect resolution speed. The results for throughput and productivity are shown in Table 5.8 and Table 5.9 respectively.

Model	Coefficient	Std. Error	t	p-value
(Constant)	-1.512	0.598	-2.529	0.015
Test Code Quality Rating	0.614	0.229	2.682	**0.010**
Team Size	0.040	0.019	2.120	**0.039**
Model Summary: $R^2 = 0.193; p \leq 0.01$				

Table 5.8: Results of multiple regression analysis for throughput

Model	Coefficient	Std. Error	t	p-value
(Constant)	-3.406	2.004	-1.699	0.095
Test Code Quality Rating	2.081	0.793	2.624	**0.011**
Model Summary: $R^2 = 0.117; p = 0.011$				

Table 5.9: Results of multiple regression analysis for productivity

The results of the multiple regression analysis for throughput qualify test code quality and team size as significant predictors of throughput. Production code maintainability was eliminated from the selection after the first step of the regression analysis as it was not a significant predictor (for an explanation of stepwise multiple regression analysis see Section 4.5). The results of the same analysis for productivity indicate test code quality alone as a significant predictor of productivity.

These results increase our confidence that the results that were presented in Section 5.3 hold after we control for the influence of production code maintainability and team size. The fact that test code quality appears to have a stronger influence on throughput and productivity than production code maintainability is an interesting finding and intrigues future research.

5.6 Threats to Validity

In this section factors that may pose a threat to the validity of the experiment's results are identified. We follow the guidelines that were proposed by Perry et al. [75] and Wholin

et al. [91], and organize the factors in four categories: construct, internal, external and conclusion validity.

5.6.1 Construct Validity

Do the variables and hypotheses of our study accurately model the research questions?

Test code quality measurement: The test code quality model was developed by following the GQM approach [91]. The most important aspects of test code quality were identified and metrics that were considered suitable were selected after studying the literature. However, more metrics can be used. For example, mutation testing (see Section 2.1.2) could be used as a metric that indicates the effectiveness of test code. Thus, the model is not complete, but it is our belief that the metrics that were used in combination with the layered structure and the benchmark-based calibration of the metrics provide a fair assessment of test code quality.

Indicators of issue handling performance: In order to measure different aspects of issue handling performance, a set of three indicators was used. Nevertheless, more aspects of issue handling can be captured. For example, the number of issues that are being reopened would indicate inefficiency in the resolution process. In addition, the used indicators are focusing on quantitative aspects of issue handling, but qualitative analysis could provide further insights. For example, the difficulty of resolving each issue could be assessed. However, such qualitative analysis is not trivial to perform. It is clear that the set of issue handling indicators that is used in this study is not complete, but it captures some important aspects of issue handling performance.

Quality of data: Our dependent variables are calculated based on data that are extracted from ITSs. The data in these repositories are not sufficiently accurate [15]. Issues may be registered as closed later than the time when the actual work on the issues has stopped. Some others may have been resolved despite the fact that it has not been registered in the ITS. More reasons like the above can lead to unreliable measurements. An additional case is when a clean-up is performed on the ITS and issues are closed massively after realising that they have been resolved but not marked as such in the issue tracker. We tried to mitigate this problem by applying extensive data cleaning in order to reduce the noise in the ITSs data.

Number of developers: The number of developers was measured both in order to calculate productivity and to measure the influence of team size as a confounding factor. The number of developers was calculated by counting the number of users that committed code at least once in the VCS of each system. This is an indication of how many people were active in the project, but it is not guaranteed to be a fair representation of the amount of effort that was put in the project. This is because (1) commits can vary significantly with regard to the effort that they represent, (2) new members of open source development teams often do not have the rights to commit code and instead, they send code changes to a senior member who performs the commit on their behalf and (3) it has been demonstrated by Mockus et al. [61] that in open source projects there is a core team that performs the majority of the effort. The first two problems remain threats to the validity of our study. For

the third problem we performed a similar analysis as done in [61] in order to calculate the number of developers in the core team.

In [61] the Pareto principle seems to apply since 20% of the developers perform 80% of the effort (measured in code churn). In our study, we applied the Pareto principle in order to calculate the number of developers that are accounted for the 80% of the commits. The commits per developer were calculated and sorted in a decreasing order. The commits were iteratively added until the number of commits reached the 80% of the total commits. Every time the commits of a developer were successfully added, the developer was considered member of the core team.

In our dataset the Pareto principle does not seem apparent. In fact, the core team is $20 \pm 5\%$ of the total number of developers in only 18.5% of the snapshots. Again, the granularity of commits is a threat to the measurements. However, the exploration of the core team concept was interesting.

After calculating productivity as number of resolved issues per month divided by the number of core team developers, we rerun the correlation test between test code quality and *team core productivity*. The results revealed no radical change in the correlation in comparison with the whole team productivity ($\rho = 0.44$ and p-value$\ll 0.01$).

5.6.2 Internal Validity

Can changes in the dependent variables be safely attributed to changes in the independent variables?

Establishing causality: The experiment's results are a strong indication that there is a relation between the test code quality model's ratings and throughput and productivity of issue handling. However, this is not establishing causality between the two. Many factors exist that could be the underlying reasons for the observed relation, factors which we did not account for.

Confounding factors: In Section 4.3.1 a subset of possible confounding factors was identified. We did not control for the granularity of the issues, the experience of the developers or the project's popularity. Additional factors possibly exist as it is impossible to identify and control all of them. However, we attempted to measure and control for the effect of production code maintainability and team size, and established that they do not influence the relation of test code quality with throughput and productivity.

Unequal representation and dependence of the subject systems: In the dataset, each snapshot is considered as a system. The systems are represented with unequal numbers of snapshots. The number of snapshots ranges from 2 for jMol to 7 for ArgoUML and Ant. Therefore the contribution of each system to the result is not equal. In addition, the snapshots of the same system are not independent with each other. We address these threats by establishing strict criteria for snapshot selection. A period of at least one year and at least 30% code churn exist between two consecutive snapshots of the same system.

5.6.3 External Validity

Can the study's results be generalised to settings outside of the study?

Generalisation to commercial systems: The data used in the experiment were obtained from open source systems. The development process in open source systems differs from that in commercial systems. Therefore, strong claims about the generalisability of the study's results for commercial systems cannot be made. Nevertheless, practices that were first adopted in open source systems, such as unit testing, the use of ITSs, continuous integration and globally distributed development appear to be applied in some commercial systems as well. Therefore, the results can be generalised to systems that are being developed using such "modern" practices according to our belief.

Technology of development: All the systems that participated as subjects in the experiment were developed in Java. It is therefore necessary to do further experiments in order to generalise the results for other technologies, especially when it comes to other programming approaches such as procedural or functional programming. On the other hand, programming languages that share common characteristics (i.e. object oriented programming, unit testing frameworks) with Java (e.g. C++, C#) follow similar development processes and therefore the results are believed to be valid for them.

The bias of systems that use ITSs: In order to collect the necessary data the existence of a systematically used ITS was established as a criterion for the selection of subject systems. The development teams of such systems appear to be concerned about the good organization of their projects and they embrace mature software engineering processes. Therefore, the relation between test code quality and issue handling cannot be generalised to systems whose teams are at lower maturity levels.

Absence of systems whose test code quality is rated 4 *and* 5 *stars:* Unfortunately, none of the snapshots in the dataset was rated above 3.5 for test code quality. Thus, we cannot claim that the correlation between test code quality, and throughput and productivity will remain positive for such systems. Future replication of the experiment with a broader range of ratings in the dataset is necessary to generalise our results for the whole scale of the test code quality rating.

5.6.4 Conclusion Validity

To which degree conclusions reached about relationships between variables are justified?

Amount of data: The correlation tests run for the three hypotheses of the experiment contained 63, 54 and 54 data points for the correlations between test code quality and defect resolution speed, throughput and productivity respectively. The number of data points is considered sufficient for performing non-parametric correlation tests, such as Spearman's ranked correlation test. The statistical power of the results for throughput and productivity can be considered highly significant, since the p-values were lower that 0.01. In addition, the correlation coefficient in both cases was approximately 0.50, an indication of a medium to strong correlation. However, it would be desirable to have a larger dataset, with snapshots of solely different systems in order to increase the strength of the results.

Chapter 6

Case Studies

Studying the application of the test code quality model on actual industrial software systems will provide valuable insight on the model's ability to provide useful information about the system's test code, as well as the strengths and limitations of the model. In this chapter three case studies are being reported. The model was applied to three different industrial systems and the ratings were compared with the opinion of an expert who has deep knowledge of the system's testing process.

For each case study, first some general information about the system under study is provided. The results of the application of the model follows. Finally, the expert's evaluation of the system's test code quality is reported and the discrepancies between the expert's opinion and the model's ratings are discussed. In the end of the chapter, the results are summarised together with the model's limitations that were revealed during the case studies.

6.1 Case Study: System A

6.1.1 General Information

The first system under study is an internal support system of an international company based in the Netherlands. The system's development started in 2002. It is programmed in Java. The system has been in maintenance since 2005. We could distinguish two phases of maintenance: 1) from 2005 to 2008, when the effort put on the system was three FTEs[1] and 2) from 2008 to 2011, when the effort was five FTEs.

The system's production code volume at the moment of the case study was $\sim$ 190 KLOC. At the same time, there were $\sim$ 230 KLOC of JUnit code. Additional testing code was used, mainly shell scripting code for integration and regression testing as well as tests written in Selenium[2], a suite of testing tools appropriate for testing web applications. However, JUnit code contributes to more than 95% of the system's test code since the scripts together with the Selenium code are $\sim$ 10 KLOC.

[1] Full-time equivalent (FTE) is a way to measure workers' involvement in a project. One FTE means that the effort is equivalent to the work performed by a full-time employee.

[2] `http://seleniumhq.org/`

It is worth mentioning that the system's development followed a strict Test-Driven Development (TDD) approach. Hudson[3] has been used for continuous integration and JUnit test reporting. Finally, the system's production code maintainability was assessed using SIG's software quality model and the SAT resulting in a final score of 4.1 stars.

6.1.2 Test Code Quality Model Ratings

The test code quality model was applied on System A. Table 6.1 shows the results.

Table 6.1: Test Code Quality Model Ratings for System A.

Properties	Value	Rating	Sub-characteristics	Rating	Test Code Quality
Coverage	66.3%	4.1	Completeness	4.3	3.6
Assert-McCabe Ratio	1.04	4.5			
Assertion Density	0.15	4.1	Effectiveness	3.7	
Directness	25.4%	3.3			
Duplication	19%	2.0	Maintainability	2.8	
Unit Size	-	2.3			
Unit Complexity	-	4.3			
Unit Dependency	-	2.4			

The model rates completeness at 4.3. This informs us that a large part of the system is tested. The lower level properties of completeness can provide more fine-grained information. In particular, code coverage was statically estimated to be 66.3%. This corresponds to the two thirds of the system. However, one third of the system is not covered by the test code. This is a considerable part of the system that is exposed to the possibility of introducing defects during code changes. Assertion-McCabe ratio is rated at 4.5. In absolute values, the ratio is greater than 1, meaning that more assertions exist than the decision points in the code. This increases the confidence that the part of the system that is covered, is covered thoroughly.

Effectiveness is rated 3.7. Assertion density is rated 4.1, indicating that the test code has an adequate number of assertions. The reason effectiveness is lower, is directness. With a direct coverage of 25.4%, it is obvious that there are a lot of parts of the system that are only covered indirectly. This possibly reduces the effectiveness of the test code in terms of facilitating the location of the cause of defects.

The third aspect of the test code quality model, maintainability, was rated at 2.8. All the metrics that are related to maintainability except unit complexity are at mediocre levels, alerting us about the need to raise the quality of the test code. Duplication is at high levels in the test code: 19% for a rating of 2.0. A very common reason for duplication in the test code is copying the fixtures. On the one hand, this makes the tests self-contained. On the other hand, a change in the code might lead to the necessity of applying the change to every single test that is related. Unit size and unit dependency are rated 2.3 and 2.4 respectively, indicating that test methods could be smaller and less dependent on collaborative

[3] http://hudson-ci.org/

objects. On the bright side, unit complexity is rated 4.3, confirming that the good practice of inserting as few decision points in the test code as possible is followed.

The overall test code quality was rated 3.6. The system would have received a higher rating if maintainability was at the same levels as completeness and effectiveness. In fact, the test code's current ability to detect defects that are introduced during code changes in most parts of the system is adequate. Nevertheless, maintaining the code to sustain this ability seems hard and expensive due to the low maintainability.

6.1.3 Test Code Quality Assessment by the Expert

For system A, an experienced developer of the system was interviewed in order to provide an evaluation of the test code from the perspective of an expert.

How completely is the system tested?

The system's code coverage is being monitored on a regular basis by the team. In particular, code coverage of 70% is being reported, as calculated by using the tool Cobertura[4]. The expert estimated that the rest of the test code adds roughly 5% of code coverage.

The expert noted that this code coverage level is not considered high enough. However, most of the critical parts of the system are covered. The defects that are found in the parts of the system that are not covered are easy to fix. In addition, it was noted that the system is developed and used only by the developers themselves. This means that the impact of the defects on the business is quite limited. Furthermore, spotted defects can be fixed and deployed quickly since there is no need to ship a patch that fixes the problem to the customers. Consequently, the expert stated that ignoring testing part of the system is an affordable risk.

How effectively is the system tested?

To answer this question, the expert combined information that is relevant to the nature of the defects that are usually reported to the ITS and to defect fixing. In particular, the majority of the defects being reported are trivial to fix. No structure related defects are detected. Therefore, defect resolution time is kept at low levels. Moreover, the test code helps substantially in the development process by pointing out defects during programming and before they reach a production release.

How maintainable is the system's test code?

After the expert put emphasis on the high commitment of the team in writing highly maintainable production code, he made clear that these high standards are not followed as strictly while writing test code. Thus, the maintainability of the test code is not at the same levels as the maintainability of the production code.

In more detail, the volume of the test code is mentioned as a significant drawback for its maintainability. There is even more test code than production code. In addition, it is often that the same piece of functionality is tested multiple times by different parts of the

[4] `http://cobertura.sourceforge.net/`

Table 6.2: Expert's ratings on system A's test code quality aspects.

Aspects	Completeness	Effectiveness	Maintainability	Overall Test Code Quality
Rating	4.0	4.0	3.0	3.5

test code. This means that some parts of the test code could be removed without an impact on the completeness and the effectiveness of the test code.

Another issue that makes maintaining the test code harder is duplication. Functional duplication is high in the system's test code, reports the expert. However, many times that was a deliberate choice, since duplication was preferred as a way to keep the test code self-contained at the test method level. This way, the expert claims, readability of the test code improves.

Finally, the expert reported that code changes lead to expensive test code adjustments. In fact, the ratio of time spent working on adjusting the test code compared to the time working on the production code that implemented a change is roughly estimated to be 3 : 1.

To which extent is the test code targeting unitary or integration testing?

The expert clarifies that the intention of using JUnit was to apply unit testing to the system. In order to isolate tested parts of the system, mocking frameworks (EasyMock and jMock) are being used partially. However, he recognizes that not all of the JUnit code is unit testing. Collaborating objects are being used often in order to serve the testing of a particular object. Thus, the tests are partly integration tests as well.

In addition, a particular characteristic of the system makes it harder to keep the tests' directness high. A large module ($\sim$ 125 KLOC) of the system has a single entry point. Therefore, whenever something in that module or an external object that communicates with that module needs to be tested, calling that entry point is required in order to set up the fixture of the test.

How would the expert rate the aspects of the test code quality of the system?

The expert's ratings of the aspects of the test code quality of the system are shown in Table 6.2.

6.1.4 Comparison between the expert's evaluation and the model's ratings

A comparison between the expert's estimations and the test code quality model's ratings for System A reveals small discrepancies. In particular, the discrepancies range from 0.1 to 0.3.

Completeness was rated by the model at 4.3 while the expert estimated it to be 4.0. In more depth, the model's code coverage metric was 66.3%. The model only analysed the test code that was written in Java in the system. The expert reported a coverage of 70% for the Java test code. This coverage measurement was obtained by using a dynamic coverage estimation tool. The percentages are in the same order of magnitude. The fact that the static estimation is lower than the dynamic one could be explained by the extensive use of the

Visitor pattern in the system. The use of interfaces and abstractions makes it harder for a static analysis to resolve the actual methods that are being called. When the tracking of a method call reaches an abstract method it is possible that this method is implemented by several others. The tracking is then stopped, since it cannot resolve which implementation method in particular is called, a disadvantage that does not exist in a dynamic code coverage tool.

However, the static estimate is very close to the dynamic one. Another point of interest here is the complexity of the tested system. System A has a rating of 4.4 in unit complexity in SIG's software maintainability model. This means that McCabe's cyclomatic complexity metric is relatively low throughout the system. Consequently, static estimation of code coverage becomes easier to be accurate. When complexity in a method is high, it means that there are a lot of different possible paths of execution within the method. Thus, the call of a method from the tests by its own does not guarantee that all of the paths are being covered by the tests. On the other hand, lower complexity means less paths of execution exist in the method. Ideally, there is just one path and the call of the method from the test code leads to the complete coverage of the code in the method. To sum up, it is expected that the accuracy of the static estimation is affected by the unit complexity of the system in such a way that the lower the unit complexity, the more accurate the static estimation of code coverage. Assert-McCabe ratio was chosen as a supplementary coverage metric. Its suitability in the model is to compensate for the aforementioned disadvantage of the static estimation of code coverage.

Regarding effectiveness, it was rated by the model at 3.7 while the expert estimated the rating to be 4.0. In particular, high assertion density combined with high completeness explains the expert's comment that the test code contributes significantly in detecting defects before they reach the production phase. The other metric that concerns effectiveness, namely directness, was rated lower. Only 25.4% of the system is being tested directly from the test code. This confirms the expectation created by the existence of a large module with extensive use of the Visitor design pattern.

The maintainability of the test code was rated 2.8. The expert estimated maintainability to be 3.0, relatively close to the rating of the model. The model's rating reflects the expert's opinion about the maintainability of the system's test code. The rating indicates that maintenance is not a trivial task in the system, something that is confirmed by the expert's comment that the ratio of time spent working on adjusting the test code compared to the time working on the production code that implemented a change is $3:1$. Furthermore, duplication of 19% shows indeed the extensive use of copying parts of the code. However, the score obtained after the benchmarking is 2.0, possibly indicating that even if duplication is a deliberate choice, 19% is too high.

Finally, the model calculated the overall test code quality as 3.6, by aggregating the subcharacteristics. This rating is very close to the experts estimation of the test code quality of system A. The expert's estimation was 3.5. Overall, this case study shows that the test code quality model was aligned with the expert's opinion on the system. Several insights about the system's test code created expectations which were reflected by the model's ratings.

6.2 Case Study: System B

6.2.1 General Information

System B is a logistics system developed by a Dutch company. The system has been in maintenance since 2006. No new functionality has been added to the system. However, since 2010 a re-structuring of the system has been performed. The programming language used is Java. In addition, the system makes heavy use of database technologies. Thus, SQL code is part of the Java code.

The system's production code volume at the moment of the case study was $\sim$ 700 KLOC. This code is automatically tested by $\sim$ 280 KLOC of JUnit code. During the re-structuring of the system extensive modularization was performed. A lot of testing code was added during this period in order to guarantee that the changes would not cause the system to break. In addition, integration testing is performed by scripting code that was not available for analysis. The testing process is completed by extensive manual testing.

Hudson was used for continuous integration in this case as well. Nevertheless, its use has not been disciplined. The principle that code in the repository should never cause tests to fail [11] has been violated for short periods of time. The system's production code maintainability was assessed using SIG's software quality model and the SAT resulting to a final score of 3.0 stars.

6.2.2 Test Code Quality Model Ratings

The test code quality model was applied to System B. Table 6.3 shows the results.

Table 6.3: Test Code Quality Model Ratings for System B.

Properties	Value	Rating	Sub-characteristics	Rating	Test Code Quality
Coverage	50%	3.1	Completeness	2.8	2.5
Assert-McCabe Ratio	0.16	2.4			
Assertion Density	0.08	2.7	Effectiveness	2.8	
Directness	17.5%	2.8			
Duplication	16%	2.6	Maintainability	2.1	
Unit Size	-	2.0			
Unit Complexity	-	2.5			
Unit Dependency	-	1.5			

Completeness is assessed at 2.8. This indicates that a big part of the system is not covered. The metrics that comprise completeness can provide more information about the weaknesses of the test code. In particular, coverage is 50%. Half of the system is tested. This leads to a rating of 3.1, intuitively higher than a system with 50% coverage should get. However, since the ratings only reveal how the system scores in comparison with the benchmarking systems, low standards can lead to these counter-intuitive ratings. The assertions-McCabe ratio is even lower (2.4).

Effectiveness is also rated at 2.8. Assertion density and directness are at the same levels (2.7 and 2.8 respectively) suggesting that the ability of the system to detect defects and locate their causes is limited.

The test code's maintainability is 2.1 according to the model. High duplication, large test methods and existence of complexity indicate that the system's test code is hard to maintain. The most important warning though is derived from the unit dependency metric. With a rating of 1.5, this implies that most of the test methods heavily use production methods that are not the ones that are being tested. This increases the dependency of the tests to many parts of the production code, thus making changes in the production code to propagate in the test code much more than in systems that avoid this kind of coupling.

With an overall rating of 2.5, the model indicates many weaknesses of the system's test code.

6.2.3 Test Code Quality Assessment by the Expert

For system B, an expert technical consultant with experience on the system was interviewed.

How completely is the system tested?

The expert reported that the code coverage in System B is poor. Initially, the system's architecture was based on Enterprise JavaBean (EJB) 2.x, a fact that made the system harder to be tested according to the expert. Only lower layers of the system's architecture are being covered by test code. The expert had in his possession available coverage data for one module of the system. However, this module comprises 86% of the whole system. The reported code coverage (dynamic estimate) of this module is $\sim 43\%$. Taking into consideration the size of the module, one can calculate that the possible range of the code coverage for the whole system is between 37% (when the other modules of the system are not covered at all) to 51% (when the other modules are fully covered).

How effectively is the system tested?

The expert reported that the system's testing effort is "immense and costly". Testing effort could potentially be reduced by developing more automated tests. However, defects are detected with satisfactory effectiveness. The expert estimates that 50% of the detected defects are due to the unit testing. Integration and manual testing adds 30% to the defect detection ability.

How maintainable is the system's test code?

Focus on test code's maintainability was not a high priority for the development team according to the expert. His own perception of the system's test code is that maintainability is hindered by high complexity and coupling between the test code and the production code. The expert further explained that, given the current maturity of average software systems, one should be satisfied if automated testing exists in the first place. He added that quality of test code is the next step for software projects' maturity.

Table 6.4: Expert's ratings on system B's test code quality aspects. Both the ratings according to the expert's ideal standards and the ones according to his evaluation of the system in comparison to the average industry standards are shown, with the latter written in parentheses.

Aspects	Completeness	Effectiveness	Maintainability	Overall Test Code Quality
Rating	2.0 (3.0)	1.0 (2.0)	2.0 (3.0)	2.0 (3.0)

To which extent is the test code targeting unitary or integration testing?

Even though the intention of the use of JUnit code was to perform unit testing, the expert reported high coupling of the tests and the production code, implying that a significant part of the test code is integration testing.

How would the expert rate the aspects of the test code quality of the system?

The expert's ratings of the aspects of the test code quality of the system are shown in Table 6.4. The expert claimed to have had significant experience in analysing systems' code. This was the reason he preferred to provide two estimations: one based on his own ideal standards for testing and one that compares the system to his perception of the average standards in the industry. In Table 6.4 both estimations are shown with the ones that take into consideration the industry average being in parentheses.

6.2.4 Comparison between the expert's evaluation and the model's ratings

Comparing the expert's ratings with the ones derived from the model, we have to consider if we expect the model's ratings to be closer to the expert's ideal rating or his rating compared to the industry average. As the model was calibrated on industrial and open source systems, we would expect the model's ratings to be closer to the expert's estimations that were based on his experience with industrial systems. However, we observe that there is no consistent alignment with either one of them.

In particular, completeness was calculated as 2.8 by the model. This rating is closer to the expert's estimation of the system's completeness in comparison with the industry average. Static estimation of code coverage for system B is 50%. This value is closer to the optimistic estimation of the system's code coverage, where the modules for which the expert had no available data about code coverage are fully covered. As this is highly improbable, it implies that in this case the static estimation is higher than the dynamic coverage that the expert reported, in contrast to what we have seen in System A. This could be explained by the rating of the production code's unit complexity. System B's production code scores 2.5 for unit complexity, an indication that the system has a considerable amount of methods that have relatively high complexity, a fact also pointed out by the expert. This makes the static estimation of code coverage less accurate.

The asserts per McCabe ratio metric is 0.16 resulting in a rating of 2.4, significantly lower than the system's coverage rating (3.1). This suggests that more assertions should be

added to the test code in order to cover more of the decision points of the system. At the same time, this case demonstrates the complementary behaviour of the two metrics: code coverage and assert-McCabe ratio. Code coverage is overestimated because of the system's high unit complexity, but assert-McCabe ratio serves as a correction for this fact by lowering the completeness rating to 2.8.

As far as effectiveness is concerned, the model rates it at 2.8, quite far from either of the expert's estimation. There are approximately 8 assertions per 100 lines of testing code in system B, leading to a rating of 2.7 for assertion density. Direct code coverage is 17.5%, leading to a rating of 2.8 for directness. The high discrepancy between these ratings and the expert's estimation owes its existence to the benchmarking nature of the model. The absolute values for the metrics assertion density and directness are indeed low. However, the system still scores ratings close to 3.0 indicating that most of the systems in the calibration set score very low in these metrics. It is important to have in mind that the ratings only reflect the system's quality in comparison with the calibration systems during the interpretation of the model's ratings.

Maintainability, on the other hand, is rated at 2.1 by the model, in the same magnitude as the expert's ideal estimation. With 16% of the system's test code being duplicated, long methods (unit size 2.0), high complexity (unit complexity 2.5) and a high unit dependency (1.5), the system's test code appears to be against the principles of clean and maintainable code.

Finally, the system's overall test code quality is rated at 2.5, a value that lies exactly between the expert's ideal estimation and his placement of the system in comparison to the industry average.

6.3 Case Study: System C

6.3.1 General Information

The last case study involves a designing system for engineers. The company that develops it is also based in the Netherlands. The system was in the maintenance phase when the case study was performed. However, implementation of new features still takes place during the maintenance of the system. The size of the development team is approximately 15 people.

The system's production code volume at the moment of the case study was $\sim$ 243 KLOC. Another $\sim$ 120 KLOC of JUnit code implement tests for the system. The development team adopted TDD during the past 18 months. It has to be noted that most of the testing effort came during the last two years. In addition to the automated tests, acceptance tests are performed annually directly by the customers of the system.

The architecture of the system has undergone some major changes. In particular, the system's main modules were rewritten, although the system is still using the old, legacy modules. This coexistence of the old and the new modules separate the system in two parts of different characteristics and quality since the two parts were developed with different processes and quality criteria. This can be reflected on the maintainability ratings of the system. SIG's quality model gave a rating of 3.3 stars for the whole system. However, applying the analysis only to the newly written modules, the maintainability score rises to

Table 6.5: Test Code Quality Model Ratings for System C.

Properties	Value	Rating	Sub-characteristics	Rating	Test Code Quality
Coverage	52.5%	3.3	Completeness	3.1	3.5
Assert-McCabe Ratio	0.29	2.9			
Assertion Density	0.12	3.7	Effectiveness	3.6	
Directness	27%	3.4			
Duplication	5.8%	4.5	Maintainability	3.7	
Unit Size	-	3.7			
Unit Complexity	-	3.3			
Unit Dependency	-	3.5			

4.0 stars, reflecting the improvement of the processes and the team's focus to increase the quality of the new modules.

6.3.2 Test Code Quality Model Ratings

The test code quality model was applied on System C. Table 6.5 shows the results.

The completeness of the test code was rated at 3.1 with coverage and assertions-McCabe ratio being relatively close (3.3 and 2.9 respectively). Coverage of 52.5% reveals that almost half of the system is not covered by the tests, a significant drawback for the defect detection ability throughout the system. Assertions-McCabe ratio suggests that additional assertions are necessary to test the various decision points in the code.

Effectiveness was rated at 3.6. This rating is higher than the completeness and indicates that the parts that are tested, are tested fairly effectively. In particular, assertion density (3.7) indicates that the system's defect detection ability in the parts that are tested is adequate. Directness falls a bit lower (3.4), with only 27% of the system being tested directly. This indicates that locating the cause of the defects could be significantly easier by increasing the percentage of directly tested code.

Maintainability of 3.7 is the highest maintainability rating met in the case studies and indicates that the system's test code is written carefully. Duplication is kept at low levels (5.8%) and unit size and unit dependency are higher than average. Unit complexity (3.3), although not too low, reveals possible space for improvement of the test code's maintainability.

Overall, the system's test code quality is assessed as 3.5. The model reveals that the system's test code is effective and maintainable, but not enough to cover the system. Therefore, adding code of similar or better quality to cover the rest of the system would lead to a system that is very well tested automatically.

6.3.3 Test Code Quality Assessment by the Expert

For system C, an expert technical consultant with experience on the system was interviewed.

How completely is the system tested?

As we saw, the system's separation in the legacy modules and the new ones has a strong impact on the system's quality. According to the expert, the legacy modules are tested weakly. Code coverage is around 15%. Moreover, the development team has stopped writing test code to cover these modules.

On the other hand, the newly developed modules are reported to have much higher code coverage: 75%. To get an overall image of the system's code coverage it is important to know the size of the legacy modules compared to the rest of the system. Legacy modules are $\sim$ 135 KLOC of the system's total of $\sim$ 243 KLOC. Thus, the fact that more than half of the system is poorly covered leads to the expectation of the system's overall coverage at $\sim 40-45\%$.

How effectively is the system tested?

The expert reported that since the development team adopted TDD a decrease in the number of incoming defect reports was noticed. In fact, during the last year the number of defects was the smallest compared to the previous years.

How maintainable is the system's test code?

The expert reported that he has no insight on the system's test code maintainability.

To which extent is the test code targeting unitary or integration testing?

The test code was developed mainly to perform unit testing. Mock testing was also used. However, the expert reports that parts of the test code serve as integration testing, calling several parts of the system apart from the one tested directly.

How would the expert rate the aspects of the test code quality of the system?

The expert's ratings of the aspects of the test code quality of the system are shown in Table 6.6.

Table 6.6: Expert's ratings on system C's test code quality aspects.

Aspects	Completeness	Effectiveness	Maintainability	Overall Test Code Quality
Rating	3.5	4.0	-	4.0

6.3.4 Comparison between the expert's evaluation and the model's ratings

The model's ratings for System C are consistently lower than the expert's opinion (where available). The difference is in the magnitude of 0.5 for each sub-characteristic and the overall test code quality.

Completeness was rated 3.1 by the model in contrast to the expert's estimation of 3.5. The static estimation of code coverage was 52.5%, higher than the expectation of $40-45\%$. The other metric concerning completeness though, was not at the same level. The existence of 0.29 assertions per decision point in the system assigns the system's assert-McCabe ratio a rating of 2.9. In this regard, a weakness of the analysis tool might have interfered with the accuracy of the measurements: the system's test code contains extensive use of user-defined assertions. These customised assertions were not being detected, thus underestimating the metrics that contain the number of assertions as a parameter.

Effectiveness was assigned the rating of 4.0 by the expert, while the model's rating was 3.6. The percentage of directly covered code was 27%, a number that surprised the expert because it was significantly lower than his expectation.

An estimation for maintainability was not given by the expert due to the lack of focus on that aspect during his assessment of the system.

Finally, overall test code quality was rated 3.5 by the model, half a unit lower than the expert's estimation. One possible explanation for the discrepancies in this case would be the role of benchmarking in the ratings of the model. The expert evaluated the system based on his own knowledge and experience. The benchmarking seems to cause the model to assign stricter ratings than the expert in a consistent way in this case.

Another possibility would be that the expert's opinion was biased towards evaluating the system according to the quality of the new modules of the system. It is interesting to see that applying the model only to the new modules the ratings converge to those of the experts. Completeness, effectiveness, maintainability and overall test code quality are 4.1, 3.9, 3.9 and 4.0 respectively.

6.4 Conclusion

In this chapter three case studies to which the test code quality model was applied were reported. The results of the model were compared to the opinion of an expert's evaluation of the system's test code quality. The case studies contribute to assessing the usefulness of the application of the model (*RQ3, RQ3.1*). In addition, the case studies enabled the detection of several limitations of the model, details that have to be carefully considered during the interpretation of the model's results, and suggested areas that can be improved by future work.

The application of the model revealed significant information about the systems that were aligned with the experts' opinions. Table 6.7 summarizes the comparison between the experts' evaluation and the model's ratings for each system. The goal of the model was achieved since in all cases we were able to receive a clear overview of the main aspects of the test code. In addition, the model enables developers and evaluators to understand points where the system's testing process is not adequate and take the corresponding actions to improve it. For example, System B lacks in completeness and it would be strongly suggested to write test code in order to cover more parts of the system, while at the same time the development team should be alerted of the test code's maintainability in order to avoid reaching a point where applying changes to the test code is consuming most of the

development effort. Finally, the model's directness metric provided a new perspective to the experts, who welcomed the insight provided by such a metric.

Table 6.7: Overview of the comparison between the experts' evaluations and the test code quality model's ratings for systems A, B and C.

System	Completeness		Effectiveness		Maintainability		Overall Test Code Quality	
	Expert	Model	Expert	Model	Expert	Model	Expert	Model
A	4.0	4.3	4.0	3.7	3.0	2.8	3.5	3.6
B	2.0 (3.0)	2.8	1.0 (2.0)	2.8	2.0 (3.0)	2.1	2.0 (3.0)	2.5
C	3.5	3.1	4.0	3.6	-	3.7	4.0	3.5

Even though the lack of data does not enable us to draw strong conclusions from a quantitative analysis of the comparison between the experts' evaluations and the model's estimates, it is still useful to perform such an analysis. When there is lack of expertise on a system, the model can be used in order to obtain an assessment of the quality of test code. Therefore, it is important to know how close to the experts' evaluations the estimates of the model are.

The experts' evaluations were given in an ordinal scale from 0.5 to 5.5 with a step of 0.5. The model's estimates are in a continuous scale from 0.5 to 5.5. To calculate the error of the model's estimates compared to the experts' evaluations, the model's estimates are converted to the same ordinal scale as the experts' evaluations. This is performed by rounding the values to the closest unit of the ordinal scale. The values and the errors can be seen in Table 6.8.

Table 6.8: Experts' evaluations, the model's estimates converted in ordinal scale and the error for systems A, B and C

System	Characteristic	Expert	Model	Error
A	Completeness	4.0	4.5	0.5
	Effectiveness	4.0	3.5	0.5
	Maintainability	3.0	3.0	0
	Test Code Quality	3.5	3.5	0
B	Completeness	3.0	3.0	0
	Effectiveness	2.0	3.0	1.0
	Maintainability	3.0	2.0	1.0
	Test Code Quality	3.0	2.5	0.5
C	Completeness	3.5	3.0	0.5
	Effectiveness	4.0	3.5	0.5
	Maintainability	-	3.5	-
	Test Code Quality	4.0	3.5	0.5

Because of the ordinal scale of the data, the median reveals the central tendency of the errors. The median is 0.5, which is exactly one unit of the ordinal scale. In particular, 9 out

of 11 comparisons have an error less or equal to 0.5. This means that in $\sim 82\%$ of the cases, the model's estimate is at most 1 unit of the scale different than the experts' evaluations. In the rest of the cases, the error is 2 units of the scale. Although 11 data points are too few to enable us draw a strong conclusion, the results of the analysis of the errors are indicating that the model's accuracy is promising. The model's estimates provide estimates that are very close to the ones that an expert would provide.

At the same time, several limitations are identified. These limitations are listed below:

- Extensive use of abstractions and complex methods can hinder the accuracy static estimation of code coverage.
- Interpretation of the model's results should take into consideration that the model is based on benchmarking. Therefore, several ratings can be counter-intuitive, e.g. directness rated at 2.8 when direct coverage is 17.5%, a value that should not satisfy any development team that intends to apply adequate unit testing on its system.
- Custom assertion methods are not detected by the tool leading to underestimation of the metrics that involve measuring the assertions in the test code (assert-McCabe ratio, assertion density).
- The current implementation of the model takes into consideration only jUnit test code. Other xUnit frameworks as well as integration tests should be included in the scope of the model in order to reflect the total test code of the system under study.

Chapter 7

Conclusions and Future Work

Developer testing has become an important part of software development. Automated tests provide the ability for the early detection of defects in software, and facilitate the comprehension of the system. We constructed a model in order to assess the quality of test code. Based on the model, we explored the relation of test code quality and issue handling performance. Furthermore, we applied the model to three commercial systems and compared the results of the model with that of experts' evaluations. In this chapter, we revisit and summarise the findings of this study. In addition, we indicate possible areas for future research that would provide answers to questions that emerged during this study.

7.1 Summary of Findings and Conclusions

In this section, we summarise the findings of the study which enable us to provide answers to the research questions.

7.1.1 RQ1: How can we evaluate the quality of test code?

The first goal of the study was to establish a method to assess the quality of test code. Towards this end, we reviewed test code quality criteria in the literature. Finally, we applied the GQM approach [91] in order to identify the main aspects of test code quality and to select a set of metrics that would provide measurements that enable the assessment of these aspects.

The results of the approach led to identifying three aspects as the main aspects of test code quality: completeness, effectiveness and maintainability. Completeness concerns the complete coverage of the production code by the tests. Effectiveness concerns the ability of the test code to detect defects and locate their causes. Finally, maintainability concerns the ability of the test code to be adjusted to changes of the production code, as well as the extent to which test code serves as documentation.

Suitable metrics were chosen based on literature and their applicability. Code coverage and assertions-McCabe ratio are used to assess completeness. Assertion density and directness are indicators of effectiveness. For maintainability, the SIG quality model was

adjusted in order to be reflective of test code maintainability. The metrics are aggregated using a benchmarking technique in order to provide quality ratings that inform the user of the quality of the system in comparison with the set of systems that was used in the benchmark.

The main aspects of test code quality were used to define corresponding sub-characteristics as a layer of the test code quality model. In the next layer, the metrics are mapped to each of the sub-characteristics. The model aggregates the metrics into sub-characteristics, and the sub-characteristics into an overall test code quality rating.

7.1.2 RQ2: How effective is the developed test code quality model as an indicator of issue handling performance?

In order to validate the usefulness of the test code quality model, we formulated and tested hypotheses based on the expected benefits of developer testing. The benefits include the localization of the cause of the defects and the removal of fear of modifying the code, since the tests serve as safety nets that will detect defects that are introduced by applying changes. Therefore, we tested whether there is positive correlation between test code quality and three issue handling performance indicators: defect resolution speed, throughput and productivity.

In order to test the aforementioned hypotheses, an experiment was conducted. Data from 75 snapshots belonging to 18 open source systems was collected, including source code, VCSs logs and ITSs data. After controlling for the effects of the maintainability of the production code and the size of the development team, we have found that test code quality is positively correlated with throughput and productivity. In other words, the higher the quality of test code, the more issues are being resolved both per team and per developer. This result provides an empirically derived indication that developer testing is beneficial for software development.

At the same time, no significant correlation was found between test code quality and defect resolution speed, a result that contrasts our expectations. However, possible explanations for this observation exist, such as the fact that the defects that are being reported in ITSs are the ones that the test code failed to detect. In addition, the ability to obtain an accurate estimation of the resolution time of an issue from ITS data is limited. Further experimentation is necessary in order to draw conclusions about the relation between test code quality and defect resolution speed.

The findings of the experiment suggest that test code quality, as measured by the proposed model, is positively related to some aspects of issue handling performance. This result increases our confidence in the assessment capability of the proposed test code quality model. We believe that the model can be used by a development team in order to monitor the quality of the test code in a system. This way, a team can maximise the benefits that developer testing offers.

7.1.3 RQ3: How can we assess the usefulness of applying the test code quality model?

Having confirmed the existence of a positive relation between test code quality and issue handling performance, we performed three case studies where we studied the usefulness of applying the model to commercial systems. The systems' source code was analysed in order to derive the ratings of the test code quality model. Based on the model's ratings, we obtained an insight about the quality of the systems' test code. Interviews with experts who were aware of the systems' test code quality were conducted, in order to assess the alignment of the model's ratings with the experts' evaluations.

The result of the case studies illustrated that the model can capture significant information about the main aspects of test code quality. The experts' evaluations were aligned to the model's ratings, with 82% of the model's ratings differing at most one unit of the rating scale from the experts' evaluations. Furthermore, the model was able to pinpoint weak aspects of the systems and indicate what further actions should be taken in order to improve the quality of test code.

Of course the model is not perfect. The case studies enabled us to identify certain limitations that hinder the accuracy of the test code quality model. However, once the limitations (see Section 6.4) are taken into consideration during the interpretation of the model's outcome, we believe that the model provides reliable assessment of test code quality.

7.2 Contributions

The contributions of this thesis are summarised as follows:

- **Constructing a model that combines metrics in order to provide a measure of test code quality.** We have proposed a model which combines metrics that can be statically obtained directly from the source code into meaningful ratings of the quality of test code. The model was calibrated based on 86 open source and commercial Java systems so that the ratings of a system's test code reflect its quality in comparison with those systems.

- **Enriching the software engineering body of knowledge concerning the relation of test code quality to issue handling performance.** Through conducting empirical research on open source systems we have provided evidence that supports the common belief that developer testing facilitates software development. In particular, our study demonstrated a significant positive correlation between test code quality and throughput and productivity of issue handling.

- **Performing three case studies to study the model's usefulness.** By performing case studies where the model was applied to three commercial systems, we demonstrated that the model provides useful insights into the quality of the test code. Furthermore, the case studies showed that our model is aligned with expert's opinion. Thus, incorporating the model as part of the development process so that the model's ratings can

be systematically consulted can be used in order to facilitate the development of high quality test code.

7.3 Future Work

During the study several questions emerged. In this section we identify interesting topics for future research.

Enlarging the data set. The dataset used in the experiment has several limitations that reduce the ability to generalise the results. In particular, only Java systems are analysed. The application of the model to systems of other languages will reduce the dependency of the results to the technology of development. In addition, none of the snapshots that were analysed received a test code quality rating of more than 3.5. Expanding the dataset with further systems so that representatives of all the quality levels are included will enable us to answer whether the results are consistent through all quality levels.

Validation of the test code quality model against mutation testing. In Section 2.1.2 we presented mutation testing as a test effectiveness evaluation method. An experiment where the dependent variable would be mutation testing score would provide further validation of the test code quality model. In fact, such an experiment was attempted to be conducted. However, several difficulties made it impossible to perform the experiment within the available time. Some of the difficulties are listed below to raise awareness for future attempts:

- Older snapshots of the used open source systems contain code that is not valid for the newest Java compilers. For example, the reserved word *enum* was used as a variable name.

- Mutation testing is performed using dynamic analysis. Thus, the source code has to be to compilable. Even though most of the open source systems make use of modern build tools (such as Apache Ant and Maven), there are still missing libraries that make the process of compiling the code non-trivial.

- A precondition of mutation testing is that all the tests should pass. However, several snapshots contain tests that fail. The intervention of the researcher by either removing the tests, fixing them or the code, has to be carefully done to avoid the introduction of bias.

- Mutation testing requires that equivalent mutants are detected. Manual inspection of each mutants to validate it is not equivalent to the initial program would be impossible in the scale of such an experiment. Mutation testing tools were explored and it was found that only Javalanche [81] features automatic detection of equivalent mutants. However, enabling this option significantly increases the execution time of the analysis, which is already significant even without enabling the option.

Replication with independent data points. As discussed in Section 5.6.2, snapshots of the same system share some dependencies. Despite our efforts to mitigate this problem,

ideally all the snapshots would belong to different systems. Once data is abundant, replication of the experiment with completely independent snapshots will enable drawing more reliable conclusions.

Assessing the relation between test code quality and further indicators of issue handling performance. In order to assess the relation between test code quality and issue handling performance we used three issue handling indicators. However, there can be indicators that reflect different aspects of issue handling. For example the percentage of reopened issues could provide an indication of issue resolution efficiency. Future research that includes additional indicators will contribute to the knowledge of which aspects of issue handling are related to test code quality in particular.

Quantification of the benefits of higher test code quality. In our study, significant positive correlation was found between test code quality, and throughput and productivity. This is an indication that higher test code quality leads to higher throughput and productivity. However, it would be interesting to quantify the magnitude of the improvement, as well as the costs that are involved. This would facilitate managers in taking decisions, such as whether investments in improving test code quality will have an adequate return on investment.

Bibliography

[1] ISO/IEC 9126-1:2001. Software engineering - product quality - part 1: Quality model. ISO, Geneva, Switzerland, 2001.

[2] A. T. Acree. *On Mutation*. Phd thesis, Georgia Institute of Technology, 1980.

[3] F. H. Afifi, L. J. White, and S. J. Zeil. Testing for linear errors in nonlinear computer programs. In *Proceedings of the 14th international conference on Software engineering*, pages 81–91. ACM, 1992.

[4] S. N. Ahsan, J. Ferzund, and F. Wotawa. Program file bug fix effort estimation using machine learning methods for oss. In *Proceedings of the 21st Int. Conf. on Software Engg. and Knowledge Engg.(SEKE09)*, 2009.

[5] T. L. Alves and J. Visser. Static estimation of test coverage. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 55–64. IEEE Computer Society, 2009.

[6] T. L. Alves, C. Ypma, and J. Visser. Deriving metric thresholds from benchmark data. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10. IEEE Computer Society, 2010.

[7] J. An and J. Zhu. Software reliability modeling with integrated test coverage. In *Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement*, SSIRI '10, pages 106–112. IEEE Computer Society, 2010.

[8] R. Baggen, J. Correia, K. Schill, and J. Visser. Standardized code quality benchmarking for improving software maintainability. *Software Quality Journal*, pages 1–21, 2011. To appear.

[9] Victor R. Basili. Software modeling and measurement: the Goal/Question/Metric paradigm. Technical report, Techreport UMIACS TR-92-96, University of Maryland at College Park, College Park, MD, USA, 1992.

[10] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[11] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[12] G. Beliakov, A. Pradera, and T. Calvo. *Aggregation functions: a guide for practitioners*, volume 221. Springer Verlag, 2007.

[13] W. G. Bently and E. F. Miller. CT coverageinitial results. *Software Quality Journal*, 2(1):29–47, 1993.

[14] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, FOSE '07, pages 85–103. IEEE Computer Society, 2007.

[15] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 308–318. ACM, 2008.

[16] D. Bijlsma. *Indicators of Issue Handling Efficiency and their Relation to Software Maintainability*. Msc thesis, University of Amsterdam, 2010.

[17] D. Bijlsma, M. Ferreira, B. Luijten, and J. Visser. Faster issue resolution with higher technical quality of software. *Software Quality Journal*, pages 1–21, 2011. To appear.

[18] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering on European software engineering conference and foundations of software engineering symposium*, Aug 2009.

[19] T. A. Budd. *Mutation Analysis of Program Test Data*. phdthesis, Yale University, 1980.

[20] T. A. Budd. Mutation analysis: ideas, examples, problems and prospects. In *Computer Program Testing*, pages 19–148. Chandrasekaran and Radicchi, Eds., 1981.

[21] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.

[22] L. A. Clarke, J. Hassell, and D. J. Richardson. A close look at domain testing. *IEEE Trans. Softw. Eng.*, 8(4):380–390, July 1982.

[23] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. Softw. Eng.*, 15(11):1318–1332, November 1989.

[24] G. Concas, M. Marchesi, S. Pinna, and N. Serra. Power-laws in a large object-oriented software system. *IEEE Transactions on Software Engineering*, 33(10):687–708, 2007.

[25] J. P. Correia, Y. Kanellopoulos, and J. Visser. A survey-based study of the mapping of system properties to iso/iec 9126 maintainability characteristics. *Software Maintenance, IEEE International Conference on*, pages 61–70, 2009.

[26] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[27] A. Van Deursen, L. Moonen, A. Bergh, and G. Kok. Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001*, pages 92–95, 2001.

[28] M. Ellims, J. Bridges, and D. C. Ince. The economics of unit testing. *Empirical Softw. Engg.*, 11(1):5–31, March 2006.

[29] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, 14(10):1483–1498, October 1988.

[30] E. Giger, M. Pinzger, and H. Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 52–56. ACM, 2010.

[31] J. S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Trans. Softw. Eng.*, 9(6):686–709, November 1983.

[32] R. Guderlei, R. Just, and C. Schneckenburger. Benchmarking testing strategies with tools from mutation analysis. In *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 360–364. IEEE Computer Society, 2008.

[33] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, 3(4):279–290, July 1977.

[34] I. Heitlager, T. Kuipers, and J. Visser. A practical model for measuring maintainability. In *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, pages 30–39. IEEE Computer Society, 2007.

[35] P. M. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3):92–96, 1976.

[36] W. C. Hetzel and B. Hetzel. *The complete guide to software testing*. John Wiley & Sons, Inc. New York, NY, USA, 1991.

[37] W. E. Howden. Methodology for the generation of program test data. *IEEE Trans. Comput.*, 24(5):554–560, May 1975.

[38] W. E. Howden. Algebraic program testing. *Acta Informatica*, 10(1):53–66, 1978.

[39] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[40] W.E. Howden. Theory and practice of functional testing. *IEEE Software*, 2(5):6–17, 1985.

[41] A. Hunt and D. Thomas. *The pragmatic programmer: from journeyman to master*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[42] S. Hussain. *Mutation Clustering*. phdthesis, King's College London, 2008.

[43] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. In *Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, pages 249–258, 28-29 September 2008.

[44] Y Jia and M Harman. An analysis and survey of the development of mutation testing. *Software Engineering, IEEE Transactions on*, PP(99):1, 2010.

[45] S. Kim and E. J. Whitehead, Jr. How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 173–174. ACM, 2006.

[46] K. N. King and A. J. Offutt. A fortran language system for mutation-based software testing. *Softw. Pract. Exper.*, 21(7):685–718, June 1991.

[47] G. Kudrjavets, N. Nagappan, and T. Ball. Assessing the relationship between software assertions and faults: An empirical investigation. In *Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 204–212. IEEE Computer Society, 2006.

[48] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. Softw. Eng.*, 9(3):347–354, May 1983.

[49] B. Luijten. The Influence of Software Maintainability on Issue Handling. Master's thesis, Delft University of Technology, 2009.

[50] B. Luijten and J. Visser. Faster defect resolution with higher technical quality of software. In *4th International Workshop on Software Quality and Maintainability (SQM 2010)*, 2010.

[51] B. Luijten, J. Visser, and A. Zaidman. Assessment of issue handling efficiency. In Jim Whitehead and Thomas Zimmermann, editors, *Proceedings of the 7th Working Conference on Mining Software Repositories (MSR 2010)*, pages 94–97. IEEE Computer Society, 2010.

[52] M. R. Lyu, Z. Huang, S. K. S. Sze, and X. Cai. An empirical study on testing and fault tolerance for software reliability engineering. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, ISSRE '03, pages 119–. IEEE Computer Society, 2003.

[53] Y. Ma, Y. Kwon, and J. Offutt. Inter-class mutation operators for java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, ISSRE '02, pages 352–. IEEE Computer Society, 2002.

[54] L. Madeyski. The impact of test-first programming on branch coverage and mutation score indicator of unit tests: An experiment. *Inf. Softw. Technol.*, 52(2):169–184, February 2010.

[55] A. P. Mathur. Performance, effectiveness, and reliability issues in software testing. In *Proceedings of the 5th International Computer Software and Applications Conference (COMPSAC'79)*, pages 604–605, 11-13 September 1991.

[56] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, 2(4):308–320, 1976.

[57] B. Meek and K. K. Siu. The effectiveness of error seeding. *ACM Sigplan Notices*, 24(6):89, 1989.

[58] T. Mens and S. Demeyer, editors. *Software Evolution*. Springer, 2008.

[59] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, 2006.

[60] H. D. Mills. On the statistical validation of computer programs. *IBM Federal Systems Division, Report FSC-72-6015*, pages 72–6015, 1972.

[61] A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: the apache server. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 263–272. ACM, 2000.

[62] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink. On the interplay between software testing and evolution and its effect on program comprehension. In *Software Evolution*, pages 173–202. Springer, 2008.

[63] G. J. Myer. *The art of software testing*. John Wiley & Sons, Inc. New York, NY, USA, 1979.

[64] N. Nagappan. *A software testing and reliability early warning (strew) metric suite*. PhD thesis, 2005. AAI3162465.

[65] N. Nagappan, L. Williams, M. Vouk, and J. Osborne. Early estimation of software quality using in-process testing metrics: a controlled case study. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, May 2005.

[66] S. C. Ntafos. On required element testing. *Software Engineering, IEEE Transactions on*, SE-10(6):795 –803, nov. 1984.

[67] S. C. Ntafos. A comparison of some structural testing strategies. *Software Engineering, IEEE Transactions on*, 14(6):868 –874, June 1988.

[68] A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. PhD thesis, University of Leiden, 2010. ISBN 978-90-9025677-1.

[69] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 5(2):99–118, 1996.

[70] A. J. Offutt, Y. Ma, and Y. Kwon. An experimental mutation system for java. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–4, September 2004.

[71] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *Proceedings of the 15th International Conference on Software Engineering (ICSE'93)*, pages 100–107, May 1993.

[72] F. Oppedijk. Comparison of the SIG Maintainability Model and the Maintainability Index. Master's thesis, University of Amsterdam, 2008.

[73] M. R. Paige. Program graphs, an algebra, and their implication for programming. *IEEE Trans. Software Eng.*, 1(3):286–291, 1975.

[74] M. R. Paige. An analytical approach to software testing. *Proceedings COMPSAC78*, pages 527–532, 1978.

[75] D. E. Perry, A. A. Porter, and L. G. Votta. Empirical studies of software engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 345–355. ACM, 2000.

[76] A. Podgurski and L. Clarke. The implications of program dependencies for software testing, debugging, and maintenance. *SIGSOFT Softw. Eng. Notes*, 14(8):168–178, November 1989.

[77] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Softw. Eng.*, 16(9):965–979, September 1990.

[78] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11(4):367–375, April 1985.

[79] S. Reichhart, T. Grba, and S. Ducasse. Rule-based assessment of test quality. In *IN PROCEEDINGS OF TOOLS EUROPE*, 2007.

[80] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, April 2009.

[81] D. Schuler and A. Zeller. Javalanche: efficient mutation testing for java. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, pages 297–298. ACM, 2009.

[82] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, 30(4):1–5, May 2005.

[83] C. Spearman. Demonstration of formulæ for true measurement of correlation. *The American Journal of Psychology*, 18(2):161–169, April 1907.

[84] R. H. Untch. Mutation-based software testing using program schemata. In *Proceedings of the 30th Annual Southeast Regional Conference (ACM-SE'92)*, pages 285–291, 1992.

[85] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *Software Engineering, IEEE Transactions on*, 33(12):800 –817, dec. 2007.

[86] J. Voas. How assertions can increase test effectiveness. *IEEE Softw.*, 14(2):118–119,122, March 1997.

[87] A. H. Watson, T. J. McCabe, and D. R. Wallace. Structured testing: A testing methodology using the cyclomatic complexity metric. *NIST Special Publication*, 500(235):1–114, 1996.

[88] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 1–. IEEE Computer Society, 2007.

[89] E.J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, 12(12):1128–1138, 1986.

[90] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Trans. Softw. Eng.*, 6(3):247–257, May 1980.

[91] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering: an introduction*. Kluwer Academic Publishers, 2000.

[92] M. R. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutationtesting issues. In *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA'88)*, pages 152–158, July 1988.

[93] M. R. Woodward, D. Hedley, and M. A. Hennell. Experience with path analysis and testing of programs. *IEEE Trans. Softw. Eng.*, 6(3):278–286, May 1980.

[94] R.K. Yin. *Case study research: Design and methods*, volume 5. Sage Publications, Inc, 2009.

[95] A. Zaidman, B. Van Rompaey, A. van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011. 10.1007/s10664-010-9143-7.

[96] S. J. Zeil. Testing for perturbations of program statements. *IEEE Trans. Softw. Eng.*, 9(3):335–346, May 1983.

[97] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., 2005.

[98] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997.