Test Code Quality and Its Relation to Issue Handling Performance

Dimitrios Athanasiou, Ariadi Nugroho *Member, IEEE*, Joost Visser *Member, IEEE Computer Society* and Andy Zaidman *Member, IEEE Computer Society*

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 to 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 Software Quality Model of the Software Improvement Group 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 to 18 open source systems. The test quality ratings are tested for correlation with issue handling indicators, which are obtained by mining issue repositories. In particular, we study the (1) defect resolution speed, (2) throughput and (3) productivity issue handling metrics. The results reveal a significant positive correlation between test code quality and two out of the three issue handling metrics (throughput and productivity), indicating that good test code quality positively influences issue handling performance.

Index Terms—Testing, Defects, Bugs, Metrics, Measurement.

1 Introduction

Software testing is well established as an essential part of the software development process and as a quality assurance technique widely used in industry [1]. Furthermore, literature suggests that 30 to 50% of a project's effort is consumed by testing [2]. Developer testing (a developer test is "a codified unit or integration test written by developers" [3]) in particular, has risen to be an efficient method to detect defects early in the development process [4]. In the form of unit testing, its popularity has been increasing as more programming languages are supported by unit testing frameworks (e.g., JUnit, NUnit, etc.).

The main goal of testing is the detection of defects. Developer testing adds to this the ability to point out *where* the defect occurs [5]. The extent to which detection of the cause of defects is possible depends on the quality of the test suite. In addition, Beck explains how developer testing can be used to increase confidence in applying changes to the code without causing parts of the system to break [6]. This extends the benefits of testing to include faster implementation of new features or refactorings. Consequently, it is reasonable to expect that there is a relation between the quality of the test code of a software system and the development team's performance in fixing defects and implementing new features.

Therefore, in this study we investigate the existence of such a relation to provide empirical evidence of the value of testing. In particular, we hypothesize that the higher the quality of the test code of a software system, the higher the development team's performance in handling issues, i.e., fixing defects and implementing new features goes faster.

To evaluate the aforementioned hypothesis, we formulate 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?

The assessment of test code quality is an open challenge [1]. 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 [7]. 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 study. In this paper we propose a test code quality model that is inspired by the Software Improvement Group (SIG)[1] quality model [8]. The model that we propose is solely based on source code metrics and does not require any other sources of information; in the proposed model test code quality has three dimensions, namely completeness, effectiveness, and

• *D. Athanasiou and A. Nugroho are with the Software Improvement Group, Amstelplein 1, 1096HA Amsterdam, The Netherlands.*
E-mail: dmitri.athanasiou@gmail.com, ariadi.nugroho@computer.org.
• *J. Visser is with the Software Improvement Group, Amstelplein 1, 1096HA Amsterdam, The Netherlands and the Model-Based Software Development Group, Radboud University Nijmegen, Heijendaalseweg 135, 6525 AJ Nijmegen, The Netherlands.*
E-mail: j.visser@sig.eu.
• *A. Zaidman is with the Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, Mekelweg 4, 2628CD Delft, The Netherlands.*
E-mail: a.e.zaidman@tudelft.nl.

1. For more information, visit http://www.sig.eu

maintainability. Several test adequacy criteria are then defined to assess those dimensions at the test code level.

To measure the issue handling performance of software development teams, Issue Tracking Systems (ITSs) can be mined. We expect that defect resolution time for a software system is reflected in its associated ITS as previous work suggests [9], [10], [11], [12], [13]. In addition, further indicators of issue handling performance, such as throughput and productivity, can be derived by studying ITS data as shown in [14]. Similar to [15], in this paper we measure issue handling performance as the speed of fixing issues.

The rest of this paper is structured as follows: we first provide some necessary background information in Section 2. Subsequently, we build our test code quality model in Section 3. In Section 4 we describe a case study to determine the alignment of our test code quality model with the opinion of experts. Section 5 explains the design of our study and Section 6 details the correlation study. Threats to validity are identified and discussed in Section 7, while related work is presented in Section 8. Section 9 concludes and identifies future research opportunities.

2 Background

Providing answers to the study's research questions requires knowledge foundations on the topics involved, namely: test code quality, issue handling and the Software Improvement Group (SIG) quality model. This section summarizes the existing literature related to these topics.

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 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 [1]. Zhu et al. [7] provide an overview of test adequacy criteria up to 1997. We now provide a brief overview of the state of the art by looking at the work of Zhu et al. and complementing it with more recent research.

The main role of test adequacy criteria is to assist software testers to monitor the quality of software in a better way by ensuring that sufficient testing is performed. In addition, redundant and unnecessary tests are avoided, thus contributing to controlling the cost of testing [7], [16].

We follow the classification of test adequacy criteria as proposed by Zhu et al. [7]. We 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. Specification 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. Program-based test adequacy criteria can be subdivided into categories for *structural testing* (Section 2.1.1), *fault-based testing* (Section 2.1.2) and *error-based testing* (Section 2.1.3). Other concepts (e.g., assertions, test smells, etc.) that can be used to measure test code quality will be discussed in Section 2.1.4

2.1.1 Structural Testing Adequacy Criteria

This category consists of test criteria that focus on measuring the coverage of the test suite upon the structural elements of the program. These criteria 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. Based on these criteria, metrics can be derived to measure the quality of the test code. Important work in this area is by Hetzel [17], Gourlay [18], Howden [19], Bently et al. [20], Myer [21] and Woodward et al. [22].

Data-flow criteria are concerned with analysing whether paths associating definitions of variables to their uses are tested. Research in this area was performed by Frankl and Weyuker [23], Rapps and Weyuker [24], Ntafos [25], Clarke et al. [26] and Laski and Korel [27].

In addition, efforts have been made to combine both of the aforementioned criteria [28].

2.1.2 Fault-based Testing Adequacy Criteria

Criteria that fall inside this category focus on measuring the defect detection ability of a test suite. *Error seeding* and *mutation analysis* are the main approaches to fault-based test adequacy criteria. These techniques can be applied to acquire test effectiveness indicators. Error seeding is the technique of planting artificial errors in a software system and subsequently testing the system against these artificial errors and counting the successfully detected ones [29]. Mutation analysis, is a more systematic way of performing error seeding [30], [31].

2.1.3 Error-based Testing Adequacy Criteria

This category of test criteria focuses on measuring to what extent the error-prone points of a program (as derived from the current knowledge level, e.g., our knowledge about how programs typically depart from their specifications) are tested [7]. To identify error-prone points, a domain analysis of a program's input space is necessary [7]. Relevant work in this area is presented by White and Cohen [32], Clarke et al. [33], Afifi et al. [34] and Howden [35]. Unfortunately, the application of error-based testing is limited when the complexity of the input space is high or when the input space is non-numerical [7].

2.1.4 Assertions and Test Code Smells

In addition to the criteria discussed in previous sections, researchers have also developed other indicators of the quality of test code.

Assertions: Kudrjavets et al. [36] 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 [37] 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 the key points of test cases at which something is *actually* tested, therefore, it is reasonable to expect assertion density to be an indicator of the effectiveness of the tests.

Maintainability: Test code has similar requirements for maintenance as production code. It is important to ensure that it is clear to read and understand, to ease its modification. 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. To detect possible points of low quality in the test code that require refactoring, van Deursen et al. [38] introduced *test smells*. Meszaros [5] extended the test smell catalogue, while Reichart et al. [39] and Van Rompaey et al. [40] worked towards automated test smell detection. Greiler et al. present strategies for avoiding test smells [41].

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 [42] and they enable developers to organise the issues of their projects. In this study, we focus on defects and enhancements.

When defects are discovered or new features are requested, they are typically reported to the ITS. Issues that are reported follow a specific *life-cycle*. Even though there is a variety of implementations of ITSs (e.g., BugZilla[2], Jira[3], GitHub[4]), they all adopt the same general process.

We now briefly describe the life-cycle of an issue report [43]. Initially, the report is formed and submitted as an *unconfirmed* issue. After it is checked whether the issue has already been reported or 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 to resolve it. The possible resolutions are:

- **Invalid** : The issue report is not valid (e.g., not described well enough to be reproduced).
- **Duplicate** : The issue has already been reported.
- **Fixed** : The issue is fixed.
- **Won't fix** : The issue will not be fixed (e.g., what the reporter thought of as a defect is actually a feature).
- **Works for me** : The issue could not be reproduced.

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 in 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 part of the process starts again.

2. http://www.bugzilla.org/
3. http://www.atlassian.com/software/jira/
4. http://github.com/

2.2.2 Defect Resolution Time

Defect resolution time is an indicator of the time that is needed to resolve a defect (enhancements are excluded from this metric). As previously discussed, 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 time necessary to resolve a defect (or a lower percentage of reopenings). However, before this claim can be evaluated, a representative measurement of the defect resolution time has to be defined.

Issue resolution is tracked by logging a series of possible actions. For each action in the ITS that changes the status of an issue, the date and time are recorded. An arguably straightforward measurement of the defect resolution time is 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 and 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 studies that involve defect resolution time. In particular, Luijten [44] showed that there exists negative correlation between defect resolution time and the software's maintainability. Giger et al. [11] worked on a prediction model of the fix time of bugs, acquiring the fix time from ITSs in the same way as described above. Nugroho [45] investigated the correlation between the fixing effort of defects related to modelled behaviours of functionalities and defects related to non-modelled behaviours of functionalities.

Ahsan et al. [13] 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.

Different approaches towards measuring the defect resolution time follow. Weiss et al. [10] predict the defect fixing time based on the exact duration of the fix as it was reported by developers in Jira, an example of an ITS that allows 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 when their ITS supports this, few developers actually supply this information (e.g., In JBoss, which was used in [10], only 786 out of the $11,185$ reported issues contained effort data).

Finally, Kim et al. [12] 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: they spot the commit that solved the defect by mining VCS logs for keywords such as "fixed", "bug" or references to the identification number of the defect report. They identify the commit that introduced the defect by applying the fix-inducing change identification algorithm by Sliwerski et al. [46], an approach based on linking the VCS to the ITS. Bird et al. [47] investigated the bias in such approaches and concluded that they pose a serious threat for the validity of the results.

There are many threats to validity for such a measurement, mainly because the information in ITSs is prone to inaccuracies. For instance, defects that have actually been resolved, sometimes remain open for a long time. Furthermore, even though it seems that a defect is being fixed during a certain time interval, the developer might not have been working on that issue continuously. Additionally, there is no information on whether more than one developer was working on the defect, increasing the actual fixing effort. Guo et al. noticed that bug reassignments happen frequently and while these reassignments add to the defect resolution time, they also observe that reassignments are not always harmful and are typically beneficial to find the best person to fix a bug [48].

Additionally, there are a number of factors that influence the lifetime of bug reports (and thus potentially also the defect resolution time). Hooimeijer and Weimer [49] report that easy-to-read issue reports are fixed faster. There are also a number of factors that influence whether bug reports are picked up sooner, namely: the presence of attachments, the presence of stack traces and the inclusion of code samples. Bettenburg et al.'s conjecture is that developers are likely to pick up on such cues since this can lessen the amount of time they have to deal with the bug [50].

2.2.3 Throughput and Productivity

Bijlsma [14] introduced additional indicators of issue handling performance, namely *throughput* and *productivity*. Unless mentioned otherwise, we discuss these measures at the level of *issues* and thus comprise both defects and enhancements. Both measures capture the number of issues that are resolved in a certain time period, corrected for respectively the size of the system and the number of developers working on the system.

When thinking of high quality test code, it seems logical to assume that developers working on a system benefiting from having high-quality test code will be able to get more done, both in terms of fixing defects, but also in terms of adding new functionality. This reasoning is instigated by the fact that these high-quality tests will make sure that the functionality that is not supposed to be changed can easily and quickly be tested through those tests that are already in place [51].

Throughput

Throughput measures the total productivity of a team working on a system in terms of issue resolution.

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

The number of resolved issues is averaged per month so that fluctuations of productivity because of events such as vacation periods, etc. have less impact. Moreover, 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.

Productivity

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. Again the number of resolved issues is averaged per month so that fluctuations of productivity because of events such as vacation periods, etc. have less impact. 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 in this study, the challenge in calculating productivity is to obtain the number of developers of the team. In [14] this is performed by mining the VCS of the system and counting the number of different users that committed code at least once. However, Mockus et al. [52] raise the 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 an investigation into the difference of the productivity indicator when the number of developers includes the whole team or just the core team.

Unless otherwise mentioned throughput and productivity are considered at the level of issues, i.e., combining defects and enhancements.

2.3 The SIG Quality Model

The Software Improvement Group, or SIG, is an Amsterdam-based consultancy firm specialized in quantitative assessments of software portfolios. It has developed a model for assessing the maintainability of software [53]. The SIG quality model (SIG QM) defines source code metrics and maps these metrics to the quality characteristics of ISO/IEC 9126 [8] that are related to maintainability.

In a first step, source code metrics are used to collect facts about a software system. The source code metrics that are used express volume, duplication, unit complexity, unit size, unit interfacing and module coupling. The measured values are combined and aggregated to provide information on properties at the level of the entire system. These system-level properties are then mapped onto the ISO/IEC 9126 standard quality characteristics that relate to maintainability, which are: analysability, changeability, stability and testability. The process described above is also depicted in Figure 1.

In a second step, after the measurements are obtained from the source code, the system-level properties are converted from metric-values into *ratings*. This conversion is performed through *benchmarking* and relies on the database that SIG possesses and curates; the database contains hundreds of systems that were built using various technologies [55]. Using this database of systems, the SIG QM model is calibrated so that the metric values can be

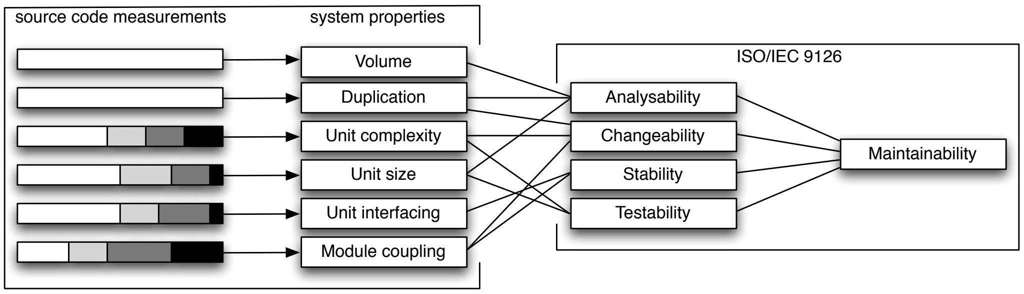

Fig. 1. The SIG Quality Model maps source code measurements onto ISO/IEC 9126 quality characteristics (image taken from [54]).

converted into star ratings that reflect the system's performance in comparison with the benchmark. This process results in a system getting attributed 1 to 5 stars (after rounding an intermediate score in the range of $[0.5\dots5.5)$).

The five 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 (i.e., top-quality) on a property performs similarly to the best 5% of the systems in the benchmark (indicated by the last value in the vector). At the same time, a 2 star rating means the system performs better than the 5% worst systems (the first value in the vector) and worse than the 65% best systems (the sum of the 3rd, 4th and 5th value in the vector) in the benchmark. More information on this quality model can be found in [53].

While the SIG QM solely takes into account the *production code*, the test quality model that we are proposing takes into account *test code*. The test code quality model will be discussed in more detail in Section 3 and is built and calibrated in a similar fashion to the SIG QM.

3 Building a Test Code Quality Model

In this section *RQ1* is addressed: *How can we evaluate the quality of test code?* Using a Goal-Question-Metric (GQM [56]) approach, we first investigate how test code quality can be measured and what information is needed to assess the various aspects of test code. Afterwards, metrics that are related to each of the identified aspects are 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 to calibrate the model and convert its metrics into quality ratings.

3.1 Research Questions

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

Q1 *How completely is the system tested?*
To answer **Q1** we can consider different ways to measure how completely a system is tested. As shown in Section 2.1.1, there are various code coverage criteria. In fact, An and Zhu [57] 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. To increase simplicity, applicability and understandability of the model, we will answer **Q1** by refining it into:

- **Q1.1** *How much of the code is covered by the tests?*
- **Q1.2** *How many of the decision points in the code are tested?*

Q2 *How effectively is the system tested?*
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 (1) to detect defects and (2) to locate the cause of these defects to facilitate the fixing process. Consequently, the following subquestions refine **Q2**:

- **Q2.1** *How able is the test code to detect defects in the production code that it covers?*
- **Q2.2** *How able is the test code to locate the cause of a defect after it detected it?*

Q3 *How maintainable is the system's test code?*
To answer **Q3**, we adopt and adapt the quality model that was developed by SIG [8], [53].

3.2 Metrics

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

3.2.1 Code Coverage

Code coverage is the most frequently used metric for test code quality assessment and there exist many tools for dynamic code coverage estimation (e.g., Clover[5] and Cobertura[6] for Java, Testwell CTC++[7] for C++, NCover[8] for C#). The aforementioned tools use a dynamic analysis

5. http://www.atlassian.com/software/clover/
6. http://cobertura.sourceforge.net/
7. http://www.testwell.fi/ctcdesc.html
8. http://www.ncover.com/

approach 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 [3]. 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 [58]. Second, dynamic analysis requires execution of the test suite, a task that is time consuming [59], [60].

Alves and Visser [58] 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. Their static estimation approach showed strong and statistically significant correlation with dynamic coverage estimation using Clover (with a mean of the absolute differences around 9%). As such, we use this approach to obtain a code coverage metric in our test code quality model.

3.2.2 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 inspired by the Cyclomatic-Number test adequacy criterion [7] and 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 McCabe's cyclomatic complexity [61] for the whole production code.

3.2.3 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, *assertion density* is defined as follows [36]:

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

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

3.2.4 Directness

As explained in Section 3.1, an effective test should provide the developers with the location of the defect to facilitate the fixing process. When each unit is tested individually by the test code, a broken test that corresponds to a single unit immediately pinpoints the defect. *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. To measure *directness*, the static code coverage estimation tool of Alves and Visser [58], which uses slicing of static call graphs, was modified so that it not only provides a static estimation of test coverage, but also outputs the percentage of the code that is *directly* called from within the test code.

3.2.5 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 [62]. The SIG quality model was designed to take into consideration the maintainability of production code.

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 Table 1. 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* [5] 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 test code smells.

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

TABLE 1
The test code maintainability model as adjusted from the SIG quality model [53]

As explained in Section 2.3, the original SIG quality 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" [62]. 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” [62]. Changes in the test code are often necessary when changes in the requirements lead to changes in the system [63].

Stability is “the capability of the software product to avoid unexpected effects from modifications in the software” [62]. 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” [62]. 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. However, testability, although applicable, implies that we would be interested in testing the test code. Such a step is not common practice as the adequacy of test code is mostly determined through other means, e.g., mutation analysis [31].

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 typically 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. We do not consider the volume of the test code to directly influence its analysability, because unit tests are typically independent of each other, which means that it is typically enough to analyze a single unit test (as captured in the *unit size* metric). We therefore chose not to use the volume metric.

Test code duplication occurs when *copy-paste* is used as a way to reuse test 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* [5]. 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.

The relation between unit size and maintainability is recognized both in the context of production code [64] and test code [5]. As unit size increases, it becomes harder to analyse. Unit size could be a warning for the *Obscure Test* and the *Eager Test* code smells [5]. 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. An eager test attempts to test too much functionality.

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, to avoid writing tests for test code, the test code should be kept as simple as possible. This is also underlined in the description of the *Conditional Test Logic* code smell [5], which advocates to keep the number of possible paths as low as possible to keep tests simple and correct. 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.

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 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 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 [5]).

To measure the dependence of a test code unit to production code we count the number of calls [65, p.29] 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 apply 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 [5]), increasing the test code’s maintenance effort.

As a conclusion of the analysis of the relevance of each system property to test code quality, the system properties that were selected for assessing test code quality maintainability are duplication, unit size, unit complexity and unit dependency. The explanation of the metrics used for duplication, unit size and unit complexity can be found in [8]. Briefly, duplication is measured as the percentage of all code that occurs more than once in identical code blocks of at least 6 lines (ignoring white lines). Unit size is measured as the number of lines of code in a unit. For unit complexity, the cyclomatic complexity of each unit is measured. Finally, unit dependency is measured as the number of unique outgoing calls (fan-out) from a test code unit to production code units, as mentioned earlier.

3.3 The Test Code Quality Model

Now that we have selected the metrics, we can present the test code quality model. The sub-characteristics of the model are derived from the questions **Q1**, **Q2** and **Q3**. In detail, they are: *completeness*, *effectiveness* and *maintainability*. The mapping of metrics to the sub-characteristics is done as depicted in Table 2, with the note that the adjusted SIG quality model combines duplication, unit size, unit complexity and unit dependency into a maintainability rating (see Table 1).

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

TABLE 2
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 3.2).

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 [66]. We chose the geometric mean:

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

3.4 Calibration

The metrics on which the test code quality model is based were calibrated to derive thresholds for risk categories and quality ratings. Calibration was done against a benchmark, following the methodology that was also used to calibrate the SIG quality model [67], [55].

TABLE 3
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

3.4.1 Set of benchmark systems

The set of systems in the benchmark includes 86 proprietary and open source Java systems that contained at least one JUnit test file. From the 86 systems, 14 are open source, while the others are proprietary.

Table 3 provides some general information on the open source systems in the benchmark. We observe that the systems' production Java code volume ranges from ~ 2 KLOC to ~ 140 KLOC. For the proprietary systems the range is entirely different: from ~ 1.5 KLOC to ~ 1 MLOC. For test code, the range for open source systems is from ~ 1.7 KLOC to ~ 130 KLOC. For the proprietary systems test code ranges from 20 LOC to ~ 455 KLOC. Further information about the proprietary systems cannot be published due to confidentiality agreements.

3.4.2 Descriptive statistics

Before applying the calibration methodology, we studied the distributions of the various metrics. Table 4 summarizes descriptive statistics for the metrics. Fig. 2 shows box-plots that illustrate the distributions of the system level metrics and a quantile plot that shows the distribution of one of the unit level metrics[9].

For the system level metrics, we observe that they cover a large range starting from values that are close to zero. Code coverage ranges up to $\sim 92\%$, with a large group of systems ranging between 40% and 70%, and the median at $\sim 46\%$. 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. Duplication in test code ranges from 0% to 23.9% with a median value of 12.9% duplicated test code in our set of 86 benchmark projects.

The unit level metrics resemble a power-law-like distribution. The summary statistics in Table 4 as well as the quantile plot in Fig. 2 show that most of the values of these metrics are low but there is a long tail of much higher values towards the right, confirming observations from earlier studies [69], [55].

9. Only the quantile plot for unit complexity is shown, the plots for unit size and unit dependency can be found in [68].

TABLE 4
Metrics and their summary statistics (all 86 systems in the benchmark)

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	0.27	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

From these observations, we conclude that the test code metrics we selected behave in similar ways to other source code metrics, in particular to those used in the SIG quality model, which means that we can apply the same calibration methodology for the test quality model as for the SIG quality model.

3.4.3 Risk categories and quality ratings

In a similar fashion to the SIG quality model (Section 2.3 and [67], [55]), we want to come to a star rating with a 5-point scale for the test code quality of software systems. We define the quality levels so that they correspond to a $<5, 30, 30, 30, 5>$ percentage-wise distribution of the systems in the benchmark. This means that a system that is rated 1 star is situated amongst the 5% worst performing projects in terms of test code quality, that a 2-star system is doing better than the 5% 1-star systems, but 65% of the systems are doing a better job at test code quality, etc. Finally, the top-5% systems are attributed a 5-star rating.

System level metrics

Having established this top-level $<5, 30, 30, 30, 5>$ distribution in terms of star-ratings, our calibration approach translates this star-rating into thresholds for system level metrics, i.e., between what thresholds should metrics for a system be to fall into a star-category.

For the system level metrics, the thresholds that result from the aforementioned calibration are shown in Table 5. All these threshold values are determined using the database of 86 proprietary and open-source Java systems as discussed in Section 3.4.1. For example, to score five stars on code coverage, a system should have coverage of at least 73.6%, while to score two stars, 0.6% is enough. Intuitively, this last threshold may seem too lenient, but it reflects the quality of the systems in the benchmark, indicating that at least 5% of the systems are tested inadequately.

Unit-level metrics

For the unit level metrics, calibration is performed in two steps. First, thresholds are derived to categorize units into four risk categories (low, moderate, high and very high). We define the thresholds for these four risk categories based on metric values that we observe in our benchmark of 86 Java systems. More specifically, we use the 70, 80, and 90 percentiles of metric-values to determine the boundaries between the four risk categories. We reused the volume percentiles 70, 80 and 90 that were successfully used on metrics with power-law distributions in the SIG quality model [55]. Alves et al. [55] settled on these 70, 80 and 90 volume percentiles after empirically assessing that these boundaries correspond to volumes of code that typically fall in the category *low risk* [0%, 70%[or needs to be fixed in long-term, to [90%, 100%[, which would indicate that this code needs to be improved in the short-term. The derived thresholds that correspond to these code volume percentiles are shown in Table 6. For instance, when we look at unit size, the 70, 80 and 90 percentile correspond to respectively 24, 31 and 48. Translating this into the risk categories means that a unit with a size of less than 24 (lines of code), is easy to maintain, thus we classify it as low risk. Similarly, when $24 < unitsize \leq 31$, we classify the unit as having moderate risk or we classify it as having high risk when unit size is between 31 and 48. A unit of size greater than 48 is classified as being very high risk, indicating that the maintenance of such a test unit can become difficult.

TABLE 5
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	5.5%	10.3%	16.4%	21.6%	-

TABLE 6
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

Secondly, thresholds are derived to map the relative volumes of the risk categories into star ratings, shown in Table 7. For example, to qualify for a 5-star rating (and hence belong to the 5% best systems) on the unit size property, a maximum of 12.3% of the units can belong to the *moderate risk* profile, 6.1% of the code can fall into the *high risk* category and 0.8% of the code can be categorized as *very high risk* (and as a consequence at least 80.8% of the code falls into the low risk category — not shown in Table 7). As such, a snapshot of a software system belongs to the highest star rating for which all thresholds hold.

4 Test Code Quality Model Alignment

We wanted to know whether the test code quality model aligns with the opinion of experts, i.e., whether a software system that gets a good score according to our test code quality model would also be seen as having high-quality tests by software engineers. To verify this, we performed a preliminary investigation involving two industrial software systems which were entrusted to the Software Improvement Group to perform a source code quality assessment

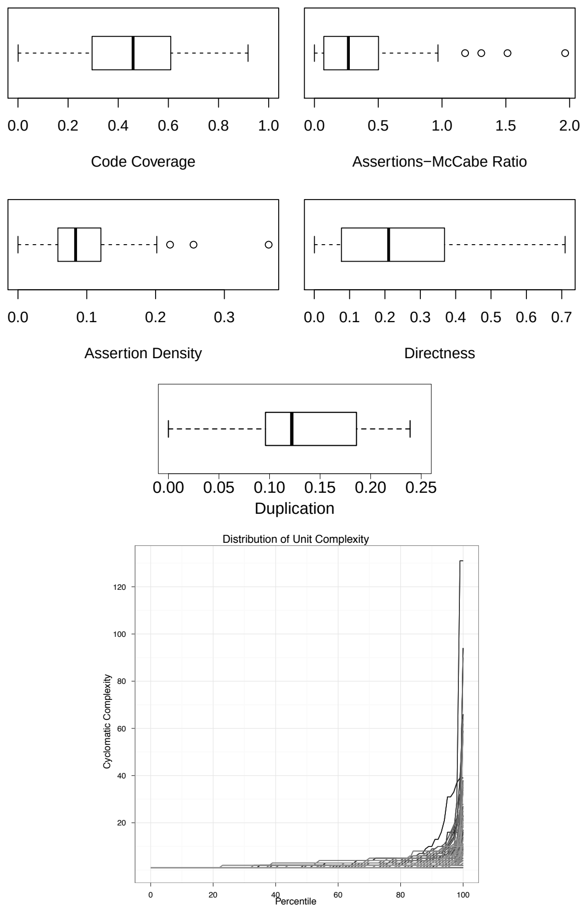

Fig. 2. The distributions of the metrics

TABLE 7
Profile thresholds for Unit Size, Unit Complexity and Unit Dependency

	maximum relative volume of code (in %)								
rating	Unit size			Unit Complexity			Unit dependency		
	moderate	high	very high	moderate	high	very high	moderate	high	very high
⋆ ⋆ ⋆ ⋆ ⋆	12.3	6.1	0.8	11.2	1.3	0.3	10.0	4.3	1.2
⋆ ⋆ ⋆ ⋆	27.6	16.1	7.0	21.6	8.1	2.5	19.3	13.9	7.8
⋆ ⋆ ⋆	35.4	25.0	14.0	39.7	22.3	9.9	33.5	24.1	14.6
⋆ ⋆	54.0	43.0	24.2	62.3	38.4	22.4	52.1	38.9	24.1
⋆	-	-	-	-	-	-	-	-	-

on them. The experts involved are consultants from the Software Improvement Group who perform source code assessments using the SIG quality model on a daily basis.

The investigation was set up as a focused interview [70] with the consultant responsible for the particular system. 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. They are however expert users of the SIG quality model, which uses the same star-based rating system. 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 models

TABLE 8
Test code quality ratings of the experts for systems A and B.

Aspect	System A		System B	
	Model	Expert	Model	Expert
Completeness	2.8	3.0	3.1	3.5
Effectiveness	2.8	2.0	3.6	4.0
Maintainability	2.1	3.0	3.7	-
Overall Test Code Quality	2.5	3.0	3.5	4.0

TABLE 9
Test Code Quality Model Ratings for System A.

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			

ratings and the experts' evaluation.

4.1 System A

System A is a logistics system developed by a Dutch company. The programming language used is Java, with a lot of SQL code embedded into the Java code. The system's production code volume at the moment of the case study was ~ 700 KLOC, with ~ 280 KLOC JUnit code.

The system has been in maintenance since 2006 with no new functionality being added. In 2010 a re-structuring of the system has been performed, with extensive modularization and the addition of a lot of testing code.

Table 9 shows the results of the application of the test code quality model. With an overall rating of 2.5, it indicates many weaknesses of the system's test code.

4.1.1 Test Code Quality Assessment by the Expert

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

How completely is the system tested? The expert reported poor code coverage for System A, with only lower layers of the system's architecture being tested. The expert had 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 $\sim 43\%$. Extrapolating this value to the whole system we obtain a code coverage level between 37% and 51%.

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. Furthermore, maintainability is hindered by high complexity and coupling between the test code and the production code.

To which extent is the test code targeting unitary or integration 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 8. The expert claimed to have had significant experience in analysing the software.

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

The system's overall test code quality is rated at 2.5, while the expert's evaluation was 3.0. The difference is only one unit in the scale of measurement which was used by the expert to rate the system's test code quality. In particular, completeness was calculated as 2.8 by the model, a rating that is aligned to the expert's evaluation (3.0). However, for effectiveness and maintainability, the model's ratings deviate from the expert's evaluation by more than one unit (2.8 against 2.0 and 2.1 against 3.0 respectively).

4.2 Case Study: System B

System B involves a designing system for engineers, developed by a Dutch-based software company. The size of the development team is approximately 15 people. The system's production code volume is ~ 243 KLOC, with another ~ 120 KLOC JUnit code. Test-driven development (TDD) was adopted over the past 18 months; most of the testing effort came during that period.

The architecture of the system has recently undergone some major changes: the system's main modules were rewritten, although the system is still using the old, legacy modules. This coexistence of old and new modules separates the system in two parts, also in terms of quality. This is reflected in the maintainability ratings of the system: the SIG quality model gave a rating of 3.3 stars for the whole system; analyzing only the newly written modules the rating rises to 4.0 stars, reflecting the team's focus to increase the quality.

4.2.1 Test Code Quality Model Ratings

Table 10 shows the results for System B. 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 is at 52.5%, while the Assertions-McCabe ratio is at a lowly 0.29.

Effectiveness was rated at 3.6, which is higher than completeness, indicating 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 System B being tested directly.

Maintainability at 3.7 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) reveals a possible space for improvement of the test code's maintainability.

TABLE 10
Test Code Quality Model Ratings for System B.

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			

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.

4.2.2 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? According to the expert, the legacy modules are tested weakly. Code coverage is around 15%. The newly developed modules have 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 ~ 135 KLOC of the system's total of ~ 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 Test-Driven Development (TDD) a decrease in the number of incoming defect reports was noticed.

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 tests, 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 8.

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

The model's ratings for System B 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. 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 B 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 when applying the model only to the new modules the ratings are aligned to those of the expert. Completeness, effectiveness, maintainability and overall test code quality are 4.1, 3.9, 3.9 and 4.0 respectively.

4.3 Discussion

Even though the lack of data does not enable us to draw strong conclusions from 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.

By looking at Table 8 we see that for the sub-characteristics of the test code quality model (completeness, effectiveness and maintainability) the ratings of the expert and the model diverge at most 0.9 (for the maintainability of System A). For most sub-characteristics, the ratings of the expert and the model are relatively close together and diverge for at most 0.5. For the overall test code quality rating, the opinion of the two experts and the test code quality model diverges for 0.5 on the quality rating scale. This does indicate that the model is relatively closely aligned to the opinion of experts and that the model's accuracy is promising.

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

- Interpretation of the model's results should take into consideration that the model is based on benchmarking, sometimes leading to ratings that can be counter-intuitive, e.g. directness rated at 2.8 when direct coverage is at a low 17.5%.
- 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.

5 Design of Study

In this section, the design of the study to answer RQ2 is discussed.

5.1 Design of the Experiment

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

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

An overview of the experiment is shown in Fig. 3.

5.2 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 Section 3 as a measurement of test code quality. We extract issue handling measurements from the ITSs of several open source Java projects.

As seen in Section 1, we assume that systems of higher test code quality will have shorter defect resolution times, and higher throughput and productivity. 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 hypotheses:

H1$_{null}$: There is no significant correlation between test code quality and defect resolution time.
H1$_{alt}$: Higher test code quality significantly correlates with lower defect resolution time.

H2$_{null}$: There is no significant correlation between test code quality and throughput.
H2$_{alt}$: Higher test code quality significantly correlates with higher throughput.

H3$_{null}$: There is no significant correlation between test code quality and productivity.
H3$_{alt}$: Higher test code quality significantly correlates with higher productivity.

All three hypotheses are formulated as one-tailed hypotheses because we have a specific expectation about the direction of the relationship between the two variables: higher test code quality correlates with higher issue handling performance.

5.3 Measured Variables

The measured variables are summarised in Table 11.

TABLE 11
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 independent variable in all three hypotheses is the test code quality, measured using the model presented in Section 3. The outcome of the model is a rating that reflects the quality of the test code of the system. The ratings are in interval scale and the values are in the range of $[0.5, 5.5]$.

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 2.2.3. To derive the resolution time of a defect, we rely on the approach presented in [15]. The summary of the approach follows.

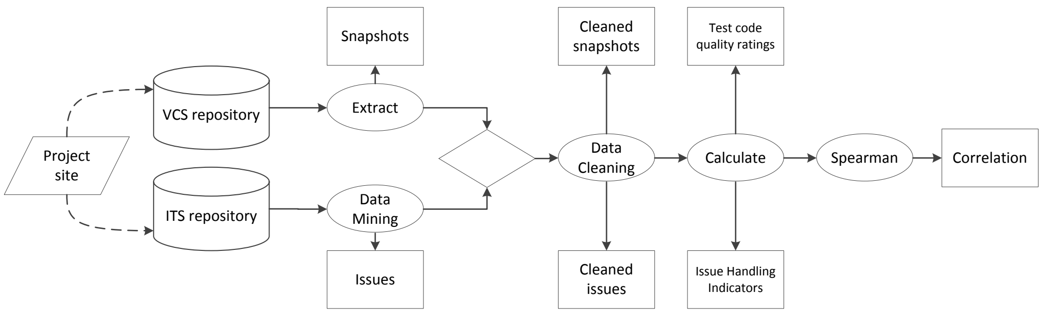

Fig. 3. Procedure Overview

5.3.1 *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.

In Section 2.2.2 we explained that to measure the resolution time of a defect, the time during which the defect was in an open state in the ITS is measured. 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 resembles a power-law-like distribution as illustrated by Bijlsma et al. [15]. 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.

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 [15]. In [15] Bijlsma et al. used 10 projects (e.g., webkit, tomcat and hibernate) to perform this calibration. The thresholds of the risk categories are shown in Table 12 and the thresholds for the quality ratings are shown in Table 13.

TABLE 12
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	

As with the test code quality model, the thresholds for the risk categories are applied on the measurement of a defect's resolution time to classify it in a risk category. Afterwards, the percentage of defects in each risk category is calculated. Finally, the thresholds for the quality ratings are used to derive a quality rating for the defect resolution speed. Interpolation is used once again 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 to measure the dependent variable for *Hypothesis 1*. It should be noted that higher rating means shorter defect resolution times.

TABLE 13
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%

5.4 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. 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 on the developers, making them resolve more issues.

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 (committers) of open source systems are many times anonymous and there is no reliable data 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 [50]. 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 Section 7 as threats to validity for the outcome of the experiment.

5.5 Data Collection and Preprocessing

To investigate the relation between test code quality and issue handling, we need (1) source code and (2) issue tracking data of systems. Open source systems provide their source code publicly and in some of them, the ITS data is also publicly available. We chose to go with open source systems because of the availability of both the source code and the ITS data, which — from our own experience — is not always the case with industrial systems.

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 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 to make sure that we have a diverse population of issues.

Further selection was applied 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. We discarded *tasks* because these kinds of issues typically also include non-technical elements, e.g., requirements analysis. Patches are not taken into account because these issues typically already contain (part) of the solution that needs to be tested and integrated [71]. While it would be interesting to see whether patches get accepted more quickly with high-quality tests in place, we defer this investigation to future work.

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 to mitigate the fact that snapshots of the same system are not independent. In particular, snapshots were taken so that (1) there is a certain period of time (1 year) and (2) there is a certain percentage of code churn [72] (at least 30%; production and test code together) between two consecutive snapshots. The actual thresholds for these criteria were defined by experimenting to find the balance between the austerity for snapshot independence and the preservation of sufficient amount of data.

Important to note is that a snapshot can be an arbitrary commit (and does not have to be marked as release).

In addition, data cleaning was necessary 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. This threshold of 50 was established after studying the distribution of the number of issues closed the same day with the same message and witnessing a very sharp rise around that point.

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 (we followed the approach from Bijlsma for this step [15]).

5.6 Data Measurement and Analysis

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

Source code was downloaded from the repositories of the subject open source systems. The necessary metrics were calculated using SIG's Software Analysis Toolkit (SAT) [8]. The test code quality model ratings were calculated by processing the metrics with the use of R[10] scripts.

10. The R project for statistical computing, http://www.r-project.org/

To obtain the issue handling performance indicators the ITSs of the subject systems were mined. For this, the tool that was created by Luijten [9] and later reused by Bijlsma [14] was used for this study. The tool supports Bugzilla, Issuezilla, Jira and Sourceforge and extracts the data from the ITSs. In addition, the VCS history log is extracted. Afterwards, the issue tracking data, the VCS log and the source code metrics are stored in a database, of which the object model is shown in detail in [44, p.15]. The database can then be used for further analyses.

Correlation tests were performed to test the formulated hypotheses. Data of test code quality and defect resolution time does not follow a normal distribution. Therefore, a *non-parametric* correlation test is suitable. We chose to use Spearman's rank-correlation coefficient. 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, to assess the influence of the confounding factors to the issue handling indicators we performed multiple regression analysis. This 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 [73].

6 Correlation Study

To address *RQ2* an experiment was conducted aiming at assessing the relation between test code quality and issue handling performance. This section presents the results.

6.1 Data

In Section 5.5, 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, ranging from 17 KLOC for the most recent snapshot of the Stripes framework to 163 KLOC for ArgoUML. It should be noted that code which was identified as generated was disregarded from the scope of the analysis as it is not maintained manually. The total number of collected issues for the 18 systems is almost $160,000$, where around $110,000$ are defects and $49,000$ are enhancements. An overview of the dataset is shown in Table 14.

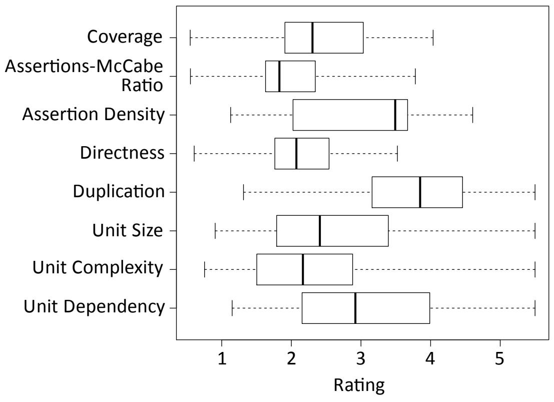

Fig. 4. Boxplots of the ratings of the test code quality model's properties

Please note that we use different open source systems in this correlation study (RQ2) as compared to the calibration of the test code quality model (RQ1). This ensures that both sets are independent.

After performing the data cleaning step described in Section 5.5, we end up with two different datasets as shown in Table 14: one for defect resolution speed (63 snapshots) and another for throughput and productivity (54 snapshots).

6.2 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. Fig. 4 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 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. 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.

Fig. 5 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.

TABLE 14
Snapshots and issues per system after selection and cleaning

Project	General information						Data for Defect Resolution Speed				Data for Throughput & Productivity			
	KLOC (latest)	Developers (max)	pCode Maintainability (latest)	tCode Quality (latest)	Earliest Snapshot Date	Latest Snapshot Date	Snapshots	Issues	Defects	Enhancements	Snapshots	Issues	Defects	Enhancements
Apache Ant	100	17	3.201	2.693	18/07/2000	13/03/2008	6	2,275	1,680	595	5	1,944	1,467	477
Apache Ivy	37	6	3.022	3.330	17/12/2006	26/09/2009	2	331	228	103	2	467	309	158
Apache Lucene	82	19	2.795	2.917	02/08/2004	06/11/2009	3	2,222	1,547	675	4	4,092	3,274	818
Apache Tomcat	159	13	2.531	1.595	21/10/2006	07/03/2009	2	295	268	27	2	275	244	31
ArgoUML	163	19	2.915	2.733	12/03/2003	19/01/2010	7	758	635	123	6	621	508	113
Checkstyle	47	6	3.677	2.413	05/02/2002	18/04/2009	6	251	248	3	4	203	200	3
Hibernate code	105	14	2.934	2.413	18/04/2005	15/08/2008	2	270	166	104	3	999	620	379
HSQLDB	69	8	2.390	2.039	06/10/2002	09/09/2009	4	295	295	0	3	356	354	2
iBatis	30	4	2.999	2.868	16/05/2005	12/10/2009	3	266	150	116	3	266	150	116
JabRef	83	17	2.574	2.727	28/11/2004	02/09/2009	4	480	480	0	4	480	480	0
jMol	92	9	2.208	1.814	06/06/2006	10/12/2007	2	64	63	1	2	64	63	1
log4j	12	6	3.966	2.365	17/05/2002	05/09/2007	4	384	323	61	2	101	86	15
OmegaT	112	6	3.278	2.448	20/06/2006	12/02/2010	3	353	192	161	3	353	192	161
PMD	35	15	3.865	2.975	14/07/2004	09/02/2009	4	176	153	23	3	98	77	21
Spring framework	145	23	3.758	3.123	13/05/2005	16/12/2009	3	5,942	2,947	2,995	1	3,829	1,923	1,906
Stripes framework	17	6	3.704	3.123	29/09/2006	28/10/2009	3	340	197	143	2	317	179	138
Subclipse	93	10	2.348	2.449	11/04/2006	11/08/2009	3	156	95	61	3	190	113	77
TripleA	99	4	2.493	2.449	21/07/2007	06/03/2010	2	204	204	0	2	187	187	0
18							63	15,062	9,871	5,191	54	14,842	10,426	4,416

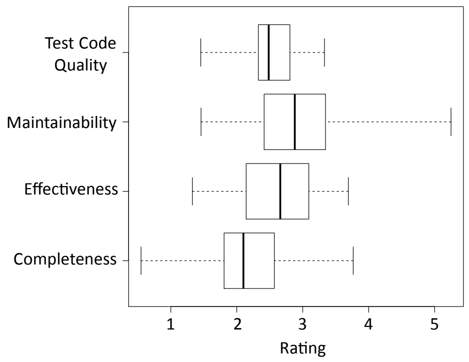

Fig. 5. Boxplots of the ratings of the test code quality model's sub-characteristics and overall test code quality

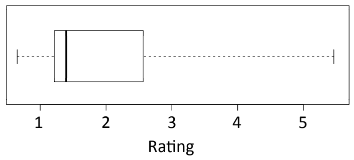

Fig. 6. Boxplot of the defect resolution speed ratings

Next, descriptive statistics about the dependent variables are presented. Fig. 6 show the distribution of the ratings for defect resolution speed and Table 15 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.

TABLE 15
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

The median for productivity is 0.99. 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.

6.3 Results of the Experiment

As discussed in Section 5, 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. Additionally, for throughput and productivity, we subdivided the issues into defects and enhancements. For each of these tests, a Spearman correlation test was performed. Table 16 shows the results of the correlation tests.

All the correlations are significant at the 99% confidence level except for 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 strong and statistically significant correlations with test code quality. The correlation coefficient is 0.50 and 0.51 for throughput and productivity

TABLE 16
Summary of correlations with the test code quality rating of the systems

	ρ	*p*-value	N
Defect Resolution Speed	0.06	0.330	63
Throughput	0.50**	**0.000**	54
Defect Throughput	0.43*	**0.001**	54
Enhancement Throughput	0.64**	**0.000**	54
Productivity	0.51**	**0.000**	54
Defect Productivity	0.45*	**0.000**	54
Enhancement Productivity	0.60**	**0.000**	54

**medium effect size; **large effect size*

respectively in the 99% confidence level. This enables us to reject $\mathbf{H2}_{null}$ and $\mathbf{H3}_{null}$ and maintain the alternative hypotheses: significant positive correlations exist between test code quality and throughput, and test code quality and productivity. In more detail, Table 16 also shows that there is significant positive correlation between test code quality and throughput and productivity at the level of defects and enhancements. In particular, the correlations are slightly stronger when only considering enhancements.

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 to acquire an indication of which aspects of test code are particularly influential on issue handling performance.

To indicate the effect size of correlations, we follow the guideline suggested by Cohen [74]. $\rho = 0.1$ is considered having a small effect size, while $\rho = 0.3$ and $\rho = 0.5$ are considered having medium and large effect sizes respectively. We indicate effect size for correlations having at least medium effect sizes.

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

Table 17 presents the correlation between the test code quality model's ratings and the defect resolution speed rating for the subject snapshots. No significant correlation is found between test code quality and defect speed rating, as such, no conclusion can be drawn regarding this hypothesis. Among the properties of test code quality, only code coverage has a significant correlation with defect resolution speed. However, code coverage is weakly correlated with defect resolution speed ($\rho = 0.28$).

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

Looking at Table 17, test code quality is significantly correlated with throughput at the 99% confidence level. Therefore, this hypothesis is maintained. In fact, it has the highest correlation coefficient ($\rho = 0.50$) compared to the sub-characteristics and properties levels. At the sub-characteristics level, completeness and effectiveness have similar, significant correlations with throughput. Maintainability is not significantly correlated with throughput. We can also 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 $\ll 0.01$).

If we consider defect throughput and enhancement throughput separately, we see that both are significantly correlated with test code quality at the 99% confidence level. The correlation is stronger for enhancement throughput.

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

Table 17 presents the correlation between the test code quality model's ratings and productivity. The overall rating of test code quality has a significant correlation with productivity ($\rho = 0.51$ and p-value $\ll 0.01$). Therefore, this hypothesis is maintained. At 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. We observe that the correlations behave similarly to those of 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.

If we again consider defect productivity and enhancement productivity separately, we observe that both show significant correlation with test code quality, with stronger correlation for enhancement productivity.

6.4 Interpretation of the Results

Defect resolution speed

Contrary to our expectations, test code quality was not found to be significantly correlated with defect resolution speed in our experiment. None of the model's properties was correlated with defect resolution speed except for code coverage, which was weakly 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 to be able to draw a more definitive conclusion. However, a close examination of the process of software development might provide a hint to explain this result.

During development changes are applied to the production code 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, 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; the developer will re-work the

TABLE 17
Correlation results for defect resolution speed, throughput and productivity.

	Defect Resolution Speed (N=63)		Throughput (N = 45)						Productivity (N = 45)					
			Defects		Enhancements		Combined		Defects		Enhancements		Combined	
	ρ	*p*-value	ρ	*p*-value	ρ	*p*-value	ρ	*p*-value	ρ	*p*-value	ρ	*p*-value	ρ	*p*-value
Code Coverage	0.28	**0.013**	0.27	**0.026**	0.39*	**0.000**	0.28	**0.021**	0.51**	**0.00**	0.43*	**0.000**	0.49*	**0.000**
Assertions-McCabe Ratio	0.01	0.480	0.41*	**0.000**	0.57**	**0.000**	0.48*	**0.000**	0.48*	**0.000**	0.56**	**0.000**	0.53**	**0.000**
Assertion Density	0.02	0.427	0.24	**0.036**	0.34*	**0.000**	0.29	**0.017**	0.26	**0.026**	0.32*	**0.008**	0.33*	**0.007**
Directness	0.08	0.260	0.26	**0.029**	0.45*	**0.000**	0.31*	**0.012**	0.32*	**0.009**	0.43*	**0.000**	0.36*	**0.004**
Duplication	−0.45	0.999	0.09	0.248	−0.04	0.611	0.10	0.246	−0.13	0.833	−0.14	0.846	−0.13	0.827
Unit Size	−0.11	0.800	0.03	0.408	−0.03	0.596	0.06	0.330	−0.11	0.786	−0.02	0.566	−0.09	0.747
Unit Complexity	−0.09	0.747	0.07	0.289	−0.10	0.771	0.06	0.321	−0.07	0.682	−0.13	0.825	−0.08	0.719
Unit Dependency	−0.17	0.905	0.06	0.327	0.02	0.442	0.10	0.236	−0.25	0.963	−0.06	0.678	−0.20	0.927
Completeness	0.12	0.182	0.38*	**0.002**	0.52**	**0.000**	0.42*	**0.001**	0.54**	**0.000**	0.53**	**0.000**	0.56**	**0.000**
Effectiveness	0.07	0.282	0.33*	**0.006**	0.57**	**0.000**	0.41*	**0.001**	0.41*	**0.001**	0.55**	**0.000**	0.49*	**0.000**
Maintainability	−0.29	0.989	0.07	0.299	−0.04	0.623	0.10	0.244	−0.26	0.971	−0.13	0.838	−0.24	0.957
Test Code Quality	0.06	0.330	0.43*	**0.001**	0.64**	**0.000**	0.50**	**0.000**	0.45*	**0.000**	0.60**	**0.000**	0.51**	**0.000**

**medium effect size; **large effect size*

source code, to make sure that his change does not make any test fail. This is how automated tests prevent defects from appearing.

Conversely, 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 listed in ITSs turns out not 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 discussed in Section 7, issues may have been resolved earlier than the moment they were marked as closed, or not marked as closed at all [50].

Throughput and productivity

On the other hand, throughput and productivity confirm our expectation that they are related to the quality of test code. Fig. 7 and 8 compare throughput and productivity with the different levels of test code quality. In particular, the snapshots were grouped in quality levels 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 Fig. 7 and 8 as circles are not considered outliers due to the lack of evidence after manual inspection as discussed in Section 6.2. Note that some of the extreme values were removed from the figures to improve readability.

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 3.4. 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 a bit higher than

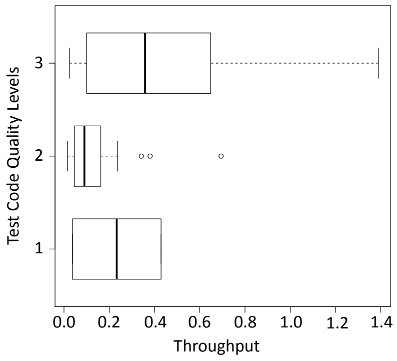

Fig. 7. Comparison between throughput and different test code quality levels

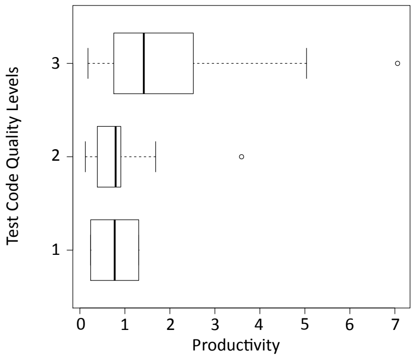

Fig. 8. Comparison between productivity and different test code quality levels

the median of 1-star snapshots, the difference is small. Productivity significantly improves for 3-star systems.

We also note that correlations for throughput and productivity are slightly stronger when only considering enhancements compared to only considering defects. This hints at the fact that high quality test code is more beneficial for implementing enhancements than it is for fixing bugs.

Sub-characteristics

As far as the influence of each of the sub-characteristics is concerned, we observe that completeness and effectiveness are both significantly correlated with 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 as it has no direct influence on the testing capability of the system. It rather focuses on the effort that is necessary to maintain the test code so that it remains as complete and as effective as it is. This may explain the lack of correlation between maintainability and issue handling performance indicators.

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 finding 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.

Finally, an interesting observation is the discrepancy of the result between defect resolution speed, and throughput and productivity. As it has already been discussed, the lack of correlation between defect resolution speed and test code quality could be explained by, among others, the expectation that the defects reported in an ITS are those that were not detected by the test code. This observation is also partly supported by the fact that when we consider throughput and productivity separately for defects and enhancements, the correlation is stronger for enhancements than it is for defects.

6.5 Controlling the confounding factors

In Section 5.1 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. 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 method involves constructing linear models that express the relationship between a dependent variable and the independent variables that influence it. The linear models under analysis are the following:

TABLE 18
Results of multiple regression analysis for throughput and productivity

	Model	Coefficient	Std. Error	t	p-value
Throughput	(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: $R^2 = 0.193; p \leq 0.01$				
Productivity	(Constant)	-3.406	2.004	-1.699	0.095
	Test Code Quality Rating	2.081	0.793	2.624	**0.011**
	Model Summary: $R^2 = 0.117; p = 0.011$				

$$\begin{aligned}&\text{Def. Res. Speed Rating} = \text{tCode Quality Rating} + \text{pCode Maintainability Rating} + \text{Team Size} + \text{c}\\&\text{Throughput} = \text{tCode Quality Rating} + \text{pCode Maintainability Rating} + \text{Team Size} + \text{c}\\&\text{Productivity} = \text{tCode Quality Rating} + \text{pCode Maintainability Rating} + \text{Team Size} + \text{c}\end{aligned} \tag{1}$$

where Def. Res. Speed Rating is the defect resolution speed rating, tCode Quality Rating is the test code quality rating, pCode Maintainability Rating is the production code maintainability rating.

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 18.

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 a weaker predictor (for an explanation of stepwise multiple regression analysis see Section 5.6). 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 our results (Section 6.3) 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.

7 Threats to Validity

In this section factors that may pose a threat to the validity of the study's results are identified. We follow the guidelines that were proposed by Perry et al. [75] and Wholin et al. [76], and organize the factors in four categories: construct, internal, external and conclusion validity.

7.1 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 [76].

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: Three indicators were used to measure different aspects of issue handling performance. Nevertheless, more aspects of issue handling can be captured. For example, the number of issues that are 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, e.g., while not trivial to perform, the difficulty of resolving each issue could be assessed. It is clear that the set of issue handling indicators used in this study is not complete, but it captures 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 [50]. 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. 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 data cleaning to reduce the noise in the ITSs data.

Number of developers: The number of developers was measured (1) to calculate productivity and (2) 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. 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, senior members of the project perform the commit on their behalf, (3) sometimes team members uses different aliases and (4) it has been demonstrated by Mockus et al. [52] 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 fourth problem we performed a similar analysis as done in [52] to calculate the number of developers in the core team.

In [52] 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 to calculate the number of developers that account for 80% of the commits. The commits per developer were calculated and sorted in decreasing order. Subsequently, we determined the cut-off point for 80% of the total commits and determined 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. This implies a large group of active developers. Again, the granularity of commits is a threat to the measurements.

After calculating productivity as the 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$\ll 0.01$).

Test coverage estimation: dynamic test coverage estimation techniques are known to provide greatly fluctuating values in the context of state-based software systems [77]. Our test code quality model does not make use of dynamic coverage estimation techniques, but rather uses the slice-based approach by Alves and Visser [58]. In theory, this approach should not be susceptible to the issue of state.

7.2 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 5.4 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.

A final remark over confounding factors concerns the correlation between test code quality and production code maintainability. Significant positive correlation of medium strength was found ($\rho = 0.42$ 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.

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.

We address these threats by establishing strict criteria for snapshot selection: a period of at least one year and at least 30% code churn between two consecutive snapshots.

Experts' experience with the case studies: The selection of the experts was mainly based on their knowledge about the measurements of the case studies. Both experts are working on analysing and evaluating the quality of software systems on a daily basis. In particular, expert for system A has more than 7 years of experience and expert for system B has 4 years of experience.

7.3 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, a number of open source practices (such as the use of ITSs, continuous integration and globally distributed development) appear to be applied in commercial systems as well [78], leading us to believe that development practices in open source and commercial systems are not that far apart.

Technology of development: All subject systems in our experiments are developed in Java. To generalize our results, in particular to other programming approaches such as procedural or functional programming, we need to carry out additional experiments. On the other hand, programming languages that share common characteristics (i.e., object oriented programming, unit testing frameworks) with Java follow similar development processes and therefore the results are believed to be valid for them.

The bias of systems that use ITSs: The systematic use of an 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. 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 replications are necessary to generalise our results for the whole scale of the test code quality rating.

Limited number of experts: In Section 4 we asked two experts to judge the test code quality of two industrial systems in order to determine how closely the model aligns with expert opinion. We realize that two expert opinions are limited to draw a strong conclusion from and we consider adding more case studies as future work.

7.4 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 to increase the strength of the results.

8 Related Work

8.1 Test code quality models

To our knowledge the efforts of assessing test code quality are mainly concentrating in the area of individual metrics and criteria such as those 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. Nagappan has similar intentions as he proposed a suite of test code related metrics that resulted in a series of studies [79].

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

To validate STREW as a method to assess test code quality and software quality a controlled experiment was performed [80]. Students developed an open source Eclipse[11] plug-in in Java that automated the collection of the STREW metrics. The student groups were composed of four or five junior or senior undergraduates. 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 of 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.

As mentioned previously, in our study we also aim at selecting a set of metrics to assess test code quality. The

11. http://www.eclipse.org/

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. This quality model is inspired on the SIG quality model and makes it easier to compare systems, which would otherwise have to be done by comparing raw metrics' values. In addition, we also identified a weak point of the STREW metrics suite, in particular, the lack of coverage-related metrics.

8.2 Software quality models

If we do not confine ourselves to test code quality models, but focus on more general software quality models, we see that a lot of work has been done in this area [81]. One of the earliest quality models was presented by McCall et al. in 1977 [82]. This model, sometimes also known as McCall's Triangle of Quality, has three major perspectives for defining and identifying the quality of a software product: product revision (ability to undergo changes), product transition (the adaptability to new environments) and product operations (its operation characteristics — correctness, efficiency, ...).

Another important step is represented by Boehm's quality model [83], which attempts to define software quality by a given set of attributes and metrics. Boehm's model is similar to McCall's model in that it also presents a hierarchical model structured around high-level characteristics, intermediate level characteristics, primitive characteristics, each of which contributes to the overall quality level.

Somewhat less known are the FURPS quality model, originally proposed by Grady [84] and later on extended by IBM Rational into the FURPS+ model and Dromey's quality model [85].

More recently, the ISO 9126 standard has been established [62], which is based on McCall's and Boehm's quality model. The standards group has recommended six characteristics to form a basic set of independent quality characteristics, namely: functionality, reliability, usability, efficiency, maintainability and portability. It is the maintainability characteristic that forms the basis for the SIG quality model that we use (also see Figure 1).

8.3 On the evolution of test code

Pinto et al. [86] investigated how unit test suite evolution occurs. Their main findings are that *test repair* is an often occurring phenomenon during evolution, indicating, e.g., that assertions are fixed. Their study also shows that test suite augmentation is also an important activity during evolution aimed at making the test suite more adequate. One of the most striking observations that they make is that failing tests are more often deleted than repaired. Among the deleted failing tests, tests fail predominantly (over 92%) with compilation errors, whereas the remaining ones fail with assertion or runtime errors. In a controlled experiment on refactoring with developer tests, Vonken and Zaidman also noted that participants often deleted failing assertions [51].

In similar style Zaidman et al. propose a set of visualization to determine how production code and (developer) test code co-evolve [3]. In particular, they observed that this co-evolution does not always happen in a synchronized way, i.e., sometimes there are periods of development, followed by periods of testing. Lubsen et al. have a similar goal, albeit they use association rule mining to determine co-evolution [87]. In response to observations of the lack of co-evolution, Hurdugaci and Zaidman [88] and Soetens et al. [89] proposed ways to stimulate developers to co-evolve their production and test code.

The aforementioned investigations can be placed in the research area of mining software repositories [90].

9 Conclusions and Future Work

Developer testing is an important part of software development. Automated tests enable early detection of defects in software, and facilitate the comprehension of the system. We constructed a model to assess the quality of test code. Based on the model, we explored the relation of test code quality and issue handling performance.

9.1 Summary of Findings and Conclusions

We now summarise the findings of the study which enable us to provide answers to the research questions.

9.1.1 *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. Subsequently, we identified the main aspects of test code quality and selected a set of metrics that would provide measurements that enable the assessment of these aspects. The three main aspects of test code quality that we identified are: completeness, effectiveness and maintainability. Completeness concerns the complete coverage of the production code by the tests. Effectiveness indicates the ability of the test code to detect defects and locate their causes. Maintainability reflects the ability of the test code to be adjusted to changes of the production code, and the extent to which test code can serve 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 to be reflective of test code maintainability. The metrics are aggregated using a benchmarking technique to provide quality ratings that inform the user of the quality of the system in comparison with the set of systems 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.

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

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.

To test the aforementioned hypotheses, we collected data from 75 snapshots belonging to 18 open source systems, 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.

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 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.

9.2 Contributions

The contributions of this study can be summarised as follows: we constructed a model that combines metrics to provide a measure of test code quality. We subsequently calibrated that model with 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. We performed an empirical investigation that demonstrated a significant positive correlation between test code quality and throughput and productivity of issue handling.

9.3 Future Work

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

The current test code quality model is solely based on source code measures. It might be interesting to extend the model with historical information that would bring additional insight as to the number of previous bugs (defects that were not caught by the test code).

In Section 2.1.2 we presented mutation testing as a test effectiveness evaluation method. An experiment with the mutation testing score as dependent variable would provide further validation of the test code quality model.

To assess the relation between test code quality and issue handling performance we used three issue handling indicators. However, other indicators reflect different aspects of issue handling, e.g., 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.

In our study, positive and statistically significant 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 related to investments in improving test code quality.

Related to the previous item of future work is to quantify the effect of test code quality on productivity and throughput and identify a *critical mass*, e.g., a minimal level of, for example, test coverage a test suite needs to fulfill in order to have a true impact on throughput and productivity.

It would also be interesting to study whether having high test code quality is more beneficial to pre-release defects or post-release defects.

Looking at Table 14 we see that several of the open source projects exhibit a greatly differing number of defects that are reported, e.g., when comparing Apache Ant to Apache Ivy we observe respectively 1,680 and 228 reported issues. In future work we want to investigate whether the number of developers, the number of installations, the number of major/minor release per year, and other process related differences could explain this difference.

Acknowledgments

This work was partly funded by (1) the NWO TestRoots project (project number 639.022.314) and (2) the RAAK-PRO project EQuA (Early Quality Assurance in Software Production) of the Foundation Innovation Alliance (the Netherlands).

References

[1] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *International Conference on Software Engineering (ICSE), Workshop on the Future of Software Engineering (FOSE)*. IEEE CS, 2007, pp. 85–103.
[2] M. Ellims, J. Bridges, and D. C. Ince, "The economics of unit testing," *Empir. Software Eng.*, vol. 11, no. 1, pp. 5–31, 2006.
[3] 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," *Empir. Software Eng.*, vol. 16, no. 3, pp. 325–364, 2011.
[4] X. Xiao, S. Thummalapenta, and T. Xie, "Advances on improving automation in developer testing," *Advances in Computers*, vol. 85, pp. 165–212, 2012.
[5] G. Meszaros, *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall PTR, 2006.
[6] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.
[7] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comp. Surveys*, vol. 29, no. 4, pp. 366–427, 1997.

[8] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Proc. Int'l Conf. on Quality of Information and Communications Technology*. IEEE, 2007, pp. 30–39.

[9] B. Luijten, J. Visser, and A. Zaidman, "Assessment of issue handling efficiency," in *Proc. of the Working Conference on Mining Software Repositories (MSR)*. IEEE CS, 2010, pp. 94–97.

[10] C. Weiss, R. Premraj, T. Zimmermann, and A. Zeller, "How long will it take to fix this bug?" in *Proc. of the International Workshop on Mining Software Repositories (MSR)*. IEEE CS, 2007, pp. 1–.

[11] E. Giger, M. Pinzger, and H. Gall, "Predicting the fix time of bugs," in *Proceedings of the International Workshop on Recommendation Systems for Software Engineering (RSSE)*. ACM, 2010, pp. 52–56.

[12] S. Kim and E. J. Whitehead, Jr., "How long did it take to fix bugs?" in *Proceedings of the international workshop on Mining software repositories (MSR)*. ACM, 2006, pp. 173–174.

[13] S. N. Ahsan, J. Ferzund, and F. Wotawa, "Program file bug fix effort estimation using machine learning methods for OSS," in *Proc. Int'l Conf. on Softw. Eng. & Knowledge Eng. (SEKE)*, 2009, pp. 129–134.

[14] D. Bijlsma, "Indicators of issue handling efficiency and their relation to software maintainability," MSc Thesis, Univ. of Amsterdam, 2010.

[15] D. Bijlsma, M. Ferreira, B. Luijten, and J. Visser, "Faster issue resolution with higher technical quality of software," *Software Quality Journal*, vol. 20, no. 2, pp. 265–285, 2012.

[16] E. Weyuker, "Axiomatizing software test data adequacy," *IEEE Trans. Softw. Eng.*, vol. 12, no. 12, pp. 1128–1138, 1986.

[17] W. C. Hetzel and B. Hetzel, *The complete guide to software testing*. Wiley, 1991.

[18] J. S. Gourlay, "A mathematical framework for the investigation of testing," *IEEE Trans. Softw. Eng.*, vol. 9, no. 6, pp. 686–709, 1983.

[19] W. E. Howden, "Methodology for the generation of program test data," *IEEE Trans. on Computers*, vol. 24, no. 5, pp. 554–560, 1975.

[20] W. G. Bently and E. F. Miller, "CT coverageinitial results," *Software Quality Journal*, vol. 2, no. 1, pp. 29–47, 1993.

[21] G. J. Myer, *The art of software testing*. Wiley, 1979.

[22] M. R. Woodward, D. Hedley, and M. A. Hennell, "Experience with path analysis and testing of programs," *IEEE Trans. Softw. Eng.*, vol. 6, no. 3, pp. 278–286, May 1980.

[23] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria," *IEEE Trans. Softw. Eng.*, vol. 14, no. 10, pp. 1483–1498, October 1988.

[24] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Softw. Eng.*, vol. 11, no. 4, pp. 367–375, 1985.

[25] S. C. Ntafos, "A comparison of some structural testing strategies," *IEEE Trans. Softw. Eng.*, vol. 14, no. 6, pp. 868–874, 1988.

[26] 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.*, vol. 15, no. 11, pp. 1318–1332, 1989.

[27] J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Trans. Softw. Eng.*, vol. 9, no. 3, pp. 347–354, 1983.

[28] 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.*, vol. 16, no. 9, pp. 965–979, 1990.

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

[30] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[31] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Softw. Eng.*, vol. 3, no. 4, pp. 279–290, 1977.

[32] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Softw. Eng.*, vol. 6, no. 3, pp. 247–257, 1980.

[33] L. A. Clarke, J. Hassell, and D. J. Richardson, "A close look at domain testing," *IEEE Trans. Softw. Eng.*, vol. 8, no. 4, pp. 380–390, July 1982.

[34] F. H. Afifi, L. J. White, and S. J. Zeil, "Testing for linear errors in nonlinear computer programs," in *Proc. of the Int'l Conference on Software Engineering (ICSE)*. ACM, 1992, pp. 81–91.

[35] W. Howden, "Theory and practice of functional testing." *IEEE Software*, vol. 2, no. 5, pp. 6–17, 1985.

[36] G. Kudrjavets, N. Nagappan, and T. Ball, "Assessing the relationship between software assertions and faults: An empirical investigation," in *17th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE CS, 2006, pp. 204–212.

[37] J. Voas, "How assertions can increase test effectiveness," *IEEE Software*, vol. 14, no. 2, pp. 118–119,122, 1997.

[38] A. van Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proc. of the Int'l Conf. on Extreme Programming and Flexible Processes in Software Engineering (XP)*, 2001, pp. 92–95.

[39] 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," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 800–817, 2007.

[40] S. Reichhart, T. Gîrba, and S. Ducasse, "Rule-based assessment of test quality," *Journal of Object Technology*, vol. 6, no. 9, pp. 231–251, 2007.

[41] M. Greiler, A. Zaidman, A. van Deursen, and M.-A. D. Storey, "Strategies for avoiding test fixture smells during software evolution," in *Proceedings of the Working Conference on Mining Software Repositories (MSR)*. IEEE / ACM, 2013, pp. 387–396.

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

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

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

[45] A. Nugroho, "The Effects of UML Modeling on the Quality of Software," Ph.D. dissertation, University of Leiden, 2010.

[46] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, 2005.

[47] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: bias in bug-fix datasets," *Proc. of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE)*, pp. 121–130, 2009.

[48] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, ""not my bug!" and other reasons for software bug report reassignments," in *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW)*. ACM, 2011, pp. 395–404.

[49] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. ACM, 2007, pp. 34–43. [Online]. Available: http://doi.acm.org/10.1145/1321631.1321639

[50] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the SIGSOFT International Symposium on Foundations of software engineering (FSE)*. ACM, 2008, pp. 308–318.

[51] F. Vonken and A. Zaidman, "Refactoring with unit testing: A match made in heaven?" in *Proc. of the Working Conf. on Reverse Engineering (WCRE)*. IEEE Computer Society, 2012, pp. 29–38.

[52] A. Mockus, R. T. Fielding, and J. Herbsleb, "A case study of open source software development: the apache server," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. ACM, 2000, pp. 263–272.

[53] R. Baggen, J. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, pp. 287–307, 2012.

[54] 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.

[55] T. L. Alves, C. Ypma, and J. Visser, "Deriving metric thresholds from benchmark data," in *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE CS, 2010, pp. 1–10.

[56] V. R. Basili, "Software modeling and measurement: the Goal/Question/Metric paradigm," Techreport UMIACS TR-92-96, University of Maryland at College Park, College Park, MD, USA, Tech. Rep., 1992.

[57] J. An and J. Zhu, "Software reliability modeling with integrated test coverage," in *Proc. of the Int'l Conf. on Secure Software Integration and Reliability Improvement (SSIRI)*. IEEE CS, 2010, pp. 106–112.

[58] T. L. Alves and J. Visser, "Static estimation of test coverage," in *Proc. of the Int'l Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE CS, 2009, pp. 55–64.

[59] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Softw. Eng. Methodol.*, vol. 6, pp. 173–210, 1997.

[60] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen, "Mining software repositories to study co-evolution of production & test code," in *Proc. of the International Conference on Software*

Testing, Verification, and Validation (ICST). IEEE Computer Society, 2008, pp. 220–229.

[61] T. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[62] I. 9126-1:2001, "Software engineering - product quality - part 1: Quality model," ISO, Geneva, Switzerland, pp. 1–32, 2001.

[63] 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*, T. Mens and S. Demeyer, Eds. Springer, 2008, pp. 173–202.

[64] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[65] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

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

[67] T. L. Alves, J. P. Correia, and J. Visser, "Benchmark-based aggregation of metrics to ratings," in *Proc. of the Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM/MENSURA)*. IEEE CS, 2011, pp. 20–29.

[68] D. Athanasiou, "Constructing a test code quality model and empirically assessing its relation to issue handling performance," Master's thesis, Delft University of Technology, 2011. [Online]. Available: http://resolver.tudelft.nl/uuid:cff6cd3b-a587-42f2-a3ce-e735aebf87ce

[69] G. Concas, M. Marchesi, S. Pinna, and N. Serra, "Power-laws in a large object-oriented software system," *IEEE Trans. Softw. Eng.*, vol. 33, no. 10, pp. 687–708, 2007.

[70] R. Yin, *Case study research: Design and methods*. Sage Publications, Inc, 2009, vol. 5.

[71] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta, "Threats on building models from cvs and bugzilla repositories: The mozilla case study," in *Proc. Conf. of the Center for Advanced Studies on Collaborative Research (CASCON)*. IBM, 2007, pp. 215–228.

[72] J. C. Munson and S. G. Elbaum, "Code churn: A measure for estimating the impact of code change," in *Proc. Int'l Conf. on Software Maintenance (ICSM)*. IEEE CS, 1998, pp. 24–31.

[73] A. Field, *Discovering Statistics Using SPSS, 2nd ed.* London: SAGE, 2005.

[74] J. Cohen, *Statistical power analysis for the behavioral sciencies*. Routledge, 1988.

[75] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: a roadmap," in *International Conference on Software Engineering (ICSE), Workshop on the Future of Software Engineering (FOSE)*. ACM, 2000, pp. 345–355.

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

[77] G. Yorsh, T. Ball, and M. Sagiv, "Testing, abstraction, theorem proving: better together!" in *Proceedings of the international symposium on Software testing and analysis (ISSTA)*. ACM, 2006, pp. 145–156.

[78] J. E. Robbins, "Adopting open source software engineering (osse) practices by adopting osse tools," in *Making Sense of the Bazaar: Perspectives on Open Source and Free Software*, J. Feller, B. Fitzgerald, S. Hissam, and K. Lakham, Eds. MIT Press, 2003.

[79] N. Nagappan, "A software testing and reliability early warning (strew) metric suite," Ph.D. dissertation, North Carolina State University, 2005.

[80] 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*, vol. 30, no. 4, pp. 1–7, 2005.

[81] B. Kitchenham and S. L. Pfleeger, "Software quality: The elusive target," *IEEE Softw.*, vol. 13, no. 1, pp. 12–21, 1996.

[82] J. A. McCall, P. K. Richards, and G. F. Walters, "Factors in software quality," in *Nat'l Tech.Information Service, no. Vol. 1, 2 and 3*, 1977.

[83] B. W. Boehm, J. R. Brown, and M. Lipow, "Quantitative evaluation of software quality," in *Proc. Int'l Conf. on Software Engineering (ICSE)*. IEEE Computer Society, 1976, pp. 592–605.

[84] R. B. Grady, *Practical software metrics for project management and process improvement*. Prentice Hall, 1992.

[85] R. G. Dromey, "A model for software product quality," *IEEE Trans. Softw. Eng.*, vol. 21, no. 2, pp. 146–162, 1995.

[86] L. S. Pinto, S. Sinha, and A. Orso, "Understanding myths and realities of test-suite evolution," in *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012, pp. 33:1–33:11.

[87] Z. Lubsen, A. Zaidman, and M. Pinzger, "Using association rules to study the co-evolution of production & test code," in *Proc. of the International Working Conference on Mining Software Repositories (MSR)*. IEEE CS, 2009, pp. 151–154.

[88] V. Hurdugaci and A. Zaidman, "Aiding software developers to maintain developer tests," in *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 2012, pp. 11–20.

[89] Q. D. Soetens, S. Demeyer, and A. Zaidman, "Change-based test selection in the presence of developer tests," in *Proc. of the European Conference on Software Maintenance and Reengineering (CSMR)*, 2013, pp. 101–110.

[90] H. H. Kagdi, M. L. Collard, and J. I. Maletic, "A survey and taxonomy of approaches for mining software repositories in the context of software evolution," *Journal of Software Maintenance*, vol. 19, no. 2, pp. 77–131, 2007.