

Assuring the Quality of Unit Testing in a Continuous Delivery Environment

Version of September 27, 2016

Maarten Duijn

Assuring the Quality of Unit Testing in a Continuous Delivery Environment

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Maarten Duijn
born in Alkmaar, the Netherlands



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



ING
Bijlmerplein 880, 1102 MG
Amsterdam, the Netherlands
www.ing.nl

Assuring the Quality of Unit Testing in a Continuous Delivery Environment

Author: Maarten Duijn
Student id: 1517279
Email: maartenduijn@msn.com

Abstract

Automated tests have always been essential for changing a piece of software. They let developers detect and locate faults early on and provide confidence in the product's quality. With the rise of Continuous Delivery (CD) in software development, changes are being deployed multiple times a day. Maintaining a high quality test suite has therefore never been more important. Based on the CD practices at ING and its delivery pipeline we have created Spectata, a tool that helps development teams assure the quality of their unit test code. Every time a change in the code is made Spectata calculates metrics that reflect the adequacy, fault finding effectiveness and maintainability of the test suite. It provides a quality verdict on those aspects, recommended refactorings and a detailed comparison with the quality of prior builds. To evaluate Spectata we perform four case studies in which we compare Spectata's verdict and recommendations to ones given by the developers and ourselves. The results demonstrate that the testing quality verdict aligns with our own opinion and that the tool is able to help development teams assess, maintain and improve the quality of their test suites.

Thesis Committee:

Chair:	Dr. A.E. Zaidman, Faculty EEMCS, TU Delft
University supervisor:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Company supervisor:	Dr. D. Romano, ING
Committee Member:	Dr. J. Pouwelse, Faculty EEMCS, TU Delft

Contents

Contents	iii
List of Figures	v
1 Introduction	1
1.1 Achieving Continuous Deployment	1
1.2 Developer Testing	3
1.3 Context	4
1.4 Problem Statement	4
1.5 Approach	5
1.6 Outline	6
2 Background	7
2.1 ING	7
2.2 Testing Quality Research	9
2.3 Overview of Related Tools	14
2.4 The SIG Quality Model	21
3 Proposed Approach to Assuring Unit Testing Quality	25
3.1 Requirements	25
3.2 Quality Characteristics	27
3.3 Descriptive Statistics	32
3.4 A Unit Test Quality Score	35
3.5 From Quality Score to Advice	39
3.6 Assuring Testing Quality	41
4 Implementation: The Spectata Tool	43
4.1 Implementation Details	43
4.2 Integration With a CI Pipeline	45
4.3 The Spectata External API	45

CONTENTS

5	Evaluation	49
5.1	Study Design	49
5.2	Experiment Result	55
6	Related Work	67
6.1	STREW and GERT	67
6.2	SIG	68
6.3	TAIME	68
7	Discussion and Threats to Validity	71
7.1	Interpretation of the Results	71
7.2	Application Limitations	72
7.3	Future work	72
7.4	Threats to Validity	73
8	Conclusion	77
	Bibliography	79
A	Threshold Values Metrics	87
B	Overview of Tables in Spectata Storage	89
C	Devhub Local Adequacy Recommendations	91
D	Metrics for Evaluated Joda-Time Classes	93

List of Figures

2.1	Overview of the ING Continuous Delivery pipeline	9
2.2	Code fragment with one of its possible mutants [6]	12
2.3	Part of a Surefire report	15
2.4	Part of a response for a request that asks what tests cover a line in a production file	16
2.5	Part of a report generated by PMD in csv format	17
2.6	The SonarQube dashboard for Apache Commons Lang	18
2.7	An overview of the SonarQube architecture [10]	18
2.8	Fragment of a pitest report in XML format	20
2.9	Bugout interface showing acceptance test results for an application	21
3.1	ROC graphs for the classification algorithms	37
3.2	Random forest feature importance for predicting the maintainability quality category, normalized to sum to one	38
4.1	Overview of Spectata's architecture	44
4.2	Mockup of BugOut with testing quality aspect	46
5.1	A test case in the Joda Time unit test suite	53
5.2	Per quality aspect the relative number of files with a particular score, shown for each of the evaluated projects.	58
B.1	The Spectata storage model	89

Chapter 1

Introduction

In this chapter we will briefly cover some of the background information that led to this study. We will formulate a problem statement including functional requirements, explain why it is a relevant problem to solve and describe how our solution will be evaluated.

1.1 Achieving Continuous Deployment

Over the last few years agile practices have become more common throughout the software engineering process. As a result we have seen a paradigm shift in how software is developed. In this transition the responsibilities of a development team have grown to include quality assurance, infrastructure and support and is now referred to as a DevOps team [47]. Ries [77] and Olsson et al. [71] explain that companies will continue to evolve towards a “lean organization” by having development teams conduct experiments with a software product as part of the development process. For the purpose of this study it is important to explain three essential steps of moving towards this new type of organization: Continuous Integration, Continuous Delivery and Continuous Deployment.

1.1.1 Continuous Integration

The essential goal of Continuous Integration (CI) is to reduce the number of integration problems and thus easing the process of integrating new code into an existing software system. Fowler [37] and Duvall et al. [34] define the practice as follows: *Continuous Integration is a software development practice where members of a team integrate their work frequently, leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible.*

Following the CI practice, a task starts with a developer checking out a working version of the project from the source repository. The developer then does whatever he or she needs to do to complete the task. Usually this means changing both production as well as test code. The new version of the code is compiled and tested locally first. Only if this succeeds, the developer is allowed to push the new version of the software back in to the source repository.

The second part of the integration is done centrally using a CI server that monitors the source repository for changes. When a developer pushes new code to the repository a build

1. INTRODUCTION

is automatically triggered that compiles, tests, and analyses the source code. If the build fails, for example because of a failing test, the developers are notified and will fix the problem by pushing new code to the repository, which in turn, will trigger a new build.

Because each change to the software is built, developers can be reasonably confident that new code does not harm the quality of the system. If it does, the error can be quickly localized and fixed instead of introducing more unexpected work later on in a project. This does however require that each piece of software is adequately tested by a suite of automated tests.

By developing software in small parts that each trigger a small integration development teams avoid a long and unpredictable process that takes place just before the software is released. Instead, a team is at all times aware of the progress, as well as the quality of the product, which removes one of the toughest barriers for the next step towards becoming a lean organization: Continuous Delivery.

1.1.2 Continuous Delivery

Continuous Delivery extends the practices of Continuous Integration to include building a release candidate and pushing it into production-like environments for further testing. Teams successfully applying Continuous Delivery can deploy the software in development to production with the push of a button, however they may choose to not deploy all release candidates.

An important concept introduced in this practice is the deployment pipeline: a series of steps through which each change needs to go before being released into production. The exact steps in the pipeline vary per company, however they usually include: build and unit tests, automated acceptance tests, manual acceptance tests, and release. Each step generates feedback, usually in the form of a report, and a change is only allowed to continue to the next step, once it has successfully passed the current step. At the end of the pipeline a change is marked as ready to release and someone on the development team may manually deploy the release candidate to production.

1.1.3 Continuous Deployment

Where Continuous Integration focuses on easing the integration hurdle, Continuous Deployment eases the release hurdle. Continuous Deployment is the practice of automatically deploying each change in the release pipeline to production, it extends the deployment pipeline with automated releases.

For automated releases to be possible a development team needs to be certain of the quality of the software in the source repository. This means that not only acceptance testing needs to be automated, but also that non functional requirements, such as performance and compliance, need to be able to run without human interaction.

Because this step requires such a high level of automation, teams can only reach this stage by instrumenting their pipeline with a set of tools that gather metrics relevant to product quality. These metrics might vary from testing coverage to the amount of time it takes for a transaction to complete.

1.2 Developer Testing

As illustrated by the pipeline in the previous paragraph, testing a newly created change happens throughout the release process. The first test execution is often done by the developer locally to get some quick feedback about the correctness of the software. Tests executed at this point are simple to execute tests that test a small part of the software, but have been shown to lower development cost significantly [88]. These tests are created and maintained by developers themselves and therefore often referred to as “developer tests” [61].

The upfront price of adding developer testing to the development process can be significant, still, ever since Beck and Gamma [41] came up with the JUnit testing framework, developer testing has risen in popularity. One of the main reasons for its popularity is because it helps developers find problems early on in the development process. When a change to the software containing a fault is made and detected by a unit test, a developer will immediately know in what part of the system the error is in and the cost of fixing the bug is considerably lower than the cost of detecting, identifying, and correcting the fault later on. The above example also illustrates how developer testing facilitates change. A well written test suite provides some confidence that a change does not have any unexpected side effects, if it would one of the tests would fail.

In addition to being able to find problems early and facilitating change, Test Driven Development (TDD) [20] uses tests as a specification as well. Following TDD, a developer will first start by writing tests before implementing a feature and because of this specifying the behavior and low level architecture. Only after the tests have been created the developer starts working on the production code, making more test cases succeed with each piece of code written. The last major use of developer testing is providing documentation, tests provide up-to-date instructions on how a part of production code should be used.

Depending on what the test intends to verify, developer tests can be said to belong to one of two categories: unit and integration tests. The Standard glossary of terms used in Software Testing [91] states that unit testing is the testing of individual software components. Integration testing is testing performed to expose defects in the interfaces and in the interactions between integrated components or systems. However what a component is differs per development team and in practice some make no distinction at all between the two types of tests. In this work we define a unit as the smallest testable part of an application, i.e., a method or class in Java. Unit tests will test these in complete isolation, independent from other units.

Both types of developer testing follow the same pattern, i.e., first the expected outcome is expressed by defining an expected state or an expected behavior. The second step is having each test execute the components it is testing in the system under test (SUT). The actual outcome of executing the components is then compared with the expected outcome defined in the first step.

Executing part of the system in order to retrieve the actual outcome is not always possible, e.g., when this part of the system relies on a database connection. In such a case, Stubs and Mock objects offer a solution, they allow a test to control the indirect inputs of the SUT. In the example of a database connection the test case could set up a stubbed connection with a pre-defined behavior and does not have to rely any more on an external database. Fowler [38]

states that the difference between Mock objects and Stubs lies mostly in how verification is done. Meszaros [61] defines them as follows: Stubs provide pre-defined answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Mocks are objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

1.3 Context

This research effort is done in cooperation with ING, specifically the group responsible for its CI Pipeline. As we will explain in Chapter 2, the company is moving towards becoming a lean organization and this group has become a vital part of the transition. The group is in the unique position to be able to help development teams on a company wide scale with verifying software quality, improving development practices and compliancy. Helping teams face these issues forms the motivation for initiating this study.

Together with the CI pipeline group we develop a solution that can be easily added to their product as well as more generic pipelines. The projects that are currently enrolled in the pipeline form both a source of inspiration as well as a source of validation for this work. In order to develop a useful solution we take into account these projects, their development teams and the way in which they are developed. Finally, we use one of the ING projects that deploys its product with the pipeline as part of the evaluation.

1.4 Problem Statement

Ever since we started developing software, quality assurance has been an essential part of the development process. In 1979 a rule of thumb in the industry stated that over 50% of the time spent on software development was spent on testing [65]. Since that day, we have developed more advanced tools, faster computers, and better ways of developing software and still, most development teams test before applying changes.

Adding to the importance of testing and automated testing in particular is the trend to develop software in an increasingly agile way. Because of the new way of work, changes in software have become common and releases are happening almost continuously. Where changes in software used to go through quality assurance before being handed over to operations, the process of rolling out a change is now almost fully automated. The responsibilities of development, quality assurance, and operations now lie with the development team instead of three different organizations. As a consequence this team needs to be confident that any errors in a change are caught before the automated release is triggered.

In “Test-Driven Development” Beck [19] states that testing, and unit testing especially, is essential to delivering “clean code that works”. He goes on to explain that testing can be used to increase confidence in applying changes to the code without causing parts of the system to break. With regard to release automation, Fowler [37] defined “Self-Testing code” as software that can verify itself with a single command, up to the point that the development team is confident that the verification will illuminate any bugs hiding in the code. He states

that self-testing code is a key part of CI, going as far as to say that “you aren’t really doing Continuous Integration unless you have self-testing code”.

However this raises a question we have faced since we started developing software: at what point is a product adequately tested? We approach this question from an incremental view point: at the very least a piece of software should be tested better after applying a change than before applying one. *The goal of this thesis* is to be able to provide both developers as well as managers insight into the quality of unit testing of a software product in an agile environment. To realize this, a solution is needed with the following functional requirements.

- **R1: Score** – A solution must be able to provide a score that reflects the quality of unit testing within a project at each code check in. The score must allow for the comparison of projects on the quality of testing, independent of their size.
- **R2: Quality over time** – A solution must be able to provide the change in score between two code check ins.
- **R3: Traceability** – A quality score or change in score between two code check ins must be traceable to the level of a specific act. When a solution finds a new threat to unit testing quality the development team must be able to see why the score reflecting unit testing quality has dropped and vice versa.

1.5 Approach

We start working towards a unit test quality evaluation tool by exploring test quality criteria defined in existing literature. There are several research efforts and tools that solve part of this problem. We describe the most relevant, what criteria they intend to cover and their drawbacks. Based on the found criteria we define a set of project-independent aspects indicative of unit testing quality over a single file containing test code. Each of these aspects is measured with a set of metrics that provide an indication of how far a specific file is from reaching that goal. The individual metrics are combined to form scores for goals, which in turn are combined to form a score for overall unit testing quality.

One of the key ideas behind agile software engineering is that software is never perfect, rather it is continuously improving. The tool we build for evaluating testing quality reflects this way of thinking. Instead of focusing on a static view of unit testing quality, it shows the differences in quality between two builds of a project. The goal of this is twofold: first of all, a developer can see which new issues have arisen due to a change that was made and improve on them if necessary. The second goal is to be able to show a manager that this new change is well tested, providing confidence that the change can be progressed to the next stage in the deployment pipeline.

Based on the pipeline at ING we implement the backend, e.g. the calculation and storage, for this tool and name it Spectata. Whenever a new build is triggered, Spectata monitors the executions of automated unit tests. In addition unit tests are statically analyzed using a number of existing tools for calculating code quality. The data is stored and analyzed so that the unit testing quality of each successful build can be compared to the unit testing quality of prior ones.

To evaluate Spectata we perform case studies with two large open source projects, one student project and one ING project. In these projects we select a number of interesting points and changes and have Spectata generate reports on them. These reports contain quality scores and advice on how the unit test suite can be improved. We then compare these with our own opinion of the test suite quality, and the advice we would give, to find out how well they align. For the student and ING projects the opinion of the developers is taken into account as well when assessing the test suite.

1.6 Outline

The thesis is structured as follows. Chapter 2 describes software development at ING, existing research into software testing and testing code quality models, as well as an overview of existing tools. We then proceed to describe the proposed approach for assuring testing quality at ING in chapter 3. Chapter 4 discusses the implementation of the approach in the form of the Spectata tool. The tool is evaluated in chapter 5 by comparing the reports it generates to manual assessment. Threats to validity and significance of the tool are discussed in chapter 7. Finally, chapter 8 summarizes the findings and contributions of this work.

Chapter 2

Background

2.1 ING

This thesis project was conducted in the context of ING Bank, the Netherlands. As a result parts of this work are influenced by how software is developed at ING. This section will describe the most relevant pieces of background information that played a role.

2.1.1 Financial Institute and Tech Company

In 1991 insurance company Nationale-Nederlanden and banking company NMB Postbank Groep decided to merge into one company, together forming ING Bank. Starting out as a Dutch company with some international business, the company has since then grown into a finance multinational with Dutch roots, serving customers in over 40 countries.

ING as a technology company on the other hand traces back to the 60s when its parent companies started using computers for record keeping. As we well know, the coming of these computers was only the start of a complete transformation in the way banks operate. The changes caused by them have made software development and maintenance vital to ING. All of its banking services are completely dependent on IT and increasingly rely on it for creating a differentiating user experience.

Customers expect nothing less than a perfectly working interface with a bank that lets them do all of their banking activities how and when they want to. Staying ahead of the competition means doing everything to make these activities fit seamlessly in daily life. A recent example of this is being able to make contactless payments with a phone. Customers being able to pay without requiring their debit or credit card could be a significant competitive advantage.

To achieve such an experience, banks are forced to almost recklessly incorporate new pieces of technology in their IT systems. Still, both users and regulatory bodies expect them to be safe, secure and available. No matter the changes in an IT system, under no circumstance can a bank allow fraud to be committed, privacy sensitive information to be leaked or laws to be broken. As a result every application used by ING or its customers needs to make a trade off between innovation and its Confidentiality, Integrity and Availability

2. BACKGROUND

(aspects of the CIA triad¹).

To make balancing the trade off easier it is standardized. Depending on the data it handles, as well as the business impact, each application is assigned a certain rating on the aspects of the CIA triad. The rating determines the requirements on the application from a security point of view. And increasingly requirements on product quality are taken into consideration.

2.1.2 Move to Agile

Over the last 50 years ING has developed and maintained thousands of software applications in The Netherlands alone. Like nearly all companies developing software ING started out by using the Waterfall method [79]. As software was given an increasingly vital role in the organization more and more structure started being applied to the development process. Over the years developers at ING started using issue trackers, a versioning system and a change management process.

Software development at ING as it is now dates back only a few years with the introduction of Scrum [84]. At the time ING realized it needed to speed up development, anticipate customer needs, and attract young talent. With over a thousand developers working on hundreds of applications this was not easy to accomplish. The conclusion drawn by the bank's management was to start developing software in an agile manner.

The company started to transform its development culture into that of a startup. Inspired by the Spotify model [57], departments were divided into tribes and squads, with leaders instead of managers. Following Humble's Continuous Deployment [48], deployments got smaller and smaller. Applications became small pieces of software that were built and deployed one feature at a time using a pipeline (see also section 2.1.3). And the walls between development, testing and operations disappeared. Creating teams responsible for a product during all phases of its life cycle.

So far the move towards agile has been relatively successful, but the transformation is not over yet. ING wants to continue to change along the lines described in section 1.1 and become a lean and learning organization. One of the requirements for this transition is to be in control of product quality. This project was originally started in order to fulfill part of this requirement by helping teams assure the quality of unit testing.

2.1.3 The ING Continuous Delivery Pipeline

As shown by fig. 2.1 the ING pipeline consists of three major steps: integration, deployment and automated testing. In the integration step code moves from a developer to a centralized artifact repository. In the deployment step the created artifact is deployed to the various testing environments. Finally, after the product is successfully tested the development team has the option to manually deploy the product to production.

Because this project focuses its effort on unit tests, the integration step is the most relevant. The Continuous Integration (CI) server in this step runs the unit tests and displays

¹The CIA Triad is a security model developed to help people think about important aspects of IT security, the letters in the acronym stand for: Confidentiality, Integrity, and Availability [72]

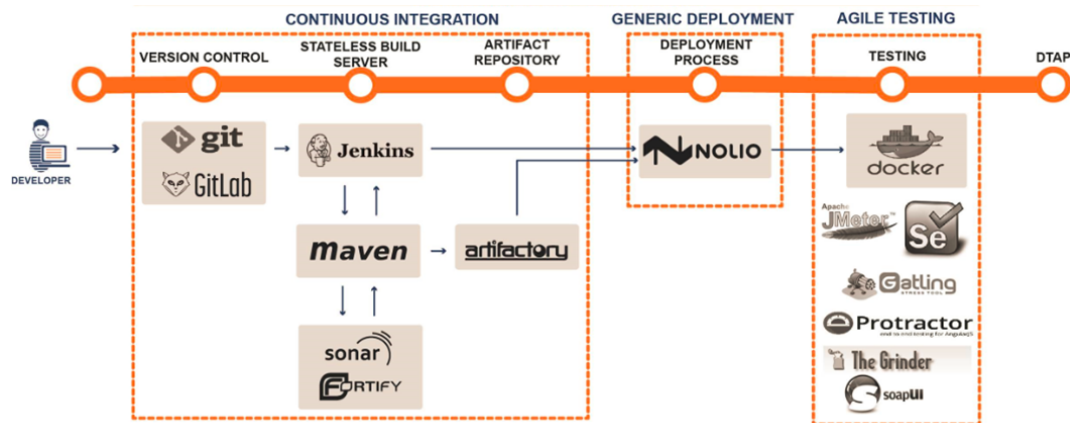


Figure 2.1: Overview of the ING Continuous Delivery pipeline

their outcome to the development team. In fig. 2.1 the CI server is referred to as the stateless build server and is implemented as a Jenkins instance. Every time a developer pushes a commit, it will pull the new code from the source repository and trigger several jobs. It triggers Maven², a tool for managing a project’s build, reporting and documentation, to build the project. Maven in turn triggers SonarQube³ to perform a code quality check. Optionally, the CI server can be configured to trigger additional jobs such as a dependency scan.

2.2 Testing Quality Research

The quality of a software product is one of the key concerns of a development team. Assessing and assuring product quality is typically done through effective unit testing. Tests are used to detect that the Software Under Test has faults and give an estimate of its quality. However testing software raises questions. First, how much testing is needed? Second, how effective is the test suite at finding faults? Third, what does a good test look like? Answers to these questions have been sought by researchers since developers started testing software. In this section we will describe known approaches for measuring adequacy and describe best and worst practices for testing, relevant to this project.

2.2.1 Unit Test Adequacy

The adequacy aspect of testing attempts to answer the question: “how much testing is needed?”. In Software Unit Test Coverage and Adequacy Zhu et al. [102] defined an adequacy criterion to be: “a stopping rule that determines whether sufficient testing has been done”. They continue to explain that adequacy criteria for unit tests can be classified according to the underlying testing approach. A criteria can belong to one of three categories: structural, fault-based, and error-based. In a master thesis project Athanasiou [16] summarizes this

²<https://maven.apache.org/>

³<http://www.sonarqube.org/>

2. BACKGROUND

work and updates it with criteria that were formulated after the paper was published in 1997. We continue to use the classification scheme, the updated criteria and discuss additional related work in the following paragraphs.

Structural Testing Adequacy Criteria

Structural adequacy criteria judge whether software is adequately tested by measuring to what extent the test suite exercises the structural elements of the program. Most adequacy criteria in this group are based on a flow-graph model of program. A flow-graph shows a piece of software as a graph in which the nodes represent a computation and the edges a transfer of control from one computation to another. Any path that flows from the start to end node corresponds to a possible execution of the software.

The most basic structural adequacy criterion is known as statement coverage and requires every statement in the program to be executed [46]. Because statements correspond to nodes in flow-graph models, this criterion can be defined in terms of flow graphs, as follows. “A set of paths P satisfies the statement coverage criterion if and only if for all nodes n in the flow graph, there is at least one path p in P such that node n is on the path p ” [102].

Note that statement coverage only has requirements for computations, it has no requirements regarding the transfer of control between them. For this we have the slightly stronger criterion: branch coverage [46]. Branch coverage requires that the test exercises all control transfers. A formal definition is as follows. “A set P of execution paths satisfies the criterion if and only if for all edges e in the flow graph, there is at least one path p in P such that p contains the edge e ” [102]. Decision coverage [65] is a commonly used variant of branch coverage. It requires that for every condition there is at least one test case that falsifies it, and one that satisfies it.

However, even if we satisfy decision coverage, we cannot be sure that all combinations of control transfers are checked. This requirement is usually called path coverage, which can be formally defined as follows. “A set P of execution paths satisfies the path coverage criterion if and only if P contains all execution paths from the begin node to the end node in the flow graph” [102].

The above definitions provide exact criteria for different ways of calculating coverage. None of them however can be applied finitely in practice. A program may contain dead statements or dead branches so that full coverage is simply not possible. The path coverage criterion in particular is not feasible because it might require an infinite number of tests. Perhaps most important is that the cost of fully satisfying the coverage criteria might not be in proportion to the gain in quality [73]. Additionally, a recent survey [27] shows that unless tests verify realistic scenarios, developers may not be convinced to fix their code based on the failing tests.

The criteria can be restated so that they become attainable by only requiring the coverage of feasible code, or limiting the number of paths tests are required to cover. As noted by [16] this leads to a range of derived coverage criteria such as Length- n Path Coverage [42] which only exercises paths with less than n edges. The Cyclomatic-Number [94] criterion which requires that the test suite covers all of these independent execution paths. And more

coverage criteria that are very rarely used in practice. In fact these criteria are not even taken into consideration when comparing coverage calculation tools [85, 98].

Research done by Runeson [80] and Smith and Williams [86] show that in practice most developers do not use the information to determine whether they have tested adequately. More often, coverage criteria are only used to show which parts of the code has been exercised by tests. At ING we observe the same pattern. There is no company wide standard for code coverage, and only some of the mission critical applications have a minimal required code coverage. While this shows that there is no simple answer to the question: “How high should my test coverage be?”. Coverage data can be a factor in assuring unit test quality. Numerous research efforts [50, 25, 63, 95] show that coverage is at least a moderate indicator for fault detection capability. They are able to correlate a higher structural coverage to a lower number of defects.

Fault-based Testing Adequacy Criteria

Fault-based adequacy criteria judge whether software is adequately tested by measuring to what extent the test suite is able to detect defects. All of the criteria in this category start by planting artificial errors in the software, generally referred to as error seeding [62]. The artificial errors are often introduced by making a tiny change in the code, e.g., changing a plus operator into a multiplication operator.

Mutation Testing is a technique first introduced in 1971 by Lipton [58] and formalized by DeMillo et al. [31]. Instead of relying on manual labor to insert errors into production code, it takes a more systematic approach. By pattern matching pieces of production code and altering their behavior slightly it creates mutants: new and faulty versions of the software. An example of this process is shown in fig. 2.2.

After the mutants are created, each is executed against the existing test suite with one of the following outcomes.

1. Killed: One of the tests in the suite failed
2. Survived by inadequate testing: tests did not detect the change
3. Survived by equivalence: a mutation yields exactly the same results as the original

The mutation adequacy score is defined as the number of killed mutations divided by the total number of mutations minus the equivalent mutations [102]. However, deciding whether a mutation is equivalent or not is an undecidable problem. In practice the mutation adequacy score is simply the number of killed mutations divided by the total number of mutations.

Mutation analysis is based on two assumptions: 1) the competent programmer hypothesis [32], and 2) the coupling effect [31]. The competent programmer hypothesis states that most software faults caused by experienced software developers are small errors. The coupling effect assumes that complex faults are a result of a series of smaller, simpler faults. Following the above two hypotheses, by testing to what extent the test suite is able to detect small errors we get an estimate of the test suite’s ability to detect all types of errors.

Considerable work has been put into validating the above two hypotheses and providing empirical support for the use of mutation testing [52]. Research by Offutt [69] and Morell

2. BACKGROUND



Figure 2.2: Code fragment with one of its possible mutants [6]

[64] finds that the mutation coupling effect hypothesis does indeed manifest itself in practice. Richard A. DeMillo [76] confirms this by finding that the simple created by mutations do indeed form a significant percentage of the faults found in production software. Continuing along this line, Daran and Thévenod-Fosse [28] found that 85% of the errors caused by mutants were also produced by real faults. And more recently, work by Just et al. [54] finds a statistically significant correlation between mutant detection and real fault detection.

Error-based Testing Adequacy Criteria

Last of the adequacy categories defined by Zhu et al. [102] is the error-based adequacy category. Error-based adequacy criteria judge whether software is adequately tested by measuring to what extent the error prone points of a program are tested [36, 65]. The error prone points can be derived from a specification of what the program is required to do. In the case of unit tests, nearly all these points can be directly derived from the program structure itself.

Evaluating error-based adequacy starts by partitioning the behavior of the software into subdomains. Ideally, each subdomain represents exactly one possible execution path of the software, as defined earlier in this section. If the software behaves correctly for one test case within a subdomain, then we can reasonably assume that the behavior of the software is correct for all data within that subdomain. How to test whether a subdomain behaves correctly depends on the chosen adequacy criteria.

An adequacy criterion that requires a single test case per subdomain and does not care about the actual value of the test data for the subdomain is equivalent to the path coverage criterion. Error-based testing requires test cases selected not only within the subdomains, but also on the boundaries of the adjacent subdomains.

Perhaps the best known strategy is the $N \times 1$ domain adequacy criterion proposed by White and Cohen [96]. The criterion requires at least N test cases on each border of each subdomain, where N is the number of the input variables of the program. Additionally it requires one test just outside of each border. There are countless other criteria but for those we refer to related work [16].

The previous criterion illustrates one of the limitations to the application of error-based adequacy criteria. If the complexity of the input space grows, the required number of test cases grows exponentially. Nevertheless these kind of adequacy criteria are growing wider applicable with the rise of property based testing [35]. Tools such as ScalaCheck⁴ allow a developer to specify boundaries of the program's subdomains. Upon execution they generate

⁴http://www.scalatest.org/user_guide/property_based_testing

appropriate test data and verify that the subdomains behave as expected. Within ING several development teams already experiment with property based testing indicating that error-based adequacy might be used in the future.

2.2.2 Testing Smells

The quality of unit testing is not just determined by its adequacy or effectiveness. Unit tests are integrated into the code base and need to be manually maintained like any other code. Tests that no longer reflect updated behavior need to be found and fixed. The larger this effort, the more reluctant developers will be to keep the test suite up to date [27].

For years now, influential authors like Beck [19], Meszaros [61] and Glenford Myers [65] have defined best and worst practices for unit testing. The advice they give does not concern themselves much with how thoroughly software should be tested. Instead it focuses on how testing should (not) be done by giving developers concrete advice on all aspects of unit testing, from strategy down to the test code itself. In this section we describe some common best practices and smells used in this thesis project for providing an indication of testing quality.

Fowler et al. [39] defined a code smell as “A surface indication that usually corresponds to a deeper problem in the system”. A smell is very easy to spot but it might not always indicate a real problem with the code. That is because most smells are not bad on their own, often they are only indicators of a larger problem in the code. An example smell is a long method. Having a long method is not desirable and warrants inspection, but some of them are perfectly fine.

The first to introduce test smells were van Deursen et al. [33]. Following the Extreme Programming (XP) methodology they worked on a project and found that refactoring test code is different from refactoring production code. Not all of the smells that indicate trouble in production code applied to test code as well. They discovered that test code has its own problems and defined 11 smells that do indicate trouble in test code. Meszaros [61] describes smells in a broader context by explaining the reasons test smells appear as well as their side effects. He also adds new smells and makes sure they can be easily understood by developers. We present an overview of the smells we use in table 2.1.

One of the first to empirically investigate the occurrence of test smells in software projects were Van Rompaey et al. [90]. They propose a set of metrics to identify the General Fixture and Eager Test smells. However, after comparing the detection efficiency with manual inspection they are forced to conclude that their metrics are not a reliable means for test smell detection.

Reichhart et al. [75] define 27 smells and do come up with a successful approach to studying the occurrence of test smells. They implement 70 dynamic and static rules to detect the smells in a tool called TestLint and evaluated the results by manually checking for false positives. Their results show that smells appear quite often, even in code written by experienced developers, albeit to a lesser degree.

Brugelmans and Van Rompaey [24] take a more pragmatic approach to detecting test smells. They introduce a reverse engineering tool called TestQ that is able to detect 12 test smells through static source code analysis. Instead of using rules in order to detect whether a

2. BACKGROUND

Name	Description
Sensitive Equality	Equality checks depend upon inappropriate methods
Test Code Duplication	Code clones contained inside the unit test suite
Assertionless Test	A test method does not have asserts
Proper Organization	A test method violates testing conventions
Long Test	A test method has too many statements
Conditional Test Logic	Control structures, such as loops and ifs, in a test method
Slow Test	A test takes too long to run
Dead code	Test code that is not used during testing
Hard-to-Test Code	Code is difficult to test
Eager Test	A test verifies too much functionality

Table 2.1: An overview of test smells used for this project

test smells, they use a set of metrics that might indicate a smell. Several others take a similar approach but often focus on a particular aspect, such as test fixtures [43], duplication [44], and co-evolution [49]. Their efforts enable developers to quickly identify test smell hot spots.

Bavota et al. [18] is one of the few to empirically investigate the presence as well as the effect of test smells. They attempt to measure the distribution of eleven test smells in both open source and industrial projects. Through an exploratory analysis they find that test smells are widely spread throughout the software systems. A follow up quantitative analysis finds that most of the test smells have a strong negative impact on the comprehensibility of test suites and production code.

2.3 Overview of Related Tools

Over the years a wide range of tools have been created to inform developers about the quality of their software. The tools usually report on a specific aspect, like code clones or style guide violations. This section presents an overview of the tools we use for collecting data on aspects of testing quality.

2.3.1 Test Execution

Measuring the quality of unit testing starts by executing the unit tests. Since we focus on projects that use Java in combination with Maven, most projects use Surefire [13] for executing unit tests. Surefire is a maven plugin that is executed during the test phase of the Maven build life cycle.

The plugin executes all of the tests specified by the developers. After execution it generates a report on each of the classes executed with details about the execution environment, duration and success or failure of its test cases. An abbreviated report is shown in fig. 2.3. Officially the plugin is only meant to execute the unit tests of an application. In practice however, most projects use it for executing integration tests as well.


```

<testsuite name="org.apache.commons.lang3.builder.SimpleToStringStyleTest" time
  ="0.176" tests="8" errors="0" skipped="0" failures="0">
  ..
  <testcase name="testLongArray" classname="org.apache.commons.lang3.builder.
    SimpleToStringStyleTest" time="0.01"/>
  <testcase name="testObject" classname="org.apache.commons.lang3.builder.
    SimpleToStringStyleTest" time="0.011"/>
  ..
</testsuite>

```

Figure 2.3: Part of a Surefire report

2.3.2 Coverage

JaCoCo

Because code coverage is such a widely used metric there are a number of tools that can measure it. We rely on the works of Shahid and Ibrahim [85] and Alemerien and Magel [14] for a comparison of the feature set. We combine this with our own experiences and the activity in the software repository for forming a verdict. Based on the criteria above we have picked JaCoCo [3] as the tool to use for measuring code coverage. It is able to detect both statement as well as branch coverage, is actively maintained and easy to integrate with Surefire.

Code coverage analysis is a runtime metric, meaning that it requires a detailed recording of the instructions that have been executed during testing. JaCoCo solves this by instrumenting the software with probes on-the-fly during class loading. The developers define probes as follows. *“A probe is a sequence of bytecode instructions that can be inserted into a Java method. When the probe is executed, this fact is recorded and can be reported by the coverage runtime. The probe must not change the behavior of the original code.”* [4]

Because each probe adds several instructions to the software, not all edges in the control flow graph (defined in section 2.2.1) of the software are instrumented. The placement of probes depends on the type of coverage that is to be measured, and some optimizations. For instance, a method with a single branch needs only one probe. If this probe is executed we can be sure that the other statements in the method are executed as well.

During execution JaCoCo stores exactly which probes have been executed using an array of booleans in a data file. In combination with the Java byte code this file can be used to generate reports on exactly which statements and branches have been executed.

SonarQube JaCoCo Plugin

By default JaCoCo does not collect information on what lines were executed per test (coverage per test). The software only stores whether a probe was executed, not by which test. In order to still collect this information we execute JaCoCo in combination with the SonarQube JaCoCo Listener.

The SonarQube JaCoCo Listener is a part of the SonarQube Java Plugin [11]. It instructs JaCoCo to execute tests consecutively and to create a coverage dump after each test is done. The coverage dump is interpreted in combination with the byte code and uploaded to the

2. BACKGROUND

```
{
  "paging": {
    "pageIndex": 1,
    "pageSize": 100,
    "total": 10
  },
  "tests": [
    {
      "id": "AVTFEQv3U2SB2-jxavzB",
      "name": "testAnnotationsOfDifferingTypes",
      "fileId": "AVTE5BybU2SB2-jxau0t",
      "fileKey": "org.apache.commons:commons-lang3:src/test/java/org/apache/commons/lang3/AnnotationUtilsTest.java",
      "fileName": "src/test/java/org/apache/commons/lang3/AnnotationUtilsTest.java",
      "status": "OK",
      "durationInMs": 9,
      "coveredLines": 3
    },
    {
      "id": "AVTFEQv3U2SB2-jxavzE",
      "name": "testBothArgsNull",
      "fileId": "AVTE5BybU2SB2-jxau0t",
      "fileKey": "org.apache.commons:commons-lang3:src/test/java/org/apache/commons/lang3/AnnotationUtilsTest.java",
      "fileName": "src/test/java/org/apache/commons/lang3/AnnotationUtilsTest.java",
      "status": "OK",
      "durationInMs": 10,
      "coveredLines": 2
    },
    ..
  ]
}
```

Figure 2.4: Part of a response for a request that asks what tests cover a line in a production file

SonarQube server. With this information the server is now able to report for each instruction which tests executed it. An example response for what tests cover a specific line on a specific production file is shown in fig. 2.4.

2.3.3 Smell Detection

PMD

PMD [8] is a widely used tool for detecting design problems. By running a set of rules against the source code it looks for code fragments that are clearly wrong. The issues it finds do not necessarily break the software but indicate that code should probably be written differently. In any case they should be looked at by a developer. Some examples of issues that PMD might find are: unused variables, empty catch blocks and unnecessary object creation.

PMD includes a standard rule set but allows developers to select which rule violations to look for. Among the rule sets is a set of rules created specifically for JUnit tests. Rules in this rule set concern themselves with obvious mistakes in test suites as well as errors in verifying object equality, e.g., a test case should include at least one assert.

```

"Problem","Package","File","Priority","Line","Description","Rule set","Rule"
"1","org.apache.commons.lang3","/repo/src/test/java/org/apache/commons/lang3/
AnnotationUtilsTest.java ","3","389","You may have misspelled a JUnit
framework method (setUp or tearDown)","JUnit","JUnitSpelling"
"2","org.apache.commons.lang3","/repo/src/test/java/org/apache/commons/lang3/
AnnotationUtilsTest.java ","3","399","Use assertEquals(x, y) instead of
assertTrue(x.equals(y)) ","JUnit","UseAssertEqualsInsteadOfAssertTrue"

```

Figure 2.5: Part of a report generated by PMD in csv format

PMD offers a substantial set of predefined rules. One of the reasons it is so popular, is that it facilitates developers to create their own rules. The documentation includes a guide for writing own rules and PMD itself includes a rule design tool. These custom rules are on equal footing with the predefined rules. They are written using the same tools and have access to same API's.

After specifying the relevant rules in an XML file the tool can be run as a command line application. Any violations to the relevant rules are reported in a report generated by PMD after execution. A fragment of a report is shown in fig. 2.5.

CPD

CPD is a tool created by the makers of PMD specifically intended to detect clones in a code base. It finds the clones by using the Karp-Rabin algorithm [55]: by comparing the hashes of sections of code. Like PMD, CPD is run as a command line utility. The report it creates specifies the duplicate locations, number of duplicated tokens and number of duplicated lines.

SonarQube

One of the tools that is already a part of the ING Continuous Delivery pipeline is SonarQube [9]. SonarQube is a platform for managing code quality. By default it shows a developer duplications, structure and some technical debt⁵ of a project. But the platform can be easily expanded with plugins. These plugins can be used to gather and display additional information such as coverage, third party code conventions and versioning information. While its uses are very wide, in the context of this project SonarQube serves as a temporary information collection tool.

A SonarQube scan starts after Maven has run the unit tests and finished the build process. A local component of SonarQube, called the SonarQube Runner, is executed. The Runner analyzes the source code and any reports generated by supported tools, e.g., JaCoCo. The gathered information is uploaded to the SonarQube server and stored in a database. An overview of the data collection process is shown in fig. 2.7.

The information on a project is presented to the developer via a website (fig. 2.6) but is also made available via a REST API [78, 12]. An example of a response given by this REST

⁵Technical debt expresses the cost that a design decision will cost in the future at the expense of a short term gain [26].

2. BACKGROUND

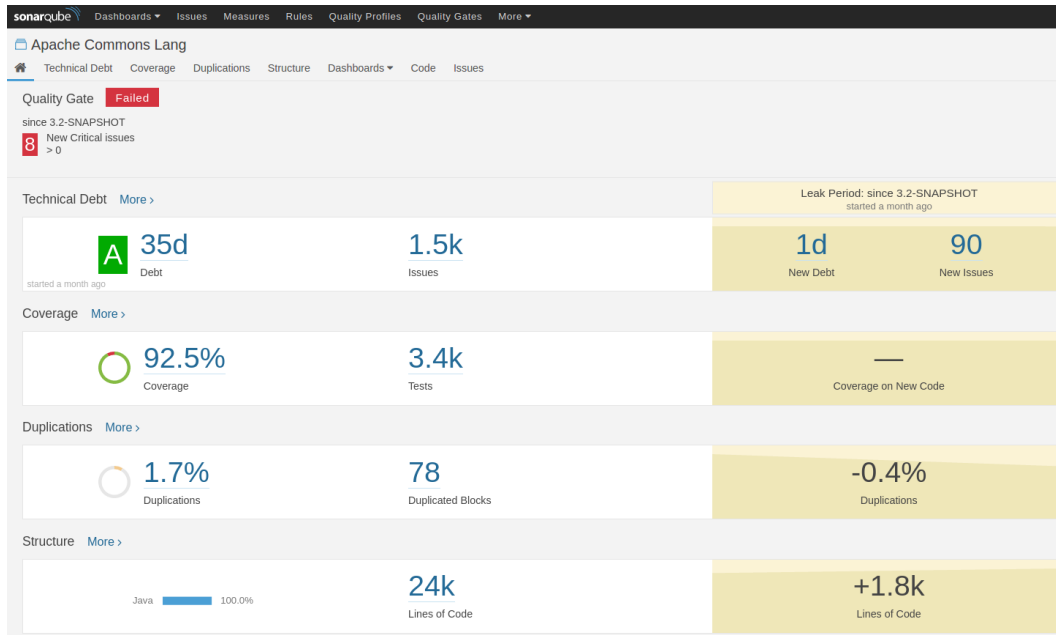


Figure 2.6: The SonarQube dashboard for Apache Commons Lang

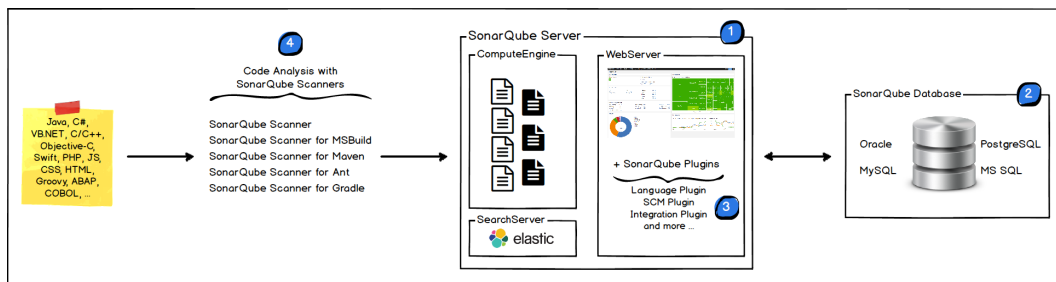


Figure 2.7: An overview of the SonarQube architecture [10]

API is given in fig. 2.4. The presentation allows a developer to compare projects to each other, see the amount of technical debt, and see exactly the issues in each module.

SonarQube provides an extensive overview of the most important issues in the production code but we found it often does not consider test code. While the tool has by default hundreds of rules for Java code, it only has a handful that deal with test code. Duplications that manifest in test code do not show up as clones and neither does the structure tab take test code into consideration. Our workaround is having SonarQube analyze a build twice. The first run the project is analyzed as it should be. The second time we tell the runner that production code includes the test code.

By analyzing every build twice we create two separate instances of the same project. It allows us to get the basics as well as the structure of the tests and a coverage report on the test suite itself.

2.3.4 Mutation Analysis

As described in section 2.2.1 mutation analysis can be used to “test the tests” by introducing faults and letting the test suite detect them. Although mutation analysis is not widely used in software development, there are a number of tools that can perform it. Delahaye and du Bousquet [30] present an overview of available mutation analysis tools for Java and recommend Pitest (also referred to as PIT) for use in industry. Even though Pitest is very limited in the type of faults it is able to introduce compared to for example μ Java, we still follow their recommendation. We need a tool that is ready for production and Pitest seems to be one of the very few that offers the required stability. Besides stability, it integrates easily with Maven, is actively maintained by the developers and performs well.

Pitest systematically introduces faults in production code by using mutators. Mutators are a configurable set of operators that each alter a specific piece of code. An example of a mutant generated by the conditional boundary mutator is shown in fig. 2.2. Besides fault injection, Pitest’s activities can be classified into three more categories [7], listed below.

1. Fault injection: The selected mutators are used to generate mutants by modifying the Java byte code.
2. Test selection: For each mutant the tests are selected based on a coverage report. Only the tests that cover code surrounding the mutant are selected to run.
3. Mutant insertion: Each mutant is separately loaded into the JVM using its instrumentation API.
4. Mutant detection: Pitest will start detecting the mutant by running the unit tests of the mutated production class. If none of these fail it will continue with the other tests until one of them does, “killing” the mutant. If there is no test that fails, the mutant will exit with one of the states shown in table 2.2.

After Pitest is run it generates a report with information on each of the mutants. The report provides details on the mutant itself as well as the test that detected the mutant. A fragment of a report is shown in fig. 2.8.

2.3.5 Internal ING

Because of the scale of software development at ING, the bank has its own department for creating development tools. Known as “building blocks” this department develops applications that help other developers create, test and maintain software. Among the applications in their portfolio is the Continuous Delivery pipeline (see section 2.1.3) and a tool called Bugout which we will discuss in this section.

Bugout was created by a group of Romanian ING developers when they realized the need for fully automating acceptance testing. Whenever they wanted to release their mobile application they were required to run their tests and present the outcome. The tests they had were already automated, but collecting data and presenting it required the same manual effort for each release. Figuring they could automate the rest of the process as well, they created

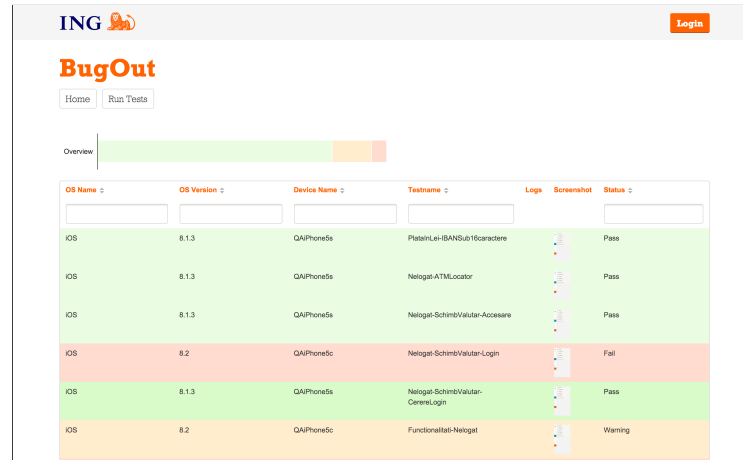
2. BACKGROUND

```
<mutations>
  <mutation detected="true" status="KILLED">
    <sourceFile>One.java</sourceFile>
    <mutatedClass>com.ing.example.One</mutatedClass>
    <mutatedMethod>methodWithMutation</mutatedMethod>
    <methodDescription>(II) Ljava/lang/String;</methodDescription>
    <lineNumber>16</lineNumber>
    <mutator>org.pitest.mutationtest.engine.gregor.mutators.
      RemoveConditionalMutator_ORDER_IF</mutator>
    <index>4</index>
    <killingTest>com.ing.example.OneTest.testMethodWithMutation(com.ing.example.
      OneTest)</killingTest>
  </mutation>
  <mutation detected="false" status="SURVIVED">
    <sourceFile>One.java</sourceFile>
    <mutatedClass>com.ing.example.One</mutatedClass>
    <mutatedMethod>methodWithMutation</mutatedMethod>
    <methodDescription>(II) Ljava/lang/String;</methodDescription>
    <lineNumber>16</lineNumber>
    <mutator>org.pitest.mutationtest.engine.gregor.mutators.
      RemoveConditionalMutator_ORDER_IF</mutator>
    <index>6</index>
    <killingTest/>
  </mutation>
</mutations>
```

Figure 2.8: Fragment of a pitest report in XML format

Name	Description
Killed	A test failed when run against the mutated version of the software
Lived	No tests failed when run against the mutated version of the software
No coverage	No tests were found that executed the line of code where the mutation happened
Non viable	The mutant invalidated the code and could not be loaded by the JVM
Timed Out	The mutant timed out, this likely occurred because it caused an infinite loop
Memory error	The mutation increased the memory use beyond the JVM's maximum
Run error	An internal Pitest error occurred

Table 2.2: The possible outcomes of a Pitest mutant



OS Name	OS Version	Device Name	Testname	Logs	Screenshot	Status
iOS	8.1.3	QAiPhone5s	PlatinLea-iBANSub10caractere			Pass
iOS	8.1.3	QAiPhone5s	NetlogATMLocator			Pass
iOS	8.1.3	QAiPhone5s	Netlogat-SchimbValutar-Accesare			Pass
iOS	8.2	QAiPhone5c	Netlogat-SchimbValutar-Login			Fail
iOS	8.1.3	QAiPhone5s	Netlogat-SchimbValutar-CerentLogin			Pass
iOS	8.2	QAiPhone5c	Functionalitati-Netlogat			Warning

Figure 2.9: Bugout interface showing acceptance test results for an application

a proof of concept and named it Bugout. Since then Bugout has received the attention of the Dutch Building blocks team. They quickly realized the possibilities such an application could offer and saw it as one every developer should use. In a joint effort both teams are now making the tool more widely applicable within ING and a permanent part of the CD Pipeline.

In essence the concept behind integrating the tool with the CD pipeline is simple. As explained in section 2.1.3 Nolio performs a deploy to some testing environment. This testing environment is owned by the application’s development team and will differ per application and the kind of testing done. In the environment the tests are run and their results as well as any other desired information is sent to the Bugout API. From here on Bugout stores the information in its database and makes it available via both a REST api and a web page. An illustration of Bugout’s interface is shown in fig. 2.9.

The ability to analyze test results before going into production is critical for achieving Continuous Deployment. Bugout offers this opportunity to managers in the form of a success/failure result per build. To developers it provides the opportunity to get a detailed look into the tests, answering questions like: “Why did a test fail?”, “How did a test perform?” and “did the performance change?”. To summarize, the tool provides insight into the quality of the product and will bring the CD pipeline one step closer to fully automating deployment.

2.4 The SIG Quality Model

The Software Improvement Group (SIG) has developed a model for assessing the maintainability of software [45]. In this work we use some of the techniques used in their model for assessing the quality of unit test code. We therefore provide an overview of the model, focusing on the parts that are used in this work.

The SIG quality model provides a 1 – 5 star rating over a project’s maintainability, more stars always meaning a better maintainability. The verdict is based on the ratings for a set of code quality characteristics taken from the ISO 25010 standard [51]: analysability,

2. BACKGROUND

changeability, stability and testability. Each of these is calculated by mapping properties of the source code to a characteristic. The properties in turn are measured by source code metrics. The next sections will explain this process in more detail.

2.4.1 Source Code Metric to Property

In this section we elaborate on how source code metrics are converted into system wide properties with a 1 – 5 star score. The conversion is based on how the metric value for a property compares to the same value for other projects. SIG uses a benchmark containing a set of industrial and open source projects [15] to calculate thresholds that allow for a quick conversion to the star rating.

For system-wide metrics, such as duplication, that are already measured on a project level, the calculation is relatively straightforward. Because the metrics are defined so that a lower value is always better, a project receives the highest scores for the lowest metric values. Concretely, the metric thresholds for a project receiving a certain score are based on a $< 5, 30, 30, 30, 5 >$ quantile distribution of the metric values in the benchmark. The thresholds are calculated so that a system needs to be among the best 5% of the projects in the benchmark to receive a 5 star score. It will receive a 4 star score if it is among the next best 30%, and so on.

For metrics calculated on a unit level, such as complexity and unit size, the property calculation is slightly more difficult. The ratings for their properties are calculated using the *quality profiles* technique, described in detail by Alves et al. [15]. It starts with thresholds that map each metric value to one of four risk levels: very high, high, moderate and low. For example, a method with cyclomatic complexity ≥ 20 might end up in the very high risk category.

A project property is assigned a rating based on the distribution of the risk categories. In order for a project to achieve a certain quality, there is a limit to the relative amount of high risk units it may have. Table 3.5 shows the thresholds a property might have for each risk category. The thresholds and property in this table are fictional, in reality these are based on benchmark data, similar to the way the system-wide metric thresholds were chosen. The thresholds are all cumulative, meaning that for a five star rating for the example property from table 3.5, at most 65% of the units in a project can have a quality score of three or lower.

2.4.2 Properties to Maintainability Characteristics

A selection performed by SIG maps each of the properties to one or more maintainability sub-characteristics. For example, the testability of a project is calculated from the unit size and complexity properties. The score of a sub-characteristic is the average rating of its properties. As a last step, the overall maintainability rating is calculated by averaging the four sub-characteristics.

Property quality rating	Maximum percentage of risk categories			
	Low	Moderate	High	Very High
5 stars	-	95%	65%	35%
4 stars	-	-	70%	40%
3 stars	-	-	75%	45%
2 stars	-	-	80%	50%
1 star	-	-	-	-

Table 2.3: Example quality profile for the SIG approach

Chapter 3

Proposed Approach to Assuring Unit Testing Quality

This chapter describes our approach to assuring unit testing quality in the form of a tool: Spectata. We start by detailing the requirements and continue with our view on unit testing quality. The last sections are dedicated to describing how the tool will help assure testing quality.

3.1 Requirements

Any solution should be designed to help assure testing quality in a Continuous Delivery environment. Its goal should be to give insight into the quality of unit testing of a piece of software. To both managers as well as developers it should provide information aimed at answering the following questions about a project.

1. **PQ1 – What is the state of unit testing?** First and foremost we want to make everybody involved in the project aware of unit testing. Spectata must be able to tell its users to what extent the unit tests actually test the production code and how this compares to other projects. In addition, the tool must provide insight into how well the tests themselves are written: whether they contain coding violations, common flaws and so on. With this information the development team will determine whether the product is ready for release, or if they should dedicate more effort to testing.
2. **PQ2 – What type of changes should be made to the test suite?** In case the quality of the test suite is not sufficient, Spectata must be able to provide insight into what should be done to improve it.
3. **PQ3 – Where should the development team concentrate its testing efforts?** If the product is not sufficiently tested, developers need to know which parts. Spectata must be able to provide the tests where the testing effort is lacking.
4. **PQ4 – How did the latest changes affect testing quality?** Because Spectata is part of a toolset made for an agile workflow, it must be able show changes in quality over

3. PROPOSED APPROACH TO ASSURING UNIT TESTING QUALITY

time. A new feature can only be accepted into production if it is tested properly so it must be able to highlight changes that cause a drop in quality.

The following sections provide more detailed requirements as to how the solution must answer the questions above.

3.1.1 Collecting Metrics

The first step towards assuring testing quality is gathering the testing metrics. Each time a build is triggered the tool must create a new report on the quality of unit testing to reflect the latest changes. Being a part of the ING pipeline means the process must be integrated with the build server. It cannot however slow the build process down or fail the build in any way. Therefore the collection process is triggered by the build server but must run on a separate compute instance.

Configuring the build server to run the collection process is done by the development team. In addition to testing and packaging, developers use the build server to run custom scripts that may be necessary before releasing, e.g., a security scan. The collection process should be configured in the same manner: as an extra step in the build process.

For this stage of the tool we may assume that all the projects are Java Maven projects that execute their unit tests with the maven surefire plugin¹. In order to speed up adoption it is highly desirable that the effort for integrating Spectata into the pipeline is as low as possible. The tool should be flexible enough to at least be able to calculate a partial verdict on products that meet these requirements.

For the initial version of Spectata high availability is not a requirement. However, because in the future it might be used in production its data storage needs to adhere to ING standards in order to prevent a large refactoring effort later. This means data needs to be stored in one of the supported data storage solutions: Cassandra², Microsoft SQL server³ or Oracle⁴.

3.1.2 Quality Verdict

The second step of assuring testing quality is calculating a quality score from one to five based on the metrics collected in the first step. Spectata needs to assign quality verdicts on three levels. First it must assign quality scores on the different aspects of unit testing over the project as a whole. These global scores must be comparable to that of other projects and be as independent as possible from differences in their relative sizes.

The global quality scores must have a basis in the local quality scores. A local quality score is calculated over a unit test class by itself on a specific aspect of unit testing. Improving a unit test in this aspect should thus be rewarded by an increase in overall project testing quality.

¹<http://maven.apache.org/surefire/maven-surefire-plugin/>

²<http://cassandra.apache.org/>

³<https://www.microsoft.com/nl-nl/server-cloud/products/sql-server/overview.aspx>

⁴<https://www.oracle.com/database/index.html>

Last of the quality verdicts is that of a change. A new release should be tested at least as well as the previous one. Spectata therefore needs to be able to compare builds, point out any new issues, and calculate quality score deltas.

3.1.3 Report

The final part of the execution is to create a report of the the quality scores as well as the individual issues that led to them. The tool should be able to provide all of this information via HTTP requests per project and per build. The reason for requiring this kind of interface is twofold. First, within ING all products work with this type of interface, allowing it to easily become a part of the existing pipeline. Second, it gives the tool the flexibility to be integrated with a broader approach for assuring product quality.

3.2 Quality Characteristics

In order to provide feedback over a project's unit tests, we first need to measure their quality. Luckily, testing and testing adequacy have been studied by researchers for decades. Based on their work, summarized in section 2.2, we have selected three characteristics of high quality unit tests: adequacy, effectiveness and maintainability. Each of them assesses a specific aspect of unit testing and complements the others. In the next paragraphs we describe the characteristics and how Spectata measures them.

3.2.1 Structural Adequacy

First of the quality characteristics is structural adequacy. Structural adequacy criteria assess software by measuring to what extent the test suite exercises the structural elements of the production code. Section 2.2.1 contains a more detailed explanation of how it is defined and some of the more prevalent criteria. The section also states that the simplest way of measuring it is through coverage.

Code Coverage

Because it is a simple and effective measure, we see coverage as a crucial feature of unit testing quality. High coverage is not necessarily indicative of a high quality, but *low* coverage can indicate low quality. A test suite that hardly exercises the production code might as well not exist and should receive a low quality score.

To a developer code coverage is important for much the same reason. It is impossible to have a decent level of confidence in the behavior of a piece of untested production code. By providing information regarding what elements remain untested, code coverage can guide the development effort.

We have chosen statement coverage and branch coverage to be at the basis of assessing structural adequacy. Ordinarily those two metrics are expressed as the percentage of elements in the production code covered by the test code. However we are providing an adequacy

3. PROPOSED APPROACH TO ASSURING UNIT TESTING QUALITY

score on individual unit tests. More importantly we want to give advice to developers where to increase their testing efforts.

By definition each unit test file is created for testing its production counterpart. Using a technique defined by Lubsen et al. [59] and Zaidman et al. [101] we relate the production and test code to each other. Convention and best practices [83] dictate that the name of a unit test file is the name of the production file with “Test” prepended or appended. Using this information we can assign each of the unit test classes to a Matching Production Class (MPC). With this connection we relate coverage data of a production class to the structural adequacy score of a unit test class.

Even though a unit class is meant to test its MPC, this does not mean that each production class is solely tested by its dedicated unit test class. The downside of this: a high coupling, we penalize in test maintainability (section 3.2.3). We take this into account by evaluating the adequacy of a unit test class by using the coverage information of the test suite as executed by Surefire. This way, a unit test class is responsible if its MPC is inadequately tested but it is not required to test all the logic in its MPC. We define the adequacy metrics as follows.

$$\text{M1: Statement Coverage} = \frac{\text{statements executed in MPC by test suite}}{\text{\#statements in MPC}} \times 100$$

$$\text{M2: Branch Coverage} = \frac{\text{branches executed in MPC by test suite}}{\text{\#branches in MPC}} \times 100$$

One last problem is that not all production classes have a dedicated unit test class, or the suite might not have any unit tests at all. In any case we still want to consider the coverage of these classes in the project adequacy verdict. Therefore, if a class does not have a dedicated unit test class, we assign the adequacy score to the production class instead. This way we can tell a developer to create a new unit test class if coverage is low and the class’ score can be taken into consideration when calculating a project score.

3.2.2 Effectiveness

The second quality characteristic is the effectiveness of a test. Effectiveness is the extent to which a test is able to find faults in the executed production code. It is important to note that effectiveness in our work is not completely a fault based criteria (section 2.2.1). What sets it apart from criteria in that category is that it only takes into account the production code that is actually exercised by the tests.

Mutation Analysis

As explained in section 2.2.1, mutation analysis has historically been used to assess the fault-finding effectiveness of a test suite [52, 64, 76, 28, 54]. For years researchers verified the effectiveness of testing strategies[56], coverage criteria [68] and more, using the faults introduced by mutation analysis. For a long time this seemed to be its primary use case. Even though it gave useful feedback, software developers considered it not worth the effort, it being either too time-consuming or too expensive to perform. Nowadays, performing

Name	Description
Conditionals Boundary	Replaces the relational operators, e.g. <, >=, with their boundary counterpart
Negate Conditionals	Mutates all conditionals found according to a replacement table ⁵
Increments	Replaces increments with decrements and vice versa
Invert Negatives	Inverts negation of numbers
Math	Replaces every binary arithmetic operation on a number with another operation
Return Values	Mutates the return values of method calls
Void Method Call	Removes method calls to void methods
Switch Mutator	Swaps labels in switch statements

Table 3.1: Mutators used for Mutation Analysis

mutation analysis on a large software project is still expensive, but only when compared to other types of dynamic analysis.

For the software developer mutation testing points out code fragments that are not being tested properly. A low mutation score can point out production code that seems tested according to coverage but is missing some checks for common coding mistakes. Additionally, mutation analysis helps a developer imagine faults earlier, which in turn will help ensure correctness [87].

What possible faults a mutation analysis helps find depends on the mutators used. Several studies have been done that recommend mutators [32, 70, 60]. We have chosen a set of mutators that both follow these recommendations and are supported by the tool we chose to use, Pitest. The mutators we use are shown in table 3.1.

The mutation score that is calculated is similar to that defined in section 2.2.1, deviating slightly by completely disregarding mutants generated in code that is not covered. The reason for disregarding them is that not covered code indicates a problem in structural adequacy rather than effectiveness. Using the same technique as we use for calculating structural adequacy we set the mutation score of a unit test class as that of its matching production class. Our metric for mutation score then becomes as follows.

$$\text{M3: Mutation score} = \frac{\#mutants \text{ killed in MPC}}{\#mutants \text{ lived in MPC} + \#mutants \text{ killed in MPC}}$$

3.2.3 Maintainability

The last of the quality characteristics is maintainability. We define maintainability of test code as the ease with which it can adapt to changes in production code. Using the test code smells described in section 2.2.2 we assess maintainability by detecting their occurrence in test classes. A class is not said to have a smell or not, we simply measure the extent to which it has the indicators of a smell.

3. PROPOSED APPROACH TO ASSURING UNIT TESTING QUALITY

Smells are very naturally correlated with the length of a unit test class. When a class is longer it will most likely contain more bad code conventions. We therefore normalize all but one of the indicators by the number of test cases that a class executes.

Length

The first smell we aim to detect is Long Test. This smell obscures tests and makes it hard for developers to localize defects. We measure both the length of a test class as well as the average length of a test case in that class.

M4a: Class Length = *Lines of code in file*

$$\text{M4b: Case Length} = \frac{\sum_{t_i \in T} \text{Lines of code of } t_i}{\# \text{Test cases in class}}$$

Where T is the set of test cases in a test class.

Asserts

The number of asserts in a test case determines how closely the behavior of a production class is tested. For fault finding effectiveness it might be desirable to put as many asserts in a case as possible. However this also obscures the purpose of the test. Having too many asserts may mean the test case verifies too much of the behavior and becomes hard to read and understand. In order to detect the Eager Test smell we therefore statically count the number of asserts. This number should be as low as possible, as a higher number means a stronger presence of the smell. The other extreme of having no asserts in a test case is measured by the JUnit Conventions metric.

$$\text{M5: Asserts} = \frac{\# \text{Asserts in class}}{\# \text{Test cases in class}}$$

Complexity

The Conditional Test Logic smell states that there should be only one execution path per test case. Having more than one will make it difficult to determine exactly what a test executes, and if it failed, why. Any if statements, loops, and try-catch blocks will make the test harder to understand. We therefore detect the Conditional Test smell by measuring the McCabe Cyclomatic Complexity in a test class.

$$\text{M6: Complexity} = \frac{(\sum_{m_i \in M} \text{Cyclomatic Complexity of } m_i) - \# \text{Test cases in class}}{\# \text{Test cases in class}}$$

Where M is the set of methods in a test class.

Dead Code

There is no good reason for having test code in a unit test class that is not executed during unit testing. Test code that is forgotten or temporarily disabled should not be present in this stage of the software delivery pipeline. The code might obfuscate and require maintenance while providing no benefits. We measure dead code by having JaCoCo install its probes in the test code as well as in the production code. Inspecting the coverage report allows us to detect any statements in the test suite that were not executed.

$$\mathbf{M7: Dead Code} = \frac{\text{Lines of unexecuted code in class}}{\#Test cases in class}$$

One thing to note for this metric is that any code between throwing and catching an exception is seen as not exercised because of the way JaCoCo calculates coverage. Test cases that catch an exception therefore always have one or more lines of what is seen as dead code.

Duplication

Many of the tests in a suite do similar things. They often verify a slightly different scenario and may need the same logic to set up, or verify behavior. It is easy for a developer to simply copy-paste the required pieces of logic instead of refactoring it in a separate method. However this severely increases the cost of making a change to production code. We therefore measure duplication as part of the maintainability of a test suite by detecting clones. A clone is a code fragment longer than 200 tokens that can be found in more than one place in the test suite. The duplication of a unit test class is the number of lines of all the cloned fragments it has.

$$\mathbf{M8: Duplication} = \frac{\sum_{c_i \in C} \text{Lines in } c_i}{\#Test cases in class}$$

Where C is the set of clones in a test class.

JUnit Conventions

All the projects used in our work use the JUnit framework for testing. This framework comes with its own set of conventions that relate to the smells: Proper Organization and Sensitive Equality. Violations to these conventions may lead to a test suite that is harder to read and understand. Using the PMD JUnit rule suite we detect the violations and we calculate a metric as follows.

$$\mathbf{M9: Conventions} = \frac{\text{Violations found in class}}{\#Test cases in class}$$

Coupling

Ideally a unit test class only exercises its matching production class. It should only fail because of an error in that class and is not responsible for verifying logic in other production classes. However in practice this is simply not feasible. All production classes are interrelated and stubbing or mocking every relation requires too much effort. In practice unit test classes exercise more than one production class. But the Highly Coupled Code smell tells us to keep this number as low as possible. Because every class and line of code exercised by a test class increases its fragility. We detect the Highly Coupled Code smell by inspecting the coverage report and verifying for each test case, what production code was executed. Both the number of lines covered and the number of classes in which those lines were executed are measured.

$$\text{M10a: Coupling Classes} = \frac{\sum_{t_i \in T} \text{Classes executed by } t_i}{\# \text{Test cases in class}}$$

$$\text{M10b: Coupling Lines} = \frac{\sum_{t_i \in T} \text{Lines executed by } t_i}{\# \text{Test cases in class}}$$

Where T is the set of test cases in a test class.

Duration

Unit tests are supposed to execute quickly in order to give feedback as soon as possible. Because of slow tests developers will waste more time verifying changes and may even be inclined to execute the tests less. In order to detect the Slow Tests smell we collect the duration of tests and calculate the average duration of test cases in a test class.

$$\text{M11: Duration} = \frac{\sum_{t_i \in T} \text{Duration of } t_i}{\# \text{Test cases in class}}$$

Where T is the set of test cases in a test class.

3.3 Descriptive Statistics

In order to get a first impression of the chosen metrics we analyzed their values in four releases of open source projects. We have chosen these projects because they are reliable pieces of software that are widely used, have an extensive test suite and are created by experienced developers. The projects are shown in table 3.2 along with some basic information about the size of the unit test suite of each. The table shows that the projects vary substantially from each other in size. It also shows that the projects are tested differently, for example, the LOC per class for Joda-Time is much higher than that of JUnit.

The second step in our analysis was studying metric distributions, shown in table 3.3. The figure shows the normalized values as calculated by the formulas in the previous sections. Some of the values that stand out are the low numbers for the JUnit convention violations. With a median value of 0 it is very unlikely to tell us anything about the quality of the test

Project	Version	Unit Test Classes	Unit Test LOC	Prod. Classes	Prod. LOC
JUnit4	4.12	61	19937	195	9317
Joda-Time	2.9.4	62	6959	164	28126
Apache Commons Lang	3.2	100	34329	133	24289
Apache Commons Math	3.6.1	394	71239	990	100364

Table 3.2: Projects used for the initial analysis, unit tests measure here all have an MPC, LOC number is the uncommented lines of code

suite and we dropped the metric from the suite as a result. Other than that, both statement and branch coverage seem to be very high. These are easily explained by our data set which contains four well tested projects at their release points.

We investigate whether the metrics are meaningful by relating them to change proneness. In this work change proneness refers to the number of times a unit test class was changed since the earliest stable version. This definition is similar to change proneness measures in related work [21, 93, 74]. For all of the projects in our benchmark the earliest stable version is simply the first registered git commit. In order to be able to use the change proneness as a dependent variable in an analysis with multiple projects we then normalize the value by dividing by the total number of changes in the test suite.

We chose change proneness because it shows which classes are likely to change more often, thus having a direct impact on the maintenance effort. Whether maintenance is caused by a defect in the software or a newly introduced feature, a development team will always want to either prevent it or minimize its changes. The three hypotheses below show how the chosen metrics can influence change proneness.

1. Classes with stronger code smells will require a larger maintenance effort, thus will be more change prone.
2. A higher coverage means more of the production class is exercised thus less error prone and the test class is less likely to change.
3. A higher mutation score means the production class behavior is tested more thoroughly thus less error prone and the test class is less likely to change.

We test these hypotheses by correlating the metric values with change proneness. This means that ideally all of the maintainability metrics should be positively correlated, while effectiveness and adequacy metrics should be negatively correlated with change proneness. For the last two aspects however, the relationship is quite indirect and we expect the correlation to be a weak one at best.

Table 3.4 shows the Spearman rho coefficient and corresponding p value for each of the metrics. It shows no significant results for statement and branch coverage. As expected, the mutation score is very weakly negatively correlated. Most of the maintainability metrics however are significantly positively correlated, albeit not very strong. The exception: duration

3. PROPOSED APPROACH TO ASSURING UNIT TESTING QUALITY

Metric	Min	Q1	Me- dian	Q3	Max	Stdev	Mean
M1: Statement Coverage	25	88.4	95	100	100	10.7	91.6
M2: Branch Coverage	0	75	87.5	96.4	100	30.8	76.2
M3: Mutation Score	0.174	0.828	0.917	1	1	0.129	0.884
M4a: Class Length	2	30	70	150	1040	163	127
M4b: Case Length	0	5.47	12	21	252	20.2	16.4
M5: Assert Count	0	0.814	2.25	3.69	22.1	3.09	2.9
M6: Complexity	0	0.0245	0.592	1.56	32	2.39	1.28
M7: Dead Code	0	0	0.448	1.29	28	3.07	1.31
M8: Duplication	0	0	0.74	13.5	2350	171	32.7
M9: Conventions	0	0	0	0	2	0.1	0.0128
M10a: Coupling Classes	0.167	3.2	6.8	10.7	38.2	6.05	8
M10b: Coupling Lines	0.333	28.5	92.7	188	892	137	134
M11: Duration	0	11.1	19	27	3030	155	38.7

Table 3.3: Distribution for the metrics defined in section 3.2

Metric	Rho	P-value
Class Length	0.589	7.82e-48
Coupling Classes	0.299	9.73e-12
Asserts	0.283	1.33e-10
Dead Code	0.254	8.56e-09
Coupling Lines	0.215	1.26e-06
Duplication	0.182	4.21e-05
Case Length	0.125	0.00518
Complexity	0.104	0.0136
Mutation Score	-0.0878	0.0412
Duration	-0.126	0.00493
Line Coverage	–	0.122
Branch Coverage	–	0.0527

Table 3.4: Metric correlations with change proneness

being significantly negatively correlated is not entirely surprising. In theory it perfectly measures the *long test* smell, in practice its value is erratic as it is heavily influenced by the environment that the tests are executed on.

Based on the correlations we chose to continue with all of the metrics except for duration. Even though coverage is not significantly correlated to test change proneness, the related work referenced in section 2.2.1 dictates that it is a key part of structural adequacy and thus unit testing quality. Similarly, section 2.2.1 also explains why mutation score is crucial for fault finding effectiveness.

3.4 A Unit Test Quality Score

With the metrics selected in the section above, we now create a quality model. The model evaluates the unit test code on the three defined quality characteristics: structural adequacy, fault finding effectiveness and maintainability. Each of these characteristics receives its own score between 1 and 5 stars, where 1 is worst, and 5 best. The following sections describe the scoring process bottom-up: starting with metric calculation and ending with the project quality scores.

3.4.1 Local Quality Scores

After the tests are executed and results collected, every unit test class has a value for each of the metrics. In the first step we combine the metrics in each characteristic to form a quality verdict over a unit test class.

Adequacy and Effectiveness

For the adequacy and effectiveness criteria this process is relatively straightforward. For all of their metrics: statement coverage, branch coverage and mutation score we can say that a higher value is better. Based on the data from the four benchmark projects listed in table 3.2 we have determined thresholds for each of the metrics. The exact threshold values can be found in appendix A. Similar to the SIG approach, as explained in section 2.4, they are based on a $< 5, 30, 30, 30, 5 >$ quantile distribution. A score for each aspect is then calculated by taking the average of the metric scores and rounding down.

Maintainability

We have implemented the score calculation for maintainability different from that of adequacy and effectiveness. The main reason for choosing another approach is that the metrics for maintainability are much less straightforward than the metrics for the other aspects. For all of the maintainability metrics we can say that in general a lower value is better, however there is rarely an ideal number a developer should aim for. Whereas coverage of a production class is ideally at a 100%, the length of a test class should be small enough so that it can be easily maintained.

There is also a trade-off between the metrics to consider. A change aimed at improving maintainability might lower one metric value but increase another. For example, introducing a mock or stub will decrease the coupling metrics but also increase the length of the test class and its cases.

Furthermore, each developer has his or her own way of writing code. Differing programming styles are not necessarily any better or worse, but will lead to other metric values. It is next to impossible to take this into account in a simple rule based scoring approach, similar to that for effectiveness and adequacy. A verdict based on a rule that says *A test class with coupling over 30 should receive a coupling score of 1* might lead to behavior where developers aim for improving a metric, instead of improving maintainability [22].

3. PROPOSED APPROACH TO ASSURING UNIT TESTING QUALITY

For the reasons above we have chosen to calculate a test class maintainability verdict using a machine learning algorithm, specifically a classifier. The classifier decides which quality category, 1 – 5 stars, to place a unit test class in by looking at its values for the maintainability metrics. The decision the classifier makes is based on example unit test classes, the training set, that it saw while it was being taught. Each class in the training set has a value for each of the maintainability metrics and an already assigned maintainability score. After teaching (training) the classifier what metric values indicate which maintainability ratings it can be used to assign a rating to classes of which we do not know the maintainability. This way, a calculated maintainability score for a new unit test class will generally approach the score for a unit test class in its training set with similar metric values.

To assign a maintainability score for each unit test class in the training set we chose to use the change proneness of the class. As we explained in section 3.3 change proneness is closely related to the maintenance effort and maintainability. A lower maintainability will lead to a larger maintenance effort and more changes to a class. Based on the change proneness of each class in the training set, a class is placed into one of five quantiles with quantile cuts: $< 5, 30, 30, 30, 5 >$. The five quantiles correspond with the five quality categories so that the 5% of classes with the largest change proneness will receive 1 star, the next 30% of classes with the largest change proneness receive 2 stars, and so on.

One argument to be made against using change proneness to base the maintainability score on is that ideally test classes only change when production classes change [80]. Work by Zaidman et al. [100] indeed shows that a large portion of the changes made to test classes are caused by them co-evolving with production classes. However there is no relation between these kinds of changes and the unit test class maintainability metrics, with the exception of the coupling metrics. All unit test classes will be changed because of changes in the production code, irrespective of their maintainability.

We have selected four classifiers that are able to calculate a maintainability score for a unit test file based on its metric values. These are: Random forest, Logistic regression, Naive Bayes and Support vector machines. As a training set we use the unit test classes from the set of benchmark projects shown in table 3.2. In terms of maintainability this set contains test classes of a varying quality and several distinct testing styles.

In order to decide which one of the four selected algorithms to use we plotted their receiver operating characteristic (ROC) in fig. 3.1. An ROC graph shows the true positive rate (or recall) as a function of the false positive rate (or fall-out). By varying the threshold that determines the probability at which a sample belongs to a certain category, a curve is created. To illustrate, a truly random classifier will have an ROC curve that moves toward a line between (0,0) and (1,1) as the sample size increases.

Because ROC curves only work for binary classifications, one classifier was trained per quality category. Each classifier is used to determine whether a sample belongs to its category or not, and thus giving binary results. The graphs show the curves for the individual qualifiers as well as a “macro-average”. This last curve is formed by averaging the true and false positives for the five categories at each point. For each of the curves we calculated their area under the curve (AUC) noted in the labels. The AUC represents the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one. It provides a decent overall score of the algorithm.

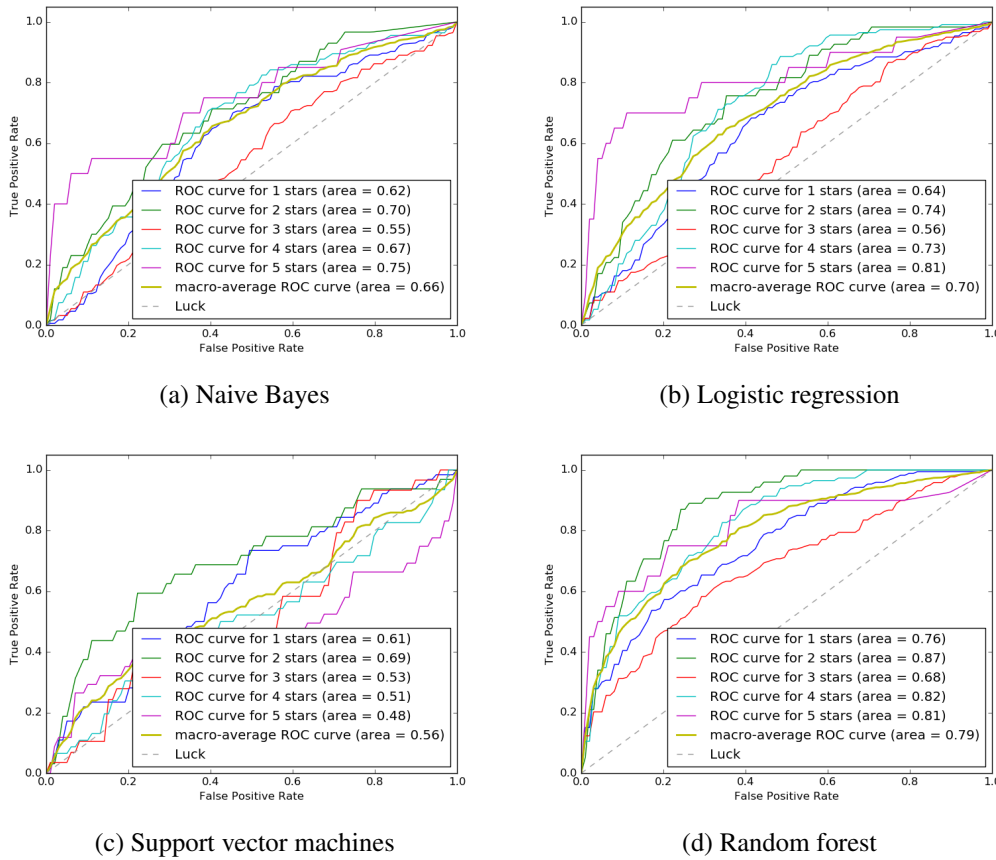


Figure 3.1: ROC graphs for the classification algorithms

We have tested the four different algorithms using 10-fold cross validation on the training data set and compared their characteristics. Based on the information provided by the ROC curves we have chosen for the Random forest algorithm. Its AUCs are highest for nearly all individual qualifiers and it has the highest average AUC. The individual curves are fairly smooth, providing an indication that the decision making process is relatively independent from this specific sample set.

The Random forest algorithm works by creating a number of decision trees when it is being taught. A decision tree can be pictured as a flowchart in which the paths from root to the leafs represent classification rules. At each internal node a decision is taken on the value of an attribute and each leaf node represents a class label (quality category in this case). The outcome of the random forest classifier is the quality category that was picked by a majority of its decision trees.

To make sure that the classifier uses all features in the decision making process we extract the feature importances of the algorithm, shown in fig. 3.2. We use the values as reported by

3. PROPOSED APPROACH TO ASSURING UNIT TESTING QUALITY

the Random forest implementation in the Spark machine learning library ⁶, which follows the method suggested by Friedman et al. [40]. The feature importances are normalized to sum to one.

Figure 3.2 shows that the class length and coupling classes metrics are considered the most important by far. For class length this is not surprising as it is much stronger correlated with change proneness than the other features. The fact that the coupling classes metric is considered important can be explained by its relation to changes in production code. Tests with a higher coupling are more dependent on production classes and thus likely to change more because of changes in production classes. Duplication is considered the least important feature. Table 3.3 shows us that the value for the duplication metric is very low for unit test classes, over 25% of them have no duplicated code at all. Because duplication occurs so little and the correlation between it and change proneness is one of the weakest, its low importance comes as no surprise.

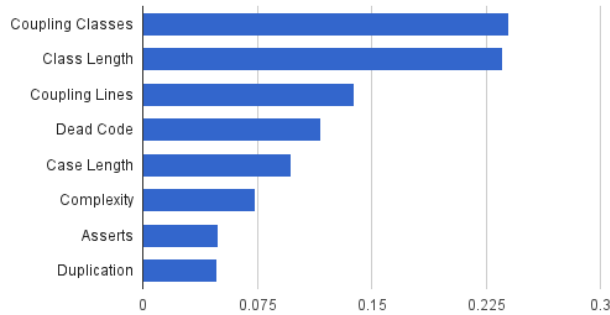


Figure 3.2: Random forest feature importance for predicting the maintainability quality category, normalized to sum to one

3.4.2 Global Quality Scores

For the global quality scores we form one score out of those of the individual test classes. All projects receive three quality scores: one for each of the quality characteristics. As with the local quality scores they are discrete and can vary from one to five. In order to calculate this score we use the SIG quality profile approach [15].

A project is assigned global quality scores based on the distribution of its local quality scores. In order for a project to achieve a certain quality, there is a limit to the relative amount of low quality tests it may have. Table 3.5 shows the exact thresholds for each global quality category. The thresholds are all cumulative, meaning that for a five star rating at most 65% of the unit tests in a project can have a quality score of three or lower. This approach differs

⁶<http://spark.apache.org/docs/2.0.0/api/scala/index.html#org.apache.spark.ml.classification.RandomForestClassifier>

Project Rating	Maximum percentage of local ratings				
	5	4	3	2	1
5	-	95%	65%	35%	5%
4	-	-	70%	40%	10%
3	-	-	75%	45%	15%
2	-	-	80%	50%	20%
1	-	-	-	-	-

Table 3.5: Quality profiles for calculating a global unit testing quality score

from the SIG approach in two ways: (1) we use aggregated metrics over classes optimized on change proneness to base the global properties on, instead of raw metrics over units. (2) In our approach the units are classified in five quality categories, instead of four risk categories.

As an example, let us assume Spectata's unit tests have the following distribution for maintainability: 5 stars: 10%, 4 stars: 19%, 3 stars: 39%, 2 stars: 30%, 1 star: 2%. According to the thresholds set in table 3.5 the project receives an overall maintainability rating of 3 stars.

The thresholds are key to this mechanism and should be based upon a large number of projects. Since we do not have access at present to such a large and varied number of projects we leave this open for future work. For now, the thresholds we set are based upon the distributions of the projects we also calculated our local quality scores over. We know these projects to be well tested with some exceptions and generally form an example of how unit testing should be done. This leads us to try to rate them with a 4 or 5 star overall quality score for each category. The thresholds were chosen so that in order to reach a 5 star rating they match the distribution we set in calculating the local quality scores. The lower scores have the same distribution, but are shifted so that they are easier to achieve.

3.5 From Quality Score to Advice

Now that we have calculated local and global quality characteristics we can provide feedback for helping development teams improve unit testing. Our approach is to provide them with factual data for test classes with low quality scores.

3.5.1 Improving Local Quality

The verdict of local quality characteristics can be directly traced back to the metrics. For effectiveness and adequacy this is straightforward: a low score can be improved by increasing coverage or the mutation score.

For maintainability this is slightly more complicated. Because of the choice for the Random forest algorithm for combining the metrics, a higher or lower metric value does not necessarily mean a higher or lower score. However table 3.4 already shows that there is a significant correlation between each of the metrics and change proneness. In table 3.6 we

3. PROPOSED APPROACH TO ASSURING UNIT TESTING QUALITY

Metric	Rho	P-value
Class Length	-0.488	1.33e-41
Dead Code	-0.232	1.09e-09
Case Length	-0.196	3.02e-07
Coupling Lines	-0.151	8.09e-05
Asserts	-0.141	0.00025
Complexity	-0.138	0.000332
Duplication	-0.128	0.000849
Coupling Classes	-0.122	0.00158

Table 3.6: Maintainability metric correlations with quality categories

have performed the same correlation but this time with the maintainability quality categories instead of change proneness. The table shows a significant negative correlation between all of the metrics and maintainability, meaning that a higher metric value will generally result in a lower maintainability score. Because of these correlations we make the assumption that for the maintainability metrics a lower value is better and base our advice on this.

The table shows that the correlation between the maintainability score and the class length metric is strongest. This makes perfect sense as the maintainability score is calculated from change proneness and the correlation between class length and change proneness was also strongest (see table 3.4). The other two metrics in the three strongest correlated ones: dead code and case length follow the same trend. Like class length they are among the metrics with the strongest correlation between them and change proneness.

Spectata provides advice by simply stating that a metric is either fine, should be higher, or should be lower. A metric is fine if it is among the lowest 5% of the values on which the classification algorithm was trained. The last step in helping the development team improve testing quality is providing the issues that led to a metric value. For instance, for the coupling metric Spectata is able to provide the classes that are exercised during test execution.

3.5.2 Build Based Progress

One of the key features of Spectata is its ability to evaluate project testing quality over time by providing each build with a new quality verdict. Of course, because of the scoring approach any global scores can be compared to each other. However its quality comparison goes further than that. The tool is able to provide an in-depth overview of how and why quality changed between builds, as long as one build is an ancestor of the other.

In order to determine whether two builds have this relationship, they are compared by the complete list of changes (revisions) that were applied to them. A build b_a is an ancestor of the current build b_c if the following condition is met. Let C be the list of commits $cr_0 - cr_x$ for b_c , and A the list of commits $ar_0 - ar_x$ for b_a , both chronologically ordered. Then b_a is an ancestor of b_c if and only if A appears in C , $cr_0 = ar_0$ and $ar_x \neq cr_x$.

Spectata provides the list of unit test classes of which a metric or a score changed. For each quality characteristic it provides a list with unit test classes sorted by importance. The

classes with the largest absolute difference in score are preferred, e.g., a rating change of 5 to 3 stars is prioritized over a change from 3 to 2 stars.

Continuing the journey from project score to source code, Spectata evaluates the changes in individual metrics and what lead to those changes. When a class receives a lower score, the metrics that changed are highlighted and the individual issues are provided. For instance, if the maintainability of a class was 5 stars and is 4 stars in the new build, that change in score can be traced back to its root cause, like a new piece of duplicated code.

3.6 Assuring Testing Quality

We end this chapter by explaining how Spectata deals with the questions set out in the requirements.

PQ1 – What is the state of unit testing? We answer this question by providing a project’s adequacy, effectiveness and maintainability scores. The scores combined give an impression of whether the current testing effort is sufficient for providing confidence in the quality of the software.

PQ2 – What type of changes should be made to the test suite? This question Spectata answers globally and locally. By providing three global quality characteristics the tool states in what aspects the testing effort was, or was not sufficient. The local quality feedback tells a developer exactly what is wrong when a unit test class receives a low score.

PQ3 – Where should the development team concentrate its testing efforts? In case the testing effort is not sufficient, we want developers to know where the problems lie, and be rewarded for fixing them. With the local scores Spectata provides a quick overview of the quality of individual unit test classes. Disregarding the best and emphasizing the worst test classes means developers get to focus on the real problems in their test code. Because of our global scoring approach, they are encouraged to improve the lowest rated areas first in order to see an increase in overall testing quality.

PQ4 – How did the latest changes affect testing quality? The last question is answered by the build comparison ability of Spectata. First, each build receives a global quality score that allows a project’s testing quality to be measured over time. Subsequently, the build comparison allows for an in depth view of the change in testing quality.

Chapter 4

Implementation: The Spectata Tool

In this chapter we describe the implementation of Spectata. The first section explains the tool's architecture and its individual components. The next sections describe the technical details of how the tool can be integrated into a software delivery workflow.

4.1 Implementation Details

An overview of Spectata's components is shown in fig. 4.1. All of the components, shown as rectangles, are separate Docker containers¹ responsible for one aspect of calculating testing quality. The figure also displays the core technologies that are used for each of the containers.

A container can be seen as a packaged application with everything needed for execution: code, runtime, system tools and system libraries. Similar to a virtual machine with a single application in it, a container provides the ability to isolate the application in a known environment. Unlike virtual machines however, containers do not emulate hardware. Instead, they share the host computer's RAM, CPUs, and kernel providing a more lightweight isolation [2].

4.1.1 Execution

Calculating unit testing quality starts by collecting data about the test suite. Each time a project triggers a Spectata analysis, an execution container is spun up. The container executes the test suite and runs tools that both statically and dynamically analyze the code base.

Using a mix of Python and Bash scripts, the container runs the tools and analyzes their results. Table 4.1 lists the information that each of the tools provides and how they relate to testing quality. To illustrate: the execution container executes PMD to detect the complexity of each method in the test code. PMD generates a report in XML format [23], similar to fig. 2.5. A Python script then parses the complexities in the report and sends them to the Spectata API.

¹<https://www.docker.com/>

4. IMPLEMENTATION: THE SPECTATA TOOL

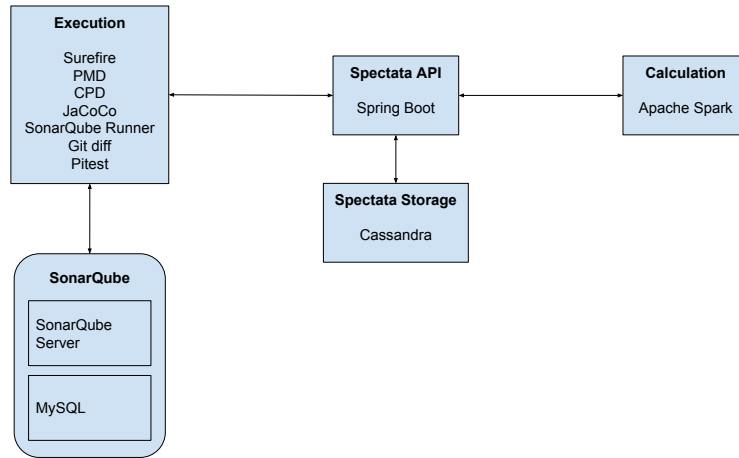


Figure 4.1: Overview of Spectata's architecture

Tool	Data	Used for
CPD	Clones in test suite	Duplication metric
PMD	JUnit violations	JUnit Conventions metric
	Number of asserts	Assert metric
	Method complexity	Complexity metric
Surefire	Test case detection	Normalization of metric values
Git diff	Changes since parent build	Change detection
SonarQube	Non commented lines of code per test class	Size metrics
SonarQube & JaCoCo	Exercised production lines per test case	Coupling metrics
	Exercised test lines per test case	Dead code metric
	Coverage percentage per production file	Coverage metrics
Pitest	Mutation results	Mutation score

Table 4.1: An overview of the data collected by the execution container and how it is used

4.1.2 Storage

Execution containers send their results to the Spectata back-end. Up to 200,000 pieces of information are collected per build, sent to the Spectata API and stored in the database. For this version of the tool we have implemented the database as a single node Cassandra cluster. It should be noted however that making the database highly available is as simple as adding more nodes, or switching to an already highly available cluster. The storage node is the only component with state, any of the other containers can be reset without consequence. An overview of the schema used for the Cassandra database can be found in appendix B.

4.1.3 Analysis

The Spectata calculation container is responsible for the most critical stage in the quality calculation: the analysis. In this stage the metric values, scores and changes are all calculated,

based on the data that was collected by the execution container. It is implemented as an Apache Spark² job that is run on a computation cluster. The job stores its results back in the Spectata back-end, awaiting retrieval by other tools via the API.

Because Spectata uses a NoSQL database, it is not desirable to let the API perform any kind of join operation to form a result. Instead, when sending data to the back-end, the calculation provides the data in a retrieval ready fashion. For information such as scores for a certain build, this is not a problem. However, for the comparison between builds we have had to devise a mechanism that is able to select changed *data items*³ between any two builds that have an ancestor relationship.

The mechanism works as follows. Data items are assigned a unique ID that lets them be traced throughout their life. If a data item ID is not present in the ancestor build, and is in the current build then it is newly introduced. During analysis, Spectata combines all data items that were present in the build's direct ancestor with the changes in the new build. The old items are updated with the changes to account for added and removed lines. Then the updated items are compared with all the items detected in the current build. Any data items in the new build that also exist in the direct ancestor build are assigned the old ID.

4.2 Integration With a CI Pipeline

Spectata is designed as just another piece of a CI Pipeline, to be used in any agile environment. Ideally, it creates a report on testing quality for every meaningful change in the software. Because a build server is used in every CI pipeline and it already executes on meaningful changes, this is the place Spectata initiates execution from.

Once the build server has successfully created a new build, it creates a new instance of the Spectata execution container and launches it on a Docker host. The execution container runs separately from the build server, and from this point on, nothing the execution container does can influence the build. Once Spectata finishes execution, collection, and analysis for this specific build, the results are published through its external API (see section below).

Missing in the tool is a front-end. Spectata provides the information a development team might want, but leaves the presentation of it to future work. We realize unit testing quality is only a small part of product quality and a development team will want a broader view when reviewing their software.

Within the ING context, we have investigated the possibility of presenting the information provided by Spectata. Figure 4.2 shows how the new information might be integrated with BugOut, combining quality of testing and test results in one source.

4.3 The Spectata External API

The key API calls that provide the unit test quality information are described in this section.

²<http://spark.apache.org/>

³ In this context, with data items we mean the data upon which the metrics are based, e.g., the detected clones, or the complexity of the methods.

4. IMPLEMENTATION: THE SPECTATA TOOL

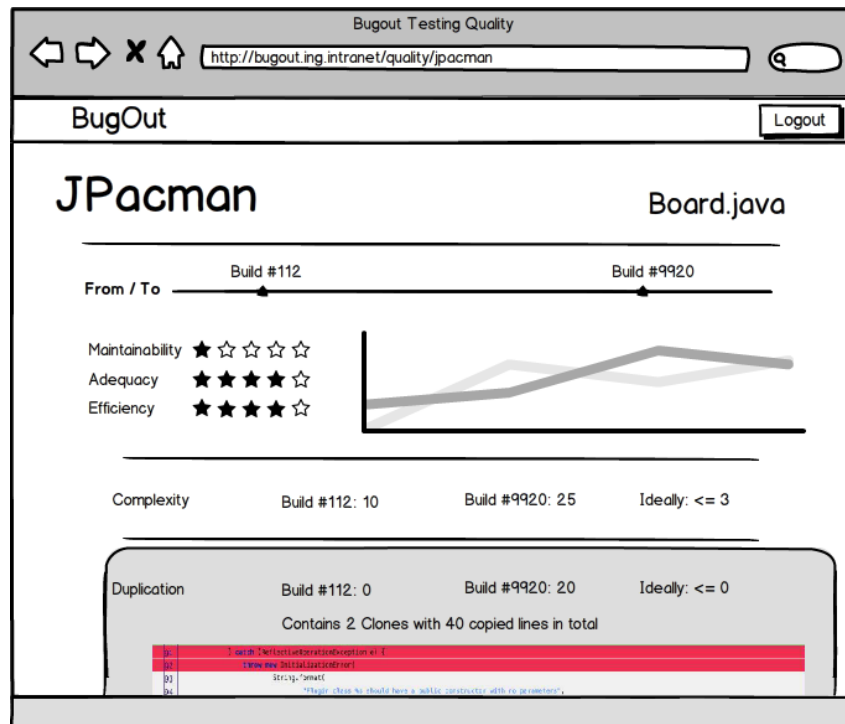


Figure 4.2: Mockup of BugOut with testing quality aspect

4.3.1 Project Score

```
/api/score/builds
```

Retrieves maintainability, adequacy and effectiveness scores for a project at a build.

Parameters:

- [Optional] project_id: ID of the project for which the builds should be retrieved
- [Optional] build_id: ID of a specific build

4.3.2 Test Class Score

```
/api/score/file
```

Retrieves the quality scores for test classes.

Parameters:

- [Required] build_ids: List of build IDs
- [Optional] file: Path of the file

4.3.3 Test Class Metrics

`/api/metric`

Endpoint for the metric values for test classes. It returns both the normalized and actual value of the metric, along with the references to the individual data items that affect it.

Parameters:

- [Required] build_ids: List of build IDs for which to retrieve the test class scores
- [Optional] file: Path of the file

4.3.4 Test Class Data Items

`/api/item`

Returns the data that forms the basis for the metrics, as explained in section 3.2. An example is the complexity of a class, or the clones that it contains.

Parameters:

- [Required] build_ids: List of build ids for which to retrieve the test class scores
- [Optional] file: Path of the file
- [Optional] metric: Metric related to the data items

4.3.5 Comparing Test Quality

`/api/compare/[file|metric]`

Presents a comparison for the quality of a class, or the metrics in that class between two builds. In case of a metric comparison, the result includes any changed data, such as a new clone in the test class.

Parameters:

- [Required] start: Build id of the ancestor build
- [Required] end: Build id of the new build
- [Required] file: Path of the file, only required in case of metric comparison

Chapter 5

Evaluation

The goal of this thesis is to be able to provide both developers as well as managers insight into the quality of unit testing of a software product in an agile environment. In this chapter we evaluate to what extent our approach meets this goal using the case study methodology [99, 81]. In four case studies, Spectata analyzes the unit test code quality of two widely used open source projects, one student project and one ING project. In each of the case studies the ratings provided by Spectata are compared to how we would judge the quality of the project, its individual unit tests and how quality changed after a modification was made. For the ING and student projects we also include the opinion of the developers in our evaluation.

5.1 Study Design

In the evaluation we seek to answer how the Spectata approach helps to answer the questions PQ1 – PQ4 as formulated in the requirements. This leads to the following research questions:

- RQ1: How does Spectata help to assess the state of unit testing (PQ1)?
- RQ2: How does Spectata help to identify the type of changes that will improve the quality of the unit test suite (PQ2)?
- RQ3: How does Spectata help developers locate the most problematic areas in the unit test suite regarding testing quality (PQ3)?
- RQ4: How does Spectata help assess the change in quality that comes with a change to the software (PQ4)?

The above questions focus on finding out *how* our approach helps assure the quality of unit testing in an agile context. In order to answer them we use the case study research method, described by Runeson and Höst [82]. We use this method as it is particularly suited for answering *how* questions and allows an exploratory evaluation of our approach.

The research questions are answered by applying the Spectata approach to assuring unit testing quality on a set of projects. This set contains two mature and stable open source projects created by experienced developers. It contains an open source project created and

used by students. Lastly, it contains an actively maintained and used project from industry. Section 5.1.1 describes these projects in more detail along with a manual assessment of the quality of their unit test suite.

The evaluation of our approach continues by applying Spectata's change assessment on a set of changes to answer RQ4. This set contains a large test suite refactoring effort and the changes that were made between a beta and the following release, explained in more detail in section 5.1.2. In addition to these changes, we follow Spectata's advice on two occasions and include changes made in an effort to improve the adequacy of a test suite and the maintainability of another test suite.

After Spectata has analyzed the projects we collect the ratings and recommendations it made. We compare the information it provides with how we ourselves and the developers would answer PQ1 – PQ4.

Some of the projects used in the evaluation are also part of the benchmark used to train the scoring algorithms. Because it only contains four projects this might skew the results. To let Spectata still form an unbiased opinion we made sure to temporarily remove any data from the project it was calculating a score over.

5.1.1 Selected Projects

Devhub

The first project under study is a software system used by the TU Delft called Devhub¹. It is designed to give students an introduction into software development by providing them with a CI pipeline. The system lets students set up their own source code repositories, CI server, and issue tracking system.

The system consists of several components, but the one we will be using for evaluation is the project's back-end. It is a Java Maven project that provides a REST API for storing and retrieving project information. It has 157 Java classes with 12K lines of code and is tested by 34 unit test classes.

This part of the system was created by two students a little over two years ago and has been actively maintained since then. All components of the system are open source, and students that use it are encouraged to submit pull requests on bugs they find or features they find lacking. The workflow of feature development is similar to what the project is meant to teach: modern and taking full advantage of an issue tracker and a CI server. A new feature is developed in a separate branch, tested locally and merged into master via a pull request, after all checks show a green light.

Recent development efforts have added the first metric tracking system in the form of Coveralls.io². The coverage information it provides is actively used when applying changes to the system as demonstrated by the comments in the pull requests. After the build status it is the most important check for deciding whether to merge a pull request or not.

Together with the current development team we discussed the quality of unit testing of the project and came to the conclusions shown in table 5.1. Due to the nature of the project,

¹<https://github.com/devhub-tud/devhub>

²<https://coveralls.io/>

testing is largely done by integration tests and thus not all of the modules are covered by unit tests. Unit tests alone cover about a quarter of the project while both unit and integration tests cover over 55%. However, even with the integration tests, adequacy is seen as fairly poor. The code itself is simple and well written, for a large part thanks to the simple nature of the project and the libraries used. Mostly because of the small size of the unit test classes we consider the code to be very maintainable.

JUnit

The second project we use for our evaluation is JUnit, specifically JUnit 4³. One of the oldest xUnit suites requires little introduction for any Java developer, but a detailed description can be found on their website [5]. It aims to make testing easier by providing a simple framework for writing unit tests.

The project was under development for over 15 years, but at the end of 2015 the developers decided to stop in favor of JUnit 5. For our evaluation we will focus on the latest release of JUnit 4: 4.12. At this point the project consisted of 9.3K lines of production code that were tested by 869 test cases in 129 test classes. Unlike Devhub the test suite is comprised completely out of unit tests executed by Surefire, it has no tests that we consider integration tests.

Recent development on the project is done by a relatively large group of developers that all follow a standard workflow⁴. Before submitting a new change, the developers execute the complete test suite locally in order to verify their change. The change is merged into master by submitting a pull request and letting the CI server verify the software an additional time.

Even though the developers do not use a centralized way of collecting metrics (i.e. SonarQube or Coveralls), we consider the production code of the project to be excellent. The code is well maintained, easily understood, and created with testability in mind. Because of this last feature effective testing is relatively easy to achieve and we rate this aspect with a score of 5 stars.

Like effectiveness we consider the adequacy of the test suite to be excellent. The project has an overall coverage of 85% and pull requests show that developers actively update tests alongside of production code. A 2012 blogpost [1] shows that most of the missing coverage is due to deprecated code. Another part can be explained because it is never executed by design, trivial or unlikely to be used in production. Thus, even though it is still 15% short of perfect, we consider achieving significantly higher an exercise in futility.

There is a bit of a trade off to be made between adequacy, effectiveness and maintainability. In most cases the more effective/adequate the test suite the longer and harder to understand the test cases. However we believe that even though the test suite contains 16K lines of code it is maintainable and deserves a perfect score in this aspect. Most test cases are short, concise and test a single aspect of the module under test. Because of this we consider the test code to be well written and understandable.

³<https://github.com/junit-team/junit4>

⁴<https://github.com/junit-team/junit4/blob/master/CONTRIBUTING.md>

Joda-Time

Before 2015, the standard Java libraries left much to be desired in their ability to handle dates and times. The de facto standard for dates and times therefore became a third party library called Joda-Time. The library became so widely used and accepted that it was decided to incorporate nearly all of its functionality as part of Java 8's STL.

In Joda-Time at the release of version 2.9.4 we find an incredibly stable project, even more so than JUnit. The majority of the work was done over 10 years ago but the project has since been actively maintained by its owner. With over three quarters of the commits done by him, development is hardly a team effort which makes the project stand out amongst open source projects and all the more interesting to us. There is no clearly defined way to contribute. The project's owner simply commits to master while everybody else is advised to submit a pull request.

Over the years the project has grown to 84K lines of code, with 56K lines of those being test code. Like JUnit, the test suite only contains unit tests executed by Surefire, which means Spectata takes all of the tests into consideration. In total the suite contains 4,114 test cases in 125 test classes.

The combination of the lack of team effort and sheer size of the test suite are reason to suspect a hard-to-maintain test suite. After even a short glimpse of the code it becomes clear that this aspect of testing has rarely received the attention we think it deserves. Test classes longer than a 1000 lines are no exceptions and quite often contain duplicates or similar logic. The test cases themselves are long, tend to verify more than one property and contain if-else and try-catch blocks. To illustrate this problem we have included a test case from the Joda-Time unit test suite, shown in fig. 5.1.

Even though the production code is not necessarily very testable, the developers have made sure the tests exercise the code to an excellent degree. The relative size of the test suite compared to the rest of the system provides an indication of the lengths they went through to make sure that the system was adequately tested. In addition, the developers have been meticulous in asserting that each part of a module behaves as specified. Therefore it is our opinion that both adequacy and effectiveness of the unit test suite should receive a perfect score.

Because Joda-Time's test suite lack in maintainability we expect Spectata to recommend improving that aspect. The list of unit tests that it will provide should contain the very worst classes in the test suite. However as explained in section 3.4 the local scoring algorithm for maintainability is not as straightforward as that for effectiveness and adequacy. Where we can say for sure that a class with low statement coverage will score low in adequacy. We cannot provide the same guarantee for maintainability and any of its metrics due to the way they are combined. For this reason we verify the local scores for maintainability with extra care. We manually inspected each unit test class and compared the recommendations we would make with those made by Spectata.

The process started with opening each one of the unit test classes identified by Spectata in an IDE. Without understanding the test class itself or even looking at its production counterpart we checked for: clones (as identified by the IDE), length, test cases, imports and complexity. Based on these indicators we made an educated guess on the maintainability

```

public void testRemoveIntervalConverterSecurity() {
    if (OLD_JDK) {
        return;
    }
    try {
        Policy.setPolicy(RESTRICT);
        System.setSecurityManager(new SecurityManager());
        ConverterManager.getInstance().removeIntervalConverter(
            StringConverter.INSTANCE);
        fail();
    } catch (SecurityException ex) {
        // ok
    } finally {
        System.setSecurityManager(null);
        Policy.setPolicy(ALLOW);
    }
    assertEquals(INTERVAL_SIZE, ConverterManager.getInstance().
        getIntervalConverters().length);
}

```

Figure 5.1: A test case in the Joda Time unit test suite

of the class and noted its score. In order to treat every class equally, we kept the process of searching for, reading, scoring and taking notes under a minute for each one.

The process described above resulted in 11 unit test classes that we gave a 1 star score and would recommend for immediate refactoring.

ING project

The last project we use for evaluating Spectata is an internal ING software component. It is part of a Java back-end application and used in a production environment. The back-end is an important component in a dynamic ecosystem with frequent changes.. At the moment, eight teams use the back-end as part of their product and ten developers have made changes to the software in the last six months.

The development teams actively use the CI pipeline and the SonarQube instance that comes with it as part of their agile way of work (see chapter 2). The developers pay some attention to the metrics provided by SonarQube, as attested by a low level of technical debt and 0% duplicated production code. SonarQube has been configured with a maximum number of critical issues it is allowed to have without triggering a build failure.

The workflow used for integrating changes is very similar to that seen in open source projects. They are developed in separate branches and integrated into the main branch using a pull request, after a code review by at least one other developer. Every two weeks, at the end of every sprint, the code in this branch is then put into production.

In total the project consists of 12K lines of production code and 16K lines of unit test code. The developers strictly follow the JUnit naming conventions and because of this we were able to match 96 out of a 100 test classes with a production class. The effects of this kind of discipline can be seen throughout the test suite: classes are short and understandable, it is immediately clear what a case tests, and even though the code base is quite old, it is kept

up to date. Because of this we expect the system to receive the maximum maintainability score.

Both the developers and we ourselves also expect adequacy and effectiveness to score this high. Even though the project is part of a back-end with a suite of integration tests, unit testing is seen as essential to product quality. The development teams are meticulous in covering production code and require changes to be tested thoroughly.

5.1.2 Selected Changes

The sections below describe the context of changes used in evaluating Spectata. In addition to the changes described in the following sections, the evaluation includes two refactoring efforts done by ourselves, but these are not discussed here.

Devhub Test Refactoring

The first change we discuss is a large test refactoring effort for Devhub. The new Coveralls results made it abundantly clear that coverage of the test suite in general was relatively poor. For this reason a team of students, headed up by one of the original creators of Devhub, made an effort to improve the quality of testing of the project, amongst other tasks. The improvements were made as part of an alternative assignment for four first year computer science students.

In a little over a month the students made around 250 commits that changed 51 test files to the Devhub back-end. During this time the main metric for measuring progress: overall coverage, went from 43% to 56%. While most of this effort went into adding integration tests, the students did see unit tests as important to testing quality. As a result 19 of the 32 unit test files were either added or changed.

Because of the limited number of changes in the unit test suite we do not expect a change in global project scores, negative or positive. For the local scores we expect to see no decrease in quality in any aspect for any file and thus no recommended additional changes. The newly added production and testing classes are a different story. These files are engineered completely during this refactoring effort and anything short of perfection should provoke a recommendation. Because of the limited experience the developers have we expect to see most of the newly added files to be imperfect. Therefore all of the recommended changes provided by Spectata should be for newly added classes.

JUnit Beta to Release

The most uneventful of change comparisons is that of the JUnit 4.12 second beta to the release of 4.12. Even though 42 commits were made in between these two points, nothing really changed because they were all fixes to tiny imperfections in the software. Nonetheless, because of the relevance of a beta release to a project we think it is important to consider. Spectata should reflect that nothing major changed in all aspects of testing. False positives are as important to avoid as false negatives and comparing these two points in a project's life cycle allows us to evaluate exactly that.

Project		Quality Aspect		
		Adequacy	Effective-ness	Maintainability
Devhub	Expected	1	-	4
	Spectata verdict	1	-	5
JUnit	Expected	5	5	5
	Spectata verdict	4	5	5
Joda-Time	Expected	5	5	1
	Spectata verdict	5	5	1
ING project	Expected	5	5	5
	Spectata verdict	5	4	5

Table 5.1: Global quality scores of the selected projects

The 42 commits changed five test files and six production files. The diff generated by Spectata shows that most of the changed lines were made in the test files. Because of this we do not expect to see any changes in adequacy or effectiveness whatsoever. The maintainability of four of the changed test files at the point of the beta was perfect, and we do not expect changes in this regard either.

5.2 Experiment Result

5.2.1 RQ1 – How does Spectata help to assess the state of unit testing?

The first requirement states that Spectata should give developers and managers insight into the current quality of unit testing of a project. It does this by providing a score for each of the quality aspects on project level. We evaluate the scores by comparing them to our own opinion and that of the developers, summarized in section 5.1.1. Both the ratings we expect and the ratings provided by Spectata are shown in table 5.1.

Devhub

The first thing we notice when evaluating Devhub is that it is missing an effectiveness score. The software is highly dependent on a database connection and supporting libraries which were not always properly stubbed for testing. Because of this the mutation analysis caused deadlocks and thus Spectata was not able to calculate an effectiveness score. Although not an actual score this result does highlight one of the down sides of Spectata: its dependence on mutation analysis results in a more fragile product.

Spectata gives Devhub’s testing adequacy the lowest possible score which matches our opinion. As shown in fig. 5.2a nearly 40% of the test classes received a quality score of one. This alone is enough to justify the score according to Spectata’s scoring algorithm. According to the developers there are two reasons for having such a high percentage of the production code so poorly tested. (1) A significant portion of the code is an API and according to the developers “simple enough to not need any testing”. (2) Because of the difficulty in testing a

REST API with unit tests, development efforts have focused on integration tests over unit tests. However, a closer inspection (see also section 5.2.3) yields that even some of the very testable business logic is barely exercised by unit tests.

With a maintainability score of 5 stars improving adequacy should be relatively easy. The score comes close to our own rating of four but deviates slightly due to a very high number of unit test classes with a score of 5 stars. The apparent lack of mocks and stubs does not cause the score to drop like expected.

JUnit

Table 5.1 shows that Spectata's score is mostly aligned with our opinion of a high quality test suite, all aspects received a 4 or 5 star score. The predicted effectiveness and maintainability scores match Spectata's judgment, while the adequacy is only one category lower than we expected.

Because of the number of files with a perfect mutation score, the distribution of local effectiveness is heavily skewed towards a 5 star score. This clearly reflects our opinion that because JUnit's production classes are well engineered, there are relatively little opportunities to apply mutators. Consequently it becomes much easier to cover all mutations and receive a 100% mutation score. Even though such a heavily skewed distribution is not one we desired, we can draw no other conclusion than that JUnit is tested extremely efficiently and Spectata's judgment is correct in this regard.

Somewhat surprisingly, the calculated adequacy score is one point off of our own score. Figure 5.2b shows that, again, the distribution of local quality scores is skewed towards five stars. For adequacy however, 7% of the files fall within the one star category. It appears that 10 production classes are largely missed during unit testing. Even though the vast majority of classes has an excellent coverage, these relatively few poorly tested classes immediately cause a drop in score. A closer inspection reveals several reasons for not, or not properly testing the classes, listed below.

- `Version` appears to be used for packaging
- `FailedBefore` and `CouldNotReadCoreException` are trivial
- `JUnitMatchers`, `MethodRoadie`, `BaseTestRunner`, `TestRunner` and `TypeSafeMatcher` are (largely) deprecated
- `Assume` and `Throwables` seem to have no discernible reason for a poor structural adequacy

For a large part the classes Spectata found match an earlier effort [1]. In itself this is a good thing, it is able to find the classes an expert would find as well. However this does mean that we have to consider its conclusion wrong. JUnit's test suite is excellent and most of the classes that were poorly tested have a valid reason for being so. Other than taking deprecated classes into account, this miss with expectations reveals one more weak point in Spectata's scoring mechanism. Small classes, even trivial ones, are weighed disproportionately in

Class	Line Coverage	Branch Coverage	Presumed reason
DelegatedDurationField	0	0	Data class
JodaTimePermission	0		Trivial
DateTimePrinterInternalPrinter	13	25	
BasicSingleEraDateTimeField	38	0	
ISOYearOfEraDateTimeField	40.9	50	
GJCacheKey	66.7	41.7	
RemainderDateTimeField	57.1	30	
StrictDateTimeField	30.8	50	Trivial
ZeroIsMaxDateTimeField	38.9	66.7	
DefaultNameProvider	60	40.3	Hard to test

Table 5.2: Joda-time classes with low coverage

calculating a global score. If the tool had used coverage as a global average instead, it would most likely have awarded JUnit a perfect score.

The maintainability aspect of the JUnit test suite is something both Spectata and manual inspection found to deserve a perfect score.

Joda-Time

As table 5.1 shows, Joda-Time’s quality scores calculated by Spectata match our opinion to a large degree. The table does however show some small differences with regard to adequacy and maintainability.

Just like with JUnit, the project’s adequacy scores four instead of the expected five stars. And just like with JUnit, this is caused by too many classes receiving the lowest possible adequacy rating. The similarity actually extends beyond this, in that the project exceeds the allowed 5% only by a very small margin. There is definitely a difference between the two however, for JUnit we found most of the low scoring classes to have a valid reason for it. For Joda-Time most of them seem to have no obvious reason for poor coverage (see also table 5.2).

The distribution of effectiveness and maintainability local scores are as expected. With 5 stars and 1 star respectively, the global scores match our opinion of a thorough test suite that cares little about the presence of test smells.

ING project

As expected the ING project scores very highly in both maintainability and adequacy. Looking at the distributions in fig. 5.2d we see that especially the latter is justified with close to 60% of the classes having a 5 star score.

For maintainability the case is slightly different. With only 5% of the files rated the highest score it barely meets the criteria for the maximum project maintainability score. The large number of 4 star rated unit test classes reflect the effort the developers put into keeping

5. EVALUATION

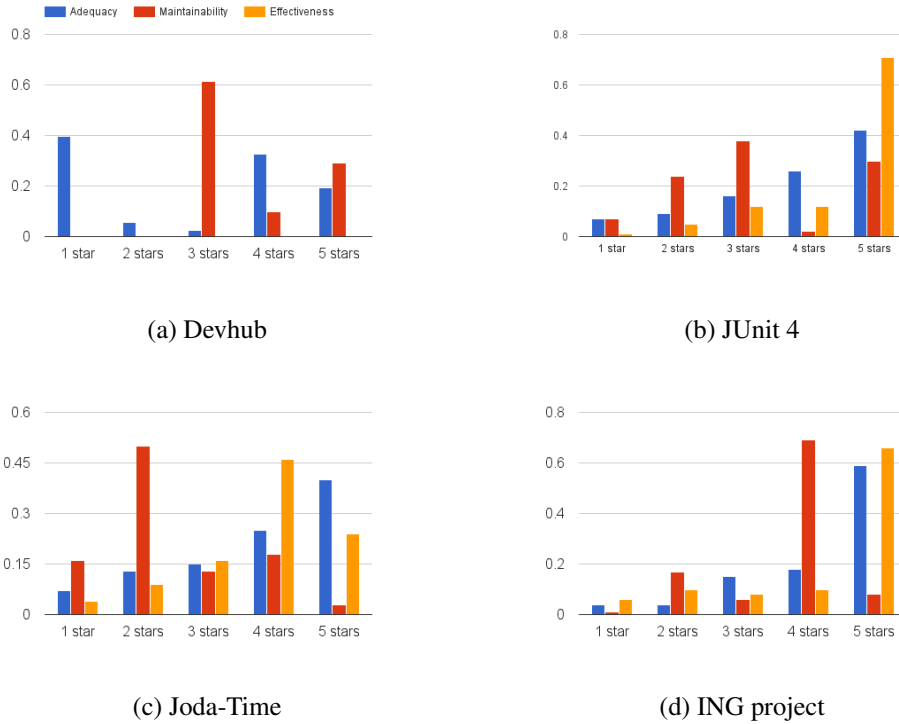


Figure 5.2: Per quality aspect the relative number of files with a particular score, shown for each of the evaluated projects.

the test suite maintainable. The overall score shows that even with a large test suite and a complex piece of production code Spectata agrees with our opinion on maintainability.

The effectiveness score is slightly misaligned with our opinion, due to too many classes with a 1 star rating. A closer inspection with a developer shows that they are two controllers and one configuration class with dedicated unit test classes. The other is a data class without much need for harsh testing. Even though the distribution would dictate a maximum score, in our opinion a 4 star score is not necessarily wrong. The findings are no false positives and Spectata is right to point out that this aspect of unit testing is not perfect.

Conclusion

In the above sections we evaluated how Spectata is able to help assess the quality of testing for four software projects. By design this question is answered by providing the state of the unit test suite on three aspects of testing: structural adequacy, fault finding effectiveness and maintainability. We compared the information provided by the tool with our own opinion of unit testing quality on each of the aspects. Table 5.1 summarizes the result and shows that in most cases our opinion matches Spectata's verdict. In only three cases the expected and calculated values are different and in those Spectata's score is only off by one.

Although four data points are too few for us to confidently say that Spectata's quality

model for projects is accurate, the case studies do provide an indication that it gives an answer to PQ1 that is very similar to one we would provide.

The case studies also revealed some of Spectata's limitations, these are listed below.

- An effectiveness score based on mutation analysis is not possible for every project.
- Deprecated classes can skew the results, a way to disregard these kinds of findings might be useful.
- The size of a class is not taken into account when calculating the project's quality score.

5.2.2 RQ2 – How does Spectata help to identify the type of changes that will improve the quality of the unit test suite?

With a global score of one out of 5 stars Spectata clearly indicates that adequacy should be improved for the Devhub project. This matches both our opinion and that of the developers. The report provides 61 classes with an adequacy score of 1 star, 59 of which do not have a dedicated unit testing class at all.

Unlike Devhub, Spectata's global scores for JUnit and the ING project show that these do not require immediate changes to their unit test suites. There are definitely improvements possible concerning adequacy and effectiveness but the number of 1 star rated classes is very low. The advice is clear and matches our view of an excellent test suite that extensively tests the product.

For Joda-Time the information provided by Spectata clearly reads improving maintainability is the way to go. The project received the lowest possible score for maintainability versus two near perfect scores for adequacy and effectiveness. However improving Joda-Time's maintainability is not as simple as improving the other two. Where a low adequacy means changes should be aimed at improving a type of coverage, low maintainability might mean the presence of any number of smells. Because of this, we cannot tell exactly what type of changes are needed based on the project's score, but need to look at local scores and metrics instead.

Whether the advice to improve maintainability is useful to the developers of Joda-Time is doubtful at best. Given the state of the product they would likely ignore it. The fact remains though that without context the advice is correct. The tool leaves the decision to act on the advice up to the development team.

Conclusion

Spectata helps to select the types of changes to make by providing comparable scores on three different aspects of testing. This results in an advice that not only suggests on what aspect a test suite is lacking, but also to what extent it is. In all of the four evaluated projects the advice matches our own opinion. Not all of it is useful however, as the Joda-Time advice shows it very much depends on the context of the project whether the development teams will act on it.

5.2.3 RQ3 – How does Spectata help developers locate the most problematic areas in the unit test suite regarding testing quality?

Devhub

As stated in the previous section, Spectata recommends the testing effort for Devhub to focus on improving adequacy. The tool prioritizes the classes that were not exercised at all with a 1 star rating. The complete list with these classes is shown in appendix C, categorized by their purpose.

Upon inspection we find 21 resource and template classes that are responsible for correctly handling HTTP requests. Although these classes can be tested with unit tests, it is often more useful to do so with integration tests. In a discussion with the developers we do indeed gather that these are at least partially tested by integration tests.

Five of the classes with the lowest possible adequacy score are data classes: they contain no logic except for generated getters and setters. Due to their nature it is understandable that these do not have dedicated unit test classes. They are generally exercised when testing the business logic of an application and automatically get a decent coverage. Because the business logic is not properly exercised this causes the Devhub project to receive an adequacy penalty twice

The remaining 27 classes are in our opinion perfectly testable and require relatively little testing effort.

JUnit

According to Spectata's judgment JUnit could benefit from improvements in adequacy. Because of the way we implemented global scoring, fixing the worst classes first is usually preferable in order to receive a higher global score. This is reflected in the improvement recommendations made for JUnit. For adequacy, 10 classes with poor coverage are listed as the most important possible refactorings. The classes, and the metrics that caused the poor score are listed in table 5.3. As explained in section 5.2.1 most of these recommendations are actually false positives due to deprecated code, trivial classes and unreachable code.

Joda-Time

Spectata's advice for improving Joda-Time maintainability points us to 10 unit test classes with a 1 star score. Comparing these classes with the ones we would have proposed ourselves we see that there are two false positives and one false negative.

First of the misclassifications is `TestIslamicChronology`. Spectata gave it 2 stars instead of the 1 star score we proposed. With 740 lines of code the class is long, but its test cases are relatively simple. There are no if statements and the tests rarely use try-catch blocks. The class does contain an unusually high number of asserts per test case but these are mostly used to extensively test one feature and not indicative of any smell. Looking at the metrics, shown in appendix D, we see that values for coupling are high, but not extremely so. Upon further inspection we do see that there is an argument to be made for rating the class with a score of two instead of one.

Class	Line Coverage	Branch Coverage
BaseTestRunner.java	42.0	36.5
Version.java	20.0	
TestRunner.java	53.9	44.4
Assume.java	47.6	40.0
CouldNotReadCoreException.java	0.0	
Throwables.java	20.0	
TypeSafeMatcher.java	0.0	0.0
FailedBefore.java	0.0	
MethodRoadie.java	46.7	38.9
JUnitMatchers.java	9.1	

Table 5.3: JUnit's production classes with an adequacy score of 1

The second misclassification is `TestDateTimeZone`. With nearly 1100 lines the class is extremely long and for us enough to justify the lowest of maintainability scores. Spectata however judged differently, giving it a score of two. The class has 47 test cases and a closer look at the code yields that most of these are actually fine. As with the previous false negative the better written test cases compensate for the length of the class.

The last of the wrongly classified unit test classes is `TestDateTimeFormatterBuilder`, a false positive. We assigned it a score of 2 stars because most of its test methods are relatively short and mostly test a single aspect of the production class. Longer, complex methods prove the exception rather than the rule. The fact remains that the class overall is still quite long and a look at Spectata's metrics show that coupling is very high. On average 441 lines of production code are executed per test case, something we were not able to consider in our inspection. After seeing Spectata's metrics we tend to agree with its conclusion, reducing the total number of false classifications to two.

ING project

For the ING project Spectata recommends improving the fault finding effectiveness of the test suite. We have presented the selection of lowest rated classes to one of the senior developers in the project and agreed upon the fact that all of these classes should have been tested better.

It was explained that the found mutations did not necessarily represent missing test cases, but that they indicated a low testing effort. Still, there was a large desire to see where the survived mutations came from and either fix them or mark them as intentionally left open.

Conclusion

The sections above show that Spectata helps developers improve their test suite by recommending classes to refactor. For two of the projects adequacy was a priority and we started by evaluating recommendations in this aspect. We found that, due to various reasons, many of them were actually not valid. Additionally, the Devhub project shows that modules might be

tested in other stages of a CI pipeline. Because Spectata focuses on unit testing we consider this out of scope.

The maintainability recommendations on the other hand were surprisingly accurate. For one class the advice actually proved more accurate than an initial glance by us. Because Spectata measures the extent to which a smell is present it cannot tell us whether one actually is. Only by combining these into a score we can say something about a class' maintainability. With our results we can say that this model for maintainability matches our own intuitive measure of this aspect to a large extent.

The case studies led us to finding the following limitations of the tool.

- Good test cases in a class that has many may obfuscate some of the worse cases. As a result a class can be made to look better than it is by adding code.
- When considering coverage it might be better to exclude some units from the result. There are valid reasons for not testing data classes and deprecated methods and classes.
- Our goal when introducing a mutation score is not necessarily to make sure all mutations are killed, but to indicate ineffective testing. In practice however, developers might see killing each of the mutants as a goal and become frustrated with the effort.

5.2.4 RQ4 – How does Spectata help assess the change in quality that comes with a change to the software?

Devhub Test Refactoring

Based on information provided by the developers we know that the changes made to Devhub were mostly aimed at improving testing quality. Spectata's change report reflects this intention in that none of the existing scores got worse and five saw an increase in adequacy score. However, the changes made were not significant enough to increase testing adequacy over the project as a whole.

Table 5.4 shows the files that Spectata recommends should be improved along with the made changes, all of them are newly created. The unit test classes that the developers did change are rated 5 stars. This indicates that Spectata's adequacy scoring strategy does indeed reflect the extra testing effort that was put into the project.

As with adequacy, the project wide maintainability score did not change during refactoring. None of the existing test classes received a worse maintainability score than they had before this period. Out of the nine newly created unit test classes Spectata recommends changes on five due to them receiving a quality score of four.

Improving JUnit Adequacy

Along with coverage statistics Spectata provides us with detailed information on which lines were and were not covered. As part of this evaluation we used this information to improve the adequacy of the JUnit test suite. With the goal of achieving a 5 star project adequacy score we assumed the role of a developer and refactored both production and test code

Class name	New adequacy score
Version	1 star
PrivateIssueResource	1 star
ProjectIssueResource	1 star
StudyNumberResource	1 star
DutchGradingStrategy	1 star
AbstractProjectIssueResource	1 star
IssueComments	1 star
Issue.java	2 stars
ToHtmlSerializer	2 stars
AbstractIssueResource	2 stars
IssueComment	3 stars
Issues	4 stars
IssueBackend	4 stars

Table 5.4: Devhub’s changes in adequacy scores

using Spectata’s recommendations. We improved coverage on the recommended classes one-by-one until the project received a 5 star adequacy score.

The process started by inspecting whether there was a reason for not testing a low rated class. The result of this process is listed in section 5.2.1, which shows that four classes really have no valid reason for poor coverage. We decided to focus our efforts on the easiest classes first: `Throwables` and `Assume`.

`Throwables` is an internal utility class that does one very simple thing: rethrow exceptions. The class is so small that poor coverage was due to a private constructor, common practice in utility classes, and a method that rethrows an exception, after which it returns null, a statement that is of course never reached. Refactoring the last method so that it returns void avoids this and fixes the perceived adequacy problem.

Unlike `Throwables`, `Assume` is part of JUnit’s current public API. It contains a set of methods that allow a developer to make sure that certain conditions are met when the test is executed. If these conditions are not met, any failed asserts in the test will be ignored. The class has nine methods, four of which are not tested at all and two of which have a branch that is missed. All of the methods in the class are small and the four missed ones consist of only a single line each. The class’ adequacy is improved to a 4 star rating by adding two cases that test two of the untested methods.

Improving the coverage of the above two classes resulted in a perfect project adequacy score and ended our development effort. After each improvement we ran the complete unit test suite locally, checked in the code and received a quality verdict from Spectata about 15 minutes later. During this time we continued our refactoring effort on the next recommended class by following the same cycle. Overall the system worked together quite nicely with our incremental approach and we found the waiting time for receiving the quality verdict not to be a hurdle. Because the refactorings took less than 15 minutes each, we simply checked in new code before receiving the quality verdict for the old code without either job being

hindered by the other.

Setting a quality goal for developers to achieve is something that could very well elicit the wrong kind of behavior from them. Instead of spending effort on improving the system's quality, it is only logical for a developer to focus on *treating the metric* [22]. However by trying to improve the metrics we actually did end up improving the product as a whole. Additionally, after spending less than half an hour on refactoring test and production code, Spectata's adequacy score reflects our own opinion.

JUnit Beta to Release

As expected, the global scores for JUnit did not change when comparing the first beta version of the software to the actual release. In fact, none of the local scores for any file for any unit testing aspect was negatively affected. Both of these results match what we expected: the commits done in the time from beta to release never decrease testing quality.

Most of the classes that changed for the release received the same scores as they had had at the beta. For adequacy one file moved from a 1 star to a 2 star rating. The same file had its effectiveness move up from a 4 star to a 5 star rating. Lastly, none of the test files had any changes in score regarding maintainability.

Improving Joda-Time Maintainability

Because of the scale of Joda-Time's test suite maintainability issues we lowered the goals for our second refactoring attempt. Instead of refactoring until the project reaches a 5 star score we settled for the best score after an hour of work. Additionally, the test suite should be at least as effective and adequate as it was before.

Spectata recommends to improve modules based on their score, which meant that in this case there were 10 options to start working on. At random we decided to improve the maintainability of the unit test class `TestConverterManager`. In essence the class contains the same tests five times because its matching production class has the same methods five times. Altering the production class so that its methods can handle all types of `Converters` would avoid this problem. However the class is public and used in other projects, therefore we cannot change its public signature.

In the first stage of refactoring we followed Fowler's [39] advice for removing duplicated code. We generalized the logic and used the Extract Method and Extract Class treatments to place it in the `SingleConverterManager` class. After this the production class became little more than a facade with its original methods directly calling the extracted class. Instead of testing the logic five times over, it is now tested once by the `TestSingleConverterManager` unit test class. As a result we were able to reduce the number of unit tests in `TestConverterManager` to 25 and its length to 251 lines of executable code.

In the second stage of refactoring we focused purely on improving the test class itself. As the code fragment in fig. 5.1 illustrates, test cases in it are often long and unnecessarily complex. We applied the following steps to counter the found smells.

- Removed test case that asserted that the class under test was a singleton

Metric	Value before	Value intermediate	Value after	Median benchmark
Class Length	592	251	83	66
Case Length	11.3	10.9	3.9	11.6
Asserts	2.68	3.1	2.4	2.09
Complexity	1.63	1.4	0	0.55
Dead Code	3.13	2.88	0	0.4
Duplication	1.06	1.87	0	0
Coupling Classes	5.73	6.73	6.73	4.27
Coupling Lines	47.5	47.8	48.9	54.6

Table 5.5: `TestConverterManager` maintainability metrics before and after refactoring

- Used a fixture to reset `ConverterManager` before each test
- Removed conditional testing logic
- Removed unnecessary and duplicate asserts

After an hour of refactoring we find the test cases to be more concise, less complex and without any duplicated code. With 25 test cases, the class is still too long to truly belong with the best, but we cannot shrink it further unless we shrink the production class signature as well. Spectata's new verdict agrees with our opinion, rating it 3 stars while keeping the adequacy and coverage the same. The old and new metrics on which the decisions are based are shown in table 5.5.

Conclusion

One of the key features of Spectata is its ability to assess changes in a project. On a project level it provides comparable scores and on a local level it recommends improving files that have seen a decrease in quality. In order to get an indication of how the tool is able to evaluate changes we evaluated its verdicts and recommendations in four separate cases.

The cases show us that a project metric is not easily changed unless it happens to be right on the edge of a quality category. Even on a local level it takes significant effort to change the score. A move from beta to release resulted in only one class with a change in an aspect of quality. Our significant effort into improving Joda-Time maintainability lead to an increase of two stars.

All of the changes do show that extra effort put into unit testing does indeed improve the perceived quality of the unit test suite. Our own efforts into adequacy show that even a *treating the metric* approach to refactoring can positively impact testing quality.

Chapter 6

Related Work

In this chapter, previous work that aimed at evaluating testing quality is discussed. Each of the research efforts is described and quickly reflected upon. For each effort we paid special attention to the similarities with Spectata.

6.1 STREW and GERT

In his doctoral dissertation Nagappan [66] proposed the Software Testing and Reliability Early Warning (STREW) metric suite. STREW is a set of nine static metrics divided into three categories: test quantification metrics, complexity and object-orientation metrics and size adjustment metrics. The metrics are known to overlap and a Principal Component Analysis [53] extracts three components. These components are then used to predict post-release reliability.

The STREW metrics are validated in a case study where 22 groups of students developed an eclipse plugin. The calculated metrics for the projects were used for creating a regression model that predicts the number of failed black-box tests per 1000 LOC. The study reveals a strong significant correlation between the two. Next, the authors predicted the quality of the software (high or low) using a logistic regression model and found that 20 of the 22 programs were correctly classified.

The research effort continued with GERT: the Good Enough Reliability Tool [67, 29]. Based on the STREW metrics and coverage information it aims to provide developers with an early estimate of reliability. It does this by providing color coded information on the state of individual metrics. Based on projects with an acceptable number of problems in production relative to their size, thresholds are calculated for each metric. A metric is green if it is higher than the mean value of the benchmark set, red if it is below a calculated lower limit, and yellow if falls between the two.

In order to validate the tool, Davidsson et al. [29] choose to validate the color coding of the metrics. They created a model that relates the number of orange or red metrics to the number of defects relative to the size of the project. In a case study with seven academic, 13 open source, and six industrial projects they found a strong positive linear association between the color-coded feedback and fault proneness.

Missing in the evaluation is the performance of the tool as it is used by developers. GERT is specifically meant to be used continuously and from within the IDE but the authors do not evaluate its use in the field. As pointed out by Athanasiou [16], the STREW study has some limitations as well, most importantly that it is hard to generalize beyond the student projects. However we do consider it a basis for evaluating unit testing quality and it clearly shows the importance of size and complexity for software quality.

6.2 SIG

Athanasiou et al. [17] use the SIG model, as explained in section 2.4, for evaluating unit test quality of software systems. Using a Goal, Question, Metric approach [97] they define a set of metrics for measuring completeness, maintainability and effectiveness of a test suite. Some of the metrics, such as coverage, are calculated on a project level. Other metrics, such as complexity, are measured per unit and are transformed into a project wide score using the quality profiles method [15]. The authors then average the scores for the metrics to come to a project wide score for each aspect, which are again combined to form a project wide testing quality score.

An initial attempt to validate the quality score is made by correlating them with issue handling performance. The authors hypothesize that increased testing quality means more solved issues per month and a higher defect resolution speed. An experiment with 75 snapshots from 18 open source systems provides support for the first of these to be true. There exists a strong correlation between the calculated project quality and the number of issues solved per month.

A follow up case study evaluates to what extent the quality aspect scores align with expert opinions. Similar to the evaluation of Spectata, an expert rates a system on each of its testing quality aspects. These ratings are then compared with the score calculated by the testing quality model. Because of the limited number of data points there is no definite conclusion that can be drawn but the case study provides an indication that the quality model provides useful results.

The work proves to be a well founded and quite successful attempt at assessing test quality. The most important difference with Spectata's approach is that the model, and the SIG approach in general, is geared towards providing management with an impression of the quality of software in their portfolio. Spectata on the other hand is designed not just for management but also for developers to use during development. Spectata is designed so that it can provide feedback on problems on a file level, recommend files for refactoring and show how changes affected testing quality. Another limitation of the work is that the code in the projects is not executed. The coverage and effectiveness metrics used are approximations based on static code analysis.

6.3 TAIME

The Test Suite Assessment and Improvement Method (TAIME) aims to aid developers in refactoring their test suite [89, 92]. TAIME is a set of development practices aimed at

improving testing quality. It revolves around setting an improvement goal and a granularity level, and incrementally improving upon the test suite. The approach offers goals in the form of Code Coverage, Efficiency and Uniqueness, expressed as five metrics. It is up to the development team to interpret these metrics and decide on which to focus.

As an example a developer might find coverage of his library insufficient. In TAIME the COV metrics expresses coverage and he decides to first improve it on a procedural level. The TAIME tool provides this metric for all modules, packages and the project as a whole. With this information the developer starts improving the test suite. A second step in improving overall coverage might be improving the COV metric on a statement level, and the process is repeated until the improvement goal is reached.

The use of the method is demonstrated by refactoring the test suite of a tool of their own making. This case study demonstrates how it might be used while highlighting its benefits. In three iterations metrics for coverage and efficiency were increased by adding tests cases and changing production code. The authors omit what led them to focus on refactoring certain units and not others and leave a recommendation system for this up to future work.

In a follow up research effort [92] the authors provide an initial evaluation of the actual metrics by calculating them over an industrial scale open source project. During this process they reflect on what might be good and bad values for a metric and use them to provide recommendations for the project. The recommendations for the kind of refactoring needed are more specific than Spectata's but requires testing expertise to understand, let alone make.

The research is an ongoing effort into defining testing quality. The papers do not provide any information on the operational details, nor how developers reacted to being forced to use the method.

Like Spectata, TAIME combines a metric-based approach for evaluating testing quality with the intent of helping developers improve their test suite. However, where Spectata was made to be part of a CI workflow, TAIME defines its own approach.

Chapter 7

Discussion and Threats to Validity

In the previous chapters we have shown how Spectata is able to help assure unit testing quality. In this chapter we reflect on these findings and position them in a broader context. We discuss their implications, limitations and possible future work. In the last sections we present the threats to validity.

7.1 Interpretation of the Results

7.1.1 Continuous Delivery

One of the key features that sets Spectata apart from other tools is that it is specifically made to be integrated with an agile way of work. Even though we were not able to connect Spectata to a continuous delivery pipeline in production, the results in our evaluation are encouraging. As demonstrated by answering RQ4, Spectata reflects the changes in quality as a result of the constant changes in the software. The answers to RQ2 and RQ3 demonstrate that it provides actionable suggestions for a development team that wants to improve testing quality.

In addition to providing developer feedback, information provided by Spectata could help further automate product delivery. Before committing to a release, the development team needs to have confidence in the quality of all aspects of the product, including its unit test suite. In the best of cases this means manually assessing whether the test suite adequately exercises the production code and is indeed able to find faults. As demonstrated with the answers to RQ1 and RQ4, Spectata provides an assessment of the testing quality and how it changed since the previous build. These assessments allow a pipeline to set a minimum testing quality threshold and ignore the build if it does not meet it, automating a labor intensive and fault prone process.

7.1.2 Application within ING

The arguments made in the previous section directly apply to the development work at ING. The build server used for Java applications could easily be fitted to initiate a Spectata analysis at each build. In addition to providing development teams with an insight into the quality of

their own test suite, they will be to compare their suite to others. Because the project scores are independent of size and comparable, it will become possible to single out teams testing particularly well and share their findings with the rest of the development teams.

Spectata also makes the quality of unit test code quantifiable. Even though it is far from perfect, the quality verdict it provides can be used as a non-functional product requirement. This forces developers to pay attention to the test suite, and managers to take testing into account when budgeting and planning.

7.2 Application Limitations

The result of this work is an early version of Spectata, and as we found in the evaluation, it still has some very basic limitations. Each conclusion in the evaluation contains the limitations we encountered then. This section describes any that were missing and reiterates the more important ones.

First and foremost is that analysis only works on Java Maven projects. The execution stage uses a set of tools that is highly language dependent, and supporting a new setup means implementing a new execution container for it.

A second limitation is the inaccuracy of some of the metrics. Both the assert, and the dead code metrics are inaccurate because of the way they are calculated. Custom assertion methods are not detected, nor are asserts generated at compile time. This leads to an underestimation of the number of actual asserts and thus a lower value of the asserts metric. Because of JaCoCo any statements between throwing and catching the exception are seen as not executed, leading to an overestimation of the amount of dead code.

The mutation analysis used limits the tool both in performance and in the projects that it accepts. As we established in section 3.2.2 mutation testing can still take quite a long time to complete. Adding to this is the fact that due to database connections some software is unsuitable for mutation analysis, thus Spectata cannot calculate an effectiveness score on them.

Finally, we observed that when evaluating changes, the granularity of the scoring mechanism was sometimes too large. When comparing two versions of a project with only some minor changes, these changes are likely to cause no change in score. This leads to Spectata not recommending any additional testing effort, even though a metric like coverage might have decreased by some percentage points.

7.3 Future work

During the study several interesting topics for future work emerged, this section provides an overview.

Evaluating the tool in use. Due to delays within ING we were not able to roll out our product and have it used by developers, as originally anticipated. This will provide insight into how the tool is used and which features specifically are successful. The usage statistics would be able to quantitatively answer whether the tool is useful to developers.

Adding data points. With only four projects used in the evaluation the test data set is relatively small, limiting the external validity. Adding different kinds of projects with languages other than Java would enable a more generalizable conclusion.

Improving metric calculation. Some of the metrics are known to be inaccurate due to the way they are calculated. With an improved metric suite we can be more confident in their effect and ease frustration with developers because of it. Examples of improvements are: adding mutators and dynamically counting the number of asserts instead of statically.

Improving test code quality assessment. Our opinion weighs heavily in the evaluation of Spectata's quality verdict and recommendations. An important part of future work should be to create a more objective view on testing quality of a project. This means adding opinions of software experts, interviewing more developers and submitting pull requests on changes recommended by Spectata.

Evaluating the metric suite. We have not made an extensive study of the relation between metrics. As a consequence the size of the metric suite can probably be smaller without losing accuracy (see also the metrics galore pitfall in section 7.4.3). In addition, related efforts define more smells [61, 75, 24, 43, 44, 18, 17, 89] and metrics worth evaluating.

Adding to the aspects that define testing quality. Our effort to define unit testing quality started out with approximately seven aspects. Related work led us to define maintainability, adequacy and effectiveness, co-evolution, understandability, adherence to TDD and CD practices as influencing testing quality. Due to time constraints we were forced to create a selection, but the pipeline could provide insight into the value of additional aspects.

Assessing the relation between calculated testing quality and indicators of product quality. In this research effort we were able to establish a relation between the maintainability metrics and the maintenance effort. It would be interesting to see how our scores in adequacy and effectiveness relate to quality indicators such as: bugs found in production, issue handling similar to the work by Athanasiou et al. [17], and faults found in other testing stages.

7.4 Threats to Validity

7.4.1 Internal

Due to limitations in the tools used some measurements are slightly off, but besides those we took care to measure correctly. This work presents metric distributions, as well as explanations when a metric value was not close to what we expected. In addition, the metrics for every evaluation were first manually verified and only after we were sure they were correct did the evaluation continue.

The test code quality used in the evaluation for comparison is of course subjective. However, by involving the developers in the student and ING projects we minimized our influence on the verdict. The open source projects presented a larger challenge as we had no relation with the developers. Issuing a pull request with the recommended changes would have been an option had the projects not been end of life, so we leave this kind of feedback open for future work.

7.4.2 External

In order to improve external validity we made sure to include industry, student and open source projects in our evaluation. For all of the projects we evaluated a global score, local scores and recommendations, as well as some change scenarios. However due to the limited number of cases in our study we cannot be confident that the results can be generalized.

All of the projects are Java projects which raises the question whether other languages show similar results. In addition, the suite of evaluated projects includes two libraries and two back-end projects, meaning other categories like projects with a UI or embedded projects are missing.

7.4.3 Metric Pitfalls

In addition to the internal and external validity we evaluate to what extent Spectata exhibits the four metric pitfalls defined by Bouwers et al. [22].

Metric in a Bubble

Not being able to explain what a given value of a metric means is a clear sign of the *metric in a bubble* pitfall. To counter this effect we made sure to place a metric in context wherever possible. A metric is always a normalized value with respect to size and Spectata's reports provide an ideal value for developers to aim for. In the case of the maintainability metrics, this relationship is supported by empirical evidence. Our findings in chapter 3 show that the presence of test smells, measured by the metrics, increases the required maintenance effort. We rely on related work [76, 28, 54, 87] for similar support for the relation between mutation score and fault finding effectiveness. And for the relationship between coverage and fault finding capability [50, 25, 63, 95].

Treating the metric

The *treating the metric pitfall* is present when changes made to the software are purely cosmetic. Since this pitfall can only be properly detected when the tool is used in production, we cannot confidently say whether Spectata causes this pitfall or not. In the evaluation we took a *treating the metric* to applying a change, showing that it is certainly possible. However, as recommended in the paper, we do make the relationship between the metric and the desired property clear. Additionally, because of the way the scoring is set up, the goal of refactoring will always be improving an aspect of the test suite, instead of improving a metric. Spectata also limits naive improvements to a single metric by providing a quality verdict on multiple aspects of the test code. For example, removing tests will improve maintainability but will decrease adequacy and effectiveness.

One-track Metric versus Metrics Galore

The third and fourth pitfalls concern themselves with the size of the metric suite. Too few metrics cause an overly simple indication of progress towards the goal and too many may demotivate the development team. Spectata's mutation score is a good candidate for the first

category but we do not think it exhibits this pitfall because the metric itself is calculated by tracking multiple kinds of mutators. Spectata’s maintainability metric suite is a good candidate for metrics galore as it uses eight of them to calculate one aspect of testing quality. We have minimized the number of metrics by trying to select only one per smell but recognize that this could be improved. Using only one metric for measuring coupling, and performing a feature selection would further decrease the suite size. A Principal Component Analysis [53] and correlations between the metrics themselves could provide insight into how the metrics relate to each other, but we leave these kinds of analyses up to future work.

Chapter 8

Conclusion

Over the last few years agile practices have become more common throughout the software engineering process. One of the key ideas behind it is that software is never perfect, rather it is continuously improving. Because changes are being applied constantly, this way of working heavily depends on automated testing for verifying a product's quality.

The goal of this thesis is to be able to provide both developers as well as managers insight into the quality of unit testing of a software product in an agile environment. Based on the development practices at ING we identified four questions (PQ1 – PQ4) regarding: the current state of the test suite, how it can be improved, where it should be improved, and how a change affects its quality. Our tool: Spectata aims at answering these questions by extending a CI pipeline so that it can evaluate test code quality. Spectata provides a 1 to 5 star score on the project and the files it is comprised of and provides an accurate view of how test code quality changed between builds.

Central in Spectata is the model for calculating testing quality. Using metrics established in related research efforts, Spectata measures quality on three aspects: structural adequacy, fault finding effectiveness, and maintainability. In an effort to explore the metrics we found a significant correlation between the maintainability metrics and the maintenance effort.

To evaluate the Spectata approach we define four research questions that enable us to evaluate how it is able to help answer PQ1 – PQ4. In four case studies, the approach was applied to two open source projects, one project from industry and a project created by students. We compare Spectata's verdicts and recommendations for a unit test suite and some common change scenarios to how the developers and ourselves would answer the four questions. The results provide an indication that Spectata's verdict on (changes in) unit testing quality aligns with manual assessment. They also demonstrate that even though it is implemented as a prototype with limited applicability, the Spectata approach is able to help assure unit testing quality in a wide variety of projects.

Thus the main contributions of this thesis are the following:

- A new approach to assessing unit test code quality based on metrics for measuring the structural adequacy, fault finding effectiveness, and maintainability of a test suite.
- Empirical data on open source systems suggesting a significant correlation between the proposed test code maintainability metrics and change proneness of test classes.

8. CONCLUSION

- The Spectata implementation, available at: <https://gitlab.com/mjduijn>.
- Four case studies demonstrating that Spectata can be used for assuring unit testing quality.

These contributions help a development team come one step closer to fully automating deployments. The adoption of continuous deployment as a best practice makes this a goal for all development teams, and Spectata a tool to be used in every software delivery pipeline.

Bibliography

- [1] Line coverage: Lessons from junit, August 2016. URL <https://avandeursen.com/tag/junit/>.
- [2] Containers explained for the novice, 2016. URL <http://www.opgenorth.net/blog/2015/09/02/docker-containers-explained-for-the-novice/>.
- [3] Jacoco, June 2016. URL <http://eclemma.org/jacoco/>.
- [4] Jacoco control flow analysis for java methods, June 2016. URL <http://eclemma.org/jacoco/trunk/doc/flow.html>.
- [5] Junit, August 2016. URL <http://junit.org/junit4/>.
- [6] Pitest, June 2016. URL <http://docs.sonarqube.org/download/attachments/6960458/SQArchitecture5.5.png>.
- [7] Pitest mutation testing systems for java compared, June 2016. URL http://pitest.org/java_mutation_testing_systems/.
- [8] Pmd, June 2016. URL <https://pmd.github.io/>.
- [9] Pmd, June 2016. URL <http://www.sonarqube.org/>.
- [10] Sonarqube architecture overview, June 2016. URL <http://docs.sonarqube.org/download/attachments/6960458/SQArchitecture5.5.png>.
- [11] Usage of jacoco with java plugin, June 2016. URL <http://docs.sonarqube.org/display/PLUG/Usage+of+JaCoCo+with+Java+Plugin>.
- [12] Sonarqube web api, 2016. URL https://sonarqube.com/web_api.
- [13] Maven surefire plugin, June 2016. URL <http://maven.apache.org/surefire/maven-surefire-plugin/>.

BIBLIOGRAPHY

- [14] Khalid Alemerien and Kenneth Magel. Examining the effectiveness of testing coverage tools: An empirical study. International Journal of Software Engineering and Its Applications, 8(5):139–162, 2014.
- [15] Tiago L Alves, Christiaan Ypma, and Joost Visser. Deriving metric thresholds from benchmark data. In Software Maintenance (ICSM), 2010 IEEE International Conference on, pages 1–10. IEEE, 2010.
- [16] Dimitrios Athanasiou. Constructing a test code quality model and empirically assessing its relation to issue handling performance. PhD thesis, TU Delft, Delft University of Technology, 2011.
- [17] Dimitrios Athanasiou, Arifin Nugroho, Joost Visser, and Andy Zaidman. Test code quality and its relation to issue handling performance. Software Engineering, IEEE Transactions on, 40(11):1100–1125, 2014.
- [18] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In Software Maintenance (ICSM), 2012 28th IEEE International Conference on, pages 56–65. IEEE, 2012.
- [19] Kent Beck. Test-driven development: by example. Addison-Wesley Professional, 2003.
- [20] Kent Beck. Test-driven development: by example. Addison-Wesley Professional, 2003.
- [21] James M Bieman, Anneliese Amschler Andrews, and Helen J Yang. Understanding change-proneness in oo software through visualization. In Program Comprehension, 2003. 11th IEEE International Workshop on, pages 44–53. IEEE, 2003.
- [22] Eric Bouwers, Joost Visser, and Arie Van Deursen. Getting what you measure. Queue, 10(5):50:50–50:56, May 2012. ISSN 1542-7730. doi: 10.1145/2208917.2229115. URL <http://doi.acm.org/10.1145/2208917.2229115>.
- [23] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (xml). World Wide Web Consortium Recommendation REC-xml-19980210. <http://www.w3.org/TR/1998/REC-xml-19980210>, 16:16, 1998.
- [24] Manuel Breugelmans and Bart Van Rompaey. Testq: Exploring structural and maintenance characteristics of unit test suites. In WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques. Citeseer, 2008.
- [25] Xia Cai and Michael R Lyu. The effect of code coverage on fault detection under different testing profiles. ACM SIGSOFT Software Engineering Notes, 30(4):1–7, 2005.

-
- [26] Ward Cunningham. The wycash portfolio management system. ACM SIGPLAN OOPS Messenger, 4(2):29–30, 1993.
 - [27] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on, pages 201–211. IEEE, 2014.
 - [28] Murial Daran and Pascale Thévenod-Fosse. Software error analysis: a real case study involving real faults and mutations. In ACM SIGSOFT Software Engineering Notes, volume 21, pages 158–171. ACM, 1996.
 - [29] Martin Davidsson, Jiang Zheng, Nachiappan Nagappan, Laurie Williams, and Mladen Vouk. Gert: An empirical reliability estimation and testing feedback tool. In Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on, pages 269–280. IEEE, 2004.
 - [30] M. Delahaye and L. du Bousquet. A comparison of mutation analysis tools for java. In Quality Software (QSIC), 2013 13th International Conference on, pages 187–195, July 2013. doi: 10.1109/QSIC.2013.47.
 - [31] Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Hints on test data selection: Help for the practicing programmer. Computer, (4):34–41, 1978.
 - [32] Richard A DeMillo, Dany S Guindi, WM McCracken, AJ Offutt, and KN King. An extended overview of the mothra software testing environment. In Software Testing, Verification, and Analysis, 1988., Proceedings of the Second Workshop on, pages 142–151. IEEE, 1988.
 - [33] Arie Van Deursen, Leon Moonen, Alex Bergh, and Gerard Kok. Refactoring test code. In Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001), pages 92–95, 2001.
 - [34] Paul M Duvall, Steve Matyas, and Andrew Glover. Continuous integration: improving software quality and reducing risk. Pearson Education, 2007.
 - [35] George Fink and Matt Bishop. Property-based testing: a new approach to testing for assurance. ACM SIGSOFT Software Engineering Notes, 22(4):74–80, 1997.
 - [36] Kenneth A. Foster. Error sensitive test cases analysis (estca). IEEE Transactions on Software Engineering, 6(3):258, 1980.
 - [37] Martin Fowler. Continuous integration. 2006.
 - [38] Martin Fowler. Mocks aren’t stubs. 2007.
 - [39] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. Refactoring: improving the design of existing code. 1999. ISBN: 0-201-48567-2.

BIBLIOGRAPHY

- [40] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. The elements of statistical learning, volume 1. Springer series in statistics Springer, Berlin, 2001.
- [41] Erich Gamma and Kent Beck. Junit, 2006.
- [42] John S Gourlay. A mathematical framework for the investigation of testing. Software Engineering, IEEE Transactions on, (6):686–709, 1983.
- [43] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Automated detection of test fixture strategies and smells. In Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on, pages 322–331. IEEE, 2013.
- [44] Benedikt Hauptmann, Maximilian Junker, Sebastian Eder, Elmar Juergens, and Rudolf Vaas. Can clone detection support test comprehension? In Program Comprehension (ICPC), 2012 IEEE 20th International Conference on, pages 209–218. IEEE, 2012.
- [45] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the, pages 30–39. IEEE, 2007.
- [46] William C Hetzel and Bill Hetzel. The complete guide to software testing. John Wiley & Sons, Inc., 1991.
- [47] Michael Httermann. DevOps for developers. Apress, 2012.
- [48] Jez Humble and David Farley. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader). Pearson Education, 2010.
- [49] Victor Hurdugaci and Andy Zaidman. Aiding software developers to maintain developer tests. In Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on, pages 11–20. IEEE, 2012.
- [50] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria. In Proceedings of the 16th international conference on Software engineering, pages 191–200. IEEE Computer Society Press, 1994.
- [51] ISO/IEC. ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. Technical report, 2010.
- [52] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. Software Engineering, IEEE Transactions on, 37(5):649–678, 2011.
- [53] Ian Jolliffe. Principal component analysis. Wiley Online Library, 2002.

-
- [54] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 654–665. ACM, 2014.
- [55] Richard M. Karp and Michael O. Rabin. Efficient randomized pattern-matching algorithms, 1987.
- [56] Sun-Woo Kim, John A Clark, and John A McDermid. Investigating the effectiveness of object-oriented testing strategies using the mutation method. Software Testing, Verification and Reliability, 11(4):207–225, 2001.
- [57] Henrik Kniberg and Anders Ivarsson. Scaling agile@ spotify. online], UCVOF, ucvox. files. wordpress. com/2012/11/113617905-scaling-Agile-spotify-11. pdf, 2012.
- [58] R Lipton. Fault diagnosis of computer programs. Student Report, Carnegie Mellon University, 1971.
- [59] Zeeger Lubsen, Andy Zaidman, and Martin Pinzger. Using association rules to study the co-evolution of production & test code. In Mining Software Repositories, 2009. MSR’09. 6th IEEE International Working Conference on, pages 151–154. IEEE, 2009.
- [60] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for java. In Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on, pages 352–363. IEEE, 2002.
- [61] Gerard Meszaros. xUnit test patterns: Refactoring test code. Pearson Education, 2007.
- [62] HD Mills. On the statistical validation of computer programs. IBM Federal Systems Division Report, pages 72–6015, 1972.
- [63] Audris Mockus, Nachiappan Nagappan, and Trung T Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In Empirical Software Engineering and Measurement, 2009. ESEM 2009. 3rd International Symposium on, pages 291–301. IEEE, 2009.
- [64] Larry Joe Morell. A theory of error-based testing. Technical report, DTIC Document, 1984.
- [65] Glenford J Myers, Corey Sandler, and Tom Badgett. The art of software testing. John Wiley & Sons, 2011.
- [66] Nachiappan Nagappan. A software testing and reliability early warning (strew) metric suite. 2005.

BIBLIOGRAPHY

- [67] Nachiappan Nagappan, Laurie Williams, Jason Osborne, Mladen Vouk, and Pekka Abrahamsson. Providing test quality feedback using static source code and automatic test suite metrics. In 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05), pages 10–pp. IEEE, 2005.
- [68] Akbar Siami Namin and James H Andrews. The influence of size and coverage on test suite effectiveness. In Proceedings of the eighteenth international symposium on Software testing and analysis, pages 57–68. ACM, 2009.
- [69] A Jefferson Offutt. Investigations of the software testing coupling effect. ACM Transactions on Software Engineering and Methodology (TOSEM), 1(1):5–20, 1992.
- [70] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology (TOSEM), 5(2):99–118, 1996.
- [71] Helena Holmström Olsson, Hiva Alahyari, and Jan Bosch. Climbing the” stairway to heaven”—a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. In Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on, pages 392–399. IEEE, 2012.
- [72] Chad Perrin. The cia triad. Dostopno na: <http://www.techrepublic.com/blog/security/the-cia-triad/488>, 2008.
- [73] Paul Piwowarski, Mitsuru Ohba, and Joe Caruso. Coverage measurement experience during function test. In Software Engineering, 1993. Proceedings., 15th International Conference on, pages 287–301. IEEE, 1993.
- [74] Jacek Ratzinger, Thomas Sigmund, Peter Vorburger, and Harald Gall. Mining software evolution to predict refactoring. In First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007), pages 354–363. IEEE, 2007.
- [75] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. Rule-based assessment of test quality. Journal of Object Technology, 6(9):231–251, 2007.
- [76] Aditya P. Mathur Richard A. DeMillo. On the use of software artifacts to evaluate the effectiveness of mutation analysis for detecting errors in production software. Technical report.
- [77] Eric Ries. The lean startup: How today’s entrepreneurs use continuous innovation to create radically successful businesses. Crown Books, 2011.
- [78] Alex Rodriguez. Restful web services: The basics. IBM developerWorks, 2008.
- [79] Winston W Royce. Managing the development of large software systems,[online] <http://www.cs.umd.edu/class/spring2003/cmsc838p.Process/waterfall.pdf>-last accessed, 8(9):2011, 1970.

-
- [80] Per Runeson. A survey of unit testing practices. Software, IEEE, 23(4):22–29, 2006.
- [81] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering, 14(2):131–164, 2008. ISSN 1573-7616. doi: 10.1007/s10664-008-9102-8. URL <http://dx.doi.org/10.1007/s10664-008-9102-8>.
- [82] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. Empirical software engineering, 14(2):131–164, 2009.
- [83] Andy Schneider. Junit best practices. Java World, 12:181, 2000.
- [84] K Schwaber and Mike Beedle. gilè software development with scrum. 2002.
- [85] Muhammad Shahid and Suhaimi Ibrahim. An evaluation of test coverage tools in software testing. In 2011 International Conference on Telecommunication Technology and Applications Proc. of CSIT, volume 5, 2011.
- [86] Ben Smith and Laurie Ann Williams. A survey on code coverage as a stopping criterion for unit testing. 2008.
- [87] Ben H Smith and Laurie Williams. Should software testers use mutation analysis to augment a test set? Journal of Systems and Software, 82(11):1819–1832, 2009.
- [88] G Tassey. Economic impacts of inadequate infrastructure for software testing, planning report 02-3. Prepared by RTI for the National Institute of Standards and Technology (NIST), 2002.
- [89] Dávid Tengeri, Arpad Beszedes, Tamás Gergely, Laszlo Vidacs, David Havas, and Tibor Gyimóthy. Beyond code coverage—an approach for test suite assessment and improvement. In Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on, pages 1–7. IEEE, 2015.
- [90] Bart Van Rompaey, Bert Du Bois, Serge Demeyer, and Matthias Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. Software Engineering, IEEE Transactions on, 33(12):800–817, 2007.
- [91] Erik Van Veenendaal. Standard glossary of terms used in software testing. International Software Testing Qualifications Board, pages 8–9, 2012.
- [92] László Vidács, Ferenc Horváth, Dávid Tengeri, and Árpád Beszédes. Assessing the test suite of a large system based on code coverage, efficiency and uniqueness. In 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 2, pages 13–16. IEEE, 2016.
- [93] Anneliese von Mayrhauser, J Wang, Magnus C Ohlsson, and Claes Wohlin. Deriving a fault architecture from defect history. In Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on, pages 295–303. IEEE, 1999.

BIBLIOGRAPHY

- [94] Arthur H Watson, Thomas J McCabe, and Dolores R Wallace. Structured testing: A testing methodology using the cyclomatic complexity metric. NIST special Publication, 500(235):1–114, 1996.
- [95] Michael Whalen, Gregory Gay, Dongjiang You, Mats PE Heimdahl, and Matt Staats. Observable modified condition/decision coverage. In Proceedings of the 2013 International Conference on Software Engineering, pages 102–111. IEEE Press, 2013.
- [96] Lee J White and Edward I Cohen. A domain strategy for computer program testing. Software Engineering, IEEE Transactions on, (3):247–257, 1980.
- [97] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. Experimentation in software engineering. Springer Science & Business Media, 2012.
- [98] Qian Yang, J Jenny Li, and David M Weiss. A survey of coverage-based testing tools. The Computer Journal, 52(5):589–597, 2009.
- [99] Robert K Yin. Case study research: Design and methods. Sage publications, 2013.
- [100] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production & test code. In Software Testing, Verification, and Validation, 2008 1st International Conference on, pages 220–229, April 2008. doi: 10.1109/ICST.2008.47.
- [101] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. Empirical Software Engineering, 16(3): 325–364, 2011.
- [102] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. Acm computing surveys (csur), 29(4):366–427, 1997.

Appendix A

Threshold Values Metrics

The table below shows the threshold values for the adequacy and effectiveness metrics required for each quality category. The values are derived from the set of benchmark projects in such a way that categorizing each of the files in our benchmark using these values approximately leads to a distribution of $< 5, 30, 30, 30, 5 >$. We have had to adjust these criteria slightly as some of the thresholds coincided with each other.

Quality Category	Line Coverage	Branch Coverage	Mutation Score
5 stars	95	95	99
4 stars	85	85	85
3 stars	70	70	70
2 stars	45	45	45
1 stars	0	0	0

Table A.1: Minimum percentages for achieving a certain quality score for the adequacy and effectiveness metrics

Appendix B

Overview of Tables in Spectata Storage

Figure B.1 shows the tables used to store Spectata's findings. The relations shown in the diagram are not enforced by the database but are verified in the analysis stage of the quality calculation.

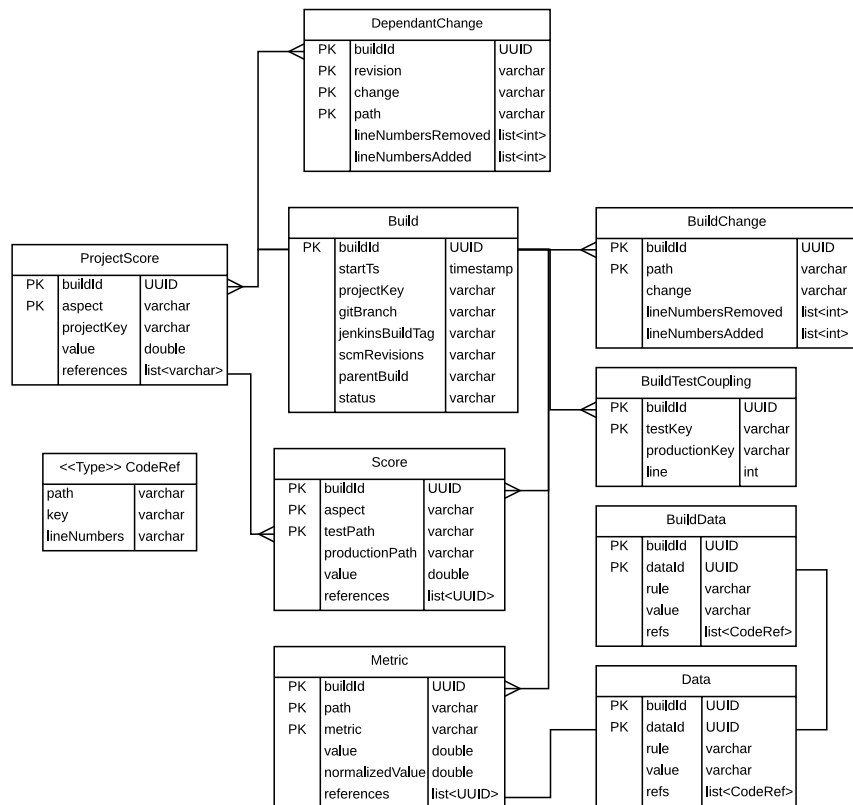


Figure B.1: The Spectata storage model

Appendix C

Devhub Local Adequacy Recommendations

Classes with no clear purpose that excuses them from being tested adequately.

```
devhub/server/DevhubModule.java
devhub/server/backend/BasicAuthenticationProvider.java
devhub/server/backend/mail/PullRequestMailer.java
devhub/server/web/jackson/ObjectMapperProvider.java
devhub/server/web/filters/UserAuthorizeFilter.java
devhub/server/web/filters/RepositoryAuthorizeFilter.java
devhub/server/database/entities/rubrics/DutchGradingStrategy.
    java
devhub/server/web/errors/ThrowableMapper.java
devhub/server/database/entities/Event.java
devhub/server/backend/CourseEventFeed.java
devhub/server/web/filters/BuildServerAuthenticationFilter.java
devhub/server/backend/CommentBackend.java
devhub/server/backend/LdapBackend.java
devhub/server/backend/mail/ReviewMailer.java
devhub/server/DevhubServer.java
devhub/server/GitServerClientModule.java
devhub/server/web/errors/NotFoundExceptionMapper.java
devhub/server/web/errors/EntityNotFoundExceptionMapper.java
devhub/server/backend/mail/BuildResultMailer.java
devhub/server/web/errors/UnauthorizedExceptionMapper.java
devhub/server/backend/mail/CommentMailer.java
devhub/server/backend/mail/MailBackendImpl.java
devhub/server/backend/LdapAuthenticationProvider.java
devhub/server/util/Highlight.java
```

Template classes

C. DEVHUB LOCAL ADEQUACY RECOMMENDATIONS

```
devhub/server/web/templating/Translator.java
devhub/server/web/templating/TemplateEngine.java
devhub/server/web/templating/WrappedReader.java
devhub/server/web/templating/TranslatorFactory.java
```

Resource classes

```
devhub/server/web/resources/repository/
    PrivateRepositoryResource.java
devhub/server/web/resources/RubricImportResource.java
devhub/server/web/resources/AccountResource.java
devhub/server/web/resources/CourseEditionResource.java
devhub/server/web/resources/ValidationResource.java
devhub/server/web/resources/PrivateRepositoryOverview.java
devhub/server/web/resources/RootResource.java
devhub/server/web/resources/StudyNumberResource.java
devhub/server/web/resources/repository/PrivatePullResource.java
devhub/server/web/resources/CoursesResource.java
devhub/server/web/resources/repository/
    AbstractProjectIssueResource.java
devhub/server/web/resources/CourseAssistantsResource.java
devhub/server/web/resources/AssignmentsResource.java
devhub/server/web/resources/repository/PrivateIssueResource.
    java
devhub/server/web/resources/BuildServerResource.java
devhub/server/web/resources/CourseEnrollResource.java
devhub/server/web/resources/repository/ProjectIssueResource.
    java
```

Data classes

```
devhub/server/web/resources/views/WarningResolver.java
devhub/server/backend/mail/MailBackend.java
devhub/server/web/models/PullCloseResponse.java
devhub/server/util/Version.java
devhub/server/web/models/DeleteBranchResponse.java
```

Controller classes

```
devhub/server/database/controllers/PrivateRepositories.java
devhub/server/database/controllers/IssueComments.java
devhub/server/database/controllers/PullRequestComments.java
```

Appendix D

Metrics for Evaluated Joda-Time Classes

Metric	TestIslamicChronology	TestDateTimeZone	TestDateTime FormatterBuilder
Class Length	569	614	470
Case Length	27.9	16.5	11.6
Asserts	19.8	4.96	3.32
Complexity	0.545	0.915	0.532
Dead Code	0.273	2.28	1.47
Duplication	8	4.98	1.96
Coupling Classes	26.5	20.6	44.1
Coupling Lines	257	368	882

Table D.1: Maintainability metrics for the evaluated Joda-Time classes