



Norwegian University of
Science and Technology

Decreasing Response Time of Failing Automated Tests by Applying Test Case Prioritization

Sveinung Dalatun
Simon Inge Remøy
Thor Kristian Ravnanger Seth
Øyvind Voldsund

Master of Science in Computer Science

Submission date: June 2011

Supervisor: Tor Stålhane, IDI

Co-supervisor: Ole Christian Rynning, BEKK Consulting AS

Problem Description

Slow running test suites are sometimes a problem in the software industry. Having a system for prioritizing test cases would to some extent solve this problem. If developers were to receive the failing tests first (fast feedback), they could save time, stay in context and not get disturbed while waiting for large test suites to run.

Assignment

- Develop several techniques for prioritizing test cases within a test suite according to their likelihood of failing.
- Implement the techniques as a tool.

You can implement the tool in whatever programming language wanted, but Ruby and/or Java is preferred.

Submission date: June 6th 2011.

Internal supervisor: Prof. Tor Stålhane, NTNU.

External supervisor: Ole Christian Rynning, Senior Consultant at BEKK Consulting AS.

Acknowledgements

We would like to thank our internal supervisor Prof. Tor Stålhane for his assistance to the thesis and several reviews of the report during the project. We also appreciate the meetings and workshops with our external supervisor Ole Christian Rynning from BEKK Consulting AS. Both Ole Christian and Christian Schwarz from BEKK have been of great help with fast replies to e-mails when we encountered problems. We thank Ole Christian and Jøran V. Lillesand for participating in the first experiment.

We would also like to thank the firms that participated in our survey. Surveys were sent out to several IT companies, and we know that at least six of them participated; BEKK Consulting, Steria, Statoil, IT-Verket, Miles and Visma Sirius.

For the research part of our project we highly appreciate the help from Jari Bakken (test developer at Finn.no) and Gregg Rothermel (professor and Jensen chair of software engineering, University of Nebraska).

Abstract

Running automated tests can be a time-consuming task, especially when doing regression testing. If the sequence of the execution of the test cases is arbitrary, there is a good chance that many of the defects are not detected until the end of the test run. If the developer could get the failing tests first, he would almost immediately be able to get back to coding or correcting mistakes. In order to achieve this, we designed and analyzed a set of test case prioritization techniques. The prioritization techniques were compared in an experiment, and evaluated against two existing techniques for prioritizing test cases.

Our implementation of the prioritization techniques resulted in a tool called *Pritest*, built according to good design principles for performance, adaptability and maintainability. This tool was compared to an existing similar tool through a quality analysis.

The problem we address is relevant for the increased popularity of agile software methods, where rapid regression testing is of high importance.

The experiment indicates that some prioritization techniques perform better than others, and that techniques based on code analysis is outperformed by techniques analyzing code changes, in the context of our experiment.

Contents

1	Introduction	1
1.1	Problem Specification	2
1.2	Motivation	4
1.3	Problem Scenarios	5
1.4	Introduction to Pritest	6
2	Methodology and Report Design	9
2.1	Methodology	9
2.2	Report Design	10
3	Literature Research	12
3.1	Related Work	13
3.1.1	Test Case Management Categories	13
3.1.2	Prioritization Techniques	15
3.1.3	Comparator techniques	17
3.1.4	Methods for Evaluating Test Case Management Techniques	18
3.1.5	Test Suite Granularity	20
3.1.6	JUnit Max	21
3.2	Industrial Survey	21
3.2.1	Introduction	21
3.2.2	Survey Theory	22

3.2.3	Design and Questions	23
3.2.4	Distribution	25
3.2.5	Results	25
3.2.6	Lessons Learned	31
3.3	Technical Prestudy and Theory	33
3.3.1	Effective Java	33
3.3.2	Big-O Analysis Method	35
3.3.3	Adaptability	36
3.3.4	Maintainability	38
3.3.5	Software and Testing	43
3.3.6	Java Source Code Analysis	46
3.3.7	Java Git Libraries	48
4	Own Contribution	51
4.1	The Pritest Tool	54
4.2	Implementing our Prioritization Techniques	62
4.3	Online Prioritization Techniques	63
4.3.1	Counting Failing Tests	63
4.3.2	Code Changes	66
4.4	Local Prioritization Techniques	70
4.4.1	Local Code Changes	70
4.4.2	Additional Method Coverage	76
4.4.3	Total Method Coverage	87
4.5	Hybrid Prioritization Techniques	90
4.5.1	Local Code Changes with Failure Counting	90
4.6	Control Techniques	94
4.6.1	Untreated Order	94
4.6.2	Random Order	94
4.6.3	Optimal Order	95

4.7	Evaluating Technique Time Complexity with Big-O Analysis	95
5	Experiment	109
5.1	Experiment Theory	109
5.2	Experiment Introduction	112
5.3	Definition	112
5.3.1	Goal Definition	112
5.3.2	Summary of Definition	113
5.4	Planning	113
5.4.1	Context Selection	113
5.4.2	Hypothesis Formulation	114
5.4.3	Variables Selection	115
5.4.4	Selection of Subjects	116
5.4.5	Experiment Design	117
5.4.6	Instrumentation	118
5.4.7	Validity Evaluation	120
5.5	Operation	122
5.5.1	Preparation	122
5.5.2	Execution	122
5.5.3	Data Validation	123
5.6	Analysis and Interpretation	124
5.6.1	Descriptive Statistics	125
5.6.2	Data Reduction	127
5.6.3	Hypothesis Testing	128
5.7	Secondary Experiment	129
5.7.1	Context	130
5.7.2	Variables	130
5.7.3	Execution	130
5.7.4	Descriptive Statistics	131

5.7.5	Hypothesis Testing	133
5.7.6	Validity Evaluation	135
5.8	Summary and Conclusions	135
6	Evaluation and Discussion	137
6.1	Comparison with Existing Techniques	137
6.2	Post-Experiment Interview	138
6.3	Quality Analysis - Comparing Pritest to JUnit Max	139
6.4	Rationale for Chosen Techniques	140
6.4.1	Improving the Existing Techniques	140
6.4.2	The Choice of Prioritization Techniques to Implement	140
6.4.3	Potential Prioritization Techniques	142
6.5	The Choice of Evaluation Metric	144
6.6	Abandoning Technique Counting Failing Tests the Last Three Days	145
6.7	Benchmark - Custom JUnit Runner	146
6.8	Continuous Integration	148
7	Conclusion and Future Work	150
7.1	Conclusion	150
7.2	Future Work	154
8	Appendices	157
8.1	Survey Free Text Replies	157
8.2	UML Diagrams for Pritest	163
8.3	Post-Experiment Interview Replies	172
8.4	Experiment Results	174
8.4.1	Minitab Hypothesis Testing	174
8.4.2	APFD Values	176
	Bibliography	179

List of Tables

3.2	Free-text question categorization labels.	30
3.3	Techniques for writing effective Java.	34
3.4	Relative dominance of common algorithm complexity terms.	36
3.5	Code Smells and mitigations.	41
3.6	Unit test benchmark projects.	45
3.7	Unit test vs. integration test run time.	45
4.1	Pritest modules properties.	53
4.2	Overview of applied technology.	54
4.3	Techniques summary.	62
4.4	Big-O Calculations.	96
5.1	Scale Types by Wohlin et al. [22].	117
5.2	Prioritization techniques numbering.	124
5.3	Hypothesis test - primary experiment.	128
5.4	Hypothesis test - primary experiment.	129
5.5	Hypothesis test - secondary experiment.	134
5.6	Hypothesis test - secondary experiment.	134
6.1	Custom JUnit runner benchmark.	148
8.1	APFD result from the primary experiment.	176

8.2 APFD result from the secondary experiment. 177

List of Figures

1.1	Relation between test suites, test cases and tests.	3
3.1	Test case selection.	14
3.2	Free-text question categorization.	31
3.3	Factory pattern example [54].	37
3.4	UML diagram - strategy pattern.	38
3.5	The V-Model of software testing.	43
3.6	The visitor pattern.	47
3.7	Supported commands by JGit library	49
4.1	Overview of Pritest.	52
4.2	Priority List.	57
4.3	UML Class diagram of the new pritest-junit-runner.	58
4.4	Organization of packages in our pritest-junit-runner.	60
4.5	Using the git status command in a bash command.	70
4.6	Technique Local Code Changes prioritization list selection process. . .	71
4.7	Additional Method Coverage illustration	78
4.8	Total Method Coverage illustration	88
4.9	Illustration of hybrid technique prioritization list.	90
4.10	Big-O illustration: Counting Failing Tests and Code Changes are red, and the two Local Code Changes techniques are green.	96

4.11	Big-O illustration of Total Method Coverage.	97
4.12	Big-O illustration of Additional Method Coverage.	97
5.1	Experiment principles [29].	110
5.2	Experiment process [22].	111
5.3	Example of APFD graph generation - using a single technique.	119
5.4	Example of APFD graph generation - using multiple techniques at once.	119
5.5	A boxplot of the results - primary experiment.	125
5.6	APFD instrumentation graph example - primary experiment.	127
5.7	A boxplot of the results - secondary experiment.	131
5.8	APFD instrumentation graph example - secondary experiment.	133
6.1	Weighted relationships between code changes and test failures.	143
6.2	Local code changes and dependencies.	144
6.3	Bash script run.	147
8.1	Minitab output - primary experiment.	174
8.2	Minitab Output - secondary experiment.	175

Chapter 1

Introduction

This report reflects the work of four master students from the computer science department. The project is carried out in cooperation with BEKK Consulting AS with Ole Christian Rynning as our external supervisor. The internal supervisor is Prof. Tor Stålhane from NTNU.

Regression testing is a common task for developers when developing software, both to ensure that new functionality works as expected, and to test that older code still work after changes [24]. Testing can be a time-consuming task, especially when the system grows big, and we have thus investigated the possibilities of reducing time spent on automated testing.

Last semester, for the specialization project [1], we developed a system named *Pritest*¹. Pritest offers a service for prioritizing test cases for developers during development of an information system. The specification of Pritest, the architecture and reasoning of applied technology are discussed in the specialization project report [1]. In the

¹In the specialization project it was called *Citrus*, but this semester we changed the name of our tool to *Pritest*. This was due to a naming conflict with another open source project.

master thesis we have focused on improving Pritest, researching prioritization techniques, evaluation methods and existing solutions as well as conducting an experiment on the final result.

1.1 Problem Specification

The project concerns automated testing in software development, with a special emphasis on unit and integration tests. We will investigate the possibilities of reducing the time spent waiting for feedback when performing automated testing. In order to achieve this, we will investigate several techniques for prioritizing test cases with respect to their likelihood of failing. We will compare these techniques in an experiment (Chapter 5), and compare the ones we develop ourselves to existing techniques.

We will implement our techniques as a tool (Pritest). Our second goal is to develop this tool according to good design principles, with focus on efficiency, adaptability and maintainability. This tool will not be compared to existing tools through an experiment, but rather through a quality analysis Chapter 6.

Goal Summary

1. Research and implement techniques for prioritizing a test suite (a set of test cases) according to their likelihood of failing. Compare these techniques through an experiment, and evaluate them against existing prioritization techniques.
2. Implement the techniques as a tool designed for efficiency, adaptability and maintainability. Compare the tool to existing solutions through a quality analysis.

The terms *test suite*, *test case* and *test* are used throughout this report, and the distinction between them is illustrated in Figure 1.1.

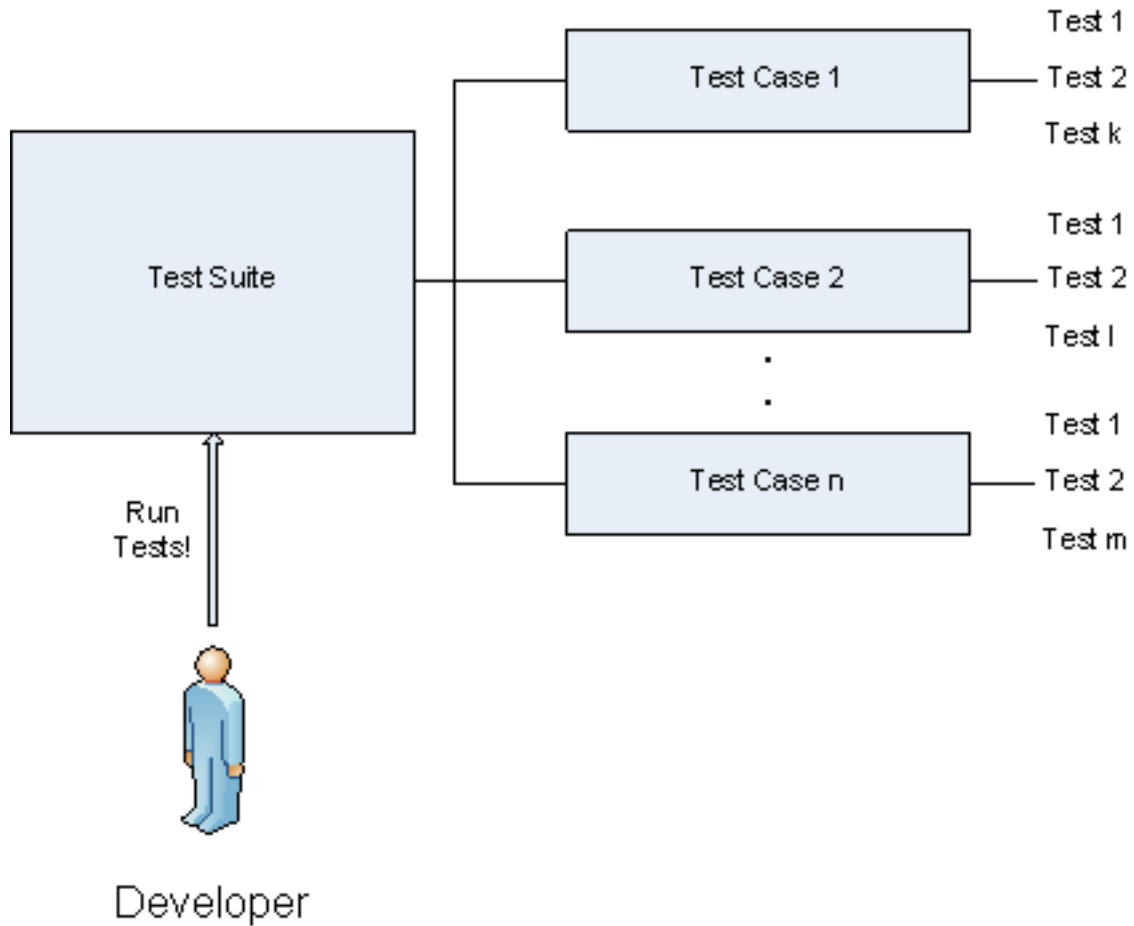


Figure 1.1: Relation between test suites, test cases and tests.

The test suite can be viewed as a set of classes, each test case is a class, and each test is a method in a class [14].

1.2 Motivation

From discussions with our external supervisor at BEKK Consulting, we identified slow automated test runs as a problem in the software industry. This problem has evolved over the last years, due to the interest in agile development methodologies and techniques like Test Driven Development (TDD). TDD practitioners want to see their tests fail before they make them succeed [64], and slow running test suites is a problem. With agile and TDD comes the paradigm of running tests both before and after writing code [28], in contrast to development methodologies built on waterfall models where the tests only are executed after writing code [15]². The need for—and problems regarding—automated testing today, was investigated through a survey in Section 3.2.

Software development is an abstract process that requires concentration, and the fact that programmers have to build up a complex mental model of the programming problem makes them vulnerable to interruptions. Interruptions often lead to the developer performing slower, and perceives the interrupted task to be more difficult to complete [5]. The severity of an interruption depends on several aspects, such as how long the interruption goes on, the person causing it, its subject and the type of work it disrupts [6]. The type of work being interrupted is an important factor, and work like designing algorithms and writing code are sensitive to interruptions [6].

This thesis address an important aspect of software development in seeking to improve the efficiency of developers with more rapid feedback from automated testing. In the field regarding the first fragment of our goal (prioritization techniques) there exists some research, especially from Gregg Rothermel [4, 9, 12, 13, 14]. For the second part

²Dr. Royce however, considered the original waterfall model as a risky development process that needed modifications to transform into a process that will provide desired products.

of our goal—tools for test prioritization—there exist only one other solution: *JUnit Max* by Kent Beck [33].

1.3 Problem Scenarios

To illustrate the problems we seek to solve, we have constructed two scenarios describing the problem and a desired situation.

Current Situation

- A developer is working on a project, and has started working on a class.
- The developer writes a unit test, implements the code, and is ready to run the tests to see if the implementation makes the test pass.
- The tests are run, but the new test is located at the end of the test suite, and thus the developer has to wait for all the older tests to be run before he is presented with the result of his new test.
- In the meantime, trying to kill the unnecessary waiting time, he lost focus and began surfing the web. Thus, he wasted even more time.

Desired Situation

- A developer is working on a project, and has started working on a class.

- The developer writes a new unit test, implements the code, and is ready to run the tests to see if the implementation makes the test pass.
- The tests are run, and since the system can evaluate the tests' likelihood of failing, the new test is placed first in the test suite. Hence, the developer can see if the test has passed or not immediately.
- The developer instantly gets feedback, and is able to continue his work with minimal distractions.

1.4 Introduction to Pritest

Pritest offers a service that can be reached through a REST interface [63] when developing software, recording several metrics which Pritest evaluate, before it delivers a prioritized list of the test cases to the test runner when called upon.

The *pritest-junit-runner* module is the client part of our system, and is responsible for executing JUnit tests in the order specified by Pritest, and then sending the results back to the server. It is implemented as a Maven plugin. The *pritest-junit-runner* can be configured to use one out of six techniques to generate a prioritized list of tests to run:

1. Counting Failing Tests (Section 4.3.1)
Favors test cases with highest amount of historical failures.

2. Code Changes (Section [4.3.2](#))

Makes use of a *post-receive hook* from the online version control host Github [37]. Github automatically sends a report to Pritest when receiving new code changes.

3. Local Code Changes (Section [4.4.1](#))

Uses a Java library for conducting a equivalent command to `git status`, and running the most recently added or modified *local* test cases first.

4. Local Code Changes with Failure Counting (Section [4.5.1](#))

Same as *Local Code Changes*, but with an additional prioritization on the remaining test cases using technique *Counting Failing Tests*.

5. Total Method Coverage (Section [4.4.3](#))

The concepts of this technique is developed by Rothermel et al., and implemented by us.

6. Additional Method Coverage (Section [4.4.2](#))

The second technique designed by Rothermel et al., and implemented by us. Our techniques will be compared to this one and *Total Method Coverage* in the experiment.

From the list above, technique *Counting Failing Tests* and *Code Changes* are online techniques—they need to contact Pritest to get prioritized lists. Technique *Local Code Changes*, *Total Method Coverage* and *Additional Method Coverage* are local techniques—only running locally on the *pritest-junit-runner* maven plugin we developed. We also implemented one hybrid technique, using both local techniques and prioritization from Pritest; *Local Code Changes with Failure Counting*.

The *pritest-server* module is responsible for storing data about changes to source code, results from tests that have been run, and for providing lists of tests that the client should run, which can be reached through a REST interface on the server.

For more details concerning Pritest and its architecture, technology and components see Chapter 4, or the specialization project report [1]. All the techniques we implemented are described in detail in Chapter 4.

Chapter 2

Methodology and Report Design

2.1 Methodology

When deciding on which methodology to use to solve our problem, we had to look at the nature of the problem at hand. We had to employ an empirical strategy to compare our designed solution to other similar solutions developed earlier. According to Claes Wohlin et al. [22] empirical strategies or investigations can be divided into three major types: *survey*, *case study* and *experiment*.

Problem Facts

- Something needs to be developed.
- There exist some similar solutions.
- A comparison of our solution and existing ones must be performed.

In the research part of this thesis (Chapter 3) we conducted a survey to gain knowledge about the problem domain. The remaining parts of the thesis is written as an empirical study using *experiment* as observation technique. Empirical studies exist in both qualitative and quantitative forms, but experiments are purely quantitative since they have focus on measuring variables, change them and measure again [22]. Our experiment measures the techniques we developed for prioritizing test cases—comparing them to existing techniques—and is presented in Chapter 5. Details regarding the experiment is also found in this chapter (such as definitions of context, hypothesis, variables, subjects and selected hypothesis testing technique).

2.2 Report Design

The thesis follows a standard structure for reports written for empirical studies [65], and the remaining chapters of this report mainly consist of *research*, *descriptions of implementations* and an *experiment*.

Chapter 3 describes the literature research we have done in advance of the implementation and experimentation. We investigate existing work in the field, conduct a survey and a prestudy of technologies needed later in the thesis.

Chapter 4 presents the implementations and design of our techniques and tool. The techniques are categorized into *online*, *local* and *hybrid* prioritization techniques, depending on their need for contacting our online service (*pritest-server*) for retrieving the prioritization lists.

In Chapter 5 we present our experiment and its results. The evaluation and discussion regarding our findings is found in Chapter 6.

Chapter 7 contains a conclusion and proposals for further improvements to the tool and the prioritization techniques.

Chapter 3

Literature Research

This chapter describes our research regarding published work on the subject, as well as some research on technologies done in advance of improving the Pritest system. There are a lot of empirical studies in this field, but none of them offer a tool like the one we plan to deliver with Pritest. The majority of the studies are conducted on the programming language C, and could not be generalized to e.g Java and testing frameworks like JUnit [45] which we are using. JUnit Max (Section 3.1.6) is the only solution similar to Pritest, but this also have some drawbacks that we have described in Chapter 3.1.6. JUnit Max is compared to Pritest in Section 6.3.

3.1 Related Work

There are several empirical studies in the field of test efficiency and test case prioritization. However, the vast majority of these are based on the programming language C, and does not consider the history of failing test cases. Our study is focused on Java technology, and uses a history-based approach to test case prioritization.

The number of studies based on selecting techniques for prioritization and selection of test cases is large. Especially Elbaum and Rothermel have written many articles on this topic [4, 9, 12, 13, 14]. All of these articles have in common that they do not generalize well to all sizes of projects, types of software or test suite characteristics. They conclude that different types of prioritization techniques should be used for different types of systems, but that using any technique at all is better than using none.

3.1.1 Test Case Management Categories

According to Yoo and Harman [3] the existing literature on the management of test cases can be categorized into three classes: *test suite reduction* (or *minimization*), *test case prioritization* and *test case selection*.

In test suite reduction, some test cases are permanently removed from the suite. The goal is to reduce the cost of running the suite, which requires that the cost of reduction is less than the gain achieved by omitting certain test cases. However, there is a possibility that by removing some test cases the ability of the suite to detect faults is reduced.

Test case selection is similar to test suite reduction, but the changes to the test suite

are not permanent. A subset of test cases is selected based on some criteria to reduce the costs (Figure 3.1).

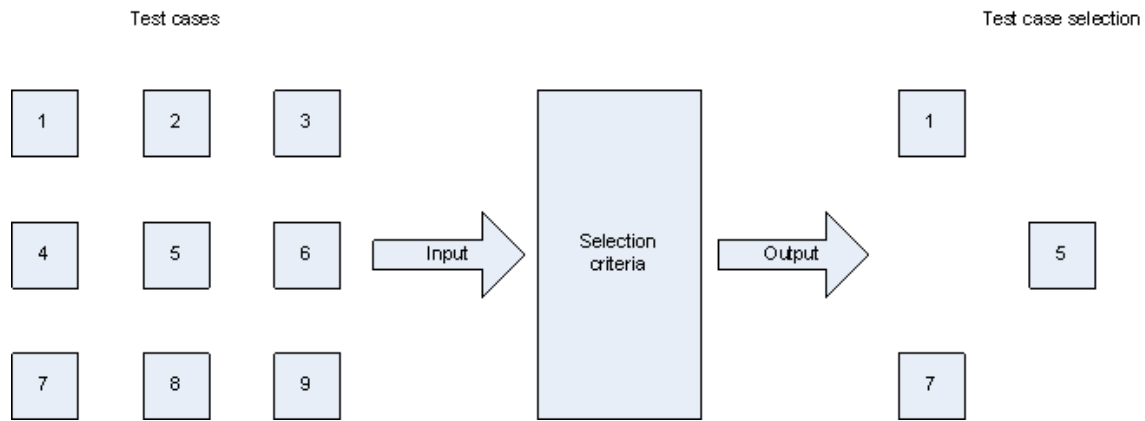


Figure 3.1: Test case selection.

In test case prioritization, the goal is to find an execution order of test cases that is optimal according to some criterion. Some prioritization techniques, along with methods for evaluating the test execution orders given by them, are explained below.

Kim and Porter [2] conducted a study on history-based test selection techniques. The study was empirical and did not result in a working tool. The study had, however, a large threat to external validity, since the experiments were conducted in strictly controlled environments and could not be generalized. The study did indicate that heuristics based on history improved efficiency through test case selection.

Many approaches have been made at ranking test cases in a test suite. Mende and Koschke [8] utilized an effort-aware model to select modules based on their related risk and the effort required to test them. Many effort-aware models assume that there is some budget involved, e.g. that we only want to run 20% of the test cases. The goal is then to select those 20% that will detect the largest amount of faults.

3.1.2 Prioritization Techniques

In most of the articles we have found, the studies are based on a common set of techniques for ranking and prioritizing tests. The article *Selecting a Cost-Effective Test Case Prioritization Technique* [9] summarizes these common techniques as *total function coverage*, *additional function coverage* and one technique family that does not involve coverage at all (the number of code statements, functions or blocks the test executes).

According to Elbaum et al., prioritization techniques can vary along several dimensions [13]. The dimensions they identified are the following:

- *Technique granularity*, which is the level at which the technique performs its analysis and gathering of information to be used as a basis for the prioritization. E.g. some techniques work at the statement-level and some at the function-level. A technique with a lower-level granularity has more information at its disposal than a higher-level technique, and can therefore make more informed decisions. It does however require more processing.
- Whether or not the technique uses feedback, or in other terms: whether it is “*total*” or “*additional*”. A “*total*” technique bases its prioritization on information available at the beginning of the process, while an “*additional*” technique gathers feedback as the test cases are prioritized, and uses this feedback to prioritize the remaining test cases.
- Whether or not the technique employs information about the modified version of the source code.

Total function coverage ranks the test cases’ importance by the order of the number of functions the individual test case covers. If multiple test cases cover the same

amount of functions, these are ranked randomly.

Additional function coverage iteratively selects the tests with the greatest function coverage, and then recursively runs additional function coverage on the remaining test cases and the remaining uncovered functions, until all of them have been covered.

There are also techniques similar to total and additional function coverage, but that consider blocks or methods instead of functions [14]. These are called total and additional block coverage, and total and additional method coverage. Blocks in this sense are sequences of statements with single entry and exit points.

The last category mentioned in the article is concerned with non-coverage techniques like code modifications. This is one of the techniques used by Pritest, in addition to a few other non-coverage techniques.

In addition to the function-level techniques mentioned above, there are also more low-level techniques such as total and additional statement coverage. *Total statement coverage* orders the test cases descendingly by the number of statements they cover. *Additional statement coverage* is to total statement coverage and statements as additional function coverage is to total function coverage and functions.

Sherriff et al. presented a *history-based prioritization technique* using three elements: association clusters, the relationships between test cases and files, and a modification vector [10]. The association clusters are made by grouping files that often are modified together as fixes of a defect. E.g. if fixing a bug requires modifying three files, those three will be placed in the same cluster. Additionally, by keeping record of the relationships between test cases and files, the test cases testing those modified files can easily be identified. A key point of this technique is that any software artifact can be subject to prioritization.

A *Requirement-based approach* was presented by Srikanth et al [11], where test cases are mapped to requirements. The prioritization is based on factors such as customer-assigned implementation complexity and priority. This method allows us to give critical requirements a higher priority, but a potential weakness is the estimation and subjectivity involved when assigning the priorities.

One of the drawbacks of test case prioritization—compared to test suite reduction and test case selection—is that its basic definition does not involve excluding test cases. If the testing activity involves a budget, running all the test cases can be unrealistic. To satisfy this need, a number of cost-aware techniques have been proposed. To evaluate cost-aware techniques, Elbaum et al. [12] developed a metric which we will discuss in Section 3.1.4.

3.1.3 Comparator techniques

For experimental control we can employ the comparison techniques used by Hyunsook Do et al. [14]. The techniques are as following:

- *Random ordering*: This technique simply reorganizes the tests randomly.
- *Optimal ordering*: If the experiment uses seeded faults, an optimal ordering can give an upper bound on the effectiveness of the techniques we use.
- *Untreated ordering*: This one is merely the ordering without any prioritization technique employed.

3.1.4 Methods for Evaluating Test Case Management Techniques

There are several techniques that can be used when evaluating a selected subset of test cases [8]. First of all, we are interested in finding as many faults as possible. One alternative for this is *recall*, which is the percentage of defective files that are detected.

$$recall = \frac{|\{\text{defective files}\} \cap \{\text{files marked as defective}\}|}{|\{\text{defective files}\}|}$$

An alternative to recall is *defect detection rate* (ddr), which is the ratio of the number of detected defects compared to the total number of defects. Mende and Koschke prefer ddr, as they argue “it better captures the cost-effectiveness of a model” [8].

Not only do we want to find as many faults as possible, we also want as few false-positives as possible. To measure this we can employ *precision*. Precision is the fraction of files marked as defective that actually are defective.

$$precision = \frac{|\{\text{defective files}\} \cap \{\text{files marked as defective}\}|}{|\{\text{files marked as defective}\}|}$$

As an alternative to precision, we have *false-positive ratio* (fdr), which is the ratio of non-defective files marked as defective. Mende and Koschke prefer precision.

One metric that can be used to evaluate prioritization techniques is the *average percentage of faults detected* (APFD) a metric developed by Elbaum et al. [4]. The APFD measures the average rate of fault detection per percentage of test suite execution, and favors orderings that detect faults early during the execution of the test

suite. APFD can be calculated using the following notation:

- T is a test suite containing n test cases.
- F is the set of m faults revealed by T .
- For a test case ordering T' , let TF_i be the order of the first test case that reveals the i th fault.

The APFD value for T' is calculated as following:

$$APFD = 1 - \frac{TF_1 + \dots + TF_m}{nm} + \frac{1}{2n}$$

Elbaum et al. incorporated the severity of detected faults and the execution cost of test cases into the APFD [12]. The resulting metric, $APFD_c$, favors prioritizations that detect severe faults at a low cost.

$$APFD_c = \frac{\sum_{i=1}^m \left(f_i \times \left(\sum_{j=TF_i}^n t_j - \frac{1}{2}t_{TF_i} \right) \right)}{\sum_{i=1}^n t_i \times \sum_{i=1}^m f_i}$$

T is a test suite of n test cases, where each has an associated cost t_1, t_2, \dots, t_n ; F is the set of m faults with severities of f_1, f_2, \dots, f_m ; and TF_i is the order of the first test case to expose the i th fault.

In our experiment we used APFD as instrumentation for evaluating the ordering of each prioritization technique. This choice is discussed in Section 6.5.

3.1.5 Test Suite Granularity

When using a test framework such as JUnit, we also need to consider the granularity of the test suite. JUnit organizes test suites in test cases—which are classes—at the top level, and test methods—which are methods—at the lowest level and are located in test cases (Figure 1.1). A study by Rothermel et al. [14] showed that test suites with finer granularity gave better support for prioritization than more coarse test suites, when using the techniques such as total function coverage and additional function coverage.

We chose to implement our solution on a “test case” level, even though this is not the finest granularity possible (that would be on a “method/test” level). The reason for this is primarily that JUnit (one test framework for Java) accepts test cases by default, and implementing a custom JUnit runner that runs on a method/test level would probably not be beneficial for our objective. Instead of jumping between specific tests inside a test case, we simply run the whole test case before proceeding to the next in the prioritization list. Also, if we were to run single tests inside a test case separately, a new JUnit runner would have to be instantiated for each test to run, which would cost more than only instantiating new JUnit runners for each test case in the test suite.

3.1.6 JUnit Max

JUnit Max [33] is a solution developed by Kent Beck for test prioritization. This is a tool that provides test prioritization based on recent history of failed tests, as well as the principle of running a lot of small tests first, and the larger tests afterwards. Beck emphasizes that test runs follow a *power law distribution* [16]; a lot of small tests and a few large ones. JUnit Max is a plugin developed for the Integrated Development Environment (IDE) Eclipse [34], and is therefore not applicable for other IDEs.

3.2 Industrial Survey

3.2.1 Introduction

To confirm that the problem we were trying to solve was a legitimate one, we conducted an industrial survey. Surveys are normally conducted when the use of a technique or tool already has taken place or before it is introduced [21]. Wohlin et al. [22] describes a survey as a snapshot of the current situation, and this was our goal for this survey. We wanted to find out to what extent our problem actually was a relevant one for developers dealing with automated testing every day.

Another motivation for this survey was to gain insight into how the industry cope with automated testing, which tools were used and which standards and routines they followed.

3.2.2 Survey Theory

Surveys are usually carried out in one of two forms: interviews or questionnaires [23]. The basic method for data collection through questionnaires is distributing it with instructions on how to fill it out, and the participants return it to the researcher when completed. According to Claes Wohlin et al. [22], there are some advantages for interviews over questionnaires:

- Interview surveys typically achieve higher response rates than, for example, mail surveys.
- An interviewer generally decreases the number of “do not know” and “no answer”, because the interviewer can answer additional questions about the survey.
- It is possible for the interviewer to observe and ask follow-up questions.

On the other hand, there are also disadvantages to using interviews over questionnaires, like time and cost. We decided to design a questionnaire, and distributed it through the social network Twitter [49], and to several well-known IT companies in Norway. The companies we know participated are mentioned in .

Claes Wohlin et al. also point out three categories—or purposes—of surveys: the *descriptive* survey, the *explanatory* survey and the *explorative* survey. These three serve different purposes in a research environment, respectively enabling assertions about some population, making explanatory claims about the population, or finally using a survey as a pre-study to more thorough investigation later on [22]. Our survey is an explorative survey.

3.2.3 Design and Questions

When designing our questionnaire we had two things especially in mind: not making it too large, and ask questions that did not “fish” for anticipated answers. The survey came with instructions, and the subjects were informed that it was an anonymous survey and that the results would be published in this thesis. The subjects were asked a series of questions regarding automated testing in software development, and a few introduction questions like experience level were included. The table below contains all the questions and options in our questionnaire.

Questions	Options
What is your current occupation / studies?	Beginner, Bachelor studies, Master studies, Professional developer
What is your experience level with Test Driven Development/Design (TDD)?	No experience, Read a book / Took a course / Learned in school, Work experience 0-1 years, Work experience more than one year
Do you use TDD or other test oriented methodologies in the project/course you are in now?	Yes, No
When using TDD—or similar techniques—how many test failures do you usually get within each test run	0-2, 3-4, 5-10, >10
Which test frameworks are you familiar with?	JUnit, Cucumber, TestNG, NUnit, Rspec, JSpec, JSUnit, PHPUnit, None, Other

When developing software, how often do you normally run through your unit tests?	After each implementation, About once each hour, Twice a day, Once a day, Other
How much do you agree to the following statement: “Waiting for slow test runs is a problem for me”?	Totally disagree, Disagree, Partially agree, Agree, To- tally agree
How many times a day do you approximately run through your unit tests?	<5, 5-10, 11-20, 21-30, >30
How much do you agree to the following statement: “The test I am interested in is often run last, or late, in the test suite”?	Totally disagree, Disagree, Partially agree, Agree, To- tally agree
If your test suite takes a long time to execute, do you get something useful done while waiting for test runs to complete?	Usually not, Yes, Some- times
What is the worst part(s) of waiting for test runs to complete?	Context switching, Annoy- ing, Time consuming, Noth- ing, Other
Which of the following automated tests would you clas- sify the slowest to run generally?	Unit test, Integration test, System test, Don’t know
Anything else? (optional)	Free text area

3.2.4 Distribution

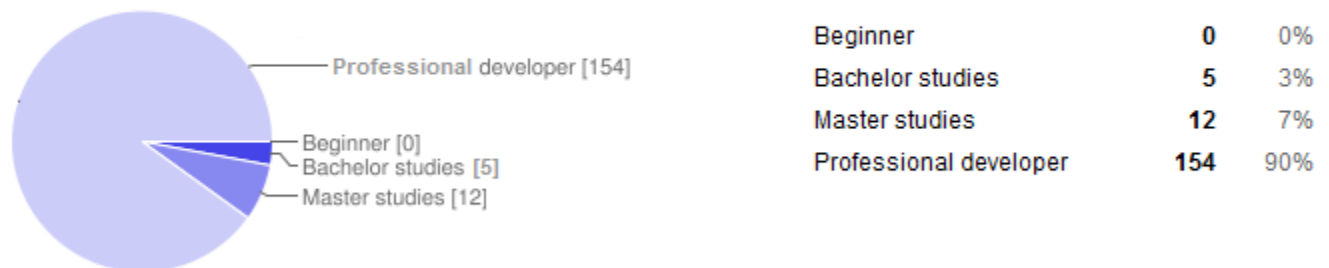
The questionnaire was first distributed using the social network Twitter [49]. By using a social network as a distribution channel we could not be sure that the subjects were actually developers that knew of automated software testing. About ten subjects answered the survey from our initial Twitter distribution.

We then decided to send e-mails to several well-known IT companies in Norway, asking them to contribute with some developers for our survey. After a short while we received a total of 172 replies. The companies we know participated, is mentioned in *Acknowledgements*.

3.2.5 Results

We found the results from the survey interesting, and they confirmed some of the assumptions we had regarding automated testing, as well as uncovered some new aspects of the problem. Most of the assumptions we had regarding automated testing and its challenges were confirmed by the multiple choice questions in the survey. The “free-text” question at the end revealed a few elements that we were not aware of. The results from the multiple choice or check-box questions are presented below:

What is your current occupation / studies?



What is your experience level with Test Driven Development/Design (TDD)?



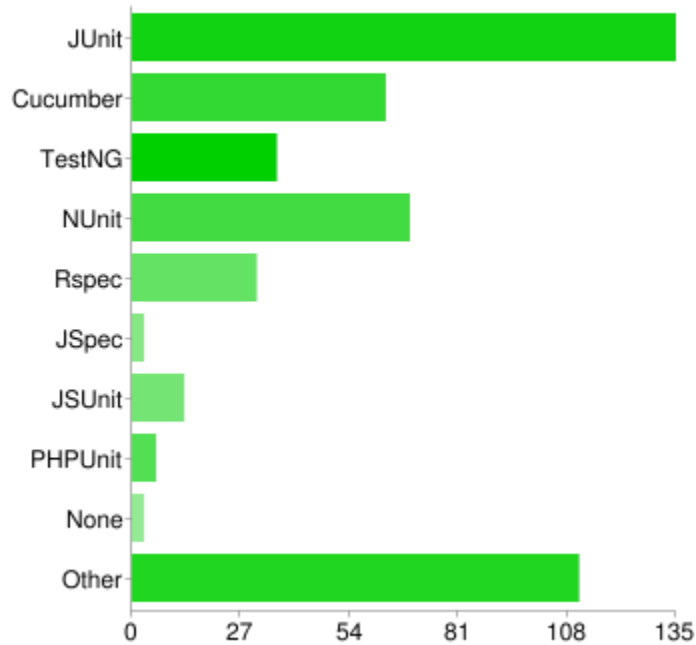
Do you use TDD or other test oriented methodologies in the project/course you are in now?



When using TDD - or similar techniques - how many test failures do you usually get within each test run?



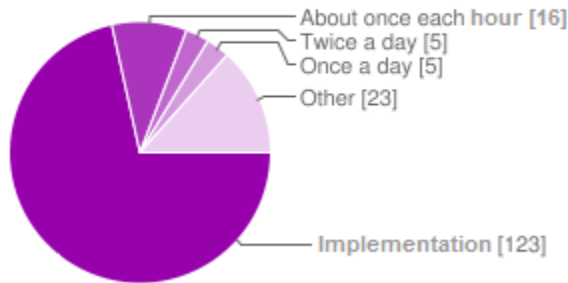
Which test frameworks are you familiar with?



JUnit	135	78%
Cucumber	63	37%
TestNG	36	21%
NUnit	69	40%
Rspec	31	18%
JSpec	3	2%
JSUnit	13	8%
PHPUnit	6	3%
None	3	2%
Other	111	65%

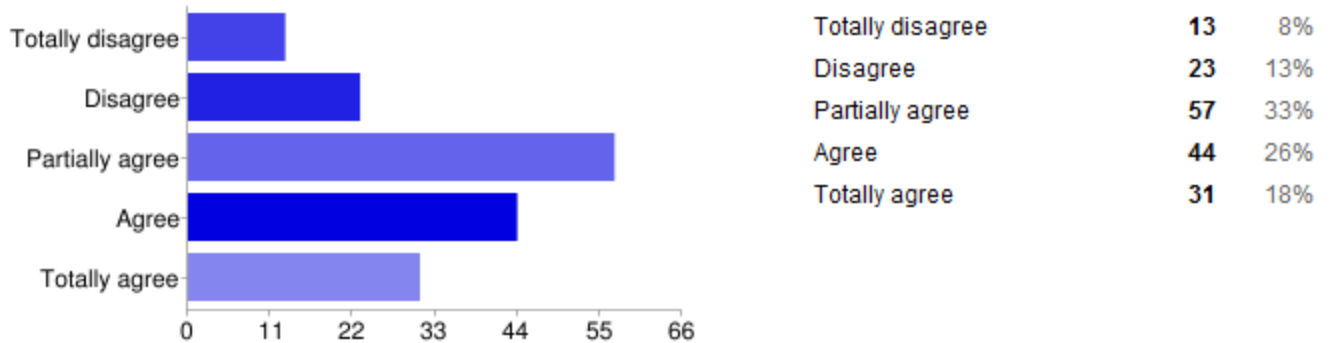
People may select more than one checkbox, so percentages may add up to more than 100%.

When developing software, how often do you normally run through your unit tests?

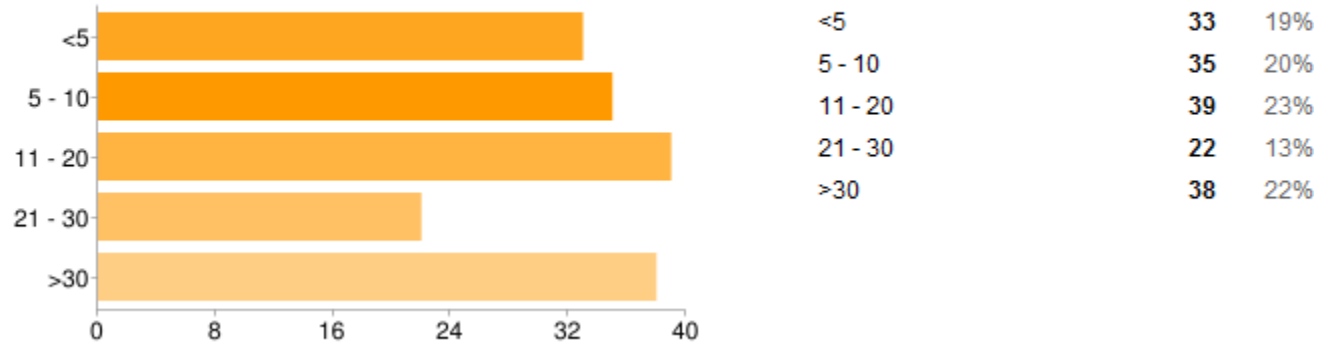


After each implementation	123	72%
About once each hour	16	9%
Twice a day	5	3%
Once a day	5	3%
Other	23	13%

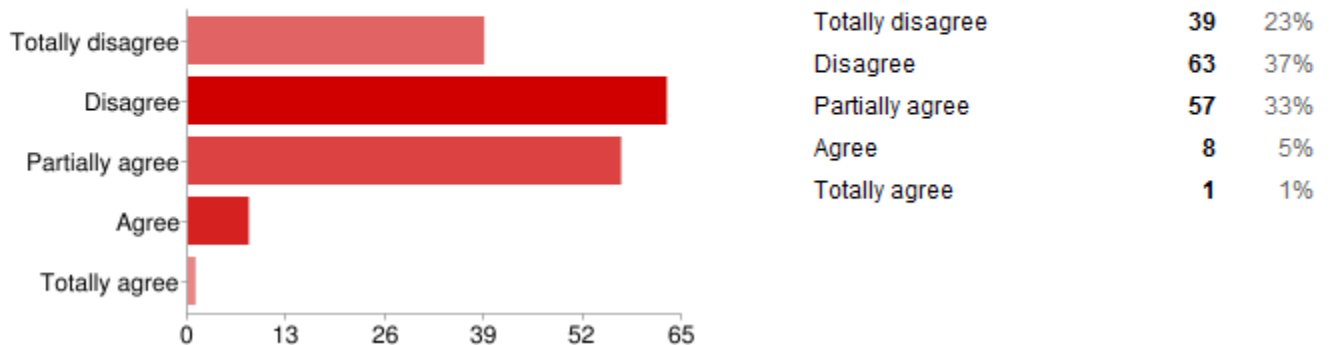
How much do you agree to the following statement: "Waiting for slow test runs is a problem for me"?



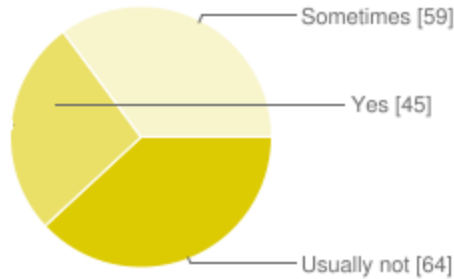
How many times a day do you approximately run through your unit tests?



How much do you agree to the following statement: "The test I am interested in is often run last, or late, in the test suite"?

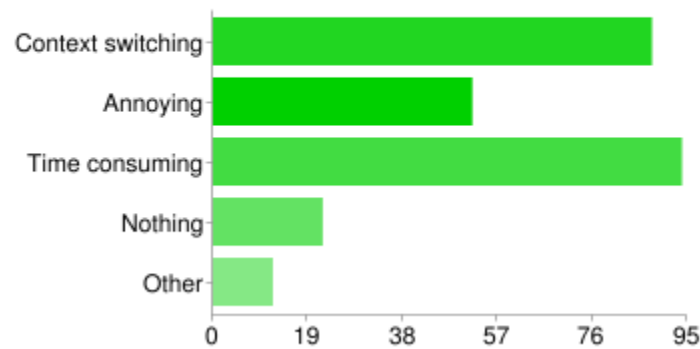


If your test suite takes a long time to execute, do you get something useful done while waiting for test runs to complete?



Usually not	64	37%
Yes	45	26%
Sometimes	59	34%

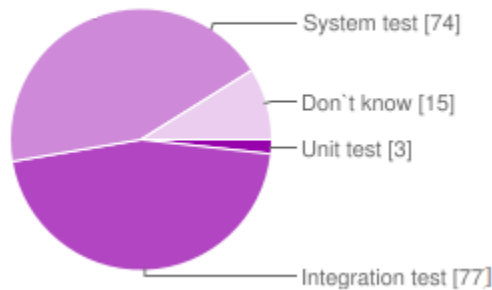
What is the worst part(s) of waiting for test runs to complete?



Context switching	88	52%
Annoying	52	31%
Time consuming	94	56%
Nothing	22	13%
Other	12	7%

People may select more than one checkbox, so percentages may add up to more than 100%.

Which of the following automated tests would you classify the slowest to run generally?



Unit test	3	2%
Integration test	77	45%
System test	74	43%
Don't know	15	9%

Free-Text Question

Included in the survey was a field where the participants could insert other issues regarding automated testing, or the survey. 32 of the subjects used this option, and a lot of interesting topics were highlighted. A list of all the replies to this question are shown in *Appendices* (Section 8.1).

To analyze the replies to the free-text question, we categorized them into nine categories, to identify the most recurring topics. We then assigned labels to the different categories (Table 3.2).

Category	# Replies	Alias
Continuous integration	10	C1
Question formulation	4	C2
TDD downsides	4	C3
Assignment praise	3	C4
Research tip	3	C5
Web test slowness	3	C6
Integration test slowness	2	C7
Test scope separation	2	C8
Test process tip	1	C9

Table 3.2: Free-text question categorization labels.

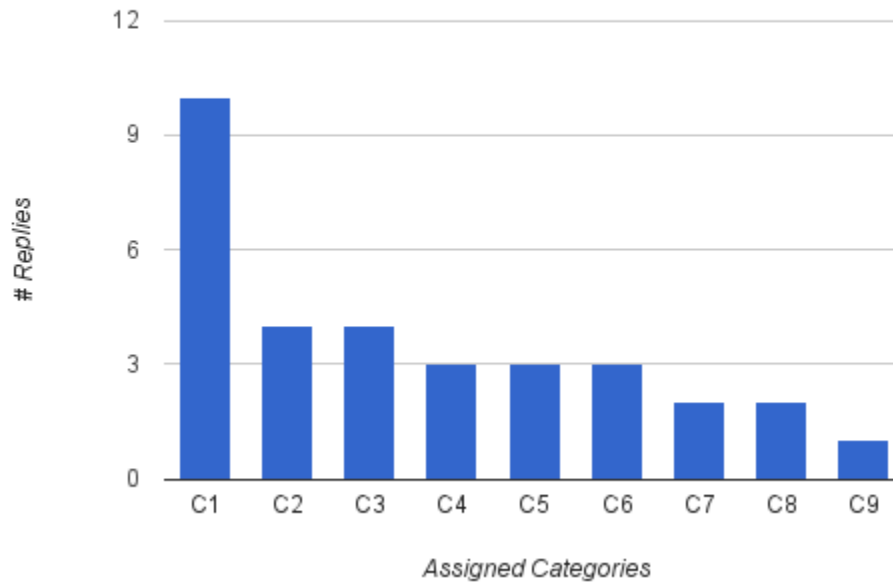


Figure 3.2: Free-text question categorization.

The category with the most mentions is *Continuous Integration (CI)*. Ten of the participants who answered the optional free-text questions mentioned CI as a partial mitigation to the problem we are trying to solve. This finding is further discussed in Section 6.8.

3.2.6 Lessons Learned

We got some feedback through the free-text question that the question "How many times a day do you approximately run through your unit tests?" was ambiguous. A lot of the subjects wondered if we meant the entire test suite, or tests related to the code being implemented at the time. We wanted to find out how often a developer

was running any unit test at all. We see that this question could be hard to interpret, and was not well formulated.

The question "How many times a day do you approximately run through your unit tests?" is unclear to me; I run parts of the test suite lots of times a day, but I only run the entire suite a few times a day. The answer I gave is how often I run a part of the suite.

- Survey participant

Also, some participants commented that TDD could not be used with tests as slow as mentioned in the survey. The survey was intended to be focused on automated software testing and not TDD as a methodology, so this was a small survey design slip from our side. The heading of the survey was not that well formulated—"Master thesis survey - Unit Testing & TDD".

What we should have done was to differentiate on unit, integration and system tests. Equally to the question "How many times a day do you approximately run through your unit tests?", we should ask the same question regarding integration tests and system tests.

However, we find the survey well enough designed, and it verified some of our earlier assumptions. We gained knowledge on how the industry treats automated testing, and how they handle the problems that follow.

Several participants commented these minor issues in the free-text field, and explained how they interpreted the questions they found unclear. From these comments, we see that most of the participants understood the questions the way we intended them to; hence, these issues have not had substantial effect on the survey results.

3.3 Technical Prestudy and Theory

We have studied several technologies as a basis for design decisions in our implementation of Pritest. The studies are mostly concerned with the effectiveness of algorithms and the applied programming language; Java, as well as techniques for writing a solution that is prepared for future development and adaptable for new environments and configurations.

The theory presented in this chapter is mostly conceptual, and cover all the areas that arose during the pre-study, while Chapter 4 present the actual implementation of the theory applied to our solution, and contains reasoning for the chosen technologies.

3.3.1 Effective Java

As Pritest is a tool aiming for rapid feedback to the user, it should have efficient techniques for prioritizing test cases and be optimized on the programming language level. There exist several literature studies on how to optimize Java code. “Effective Java” by Bloch [18]—one of the best known books in this field—contains a total of 78 techniques for writing effective Java, considering both run-time efficiency of code and development efficiency (number of features implemented per time unit).

The book is intended for experienced Java developers trying to becoming even better developers, but the techniques are based on fundamental principles, and clarity and simplicity is especially emphasized by the author. Bloch stress that the book is not primarily about writing high performance code, but about writing code that is clear, correct, usable, robust, flexible and maintainable, and if one can do that, it is usually a relative simple matter to get the performance needed. Figure 3.3 represent a

selection of the techniques presented in the book.

Avoid creating unnecessary objects
Consider implementing <code>Comparable</code>
Minimize the accessibility of classes and members
In public classes, use accessor methods, not public fields
Prefer interfaces to abstract classes
Use interfaces only to define types
Use function objects to represent strategies
Eliminate unchecked warnings
Prefer lists to arrays
Favor generic types
Favor generic methods
Use enums instead of <code>int</code> constants
Consistently use the <code>Override</code> annotation
Design method signatures carefully
Return empty arrays or collections, not nulls
Prefer for-each loops to traditional <code>for</code> loops
Know and use the libraries
Avoid <code>float</code> and <code>double</code> if exact answers is required
Prefer primitive types to boxed primitives
Adhere to generally accepted naming conventions
Avoid unnecessary use of checked exceptions
Favor the use of standard exceptions
Do not ignore exceptions
Use lazy initialization judiciously

Table 3.3: Techniques for writing effective Java.

Other research papers exist on the field of high performance Java code, but these are mostly concerned with concurrency systems, and systems that require atomic behavior, like a service intended to handle multiple user requests at the same time. Such features are not required of Pritest at the moment.

3.3.2 Big-O Analysis Method

Big-O Analysis (O: Order of Magnitude) is a method used to evaluate functional relationships that arise in all fields of engineering [19]. In computer science it can be used to evaluate, either the relative speed or the absolute speed of an algorithm. Relative speed is used to compare two or more algorithms, while the absolute speed is the actual execution time. Which measure to use depends on the application of the algorithms, and the goal of the evaluation. The analysis of relative speed is regarded as the simplest to perform, since absolute speed depends on variables like platform and hardware.

Big-O evaluate the upper limit of a mathematical function, and is based on the principle of having a *dominant term* present in a function of several terms. When values get large, the dominant term will be sufficient to represent the approximate value of the original function being evaluated.

Table 3.4 present the most common occurrences of dominant terms in algorithms [19]. The table is ranked according to dominance, with the least dominant term first and the most dominant term in the bottom of the table.

Dominant Term	Name of Dominant Term
c	Constant
$\log_2 n$	Logarithmic
n	Linear
$n \log_2 n$	Linear Logarithmic
$n^2 < n^3 < \dots < n^i (i < n)$	Polynomial
c^n	Exponential
$n!$	Factorial

Table 3.4: Relative dominance of common algorithm complexity terms.

To get a picture of the time complexity of each technique in our solution, we performed a big-O Analysis in Section 4.7.

3.3.3 Adaptability

In addition to writing efficient Java code, we would like to implement our solution as adaptable as possible. We found that most of the techniques for achieving adaptability were through implementation of one or more suitable design patterns. A design pattern is a programming specific common solution to a well known problem or task [20].

One way of obtaining adaptability is the use of one or more abstract factory patterns. The classic example of an abstract factory is painting of operating system specific GUI elements. The application simply calls a `createButton()` method in the factory, not specifying which operating system the application is running on, but depending on the actual operating system, the factory creates the correct button design. This is one way of achieving adaptability, and could be used in areas like database access, database determination and GUI applications. Another example can be seen in Figure 3.3.

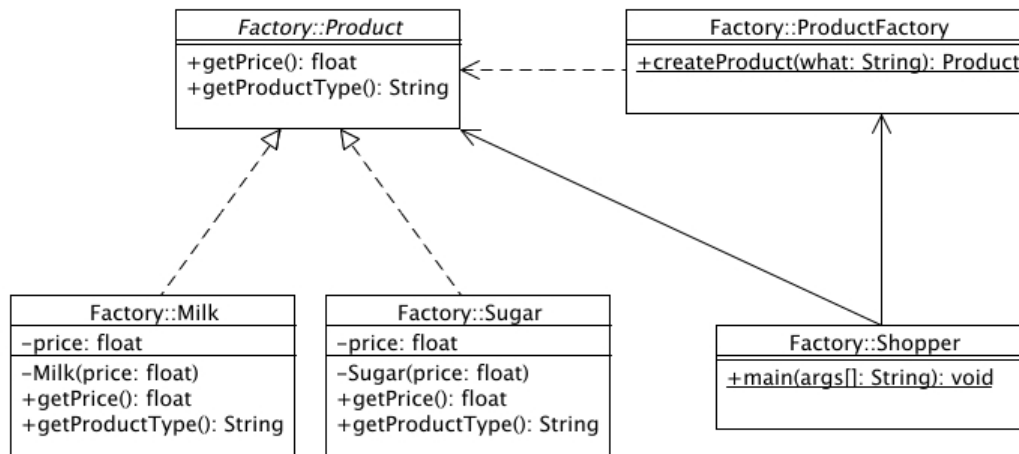


Figure 3.3: Factory pattern example [54].

The use of configuration files is another tactic which enhance both the adaptability and the modifiability of software [20]. One way of implementing this, is instantiating the `Configuration` class in Java, and passing the singleton class `PropertiesHolder` as parameter. `PropertiesHolder` is reading a `“.properties”` file from the file system, where the configuration is set.

We discovered one last pattern that was particularly interesting: the strategy pattern.

We were advised to take a look at this pattern by our external supervisor. This pattern is intended to alter the behavior of an application with respect to what context it is operating in. This is a way to adapt the run-time behavior, and this enables us to swap between different techniques based on the input at runtime, and select a strategy for running the tests based on the context. Our implementation is described in Section 4.1. Figure 3.4 shows a conceptual UML diagram of the strategy pattern.

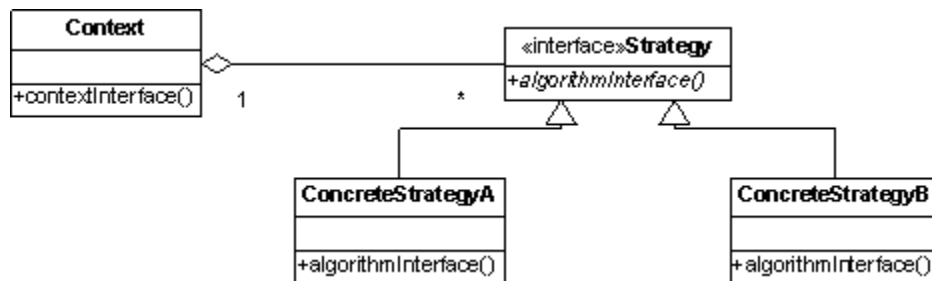


Figure 3.4: UML diagram - strategy pattern.

3.3.4 Maintainability

Maintainability is “a set of attributes that bear on the effort needed to make specified modifications” (ISO/IEC 9126-1:2001). The factor consists of four criteria [17]:

- *Analyzability*, the effort needed to analyze the software, e.g. to identify what parts to be modified.
- *Changeability*, the effort needed to change the code in order to attain the desired effect, e.g. fault removal.
- *Stability*, the effort needed to modify the code in such a way that the modifications do not have any unintended side effects.

- *Testability*, the effort needed to validate modifications.

When software goes from development to production, it needs to be maintained. The main reasons for this is summarized by Vandegriend [55]:

1. Defect fixing.
2. Changed or new business requirements.
3. Changes to the software execution context.

When researching techniques for building maintainable software, we found that this is a somewhat subjective matter, and that there is no clear theory on how to obtain this software quality factor.

One recurring topic is to write readable code [18, 56], Bloch [18] emphasizes the use of generally accepted naming conventions, and using logical naming of classes, variables and methods. When designing for maintainability, one of the goals is that developers should be able to modify code written by others, and this is simplified by adhering to conventions and standards.

The use of automated testing is a factor to improve modifiability. Automated test suites does not only serve as a safety net when implementing new features, but also as a form of *documentation* for new developers for understanding the work flow and intentions of the system being tested [56].

Riley has written an article named “The Four Pillars of Maintainable Software” [61], where he points out the most important factors for writing maintainable software.

- Pillar 1: Keep it simple, stupid (KISS¹).
- Pillar 2: You ain’t gonna need it.
- Pillar 3: Don’t repeat yourself (DRY²).
- Pillar 4: Stay organized.

Riley describes that maintainability is driven by soft factors—factors that often cannot be quantified or easily measured. Pillar one of Rileys four pillars urge simplicity. You should always strive for simplicity to every aspect of your product, from design, to architecture and implementation.

Pillar two tells us that we should not implement features just in case we should need them later on. Only what is needed at the moment should be implemented, and with the least effort possible. One should always ask if the functionality really is needed for the application to perform its tasks properly.

The next pillar emphasizes the use of design patterns and recommends refactoring of repeated tasks in the program. And finally, pillar four says *stay organized*. Organization should be in all parts of the project, both in processes and code structure.

Code Smell was a term introduced in an essay by Beck and Fowler published as chapter 3 in the book “Refactoring: Improving the Design of Existing Code” [27]. Deciding

¹The acronym was first coined by Kelly Johnson.

²The principle has been formulated by Andy Hunt and Dave Thomas in their book “The Pragmatic Programmer” [30].

whether or not code falls into this category is, however, a subjective matter depending on the developer, programming language and development methodology.

A quick reference guide to “code smells” was developed by *Industrial Logic* [57]. Reducing “code smell” increase readability and thereby maintainability. The “code smells” we have focused on when developing our solution is summarized in Table 3.5.

Code Smell	Mitigation
<i>Long Method</i> : Fowler and Beck means that short methods are superior to large methods, and have some reasons for this. The most important one is sharing of logic. Two long methods may contain duplicated code, and by dividing methods into defined areas of responsibility, and keeping this responsibility at a minimum, duplications may be mitigated.	Extract method, compose method, introduce parameter object.
<i>Large Class</i> : Too many instance variables, often indicate that a class is trying to do too much, and in general too large classes contain too much responsibilities	Extract class, extract subclass, extract interface.
<i>Comments</i> : Comments are described by Fowler and Beck as used to cover existing “smell”, and they advice trying to rename variables, classes and methods to make comments superfluous.	Rename method, extract method, introduce assertion.

Table 3.5: Code Smells and mitigations.

When we were developing our solution, we put a lot of effort into making the code as readable and understandable as possible. This meant that we avoided writing comments as much as possible, and would rather extract the unreadable code and place it in a method with a descriptive name. Consider Listing 3.1:

```
1 if (classesInProject.get(variable.getType()) != null) {  
2     ...  
3 }
```

Listing 3.1: Code smell example.

It is not immediately clear what the *if* expression means. However, if we extract a method, like in Listing 3.2, the *if* expression becomes more comprehensible.

```
1 if (isClassInProject(variable)) {  
2     ...  
3 }  
4 ...  
5 private boolean isClassInProject(ReferenceType variable) {  
6     return classesInProject.get(variable.getType()) != null;  
7 }
```

Listing 3.2: Code smell mitigated.

The alternative to using method names for making the code readable is using comments. We also made an effort of keeping the number of *Long Methods* and *Large Classes* to a bare minimum.

3.3.5 Software and Testing

We have previously stated that our thesis has a special emphasis on unit tests, but what about all the other types of tests? It is normal to separate software tests with respect to the scope of the test. On the lowest level—and closest to implementation—we find the unit tests. The V-Model [24] of software development divides tests into *development tests*, *system tests* and *acceptance tests*.

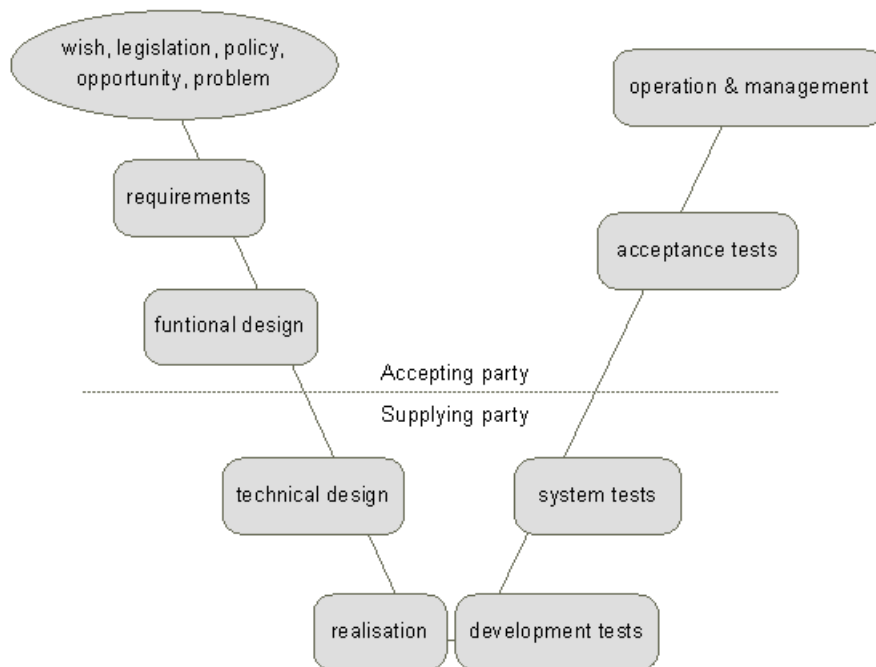


Figure 3.5: The V-Model of software testing.

The unit and integration tests (development tests) are the tests closest to the developer, and is normally written by the developer during implementation. These are also commonly fully automated. The focus of this thesis is on unit and integration tests, partially because system and acceptance tests are not always fully automated, and partially because such tests are not always performed by the developer itself. Pritest

is a tool for the developer to use during development. When using agile development methodologies, the V-Model can be applied for each iteration of development, but normally only using the development test phase; unit and integration tests [24].

In the specialization project we decided to use the frameworks JUnit for unit testing and Cucumber for integration testing [1]. We found that the next version of Cucumber would comply with JUnit, so we only needed to make support for JUnit in our solution—Pritest, and integration tests written in Cucumber will automatically be supported in the next update of Cucumber. We also found that integration tests are often written in JUnit as well, and Cucumber is often used for writing automated system and acceptance tests [25]. Thus, Pritest would support most of the V-Model test scopes with the next update of Cucumber.

From the survey (Section 3.2) we see that integration testing is generally more time consuming than unit testing. The reason for this is that integration tests test interaction between modules and systems, and normally include establishing connections between components. However, we conducted a small benchmark case study on this field as well. We collected a few open source projects (Table 3.6) containing a lot of unit tests, and compared the average test run time to the integration tests in our *pritest-server* module. To see a general tendency of time consumption of the two test scopes, we simply calculated the average run time value of all the unit tests, and the average run time value of our integration tests. Table 3.6 present the projects we found to represent the unit tests.

Project Name	Number of Unit Tests	Run Time
apache-commons-codec	380	9 seconds
apache-commons-lang3	1850	20 seconds
maven-core	236	48 seconds
Total Tests	2466	77 seconds
Average test run time		0.03 seconds

Table 3.6: Unit test benchmark projects.

In *pritest-server* we only have a total of three integration tests. They ran in a total of 11 seconds, resulting in an average test run time of 3.66 seconds.

Unit tests	Integration tests
0.03 seconds	3.66 seconds

Table 3.7: Unit test vs. integration test run time.

Based on this we believe that the average run time for an integration test is substantially larger than for a unit test, as suspected. We selected several types of projects when collecting our sets of unit tests, but the results should not be generalized from this comparison. Still, it seems to be well known among software developers that integration tests take longer time to run than unit tests, but it is not always a clear distinction between the two test types.

3.3.6 Java Source Code Analysis

To be able to implement techniques such as total and additional function coverage, we need to have a library capable of analyzing code. The specific requirements are as follows: isolating the individual classes and their member methods and fields, finding references for each method and method calls to other classes implemented in the development project of study. We have looked at the following tools:

Byte Code Engineering Library (BCEL)

BCEL is a library for static code analysis, and dynamic creation and manipulation of java byte code [50]. Since BCEL uses byte code in its analysis of code, the source code must be compiled first. The library can parse java class files and retrieve methods and fields; however, BCEL is seemingly unable to give information about method calls within the member methods of a class. Therefore, BCEL is not suitable for the function coverage techniques.

Eclipses ASTParser

Eclipse's ASTParser class is used by the Eclipse IDE to generate abstract syntax trees (AST) from java source code [51]. These ASTs are used by Eclipse to provide functionality such as code completion and syntax highlighting. The ASTParser is found in the package *org.eclipse.jdt.core.dom*.

To give developers the possibility of adding functionality to the parser, ASTParser implements the *visitor pattern* (Figure 3.6). The visitor pattern is a behavioral design pattern—like the strategy pattern. The library exposes a set of *visit* methods

which are triggered at certain points throughout the parsing of a class: e.g. there are one visit method that is triggered at each class declaration, and one that is triggered at each method call. Developers can then add functionality by implementing a subclass of a visitor and overriding the desired visit methods. Objects that require extendable behavior can then be given an implemented visitor via the `accept` method.

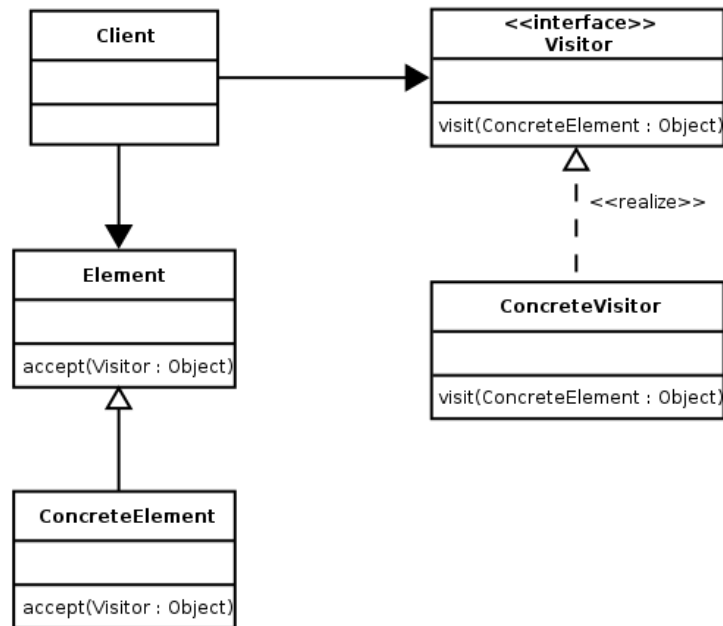


Figure 3.6: The visitor pattern.

The drawback of the Eclipse ASTParser is that it can only be used within an Eclipse plugin. Since our solution is not an Eclipse plugin, ASTParser cannot be employed.

JavaParser

The *JavaParser* is a parser with similar capabilities as the Eclipse ASTParser; that is, AST generation and visitor support [52]. The project is licensed under the LGPL license, and is maintained by Júlio Vilmar Gesser.

According to the project homepage, the main features of JavaParser are that it:

1. is light-weight,
2. has good performance,
3. is easy to use,
4. can modify and build ASTs from scratch.

The three first points are of special importance to us, as the technique using the parser must be quick so as not to offset the gain we get by prioritizing the test cases. The fact that our thesis project is operating on a limited time span also makes it all the more important that the libraries and frameworks we use are easy to use.

3.3.7 Java Git Libraries

In the specialization project [1], we identified an area of improvement regarding technique *Code Changes* (Section 4.3.2) for retrieving prioritization lists. Technique *Code Changes* is concerned with using the version control system GitHub [37] for prioritizing the test cases to be run. GitHub sends a message to Pritest after receiving a push commit from a developer, whereas Pritest records this, and the technique *Code Changes* uses these recordings to run the most recently edited classes' tests first.

However, in most cases there will exist some local classes that Pritest already know about, that have been edited, but not yet committed—thus Pritest does not know that they have been edited yet. This is handled by a new technique called *Local Code Changes* (Section 4.4.1).

To be able to implement this feature, we needed a Java Library for detecting the altered classes locally by using the equivalent command to `git status`. This command returns a list of files that is ready to be committed (the files that have been edited and added). We found only *one* Java Library for this task: JGit [53]. The figure below present all the common git commands that is available in this Java library.

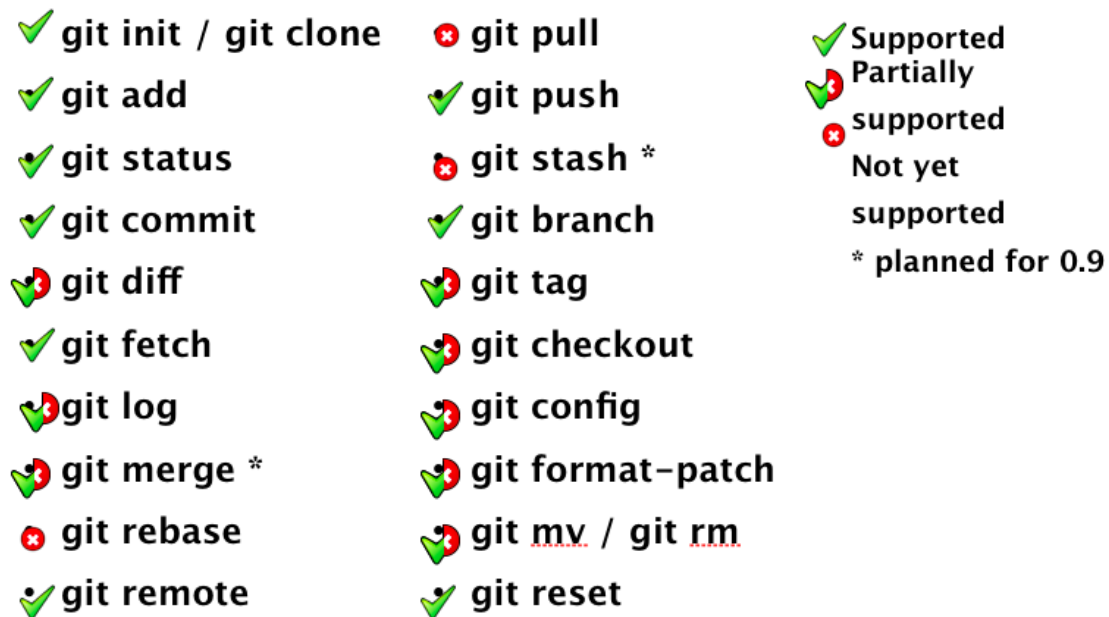


Figure 3.7: Supported commands by JGit library

Note that this illustration displays the features supported by version 0.9, and that the library is now at version 0.12.1 (May 18th 2011). No diagram was found for the present version of the library, but nevertheless the `git status` command—which we

will need—is supported in both versions. Since this was the only library we found for git support in Java, and it supported the simple operation we needed, we chose this one for further implementation of the technique.

Chapter 4

Own Contribution

The resulting product from this thesis and the previous specialization project is a system called Pritest. Pritest is a tool that provides recording of test runs and code changes for a project under development, analysis of recorded data, and generation of prioritized lists of test cases. As developers are about to run the project's tests, the Pritest JUnit runner contacts Pritest server to retrieve the list of tests ranked according to their likelihood of failing. The solution uses several techniques for calculating the prioritized lists. These are evaluated in Chapter 5. The plugin can also be run in offline mode, without using the Pritest server, rather using local prioritization techniques like *Local Code Changes*, *Total Method Coverage* and *Additional Method Coverage*. A big-O analysis will be performed on the techniques in Section 4.7.

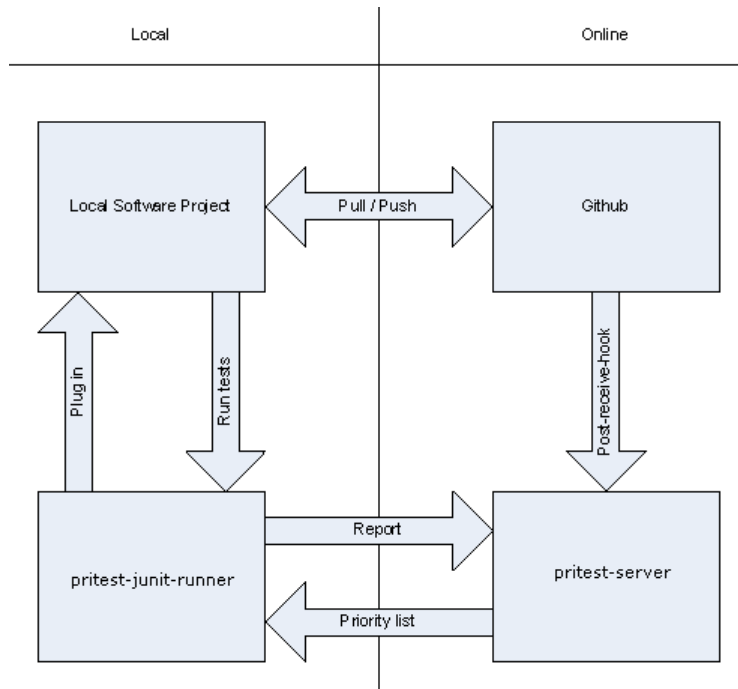


Figure 4.1: Overview of Pritest.

Figure 4.1 displays the overall architecture of Pritest and its modules. *pritest-server* is the online service of Pritest, and is responsible for retrieving reports from a local project and sending prioritized lists to requesters. The *pritest-junit-runner* is a Maven [38] plugin to be used in local Maven projects on a developer’s computer when developing software. The runner is responsible for running the project’s tests. Before this, it contacts the *pritest-server* to get the list of prioritized tests, and afterwards it sends a report from the test run to *pritest-server*.

A lot of time was spent developing the tool, and designing our prioritization techniques. To illustrate this, we have summarized some properties of the Pritest modules we developed in Table 4.1. All the metrics in the table are based on source code, excluding comments and test code such as unit and integration tests written while developing the modules.

	pritest-server	pritest-core	pritest-junit-runner
SLOC (Source Lines Of Code)	795	516	1847
Packages	4	2	8
Classes	20	10	46
Methods	56	112	145

Table 4.1: Pritest modules properties.

Another part of the Pritest architecture is the use of the online version control host Github [37], which is required by the prioritization technique *Code Changes*. The developer can configure his Github account to forward a “post-receive-hook” to a Pritest server after code has been pushed to the repository. The information is sent in the JSON format [42] to Pritest, and stored for future use. In addition, we implemented some local test prioritization techniques (Section 4.4) that construct prioritization lists directly in our *pritest-junit-runner* (the plugin).

4.1 The Pritest Tool

The main concepts of Pritest were partially implemented during the specialization project, and the system was further improved during our work with the master thesis. Several areas of improvement were identified in retrospect of the specialization project, and the improvements are described in detail in this chapter. For a full view of the Pritest architecture, technology and reasoning behind the applied technology, we refer to the specialization project report [1]. Table 4.2 present the chosen technologies for the different areas of our application.

Area	Chosen Technology
Programming Language	Java [39]
Build Server	Hudson [40]
Project Management Tool	Maven [38]
Web Server and Web Service	Jersey [41]
XML and JSON Parsing	JAXB [42]
Analytics and Statistics	Sonar [43]
Issue Tracking	Github [37]
Version Control	Git [44]
Test Frameworks	JUnit and Cucumber [45], [46]
Storage	MongoDB [47]

Table 4.2: Overview of applied technology.

Some low-level improvements have been done this semester, most of them regarding Java implementations, and improvements of the Pritest code. The improvements are based on the post-mortem analysis from the last report, and from the research chapter in this thesis. One improvement is the use of dependency injection. We made use of

the `PropertiesHolder` class and the `Configuration` class in Java. By instantiating the `Configuration` class and passing a `PropertiesHolder` object as parameter, we set the configuration in a file that is read by `PropertiesHolder` at instantiation.

```
1 private PropertiesHolder() {
2     try {
3         FileInputStream inStream = new FileInputStream("./pritest.
4             properties");
5         properties = new Properties();
6         properties.load(inStream);
7         inStream.close();
8     } catch (FileNotFoundException e) {
9         e.printStackTrace();
10    } catch (IOException e) {
11        e.printStackTrace();
12    }
13 }
```

Listing 4.1: The constructor of the `PropertiesHolder` class.

Listing 4.1 shows the implementation of the `PropertiesHolder` constructor. As we can see, the file “pritest.properties” is read, and used as a configuration file.

```
1 public static PropertiesHolder getInstance() {
2     if(instance == null){
3         instance = new PropertiesHolder();
4     }
5     return instance;
6 }
```

Listing 4.2: The `getInstance` method of `PropertiesHolder`.

The object of `PropertiesHolder` is a singleton object, and the implementation in

Listing 4.1 assures that only one object of the class is made.

Listing 4.3 shows how the configuration is used in practice. The example is from the instantiation of a Mongo Database [47], based on the information found in the instance of the `PropertiesHolder`.

```
1 private MongoDBProvider() throws MongoException, UnknownHostException {  
2     Configuration config = new Configuration(PropertiesHolder.  
3         getInstance());  
4     Mongo mongo = new Mongo(config.getDatabaseURL(), config.  
5         getDatabasePort());  
6     db = mongo.getDB(config.getDatabaseName());  
7 }
```

Listing 4.3: Using the configuration file.

As an improvement we also decided to write a new custom JUnit Runner module that works as a plugin to a Maven project. We called our new runner *pritest-junit-runner*. The runner we used in the specialization project was somewhat inefficient and bloated with a lot of unnecessary code re-used from the surefire-maven plugin [48]. As a comparison the old runner module contained 1337 source lines of code (SLOC), while the new runner contains 294 SLOC¹. The results from a benchmark study of the new runner compared to the old one is discussed in Section 6.7.

The new runner is also improved by implementing the strategy pattern (Section 3.3.3). This pattern consist of an *online* strategy, and an *offline* strategy. The prioritization lists generated by the *pritest-junit-runner* will depend on the internet connection and the connection to Pritest.

¹Source code lines of the actual runner, excluded the implementation of the local prioritization techniques located in this module.

The new runner localizes the folder where the unit tests are placed in a Maven project, and checks if the classes in that folder actually are JUnit-4 test cases. This is done by looking through the classes for the presence of the `@Test` annotation, using the method `isAnnotationPresent`² which takes an annotation as argument. All test cases in JUnit-4 contain the annotation `@Test` before each test method.

The runner also detect test cases where the annotation `@RunWith` is present. This annotation indicates that this test case should be run with another custom JUnit runner, and it is therefore excluded from our further prioritization evaluation. Ideally, our runner should have a spawn mechanism for the results from this test case, and measures should be recorded for future prioritization of such test cases. This improvement is described in Section 7.2.

The tests are organized by our custom runner. The classes that exist locally, but are not found by the runner on the *pritest-server* are placed first in the list³ (Figure 4.2). The reason for this is that these classes are probably newly created (since Pritest server does not know of them yet), and are most likely to fail.



Figure 4.2: Priority List.

²Present in the `java.lang.reflect.Method` class.

³This applies only for our online techniques (Section 4.3).

However, we do not fully know that local test cases are the ones most likely to fail. It could be possible to maintain a local database of test cases that are sent to Pritest in advance of a test run, and is equally evaluated on *pritest-server* to be merged with the test cases present in *pritest-server*. This idea is further discussed in Section 6.4.

Figure 4.3 illustrates the classes in the new custom Junit Runner implementation.

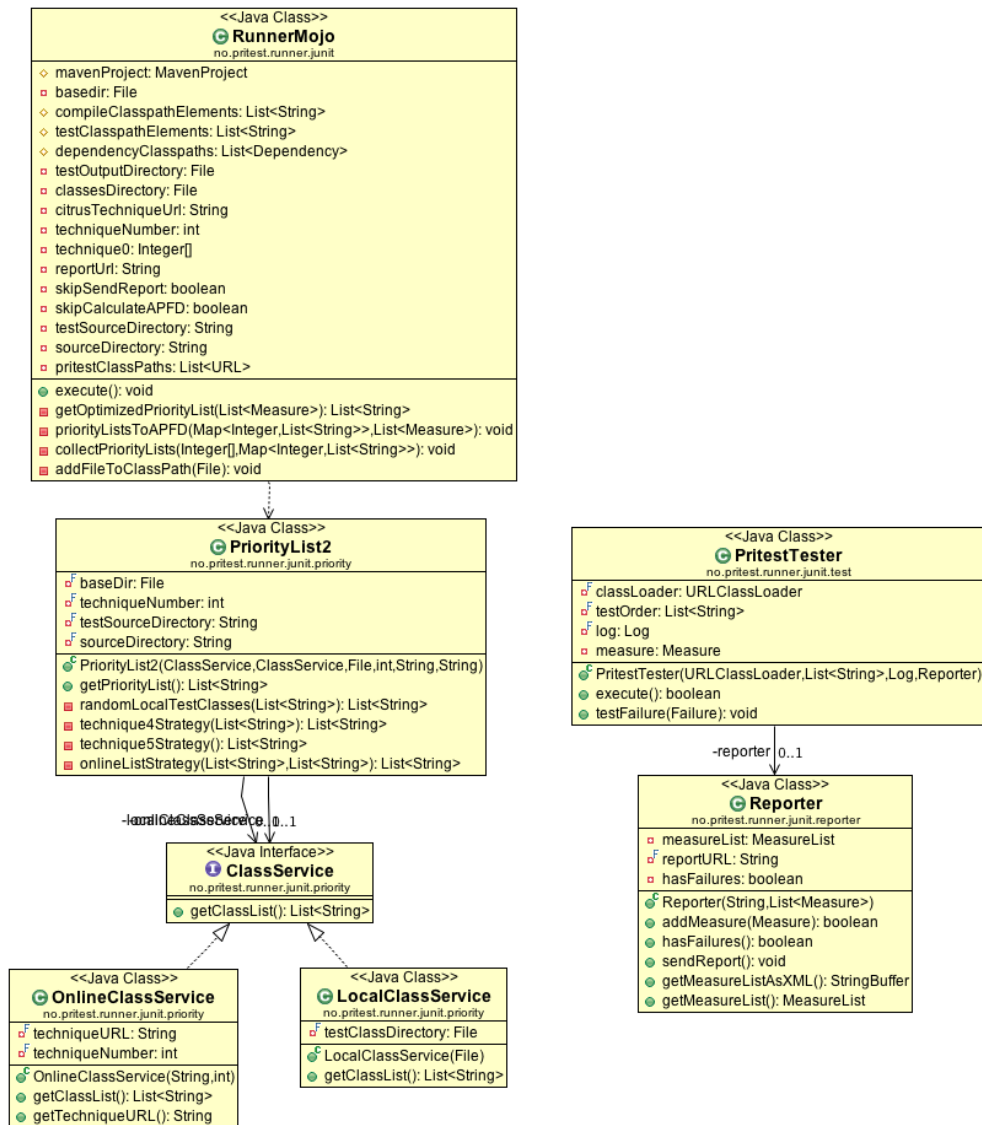


Figure 4.3: UML Class diagram of the new pritest-junit-runner.

To obtain a maintainable solution we applied principles both from “The Four Pillars of Maintainable Software” [61], and “Clean Code: A Handbook of Agile Software Craftsmanship” by Martin [26]. He introduced principles such as good naming conventions regarding classes, interfaces, methods and variables, following policy throughout the entire project, organizing modules in a logical manner, keeping it simple and not implementing features “just in case” we may need them later. Our representation of maintainability relies a lot on the *readability* of our code. Also the configuration file patterns presented previously is part of our effort to attaining maintainability. We developed and applied the maintainability policy described below:

- Naming conventions:

Classes must start with an upper-case letter, and by camel-casing⁴ every new word in the method name. Naming classes to tell what they do is important.

Methods must start with a lower-case letter, and by camel-casing every new word in the method name. The method names should be descriptive, and tell what they do.

Variables must start with a lower-case letter, and by camel-casing every new word in the variable name. The variable name must be descriptive enough to tell what the variable actually holds.

Examples: `TotalMethodCoverage.java`, `getPriorityList()`, `getMeasureListAsXML()`, `fileName`, `listToAddStringTo`.

⁴Also known as medial capitals - the practice of joining words in phrases without spaces, but with the initial letter of each element capitalized. E.g. *thisTextIsCamelCased*.

- Keeping it organized:

One important factor for high maintainability is to keep the project organized, both on a process level and code level. By dividing our classes and implementation into logically named packages, we improved the organization of the code.

Example:

```
|___localprioritization
| |___model
| | |___ClassType.java
| | |___MethodCall.java
| | |___MethodDecl.java
| | |___ReferenceType.java
| |___TotalMethodCoverage.java
| |___visitor
| | |___ClassOrInterfaceDeclarationVisitor.java
| | |___CompilationUnitVisitor.java
| | |___FieldVisitor.java
| | |___MethodCallVisitor.java
| | |___MethodDeclarationVisitor.java
```

Figure 4.4: Organization of packages in our pritest-junit-runner.

- Unit testing conventions:

To address our goal concerning maintainability, we decided that the words of the test case names should be separated with underscores instead of using capital letters. This was a tip we got from our external supervisor in order to write readable unit tests. This is merely a matter of taste, but based on our own experience, this makes test method signatures more readable.

Example:

`should_support_test_cases_not_covering_any_methods()` instead of
`shouldSupportTestCasesNotCoveringAnyMethods()`

We use standard JUnit4 naming conventions concerning annotation.

4.2 Implementing our Prioritization Techniques

In this section, we will present all the techniques implemented for prioritizing test cases. Most of the techniques are designed by us (Table 4.3). In addition—based on research (Section 3.1)—we implemented two techniques designed by Rothermel et al. (*Total Method Coverage* 4.4.3 and *Additional Method Coverage* 4.4.2). We will run an experiment to compare the techniques (Chapter 5).

We also developed a set of control techniques (Section 4.6), to be used in the experiment. These prioritize the test cases in *original*, *random* and *optimal* order.

Our techniques	Techniques designed by others
Counting Failing Tests	Total Method Coverage
Code Changes	Additional Method Coverage
Local Code Changes	
Local Code Changes with Failure Counting	

Table 4.3: Techniques summary.

For choosing the design of our prioritization techniques, we performed several brain storming meetings and discussions, with the intention of inventing concepts for high precision techniques. The discussions and decisions were based on our own experience, the industrial survey conducted in Section 3.2 and with help from our external supervisor.

4.3 Online Prioritization Techniques

4.3.1 Counting Failing Tests

This technique was added to Pritest because of its simplicity. The rationale is that a quick technique giving suboptimal prioritizations can be better than a slow technique giving optimal prioritizations.

The Algorithm

With this technique, test cases are prioritized descendingly by the number of test failures they have caused in the past (Procedure 1).

Procedure 1 Prioritize based on number of failures

Input: A list of test cases t_i in T , a list of the numbers of failures for each test case f_i in F

Output: A list of prioritized test cases T'

An empty PriorityQueue pq

An empty list of test cases T'

$T' \leftarrow null$

for $t_i \in T$ **do**

$pq.insertWithPriority(t_i, f_i)$

end for

while pq is not empty **do**

$t \leftarrow pq.pullHighestPriority()$

$T'.insert(t)$

end while

return T'

The Implementation

As this technique must store the number of test failures for each test case over time, some sort of persistence is needed. Currently we only support MongoDB [47], but adding support for other types is made easy by using the factory pattern.

The implementation of the technique itself can be seen in Figure 4.7 below.

```
1 public class TestOrderResource {
2     ...
3     private List<String> method1() {
4         TestDataDAO tdDAO = DAOFactory.getDatabase().getTestDataDAO();
5         List<TestData> tests = tdDAO.getList();
6         Collections.sort(tests);
7
8         List<String> testNames = new ArrayList<String>();
9         for (TestData test : tests) {
10            testNames.add(test.getClassName());
11        }
12
13        return testNames;
14    }
15    ...
16 }
```

Listing 4.4: The technique Counting Failing Tests.

All sorting is done by the method `sort(List list)` in the utility class `Collections` found in the standard Java library. This requires that the sorted objects implement the `Comparable` interface, as shown in Listing 4.7.

```
1 public class TestData implements Comparable<TestData> {
2     private String className;
3     private int fails;
4
5     ...
6     // Constructors, getters and setters
7
8     @Override
9     public int compareTo(TestData arg) {
10        return this.fails - arg.getFails();
11    }
12 }
```

Listing 4.5: The TestData class.

A reporting mechanism is also necessary, since the results from the test run must be stored for future use. This is done by sending a HTTP POST [59] to the Pritest server with the results formatted as XML [60]. Upon receipt, the results are used to update the database.

```
1 @Path("/measure")
2 public class MeasureResource {
3     @POST
4     @Consumes({"application/xml"})
5     public Response post(MeasureList measures) {
6         ...
7         return Response.ok().build();
8     }
9 }
```

Listing 4.6: The MeasureResource REST interface.

4.3.2 Code Changes

This technique utilizes data about changes made to the code base when prioritizing test cases. A corollary of this is that code change data should be available for every test case for this technique to function properly. This again means that the technique might give better results when being used from the beginning of a project, than if it is introduced in the later stages of the project's lifecycle (Section 5.4.7).

The Algorithm

As can be seen in the algorithm below (Procedure 2), this technique takes as its input a set of test cases and data about when they last were affected by a change, and then prioritizes them descendingly by the date of the change. By being affected by a change we mean that it does not matter whether it is the test case itself that is changed, or the class being tested by it.

Procedure 2 Prioritization algorithm based on code changes.

Input: A list of test cases t_i in T , a list of the most recent code changes c_i in C for each class

Output: A list of prioritized test cases T'

An empty list of test cases T'

$C' \leftarrow \text{sortedDescendinglyByDate}(C)$

for $c'_i \in C'$ **do**

$T'.\text{insert}(t_i)$

end for

return T'

The Implementation

Like the *Counting Failing Tests* technique in Section 4.3.1, this one requires a database, this time for storing the time and date a test case was last affected by a change. The prioritization can be seen in Listing 4.7.

```
1 public class TestOrderResource {
2     ...
3     private List<String> method3() {
4         ChangeDataDAO cdDAO = DAOFactory.getDatabase().getChangeDataDAO();
5         List<ChangeData> changes = cdDAO.getList();
6         Collections.sort(changes);
7
8         List<String> testNames = new ArrayList<String>();
9         for (ChangeData change : changes) {
10            testNames.add(change.getSource());
11        }
12
13        return testNames;
14    }
15 }
```

Listing 4.7: The Code Changes technique.

This technique also implements the `Comparable` interface to make it easier to sort the test cases (Listing 4.8).


```
1 public class ChangeData implements Comparable<ChangeData> {
2     private String source;
3     private Date lastChange;
4
5     ...
6     // Constructors, getters and setters
7
8     @Override
9     public int compareTo(ChangeData arg) {
10        return lastChange.compareTo(arg.getLastChange());
11    }
12 }
```

Listing 4.8: The ChangeData class.

In this case, the reporting must be done by the version control system, and a report must be sent every time someone commits source code to the central repository. With Git, this can be done with a *post-receive hook*⁵. The `git push` command sends local changes to a remote repository. GitHub provides its own post-receive hook, which given a URL sends information about the *push* to the specified destination. GitHub is an online service that provides public Git repositories for free, and private repositories for a fee. The information sent by the hook includes lists of added, removed and modified files. Upon receipt this information is converted to a `Change` object (Listing 4.9).

⁵A post-receive hook is a little program that is triggered each time someone *pushes* code to the remote repository.

```
1 public class Change {
2     public String after;
3     public String before;
4     public no.citrus.restapi.model.Repository repository;
5     public String ref;
6     public String compare;
7     public boolean forced;
8     public Pusher pusher;
9     public List<Commit> commits;
10    // ... Constructors, getters and setters
11 }
```

Listing 4.9: The Change class.

Each **Change** consists of—among other things—a list of the commits made by the developer (Listing 4.10).

```
1 public class Commit {
2     public List<String> added;
3     public String id;
4     public String message;
5     public List<String> modified;
6     public List<String> removed;
7     public Date timestamp;
8     public String url;
9     public Author author;
10    // ... Constructors, getters and setters
11 }
```

Listing 4.10: The Commit class.

A **Commit** object has lists of the removed, added and modified files.

4.4 Local Prioritization Techniques

4.4.1 Local Code Changes

```
EnterCommandHere---git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   pom.xml
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .classpath
#       .project
#       .settings/
#       target/
no changes added to commit (use "git add" and/or "git commit -a")
EnterCommandHere---
```

Figure 4.5: Using the git status command in a bash command.

Figure 4.5 shows a typical result of a `git status` command. We implemented a technique for running the test cases of the most recent modified and untracked (not yet committed) classes. This was implemented using the JGit [53] library as mentioned in the research (Chapter 3).

The idea is to make a list of three sequential parts: untracked classes, modified classes and all the local test cases in the project. The untracked classes are placed first in the final prioritization list, then the modified classes, and at the end all the remaining local test cases that have not yet been added to the list. The process is visualized in Figure 4.6. Thus, this is our first technique that does not require contact with Pritest. It runs locally just using the *pritest-junit-runner* as a plugin to Maven.

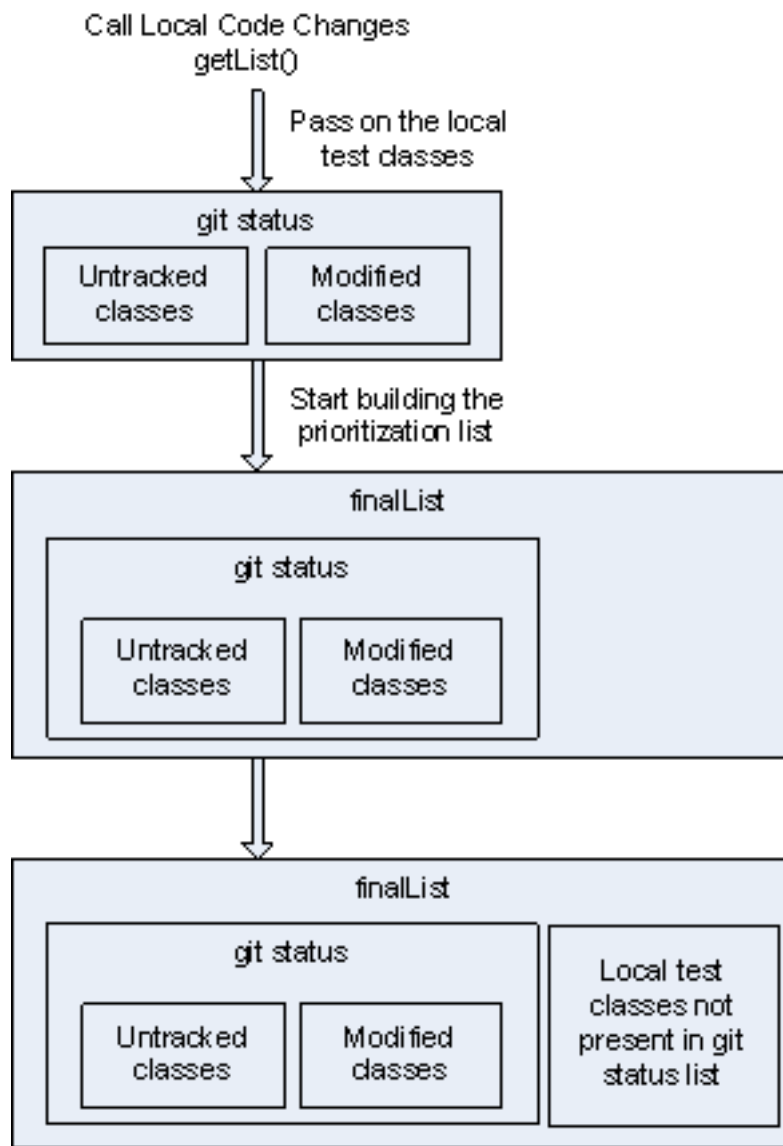


Figure 4.6: Technique Local Code Changes prioritization list selection process.

The Algorithm

Procedure 3 addTestCaseToList(c, L)

Input: A class c , a list of test cases L

Output: A list of test cases L

if c is a test case **then**

$L.add(c)$

else

$t \leftarrow$ test case testing c

if $t \neq null$ **then**

$L.add(t)$

end if

end if

return L

Procedure 4 Prioritize based on local changes

Input: A list of classes C **Output:** A list of prioritized test cases T' An empty list of test cases T' An empty list of untracked test cases U An empty list of modified test cases M An empty list of unchanged test cases R **for** $c_i \in C$ **do** **if** c_i has not yet been added to the remote repository **then** $U \leftarrow \text{addTestCaseToList}(c, U)$ **else if** c_i has been modified **then** $M \leftarrow \text{addTestCaseToList}(c, M)$ **else** $R \leftarrow \text{addTestCaseToList}(c, R)$ **end if****end for** $T'.\text{append}(U)$ $T'.\text{append}(M)$ $T'.\text{append}(R)$ **return** T'

The Implementation

The implementation of this technique will currently only support Git [44]; though ideally it should be version control system *agnostic*. It should not care whether it is Git, SVN or any other version control system. Every call to JGit is made in the method `callGitStatus()`. JGit starts by finding the local Git repository, and proceeds by retrieving the *status*. This status contains, among other things, one list

for *untracked* files and one for *modified* files. Both of these lists are then traversed, and their content added to a list of test cases, assuming they are Java classes. If they are not test cases, “Test” is appended to the class name. We do this as many developers name their test cases like this: “[class name]Test.java”, where “class name” is the name of the class being tested by the given test case. If no test case with that name is found, it will be ignored when running the test cases in the list. The remaining test cases unaffected by any change, are added to the list by iterating through all local test cases, and adding them if they are not already added as *untracked* or *modified*.

The direct mapping from class name to test case by simply adding “Test” at the end, is a potential shortcoming to the implementation of this technique. In some cases, altered classes would have an effect on test cases that are not directly coupled to the class itself, and then this technique would perform poorly. An improvement to this is discussed in Section [6.4.3](#).

```
1 public class Technique4Ranker {
2
3     private List<String> localTestClasses = new ArrayList<String>();
4     private File basedir;
5
6     // Constructor
7
8     public List<String> getTechnique4PriorityList() throws
9         NoWorkTreeException, IOException {
10        List<String> gitStatusList = new ArrayList<String>();
11        gitStatusList = callGitStatus();
12        List<String> finalList = new ArrayList<String>();
13
14        finalList.addAll(gitStatusList);
15
16        for (String localTestClass : localTestClasses) {
```

```
16     if (!finalList.contains(localTestClass)) {
17         finalList.add(localTestClass);
18     }
19 }
20 return finalList;
21 }
22
23 public List<String> callGitStatus() throws NoWorkTreeException,
24     IOException {
25     List<String> gitStatusList = new ArrayList<String>();
26
27     File repoPath = new File(basedir.getAbsolutePath() + "/.git");
28     RepositoryBuilder repoBuilder = new RepositoryBuilder();
29     Repository repo = repoBuilder.setGitDir(repoPath).build();
30     Git git = new Git(repo);
31     Status status = git.status().call();
32
33     for (String untrackedFile : status.getUntracked()) {
34         addIfJavaSuffix(untrackedFile, gitStatusList);
35     }
36     for (String modifiedFile : status.getModified()) {
37         addIfJavaSuffix(modifiedFile, gitStatusList);
38     }
39     return gitStatusList;
40 }
41
42 private boolean addIfJavaSuffix(String fileName, List<String>
43     listToAddStringTo) {
44     // Adds the file to the list if it is a java file,
45     // appends "Test" if it is not a test case
46 }
```

Listing 4.11: The Technique4Ranker class; prioritization based on local changes.

The technique can be called like this, as in the `PriorityList2` class:

```
1 public class PriorityList2 {  
2     ...  
3     private List<String> technique4Strategy(List<String> localTestClasses)  
4         throws NoWorkTreeException, IOException {  
5         Technique4Ranker t4 = new Technique4Ranker(localTestClasses, this.  
6             baseDir);  
7         return t4.getTechnique4PriorityList();  
8     }  
9     ...  
10 }
```

Listing 4.12: Calling the technique.

4.4.2 Additional Method Coverage

According to Rothermel et al., techniques using feedback tend to give the best results more often than other techniques [13]. Additionally, they argue that techniques having a low-level granularity, such as statement coverage, tend to be more efficient. However, implementing techniques operating at a low level will probably require more code-writing, and we will therefore focus on high-level techniques, such as *Additional Method Coverage*.

The Algorithm

Additional Method Coverage iteratively selects the test case that covers the most methods that have not yet been covered. If every statement have been covered by a test case and there are still test cases to prioritize, all statements are set to “not covered”, and then covered by calling *additionalMethodCoverage* recursively with the same statements and the remaining test cases.

Procedure 5 Prioritize based on Additional Method Coverage

Input: A list of test cases T , a list of possible methods M **Output:** A list of prioritized test cases T' An empty list of test cases T' **while** T is not empty **and** every method in S is not yet covered **do** $t \leftarrow \text{testCaseCoveringTheMostMethods}(T, M)$ $T'.\text{insert}(t)$ $M \leftarrow \text{markCoveredMethodsAsCovered}(T', M)$ **end while****if** Every method in M is covered **and** T is not empty **then** set every method in M as not covered $T' \leftarrow \text{additionalMethodCoverage}(T', M)$ **end if****return** T'

An example of *Additional Method Coverage* being used on a test suite is shown below (Figure 4.7). The test suite has 3 test cases and 7 methods. First, test case A will be chosen, as it covers 4 methods versus B 's 3 and C 's 2. Those 4 methods are then marked as “covered”, which leaves B with 1 method. Secondly, C is selected, and then lastly, B is selected.

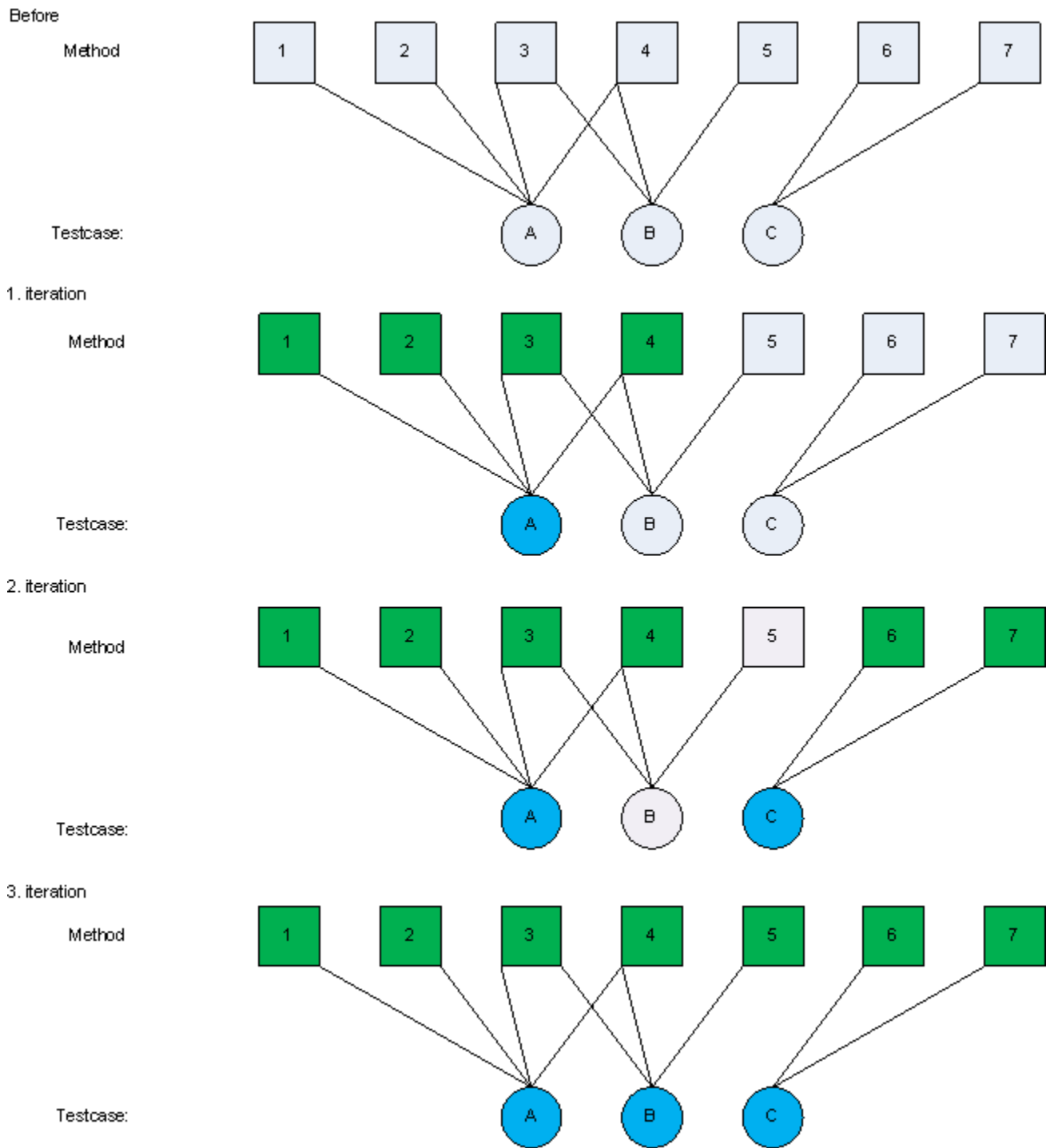


Figure 4.7: Additional Method Coverage illustration

The Implementation

Before the technique can be executed, the source code must be analyzed. That is, each test case must be associated with a set of covered methods. Since we are mainly interested in methods declared in the development project we are currently testing, calls to methods belonging to standard or third party libraries will not be included. The reason for this is that these libraries most likely will have their own test suites, and therefore will be tested separately.

This analysis process is divided into three successive steps:

1. Finding each class (and test case) in the project, along with their field variables and method declarations.
2. For each of the classes discovered in the previous step, and for each of their declared methods: find calls to the methods belonging to this project within that method.
3. For each test case in the project, recursively find the set of methods covered by that test case. E.g. if test case **A** covers method **a()**, and **a()** covers **b()**, then **A** covers both **a()** and **b()**.

The data collected by these three steps are used both by this technique, and *Total Method Coverage* (Section [4.4.3](#)).

The source code of *Total* and *Additional Method Coverage* is located in the package `no.pritest.localprioritization`, with the visitors in `no.pritest.localprioritization.visitor`, model classes in `no.pritest.localprioritization.model` and algorithms in `no.pritest.localprioritization.algorithm`. The visitors are used in step 1 and 2 to extract data from the source code being tested, such as: method calls,

method declarations, variables and class declarations. `no.prietest.localprioritization.algorithm` contains the actual techniques: *Total Method Coverage* and *Additional Method Coverage*, contained in the class `MethodCoverageAlgorithm`. The algorithms in this class have been decoupled from the visitor classes, which requires I/O, to make them more testable and modifiable.

```
1 public class MethodCoverageAlgorithm {
2     ...
3     public static List<SummarizedTestCase> additionalMethodCoverage(
4         Map<String, ClassCover> testSuiteMethodCoverage,
5         Map<String, ClassCover> sourceMethodCoverage) {
6         List<SummarizedTestCase> prioritizedTestCases =
7             sortTestCasesByCoverage(testSuiteMethodCoverage,
8                 sourceMethodCoverage);
9
10        List<SummarizedTestCase> results =
11            new ArrayList<SummarizedTestCase>();
12
13        results.addAll(additionalMethodCoverageHelper(
14            sourceMethodCoverage, prioritizedTestCases));
15
16        return results;
17    }
18
19    private static List<SummarizedTestCase> additionalMethodCoverageHelper
20        (
21        Map<String, ClassCover> sourceMethodCoverage,
22        List<SummarizedTestCase> prioritizedTestCases) {
23
24        List<SummarizedTestCase> results =
25            new ArrayList<SummarizedTestCase>();
26
27        int amountOfCoveredMethods =
28            coveredMethodsInSource(sourceMethodCoverage);
```

```
25
26 while (!prioritizedTestCases.isEmpty() && amountOfCoveredMethods >
27         0) {
28     SummarizedTestCase mostCoveringTestCase =
29         Collections.max(prioritizedTestCases);
30     prioritizedTestCases.remove(mostCoveringTestCase);
31
32     results.add(mostCoveringTestCase);
33
34     Map<String, MethodCover> alreadyCoveredMethods =
35         mostCoveringTestCase.getSummarizedCoverage();
36     amountOfCoveredMethods -= mostCoveringTestCase.coveredMethods();
37
38     markMethodAsCovered(prioritizedTestCases, alreadyCoveredMethods);
39 }
40
41 if (!prioritizedTestCases.isEmpty() && amountOfCoveredMethods == 0)
42     {
43     unMarkCoveredMethods(prioritizedTestCases);
44
45     results.addAll(additionalMethodCoverageHelper(
46         sourceMethodCoverage, prioritizedTestCases));
47     }
48
49 return results;
50 }
```

Listing 4.13: The MethodCoverageAlgorithm class: the Additional Method Coverage part.

```
1 public class MethodCoverageAlgorithm {
2     ...
3     private static List<SummarizedTestCase> sortTestCasesByCoverage(
4         Map<String, ClassCover> testSuiteMethodCoverage,
5         Map<String, ClassCover> sourceMethodCoverage) {
6         List<SummarizedTestCase> prioritizedTestCases = new ArrayList<
7             SummarizedTestCase>();
8
9         Collection<ClassCover> testCaseCollection = testSuiteMethodCoverage.
10            values();
11         for (ClassCover testCase : testCaseCollection) {
12             MethodCoverageSummarizer mcs = new MethodCoverageSummarizer(
13                 sourceMethodCoverage, testCase);
14             Map<String, MethodCover> summarizedCoverage = mcs.
15                getSummarizedCoverage();
16             SummarizedTestCase summarizedTestCase = new SummarizedTestCase(
17                 testCase, summarizedCoverage);
18             prioritizedTestCases.add(summarizedTestCase);
19         }
20
21         Collections.sort(prioritizedTestCases);
22         Collections.reverse(prioritizedTestCases);
23
24         return prioritizedTestCases;
25     }
26     ...
27 }
```

Listing 4.14: The MethodCoverageAlgorithm class: sorting test cases by the number of covered methods.

The implementation for the *Additional Method Coverage* algorithm can be found in Listing 4.13 and Listing 4.14. The method `additionalMethodCoverage()` takes as

parameters the classes in the source code and the test cases, and returns a list of test cases. These classes and test cases are contained within two maps, where each test case and class is represented by a `ClassCover` object. The `ClassCover` contains information about the method coverage of a given class or test case. As described in Section 3.1.2, the technique runs recursively if all methods have been covered and there are still unprioritized test cases. This recursive part of the algorithm is located within the method `additionalMethodCoverageHelper()`.

To analyze the source code we use the *JavaParser* library (Section 3.3.6). Initially, the class `ClassListProvider` will give a list of all the files in a given directory of a given type (e.g. Java files). These files will then be given to `CompilationUnitProvider`, which parses them using *JavaParser* and returns a list of `CompilationUnits`, which are ASTs capable of accepting visitors.

In the analysis process defined above, the `CompilationUnits` will only be used in the first two steps: firstly, in step 1 by `ClassTypeProvider`, and secondly in step 2 by `MethodCoverageProvider`. `ClassTypeProvider`'s responsibility is to retrieve information about field variables and method declarations in each class, and return that information as a list of `ClassTypes`. This information is then used by `MethodCoverageProvider` to produce a list of `ClassCover` objects, which store information about method calls within a given class. The sequence of these method calls can be seen in Listing 4.15, which shows how a map of `ClassCover` objects is retrieved given a path to the source code.


```
1 public abstract class MethodCoverage {
2     ...
3     private Map<String, ClassCover> sourceMethodCoverage;
4     ...
5     private void retrieveClassCoverage(String pathToProjectSource)
6         throws ParseException, IOException {
7
8         List<File> fileList =
9             ClassListProvider.getFileList(
10                new File(pathToProjectSource), new String[] {".java"});
11
12        List<CompilationUnit> compilationUnits =
13            CompilationUnitProvider.getCompilationUnits(fileList);
14
15        ClassTypeProvider classTypeProvider =
16            new ClassTypeProvider(compilationUnits);
17        projectSourceClassTypes = classTypeProvider.getClassTypes();
18
19        MethodCoverageProvider mcp =
20            new MethodCoverageProvider(projectSourceClassTypes,
21                compilationUnits);
22
23        sourceMethodCoverage = mcp.getMethodCoverage();
24    }
25    ...
26 }
```

Listing 4.15: MethodCoverage: getting method coverage.

This coverage information will then be used by `MethodCoverageSummarizer` to find all transitive method calls executed by a given test case. Afterwards, the test cases are ready to be prioritized, as shown in Listing 4.13.

Shortcomings of the Implementation

As mentioned earlier, we implemented our own visitors for extracting method calls from the code being tested. As a result, we would have to consider every syntactic rule in the language if we desired a complete method coverage. Unfortunately, we were—due to time limitations—only able to implement some of the most basic and common rules. These shortcomings also apply for *Total Method Coverage* (Section [4.4.3](#)).

These are the shortcomings we have been able to identify:

- Only calls to methods declared in the class being referenced are discovered. E.g., calls to inherited methods are not discovered.
- Method calls must have an adjoined object. E.g. `a.doSomething()` is discovered, while `doSomething()` is not.
- Method calls with parameters belonging to a subtype of the type referenced in the method declaration. E.g. given a method declaration `doSomething(A a)` and two classes `A` and `B`, where `B` inherits `A`, and an object `b` of type `B`; then the method call `doSomething(b)` will not be found.
- Method calls with a `null` parameter are not found.
- Polymorphism is not supported. Dynamic binding makes it hard to find the correct method calls during a static code analysis. E.g., an abstract class `A`, an object of that type calling a method `a.doSomething()`, and some subclasses `B` and `C` overriding that method. The code analysis is currently done on a per-module basis, i.e. analysis data from one module are not accessible during the analysis of another module, which makes it impossible to support dynamic

binding across modules. For example, if we have a module which updates some database, but the database driver implementation is given by the calling module—and is therefore unknown at compile time by the former module—then method calls to the database driver will not be discovered.

- Dependency injection is not supported, as this requires support for polymorphism.
- Mocking in tests can also present some problems for the code analysis, at least when Mockito [67] is used to generate mock objects at run-time.
- Methods and fields declared within enumerations are not supported.

Implementing analysis support for every syntactic rule regarding method calls in a language might not even be desirable. Code analysis requires time for processing, and if support for too many rules is implemented, the prioritization process might negate any benefit earned by finding failures earlier.

4.4.3 Total Method Coverage

Additional Method Coverage was the primary technique we wanted to implement, but as *Total* and *Additional Method Coverage* have a lot of code in common, we decided to implement this one as well. In fact, *Additional Method Coverage* uses every bit of code that *Total Method Coverage* uses.

The Algorithm

Procedure 6 Prioritize based on Total Method Coverage

Input: A list of test cases T , a list of possible methods M **Output:** A list of prioritized test cases T' An empty list of test cases T' **while** T is not empty **and** every method in S is not yet covered **do** $t \leftarrow \text{testCaseCoveringTheMostMethods}(T, M)$ $T'.\text{insert}(t)$ **end while****return** T'

An illustration of a *Total Method Coverage* run can be seen in Figure 4.8. We have three test cases: A, B and C, where A covers 4 methods, B covers 3 and C covers 2. Therefore, the test suite execution order will be: A, B and C.

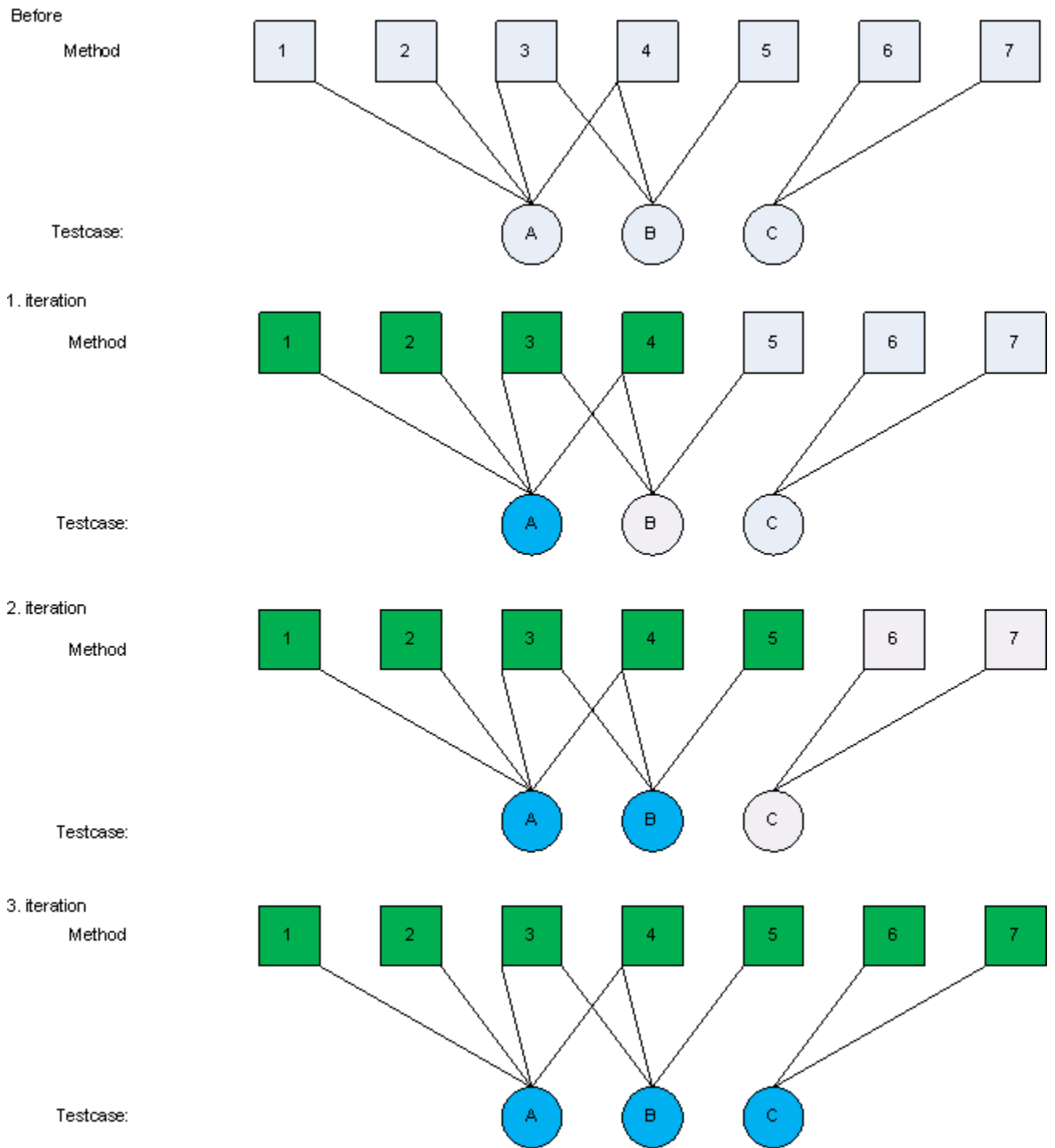


Figure 4.8: Total Method Coverage illustration

The Implementation

Listing 4.16 shows the method in `MethodCoverageAlgorithm` that returns a list prioritized based on this technique.

```
1 public class MethodCoverageAlgorithm {
2     ...
3     public static List<SummarizedTestCase> totalMethodCoverage(
4         Map<String, ClassCover> testSuiteMethodCoverage,
5         Map<String, ClassCover> sourceMethodCoverage) {
6
7         return sortTestCasesByCoverage(testSuiteMethodCoverage,
8             sourceMethodCoverage);
9     }
}
```

Listing 4.16: The `MethodCoverageAlgorithm` class: the Total Method Coverage part.

4.5 Hybrid Prioritization Techniques

A hybrid is a technique that uses the principles of at least two other techniques to prioritize test cases.

4.5.1 Local Code Changes with Failure Counting

When considering Figure 4.6, you can expect to get the desired test cases from the `git status` command, either from the untracked files or from the modified ones. If that is not the case (e.g. if you are looking for an old test to see if it still passes after the recent code changes), it would improve the prioritization to rank the *local test cases* internally. This technique builds the prioritization list by putting untracked files first in the queue, then modified files, followed by the remaining test cases present in the project, which are ranked internally according to technique *Counting Failing Tests* (Section 4.3.1).

This will enhance the prioritization in precision, but not necessarily when considering efficiency (since the technique contacts our online service: *pritest-server*). This will be further investigated in the experiment in Chapter 5. Figure 4.9 illustrates the prioritization list retrieved by this technique.

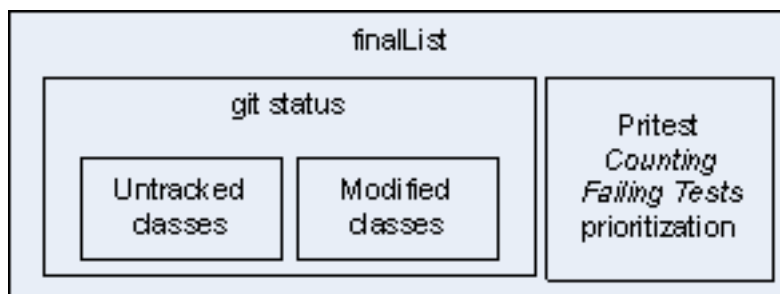


Figure 4.9: Illustration of hybrid technique prioritization list.

The Algorithm

The algorithm for this technique is quite similar to the algorithm for *Local Code Changes* (Procedure 4), except that the list of unchanged test cases is prioritized descendingly by the number of failures for each test case.

Procedure 7 Prioritize based on local changes.

Input: A list of classes C **Output:** A list of prioritized test cases T' An empty list of test cases T' An empty list of untracked test cases U An empty list of modified test cases M An empty list of unchanged test cases R **for** $c_i \in C$ **do** **if** c_i has not yet been added to the remote repository **then** $U \leftarrow \text{addTestCaseToList}(c, U)$ **else if** c_i has been modified **then** $M \leftarrow \text{addTestCaseToList}(c, M)$ **else** $R \leftarrow \text{addTestCaseToList}(c, R)$ **end if****end for** $R \leftarrow \text{prioritizeTestCasesByCountingFailingTests}(R)$ $T'.\text{append}(U)$ $T'.\text{append}(M)$ $T'.\text{append}(R)$ **return** T'

The Implementation

The main difference between this technique and Local Code Changes (Section 4.4.1), is that this one does not merely append the remaining test cases to the list, as is done in the latter (Listing 4.11), but uses a Pritest server to prioritize them beforehand (Listing 4.17). The list is retrieved from the server by calling the method `getClassList()` on the object `onlineClassService`, which is of the type `ClassService`.

```
1 public class PriorityList2 {
2     ...
3     private List<String> technique5Strategy() throws NoWorkTreeException,
4         IOException {
5         Technique5Ranker t5 = new Technique5Ranker(this.baseDir);
6         List<String> t5GitStatusList = t5.getTechnique5PriorityList();
7
8         List<String> contactCitrusList = new ArrayList<String>();
9         try {
10            contactCitrusList = this.onlineClassService.getClassList();
11        } catch (Exception e) {
12            // Handling exceptions
13        }
14
15        for (String s : contactCitrusList) {
16            if (!t5GitStatusList.contains(s)) {
17                t5GitStatusList.add(s);
18            }
19        }
20        return t5GitStatusList;
21    } ...
}
```

Listing 4.17: The TestOrderResource class.

Listing 4.18 shows the part of the server that accepts requests for prioritized lists. If a request with the parameter “5” is made, the server returns a list prioritized by the technique *Counting Failing Tests* (Section 4.3.1). The plugin will then append all test cases from this list that are not appended.

```
1 public class TestOrderResource {
2     ...
3     public List<String> get(@PathParam("method") int method) {
4         List<String> testClasses = new ArrayList<String>();
5
6         switch (method) {
7             ... // The other online techniques
8             case 5:
9                 testClasses = method1();
10                break;
11            }
12
13            return testClasses;
14        }
15        ...
16    }
```

Listing 4.18: Calling the technique

4.6 Control Techniques

Rothermel et al. introduced a group of ordering techniques called *control techniques* in his articles [4, 7]. This group contains the ordering techniques called *untreated*, *random* and *optimal*. Each of these orderings techniques will be described in the following sections.

By having these techniques we can compare them with our own techniques. This will give us a measure of how well our techniques sort the test cases relative to the control techniques. The control techniques are included in our experiment.

4.6.1 Untreated Order

This technique returns the list of test cases in a specific order. In this case, we will just use the local test cases.

4.6.2 Random Order

This technique uses the shuffle method in Java's `Collections` class. The result is a randomized list of the local test cases.

```
1 private List<String> randomLocalTestClasses(List<String>
   localTestClasses) {
2     Collections.shuffle(localTestClasses);
3     return localTestClasses;
4 }
```

Listing 4.19: Method for randomize local test classes.

4.6.3 Optimal Order

The optimal test case order is calculated after the test suite run. The measure objects, equal to test case, is then sorted by the measure “containing most failed tests”.

```
1  private List<String> getOptimizedPriorityList(List<Measure> list) {  
2      Collections.sort(list);  
3      Collections.reverse(list);  
4      List<String> optimizedList = new ArrayList<String>();  
5      for(Measure measure : list){  
6          optimizedList.add(measure.getSource());  
7      }  
8      return optimizedList;  
9  }
```

Listing 4.20: Method returning optimized test order.

4.7 Evaluating Technique Time Complexity with Big-O Analysis

To get a picture of the time complexity of each technique in our solution, we performed a big-O Analysis (introduced in Section 3.3.2). The calculations are described in detail in the following sections, and the results are summarized in Table 4.4 below:

Technique	Time complexity
Counting Failing Tests	$O(n \log n)$
Code Changes	$O(n \log n)$
Local Code Changes	$O(n^2)$
Total Method Coverage	$O(m + mn + n \log n)$
Additional Method Coverage	$O(n^3 m)$
Local Code Changes with Failure Counting	$O(n^2)$

Table 4.4: Big-O Calculations.

The big-O orders of techniques are illustrated in Figure 4.10, Figure 4.11 and 4.12.

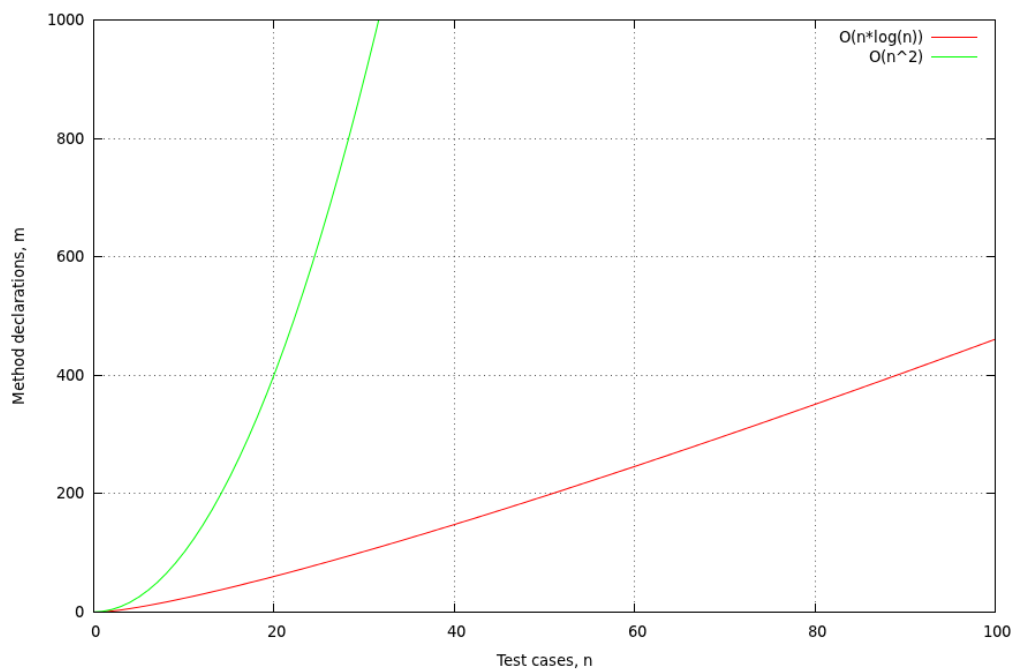


Figure 4.10: Big-O illustration: Counting Failing Tests and Code Changes are red, and the two Local Code Changes techniques are green.

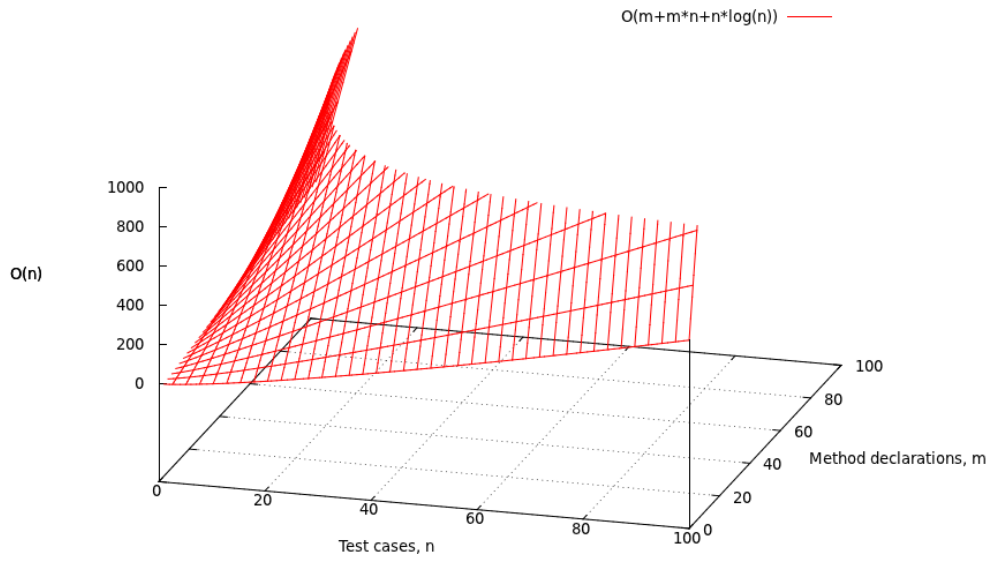


Figure 4.11: Big-O illustration of Total Method Coverage.

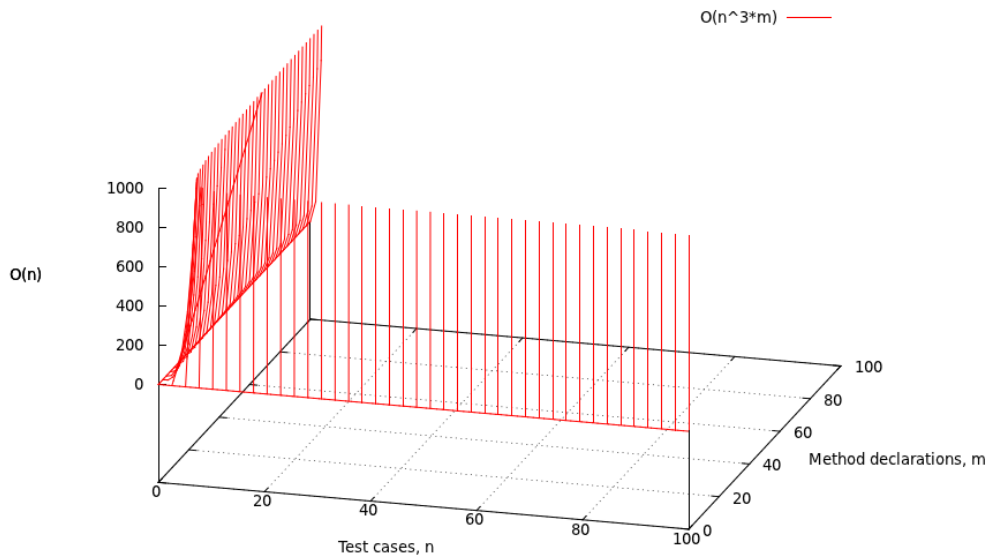


Figure 4.12: Big-O illustration of Additional Method Coverage.

As can be seen in the illustrations, the order of *Additional Method Coverage* and *Total Method Coverage* (especially the former) grows much faster than the others. However, these time complexities are both worst-case, so we did not deem it necessary to remove them.

Counting Failing Tests

```
1 private List<String> method1() {
2     TestDataDAO tdDAO = DAOFactory.getDatabase().getTestDataDAO();
3     List<TestData> tests = tdDAO.getList();
4     Collections.sort(tests);
5     List<String> testNames = new ArrayList<String>();
6     for (TestData test : tests) {
7         testNames.add(test.getClassName());
8     }
9     return testNames;
10 }
```

Listing 4.21: Technique summary Counting Failing Tests.

Technique *Counting Failing Tests* mainly consist of three parts:

1. Get a list from the database.
2. Sort the list.
3. Add the class names to a list.

The first part would result in an complexity of n . The second part guarantees a complexity of $n\log(n)$ [58]. The third part results in a complexity of n .

$$n + n\log(n) + n = 2n + n\log(n) \Rightarrow O(n\log(n))$$

Code Changes

```
1 private List<String> method3() {
2     ChangeDataDAO cdDAO = DAOFactory.getDatabase().getChangeDataDAO();
3     List<ChangeData> changes = cdDAO.getList();
4     Collections.sort(changes);
5
6     List<String> testNames = new ArrayList<String>();
7     for (ChangeData change : changes) {
8         testNames.add(change.getSource());
9         System.out.println(change.getSource());
10    }
11
12    return testNames;
13 }
```

Listing 4.22: Technique summary Code Changes.

Technique *Code Changes* mainly consist of three parts:

1. Get a list from the database.
2. Sort the list.
3. Add the class names to a list.

The first part would result in an complexity of n . The second part guarantees a complexity of $n\log(n)$ [58]. The third part results in a complexity of n .

$$n + n\log(n) + n = 2n + n\log(n) \Rightarrow O(n\log(n))$$

Local Code Changes

```
1 private List<String> technique4Strategy(List<String> localTestClasses)
   throws NoWorkTreeException, IOException {
2     GitStatusProvider gsp = new GitStatusProvider(baseDir,
           sourceDirectory, testSourceDirectory);
3     List<String> gitStatusPriorityList = gsp.getGitStatusPriorityList();
4
5     for(String localTestClass : localTestClasses) {
6         if(!gitStatusPriorityList.contains(localTestClass)) {
7             gitStatusPriorityList.add(localTestClass);
8         }
9     }
10    return gitStatusPriorityList;
11 }
```

Listing 4.23: Technique summary Local Code Changes.

Technique *Local Code Changes* mainly consist of three parts:

1. Get a list of the local test classes recently modified or added.
2. For each local test class sent in as parameter, check if it exists in the list of modified or added classes.
3. If it does not exist, add it to the list.

The first part would result in an complexity of n . The second and third part combined result in a time complexity of n^2 .

$$n + n * n = n + n^2 \Rightarrow O(n^2)$$

Total Method Coverage

As already mentioned in Chapter 4, the implementation of *Total Method Coverage* and *Additional Method Coverage* does some static code analysis prior to executing the techniques. This part will be analyzed separately from the implementation of the techniques.

The Time Complexity of the Static Code Analysis Phase

A step-wise description of the process follows, the time complexities are at the end of the lines in parentheses. A given class is denoted with the index i , and a given method declaration by the index j .

1. Retrieve Java files in project using `ClassListProvider.getFileList(...)`.
(m)
2. Retrieve `CompilationUnits` using `CompilationUnitProvider.getCompilationUnits(...)`.
(m)
 - (a) Parse code into `CompilationUnits` using `JavaParser.parse(...)`⁶.
3. Iterate through each `CompilationUnit` with `ClassTypeProvider.getClassTypes()`.
(m)
 - (a) Retrieve `imports`. (t)
 - (b) Retrieve `package` declaration. (1)

⁶It should be noted that the time complexity of `JavaParser.parse(...)` is unknown

- (c) Retrieve `ClassTypes` with `ClassOrInterfaceDeclarationVisitor`. (n)
 - i. Retrieve class name. (1)
 - ii. Retrieve `extend` declaration. (1)
 - iii. Retrieve fields with `ClassOrInterfaceDeclarationVisitor.visit(FieldDeclaration ...)`. (f_i)
 - iv. Retrieve member method declarations with `MethodDeclarationVisitor`. (k_i)
 - A. Retrieve return type. (1)
 - B. Retrieve parameters with `MethodDeclarationVisitor.extractParameter(...)`. ($p_{i,j}$)
- 4. Iterate through each `CompilationUnit` and retrieve method coverage information with `MethodCoverageProvider` and `MethodCoverageVisitor`. (m)
 - (a) Retrieve package declaration. (1)
 - (b) Retrieve a `ClassCover` for each class using `MethodCoverageVisitor.visit(ClassOrInterfaceDeclaration ...)`. (n)
 - i. Retrieve a `MethodCover` for each declared method in that class using `MethodCoverageVisitor.visit(MethodDeclaration ...)`. (k_i)
 - A. Retrieve a `MethodDecl` object for each method declaration, consisting of: the method name (1), return type (1) and parameters ($p_{i,j}$). ($1 + 1 + p_{i,j}$)

- B. Retrieve local variables. ($v_{i,j}$)
- C. Retrieve raw method calls (as `RawMethodCalls`) within this method body using `MethodCallVisitor`, with each raw method call consisting of: method name (1), parameters ($p_{i,j}$) and scope (1) (e.g. the object reference the method call is made on). ($1 + p_{i,j} + 1$)
- D. For each raw method call find the class the called method belongs to, using `processRawMethodCall(...)`. (1)

The steps above are the same for the analysis of both source code and test source code.

$$m + m + m * (t + 1 + n * (1 + 1 + f + k * (1 + p))) + m * (1 + n * (k * (1 + 1 + p + v + 1 + p + 1 + 1))) = 2 * m + m * (t + 1 + n * (2 + f + k * (1 + p))) + m * (1 + n * (k * (2 * p + v + 5)))$$

As we are more interested in classes than java files, we will ignore m , which is the number of `CompilationUnits`, and also import statements:

$$1 + n * (2 + f + k * (1 + p)) + 1 + n * (k * (2 * p + v + 5))$$

Then we introduce indices for classes and method declarations, i for classes and j for method declarations:

$$1 + n * (2 + f_i + k_i * (1 + p_{i,j})) + 1 + n * (k_i * (2 * p_{i,j} + v_{i,j} + 5))$$

A method will usually not have more than a few parameters; therefore, we will ignore the number of parameters a method declaration has. We will also ignore the number of field variables and local variables, as a high number of these can indicate the presence of “code smell”. Large Class and Long Method respectively (Chapter 3).

$$1+n*(2+k_i*1)+1+n*(k_i*(2+5)) = 2+n*(2+k_i)+n*k_i*7 = 2+n*2+n*k_i+n*k_i*7 = 2+n*2+8*n*k_i$$

Asymptotically, this becomes:

$$O(n * k_i)$$

Or, the number of method declarations in the project being tested:

$$O(|k|)$$

The Time Complexity of the Technique

When executing the *Total Method Coverage* technique, the total set of methods being called by each test case will be found (Listing 4.14 and Listing 4.16). Since each method call will only be analyzed once, the total number of covered methods a test case can have equals the number of methods in the project being tested. The other extreme is zero method calls found, something that can happen if the test case uses a feature not supported by the code analysis. Hence, we have a best case:

$$O(1),$$

and a worst case, where m is the number of method declarations:

$$O(m),$$

when counting method calls of a single test case. The best and worst cases for n test cases are then, respectively:

$$O(n) \text{ and } O(nm)$$

Afterwards, the list of test cases is sorted by the number of covered methods using `Collections.sort(...)`, which has a time complexity of $O(n \log n)$. If we substitute $O(|k|)$ with $O(m)$ the best and worst case for Total Method Coverage becomes:

$$O(m + n + n \log n) \Rightarrow O(n \log n) \text{ and } O(nm + n \log n)$$

Additional Method Coverage

In addition to the static code analysis and sorting the test cases by the number of covered methods, *Additional Method Coverage* must recalculate the coverage for each chosen test case (Listing 4.13).

The process is described below; n is the number of test cases, and m is the number of declared methods.

1. Run the static code analysis and sort test cases by coverage.
2. Count the number of method declarations in the project, using `coveredMethodsInSource(...)` in the class `MethodCoverageAlgorithm` (m)
3. Iterate through the test cases descendingly by the number of covered methods for as long as there are unprioritized test cases and uncovered methods. (n)
 - (a) Select test case with the highest number of covered methods using `Collections.max(...)` from the standard library. (n)
 - (b) Remove the selected test case using `List.remove(...)`. (n)
 - (c) Decrement the number of uncovered methods. (1)

- (d) Mark the covered methods using `markMethodAsCovered(...)`. (nm)
- 4. Run Additional Method Coverage again on any remaining test cases if all methods have already been covered. In the best case scenario none remains, and in the worst case scenario $n - 1$ test cases remains.
 - (a) Mark all methods as *not* yet covered using `unMarkCoveredMethods(...)`.
($(n - 1) * m$)
 - (b) Run `additionalMethodCoverageHelper(...)` with the remaining test cases. Will be called recursively at most $n - 1$ times, which means that every test case covers the same methods.

If we summarize this, and exclude everything not specific to the *Additional Method Coverage* implementation, we get in the best case and worst case, respectively:

$$m + n * (n + n + 1 + nm) = m + 2n^2 + n + n^2m$$

and

$$(m + n * (n + n + 1 + nm) + (n - 1) * m) * n = nm + n^2(n + n + 1 + nm) + nm(n - 1) = nm + 2n^3 + n^2 + n^3m + n^2m - nm = 2n^3 + n^2 + n^3m + n^2m$$

Then we introduce the time complexities of the static code analysis and *Total Method Coverage* and get:

$$O(n \log n) + m + 2n^2 + n + n^2m \Rightarrow O(2m + n \log n + 2n^2 + n + n^2m) \Rightarrow O(n^2m)$$

and

$$O(nm + n \log n) + 2n^3 + n^2 + n^3m + n^2m \Rightarrow O(n^3m)$$

Local Code Changes with Failure Counting

```
1 private List<String> technique5Strategy() throws NoWorkTreeException,
   IOException {
2     GitStatusProvider gsp = new GitStatusProvider(baseDir,
           sourceDirectory, testSourceDirectory);
3     List<String> gitStatusPriorityList = gsp.getGitStatusPriorityList();
4
5     List<String> contactPritestList = new ArrayList<String>();
6     try {
7         contactPritestList = this.onlineClassService.getClassList();
8     } catch (ConnectException e) {
9         e.printStackTrace();
10    } catch (JSONException e) {
11        e.printStackTrace();
12    } catch (Exception e) {
13        e.printStackTrace();
14    }
15
16    for (String s : contactPritestList) {
17        if (!gitStatusPriorityList.contains(s)) {
18            gitStatusPriorityList.add(s);
19        }
20    }
21
22    return gitStatusPriorityList;
23 }
```

Listing 4.24: Technique summary Local Code Changes with Failure Counting.

Technique *Local Code Changes with Failure Counting* mainly consist of four parts:

1. Get a list of the local test classes recently modified or added.
2. Contact Pritest for remaining internal prioritization.
3. For each class retrieved from Pritest, check if it exists in the list of modified or added classes.
4. If it does not exist, add it to the list.

The first part would result in a time complexity of n . The second part results in a time complexity of n . The third and fourth part combined result in a time complexity of n^2 .

$$n + n + n * n = 2n + n^2 \Rightarrow O(n^2)$$

Chapter 5

Experiment

5.1 Experiment Theory

In order to perform a complete and good experiment, we applied principles from “Experimentation in Software Engineering - An Introduction” by Wohlin et al. [22].

According to Wohlin et al., there are two types of research approaches to empirical studies: *qualitative* and *quantitative*. Qualitative research is concerned with studying objects in their natural setting, whereas quantitative research focus on quantifying a relationship or compare two or more groups. An experiment is a quantitative empirical research method.

Experiments are often carried out in a laboratory environment, and provide a high level of control throughout the execution. The purpose is to manipulate a few variables with different treatments to the subject, and observe how the subject reacts to these treatments. The variables not being manipulated, must be controlled to ensure that the reactions from the subject are due to the variables being manipulated and

not anything else.

The experiment process must be planned in detail to ensure valid results. A theory must be developed, and a cause-effect relationship between treatments and expected outcome must be constructed (Figure 5.1).

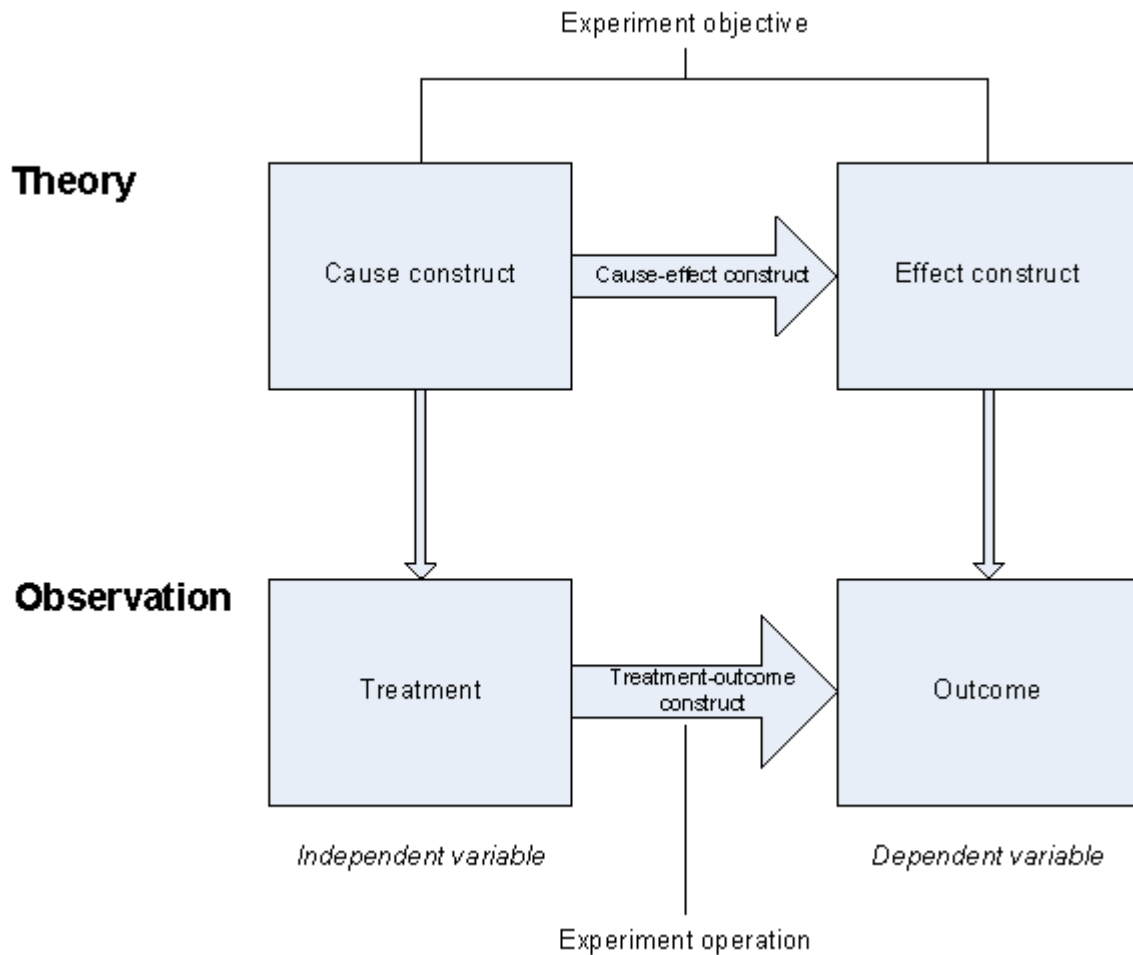


Figure 5.1: Experiment principles [29].

The process proceeds with selecting independent and dependent variables, context, subjects, hypotheses, experiment design, etc. (Figure 5.2).

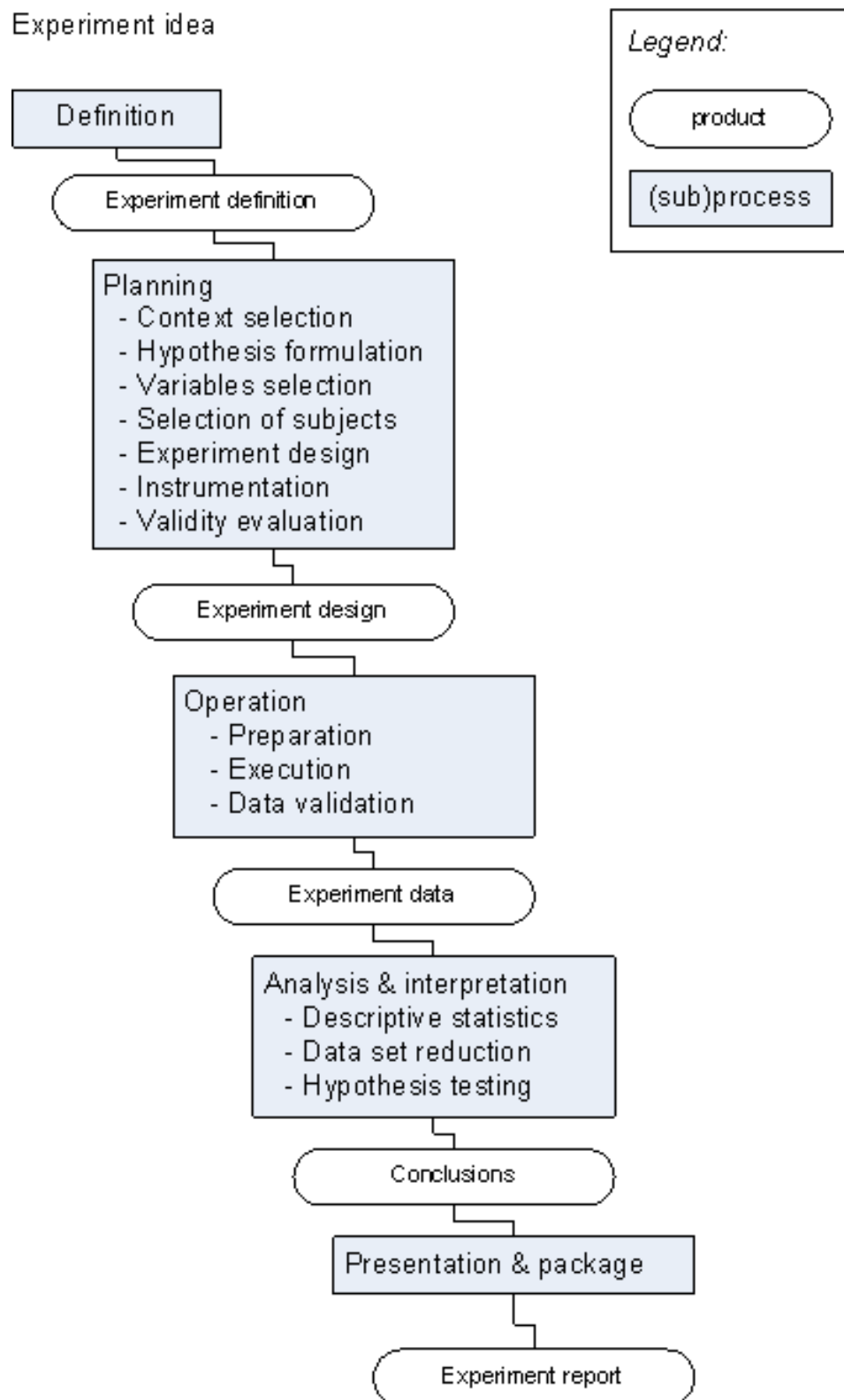


Figure 5.2: Experiment process [22].

5.2 Experiment Introduction

The purpose of the experiment is to test whether or not sorting the test suite according to one of the techniques described in Chapter 4, will make the failing tests occur earlier in the test suite.

5.3 Definition

5.3.1 Goal Definition

- **Object of study:**

The object of study is the APFD values recorded in each test run.

- **Purpose:**

The purpose of the experiment is to evaluate the performance of each prioritization technique for a test suite.

- **Perspective:**

The perspective is from the point of view of the researchers, i.e. we want to know if there is any significant difference in APFD values depending on which prioritization technique is used.

- **Quality focus:**

The effect studied is the APFD values of the test runs.

- **Context:**

The experiment is run within the context of professional developers performing a development project, and using JUnit for testing.

5.3.2 Summary of Definition

Analyze *the execution of test suites*

for the purpose of *evaluation*

with respect to *the test case ordering*

from the point of view of the *researchers*

in the context of *professional developers using Pritest over a period of time.*

5.4 Planning

5.4.1 Context Selection

The context of the experiment is industrial software development (online). The experiment will be run by two developers at BEKK, on the actual development project they are employed at the moment. As the experiment is run in this context, we will gain external validity which is elaborated below when discussing threats to validity. This context enables us to test our tool on real problems.

5.4.2 Hypothesis Formulation

One of the most important parts of an experiment is to know, and formulate the hypotheses. The hypotheses tell us what we want to evaluate in the experiment, and define the success factors. We want to observe if there is any difference in how the prioritization techniques rank the test cases with likelihood of failing; if there is difference in feedback quality of the different techniques.

1. Null hypothesis:

H_0 : There is no difference in the feedback quality (APFD value) resulting from each prioritization technique.

2. Formal null hypothesis:

$$H_0 : APFD(technique_1) = APFD(technique_2) = \dots = APFD(technique_n)$$

3. Alternative hypothesis:

There exist at least one APFD value that is different from the others.

4. Formal alternative hypothesis:

$$H_1 : \exists x_i APFD(x_i) > Average(T), T = \{APFD(x_1), APFD(x_2), \dots, APFD(x_n)\}, x_i \notin T.$$

If at least one of the techniques provides a different APFD value than the others, the null hypothesis can be rejected. We define risks to the hypothesis testing during the experiment as *Type-1-error* and *Type-2-error* [22]:

- Type-1-error:

$$P(\text{Type} - 1 - \text{error}) = P(\text{reject}H_0 \mid H_0\text{true})$$

- Type-2-error:

$$P(\text{Type} - 2 - \text{error}) = P(\text{notreject}H_0 \mid H_0\text{false})$$

In our case a Type-1-error occurs if we say that there is a difference in the APFD values to the different technique, even though it is not. A Type-2-error would occur if we find that there is no difference in the feedback quality using the different techniques, when it actually is. We consider a Type-1-error the most critical since it confirms that our system improves feedback from automated tests, when it actually does.

5.4.3 Variables Selection

The independent variable, or variables, are the ones that can be controlled and changed during the experiment. A factor is an independent variable that can receive a treatment.

The independent variables in our experiment are: *applied prioritization technique* to the test suite run (subject), *how the developers work* and *the development project*. The way the developers work can affect the performance of some of the prioritization techniques, e.g. the *Local Code Changes* technique (Section 4.4.1) might perform differently depending on how often the developers commit code. This variable is to a

degree controlled, as the experiment will only have two developers. The development project can also affect the outcome, as the prioritization techniques might perform differently when used in different projects (Section 5.4.7).

The dependent variable is the *APFD value*. One should be able to draw the dependent variable directly from the hypothesis.

5.4.4 Selection of Subjects

A clear definition of subjects in the experiment is important. When selecting subjects, it is important to remember that to be able to generalize to the desired population, the sample from the population (group of subjects) must be a representative sample from this population.

Our subjects in this experiment are *test runs*. Our developers will use our system in a real-life project over a period of time, and every time they run their test suites, we will record a new subject and results from the dependent variable. Hence, we do not know yet how many subjects we will have, but the longer our developers run our application, the more subjects—and more valid—our results will get.

Whether or not the sample is representative for the population is hard to tell. Our population is every kind of test suite run imaginable, and whether the test runs in our experiment project are representative samples is an area for discussion. However, there are some test suite characteristics that are more “common” than others, and the project we will perform our experiment on is a rather “common” Norwegian software project (large scale, public sector and hired consultants).

5.4.5 Experiment Design

As previously mentioned, a factor is an independent variable that can receive a form of treatment. In our case that is *prioritization techniques*. We have five prioritization techniques that will be used in this experiment. The choice of experiment design limits the statistical analysis methods that can be applied to the hypothesis later on.

When determining which statistical test to use, we have to look at the data represented in the dependent variable after retrieving different treatments. This is the APFD value, and can be classified as an interval scale or ratio scale (Table 5.1). Statistical hypothesis testing can again be divided into parametric and non-parametric tests. When having an interval scale or ratio scale parametric tests can be applied.

Scale Type	Characteristics
Nominal Scale	Maps the attribute of the entity into a name or symbol. Typically classification and labeling.
Ordinal Scale	Ranks the entities after an ordering criterion. Typically grades and software complexity.
Interval Scale	The difference between two measures are meaningful, but not the value itself. Typically Celsius temperature or Fahrenheit.
Ratio Scale	There exists a meaningful null value, and the ratio between two measures is meaningful. Typically Kelvin temperature.

Table 5.1: Scale Types by Wohlin et al. [22].

In our experiment it is clear that we have a design consisting of *one factor* with *more than two treatments*. And since we have a dependent variable with (at least) an interval scale, we can apply a parametric hypothesis test. We will perform an ANOVA test on our data to check our hypothesis.

5.4.6 Instrumentation

The instrumentation in an experiment can be of three types: *objects*, *guidelines* and *measurement instruments* [22].

The executor of the experiment, our external supervisor, will be supplied with guidelines on how to set up, execute and handle the responses the application gives through the experiment.

The measurement instruments are implemented as a feature in the application, and are handled automatically by our plugin. We record APFD values to files when the test suites are run. This feature is provided with information from the test run, such as failing tests, and calculates a APFD value. We also implemented a feature for generating graphs from the APFD values, an example can be seen in Figure 5.3 and 5.4.

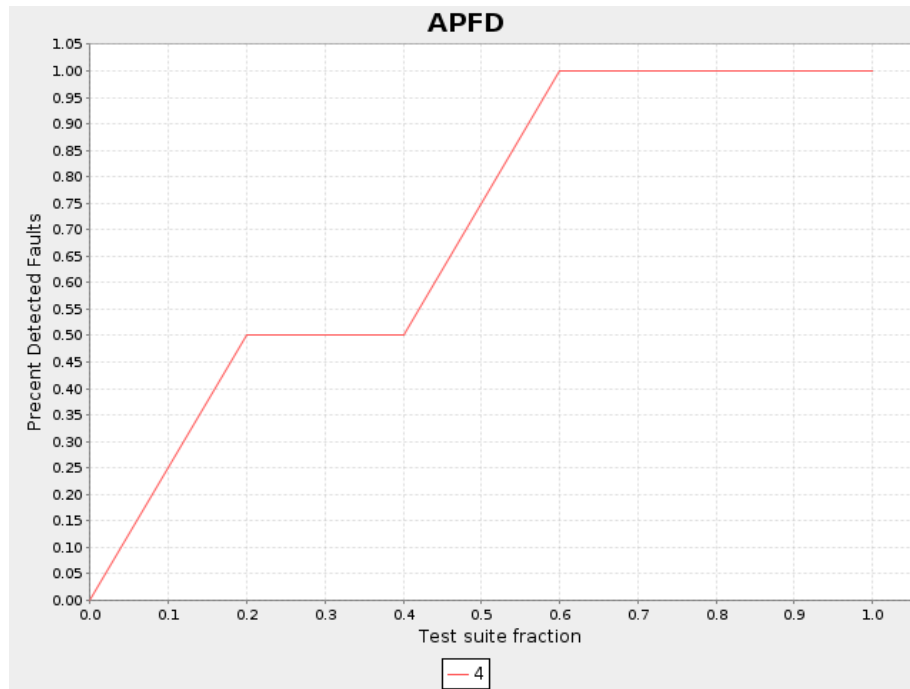


Figure 5.3: Example of APFD graph generation - using a single technique.

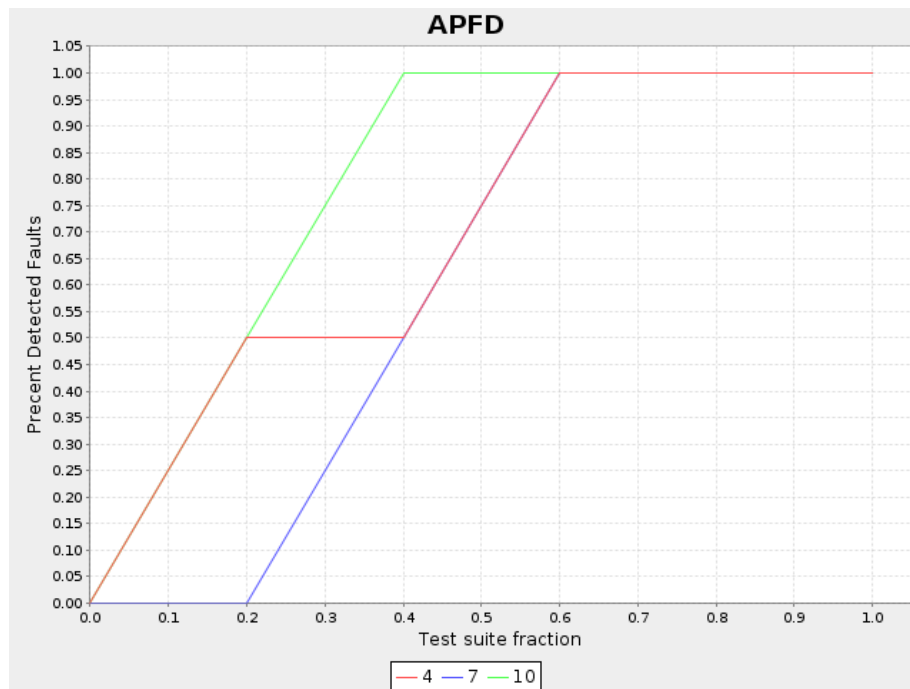


Figure 5.4: Example of APFD graph generation - using multiple techniques at once.

5.4.7 Validity Evaluation

It is essential to evaluate the question of validity of the results we can expect. Examples of threats to the validity of the results are maturation among the subjects, subjects guessing what the hypothesis might be or if the experiment is designed in a way that it sort of “fishes” for anticipated answers. When evaluating validity of an experiment, you can divide it into to four categories *conclusion validity*, *internal validity*, *construct validity* and *external validity* [22].

Conclusion Validity

- *Reliability of measures.* APFD might not be a perfect measure, as it does not consider the processing time a technique requires. This is however mitigated to some extent by the big-O analysis we performed (Section 4.7). In addition, as we only support tests written as JUnit test cases, errors in layers that are not covered by unit tests will then not give any results.
- *Random irrelevancies in experiment setting.* The developer performing the experiment can get distracted with other work during the experiment execution.
- *Low statistical power.* The development project might not generate enough test failures, giving us too few data.

Internal Validity

- *Selection of developers.* As mentioned in Section 5.4.3, the way the developer work can affect the results of the test executions. The developer should ideally be representative of the whole population. The fact that our experiment has only two developers can therefore be a threat. Although they should be representative of BEKK.

- *Selection of development project.* The development project can also affect the outcome of the experiment. The development methodology, architecture, and the type of the project can all have an impact on the performance of each technique. Like the selection of developers above, we are only executing the experiment in one project; though it should be representative for the projects in BEKK.

Construct Validity

- *Erroneous implementation of techniques.* If the implementation of a prioritization technique contains faults the results would potentially also contain faults. We have, however, tested the techniques on some dummy projects, so we have at least reduced the amount of possible faults. The shortcomings that we know about, but are unable to correct at the current time, are described in Chapter 4.

External Validity

- *Interaction of selection and treatment.* As the experiment only has two developers, we get a limited amount of subjects (i.e. test executions). It is therefore a possibility that these are not representative of test executions in general.
- *Interaction of setting and treatment.* A threat is that the project used in the experiment is different from the projects usually performed at BEKK.
- *Interaction of history and treatment.* If the experiment is conducted at a special time in project, the results might not be what they would have been at another time. E.g. if the experiment is conducted at a time when the coding is not very unit test oriented.

5.5 Operation

5.5.1 Preparation

The experiment executors were given instructions on how to set up the system, and install instructions. They were briefed on how the system works, and what had to be in place for the system to work properly. In advance of the experiment period, we tested the techniques ourselves to ensure that the techniques were properly implemented according to their intentioned heuristics, and that they gave reasonable results.

We prepared the APFD calculator in our tool, to automatically generate APFD values for each technique, in addition to the APFD values for the untreated ordering, random ordering and the optimal ordering (Section 3.1.3). These values will be used for analysis later on in the experiment.

When running a test suite in the experiment project, the test suite is only run once, and the APFD values for every technique are then calculated using the results from the test suite. The experiment executors should work as they normally would in the project.

5.5.2 Execution

The experiment was executed on a real industry software project over three days, and data were collected automatically through our own implementation for collecting APFD values (Section 3.1.4).

The techniques that were evaluated in the experiment were the following:

- Counting Failing Tests
- Local Code Changes
- Local Code Changes with Failure Counting
- Total Method Coverage
- Additional Method Coverage

These techniques were evaluated against the following orderings:

- Untreated ordering
- Random ordering
- Optimal ordering

Technique *Code Changes* was excluded from the experiment due to firewall issues at the project site where the experiment was executed. The firewalls did not allow Github to send the “post-receive hooks” to Pritest needed by this technique.

5.5.3 Data Validation

To ensure that the data being collected was reasonable, and gave expected values, we implemented a logging feature in the experimental source code. These loggings were inspected before the actual experiment was initialized to see that values in fact were recorded, and that the source code analysis part of technique *Additional*

Method Coverage (Section 4.4.2) and *Total Method Coverage* (Section 4.4.3) were correct.

Failing test runs should result in recording of APFD values between 0 and 1, whereas test runs without failures should record APFD values saying “NaN”. The inspections in advance of the experiment confirmed this rule.

5.6 Analysis and Interpretation

Several diagrams and statistics are presented in this sections. The diagrams refer to our techniques with the numbering displayed in Table 5.2.

Technique	Alias #
Counting Failing Tests	1
Code Changes	3
Local Code Changes	4
Local Code Changes with Failure Counting	5
Total Method Coverage	6
Additional Method Coverage	7
Untreated Order	8
Random Order	9
Optimal Order	10

Table 5.2: Prioritization techniques numbering.

5.6.1 Descriptive Statistics

The results from the experiment are shown in the boxplot in Figure 5.5 below.

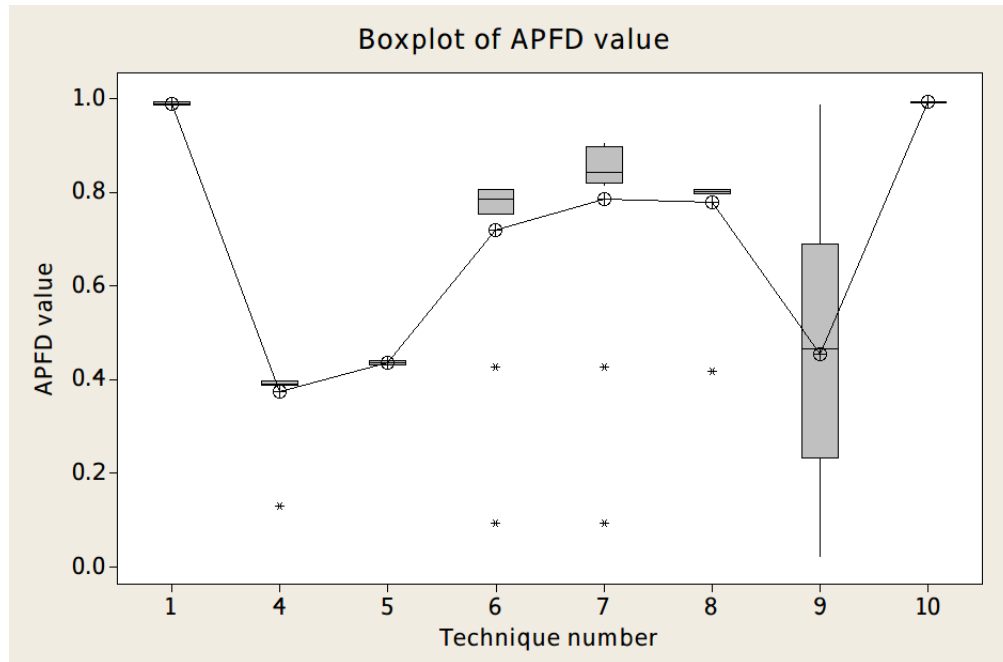


Figure 5.5: A boxplot of the results - primary experiment.

Technique 1 (*Counting Failing Tests*) has the best performance among our techniques. The reason for this is likely that test failures were occurring in some of the same test cases during the experiment.

Technique 4 and 5 (*Local Code Changes* and *Local Code Changes with Failure Counting* respectively) gave suspicious results—something that must be investigated further. Giving further proof that something was not right with the results, was that the two control techniques *Untreated* and *Random Order* performed better.

Technique 6 and 7 (*Total Method Coverage* and *Additional Method Coverage*) did well, compared with the control techniques. According to the boxplot, *Additional*

Method Coverage did better than *Total Method Coverage*, which complies with the findings of Do et al. [14].

These two techniques each have two outliers, which have the same values: 0.0937500000 and 0.4279279279. The former value is almost the same as the outlier in the results of *Local Code Changes*, which might be related. These are possibly caused by code changes resulting in errors in other parts of the system than usual. Another possibility might be actual code changes in another part of the system. In any case, these outliers can happen again, and will therefore not be eliminated.

In the instrumentation definition (Section 5.4.6) we explained how we implemented an automatic APFD value graph generator, being generated after each complete test run. These were mainly used by us to inspect the results faster when receiving APFD values from our test executor, but an example from a experiment test run can be viewed below (Figure 5.6).

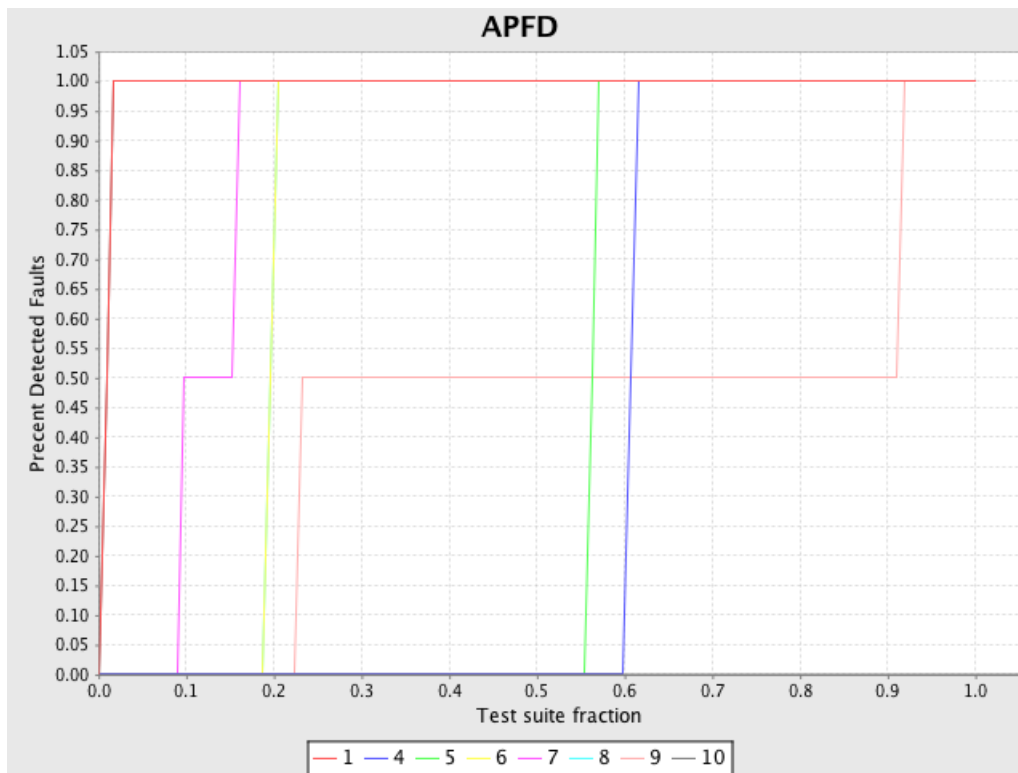


Figure 5.6: APFD instrumentation graph example - primary experiment.

5.6.2 Data Reduction

The very first APFD values recorded for each technique was removed from our dataset. The reason for this is that the first test run using our system will not have the history of former test runs (which some of the techniques base their prioritization on), and is therefore representative to measure the prioritization quality of these techniques.

In addition, test runs without failures record *NaN* as APFD value. These were also removed from our dataset. The experiment is concerned with examining the prioritization quality of failing tests, and test runs without failures are not of interest.

5.6.3 Hypothesis Testing

Table 5.3 summarize our hypothesis test. It was a one-way ANOVA test, with *technique number* as factor, and *APFD value* as the dependent variable. The chosen significance level was 5%. From the rightmost column, we see that the p-value is 0.000, so the results are highly significant, and the null hypothesis H_0 (Section 5.4.2) can be rejected.

Source	Degrees of freedom (DF)	Sum of Squares	Mean Square	F-value	p-value
Technique Number	7	6.7022	0.9575	45.37	0.000
Error	120	2.5324	0.0211		
Total	127	9.2346			

Table 5.3: Hypothesis test - primary experiment.

We had a total of 16 subjects (test runs) in this experiment. Table 5.4 sum up the mean APFD values for each technique, and its standard deviation.

Since technique 10 is a control technique, technique 1 —*Counting Failing Tests*— performed best in this experiment.

Technique	# Occurrences	Mean APFD	Standard Deviation
1	16	0.9911	0.0040
4	16	0.3763	0.0656
5	16	0.4375	0.0044
6	16	0.7213	0.1910
7	16	0.7860	0.2162
8	16	0.7792	0.0961
9	16	0.4550	0.2683
10	16	0.9946	0.0017

Table 5.4: Hypothesis test - primary experiment.

5.7 Secondary Experiment

From the experiment, we found the results from using technique *Local Code Changes* suspicious. We predicted it to do well in the experiment, due to its ability to run newly added and modified test cases first. In most cases these would be the ones containing test failures.

After code review, we found the implementation error, and we performed an additional experiment on the hypothesis with a different context than defined in our primary experiment in this chapter. All the definitions of hypotheses, subjects, experiment design, instrumentation still account for this secondary experiment. The context is different in the way that it is not an industrial online experiment, rather an offline laboratory study.

We ran a secondary experiment partly to examine if we are able to generalize the

results better to different types of software projects, and partly due to the suspicious results of technique *Local Code Changes* in the main experiment. Hence, this experiment has a different design than the primary experiment. We focus on inserting errors in an existing open source project (*sonar-squid* [66]), instead of developing a system using unit tests.

5.7.1 Context

The context of this experiment is offline and performed as a laboratory experiment.

5.7.2 Variables

The independent variable is *applied prioritization technique*. The dependent variable is the resulting APFD values of a test run.

5.7.3 Execution

We will randomly¹ insert errors in the test cases and classes already present in *sonar-squid*. Afterwards, the test suite is run with the same prioritization techniques as in the primary experiment. APFD values are recorded and used as instrumentation to analyze the results. After the test run, we will discard the changes, and randomly pick new test cases or classes to alter, and run the tests again. Before each test run

¹We made a numbered list of all the Java classes in the project. We used a random number generator to pick random classes from this list.

we insert a random number of errors² between one and three in randomly selected classes and/or test cases present in the project.

5.7.4 Descriptive Statistics

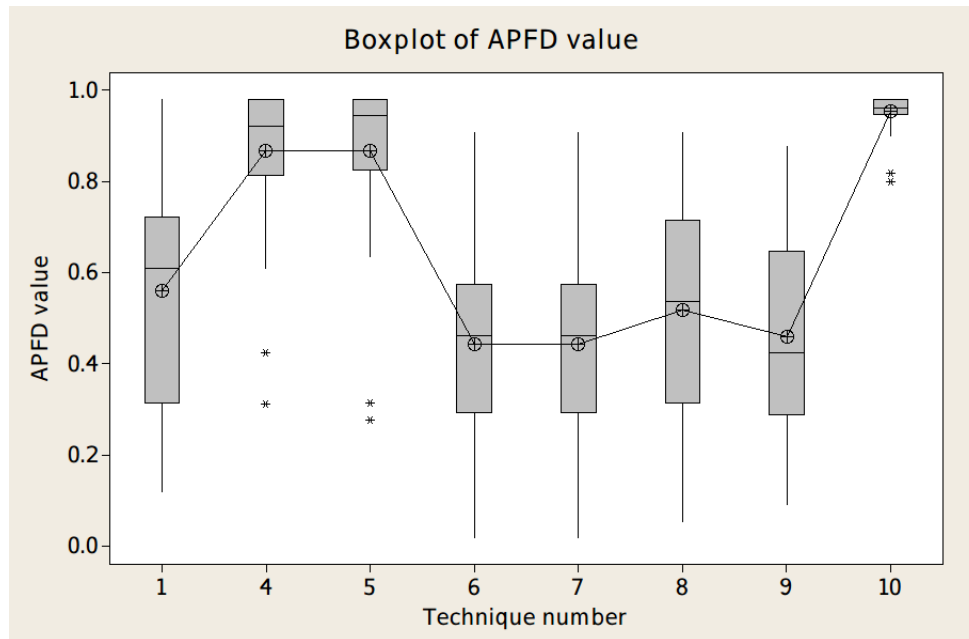


Figure 5.7: A boxplot of the results - secondary experiment.

The boxplot (Figure 5.7) gives us an overview of the dispersion and skewedness of the samples. There are some values outside the lower and upper tails of technique 4 (*Local Code Changes*), technique 5 (*Local Code Changes with Failure Counting*) and technique 10. The reason for the outliers on technique number 4 and 5, are most likely that Pritest did not find any test cases covering the given classes. Pritest assumes that every class `SomeClass` has a test case `SomeClassTest`. If that test case either does not exist or is named differently, the error may show up as a test failure

²We used a random number generator to determine how many classes to insert errors to for each test run.

that Pritest was unable to give a fitting priority. How Pritest looks for the test case covering a given class is described in Section [4.4.1](#).

The two outliers of technique number 10 are caused by some inserted errors causing a chain reaction of failures in a series of test cases. None of the outliers in the two cases will be removed, as they are likely to happen again and must therefore be considered.

Considering the APFD values, we see that technique 4 and 5 (the techniques based on local code changes) performed better than all the others, except technique 10 (Optimal Order).

In Figure [5.8](#), you can see an example of the auto-generated APFD value graphs we used for instrumentation and inspection of results.

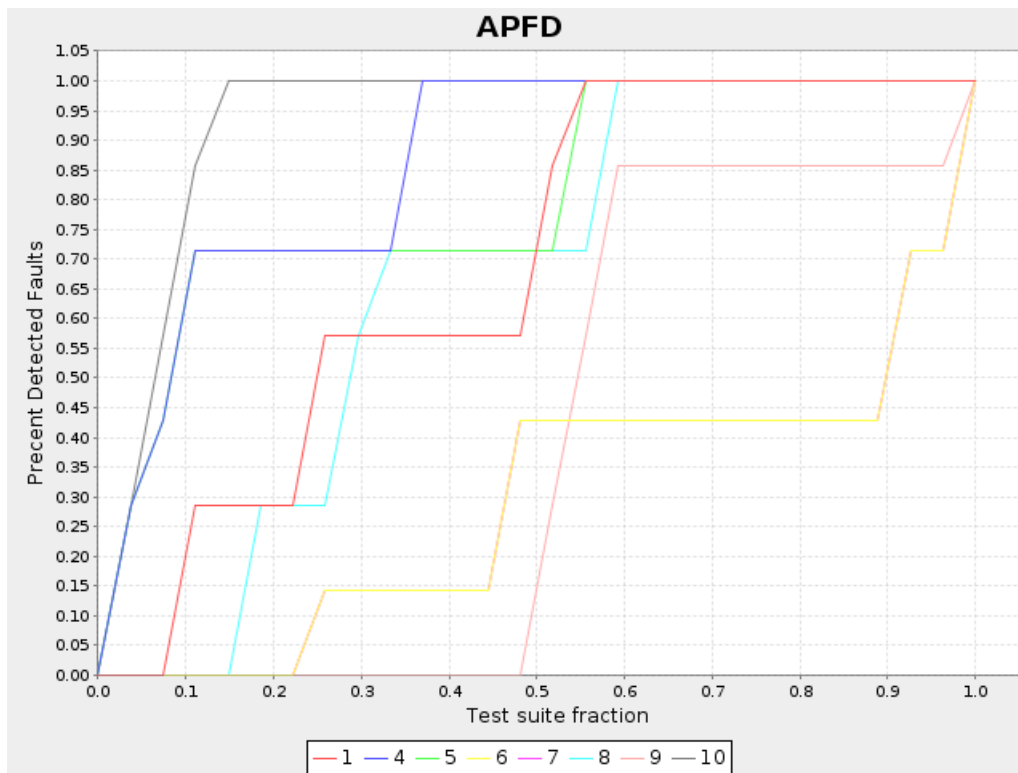


Figure 5.8: APFD instrumentation graph example - secondary experiment.

5.7.5 Hypothesis Testing

Table 5.5 summarize our hypothesis test. It was a one-way ANOVA test, with *technique number* as factor, and *APFD value* as the dependent variable. The chosen significance level was 5%. From the rightmost column, we see that the p-value is 0.000, so the results are highly significant, and the null hypothesis H_0 (Section 5.4.2) can be rejected.

Source	Degrees of freedom (DF)	Sum of Squares	Mean Square	F-value	p-value
Technique Number	7	11.6796	1.6685	39.92	0.000
Error	272	11.3679	0.0418		
Total	279	23.0476			

Table 5.5: Hypothesis test - secondary experiment.

We had a total of 35 subjects (test runs) in this experiment. Table 5.6 sum up the mean APFD values for each technique, and its standard deviation. We see that technique 5 and 4 performed well in this experiment.

Since technique 8, 9 and 10 are control techniques, technique 4 and 5 — respectively *Local Code Changes* and *Local Code Changes with Failure Counting* — performed best in this experiment.

Technique	# Occurrences	Mean APFD	Standard Deviation
1	35	0.5615	0.2436
4	35	0.8676	0.1574
5	35	0.8687	0.1764
6	35	0.4429	0.2252
7	35	0.4429	0.2252
8	35	0.5195	0.2560
9	35	0.4607	0.2249
10	35	0.9547	0.0406

Table 5.6: Hypothesis test - secondary experiment.

5.7.6 Validity Evaluation

As this experiment was not conducted on a project during development, but by inserting random errors into an open source project, some special considerations must be made. Most of these concern external validity, more specifically the *interaction of setting and treatment*. The main issue with this experiment is that the test failures are not caused by naturally occurring errors in the code, and generalization of the results should be done with some care.

Due to the randomized errors, the technique *Counting Failing Tests* will probably perform badly in this experiment. As the errors are randomized it is possible that a given test will not fail again.

5.8 Summary and Conclusions

Both experiments have a p-value of 0.000; hence, the results are significant, and the null hypothesis H_0 can be rejected. The null hypothesis stated that “there is no difference in the feedback quality (APFD value) resulting from each prioritization technique”, but the experiment results show that there is.

In both the experiments, we included a set of control techniques (technique 8, 9 and 10). Besides from these, we see that technique *Counting Failing Tests* performed the best in the first experiment, and technique *Local Code Changes with Failure Counting* performed slightly better than technique *Local Code Changes* in the second experiment.

The two experiments do however show quite different results. Techniques that performed well in the first experiment, performed poorly in the second, and vice versa.

We know that we had an implementation failure in technique *Local Code Changes* and its hybrid version in the first experiment, and this was partly the reason for running an additional experiment. The second experiment had a different design; inserting errors to an existing system, rather than developing one from the beginning. This resulted in some differences in performance for the techniques from the first experiment to the second. *Local Code Changes* and *Local Code Changes with Failure Counting* performed really well in this experiment, and *Counting Failing Tests* performed poorly. This is natural due to the heuristic nature of the technique—as it uses previous history of failing tests—and the failures were inserted randomly in the second experiment.

The techniques from Rothermel et al. (*Total Method Coverage* and *Additional Method Coverage*) did well in the first experiment, and not that well in the second. This was somewhat expected as well, since inserting errors randomly in an existing system does not provide any new method calls, and these techniques will therefore have a “random performance”.

Since there was an implementation mistake in technique *Local Code Changes* and *Local Code Changes with Failure Counting* in the first experiment, and that we implemented the techniques from Rothermel et al. ourselves, we will be careful to draw any substantial conclusions. However, from the experiment results, we see that we have one technique performing better than *Total Method Coverage* and *Additional Method Coverage* in the first experiment, and three of our techniques performed better in the second experiment. We will emphasize that our implementations of techniques designed by Rothermel et al. is far from optimal, and their shortcomings are discussed in Section [4.4.2](#).

Chapter 6

Evaluation and Discussion

6.1 Comparison with Existing Techniques

The techniques developed by Rothermel et al. that we employed seems to be better at detecting a large amount of failures as early as possible, than detecting errors caused by recent changes—which is often the case when writing code in a TDD fashion [32]. The prioritizations made by techniques such as *Total Method Coverage* does not change much when a single new method is implemented; because of this, it is possible for two prioritizations to be identical (or close to identical) even when changes have been made to different parts of the system. Our implementations of *Total* and *Additional Method Coverage* had their shortcomings though (Section 4.4.2), which resulted in test cases without any discovered method calls, which again resulted in the same test cases not being prioritized and merely added to the end of the prioritization list.

6.2 Post-Experiment Interview

To find out how the experiment executors experienced using Pritest, we carried out an interview in retrospect of the experiment. Some of the replies are highlighted below, and the rest are found as appendices (Section 8.3).

Question: Do you see any use for a product like Pritest?

“Absolutely. It does need some IDE support (IDEA & Eclipse-plugins) in order to get mainstream usage. The runner should probably be decoupled from maven. I think projects with a very long runtime would love to use it on their CI servers to get quicker responses.”

Question: Do you have any suggestions to improvements of Pritest?

- Plugin for IDEA and Eclipse.
- Decouple the runner from Maven (makes it easier to develop said plugins).
- Support for Maven multi-module projects.
- Fail-fast mechanism.
- Fail-after-running-all-previously-failing-tests (if any failed) mechanism.
- Support for tags/annotations for the test running phase.

6.3 Quality Analysis - Comparing Pritest to JUnit Max

Based on our research, we found that JUnit Max was the only tool similar to Pritest. The two have quite differing features, which makes them difficult to compare and evaluate against each other. Pritest needs Maven and JUnit 4 to work, the way it is designed today. JUnit Max is an Eclipse plugin, and is not applicable for other IDEs. Also, it runs tests when the test case is saved in Eclipse, which is quite a handy feature.

As mentioned earlier, Max runs a lot of small tests first, and the larger, slower ones in the end, in addition to running recently failed tests before tests that have passed several times in the past. Thus, Max does not allow configuring which techniques to use (e.g., if different prioritization techniques is best with different types of projects). The selection of prioritization techniques is an advantage of Pritest, and the tool allows the users to develop their own custom techniques for developers seeing special needs for their project.

To summarize, a drawback to JUnit Max is that it can only be used with the Eclipse IDE. An advantage is that it is well integrated with this IDE, and it is seamless to run also during development of non-Maven projects. Pritest offers features like choosing what prioritization technique to use when configuring the plugin, as well as custom development of prioritization techniques. In addition, it runs simply by typing `mvn test` in console from within a Maven project directory. When running Maven projects—maybe larger projects with a lot of test case—and when in need for special types of prioritization algorithms, Pritest would be the best choice.

6.4 Rationale for Chosen Techniques

6.4.1 Improving the Existing Techniques

In Section 4.1 we described how the Pritest JUnit Runner treats test cases that are unknown to the Pritest Server when running online techniques (Section 4.3), by running them before the other test cases. This is not necessarily the best solution. A better alternative could be to send the necessary information about each test case to the server, and utilize it together with the information already available at the server.

With the technique *Counting Failing Tests* (Section 4.3.1) this could be realized by having a local database where data about the failures of test cases not yet committed to the central repository are stored. These local failure data would be sent to the server when requesting a prioritized list of test cases. That being said, the data should not be stored on the server before it is certain that the new code is to be included in the code base. That is, not before it has been committed to the central repository.

6.4.2 The Choice of Prioritization Techniques to Implement

The techniques we chose to implement were selected for different reasons. The online techniques, *Counting Failing Tests* (Section 4.3.1) and *Code Changes* (Section 4.3.2), were selected for their simplicity. The rationale behind this is that a sub-optimal technique might give good enough results.

The *Local Code Changes* technique (Section 4.4.1) and its derivative (Section 4.5.1)

were developed, and selected, based on our experience with developing software. When developing software, we tend to use a version control system—like Git—and work in short iterations. We often commit code to the central repository, for every new feature implemented or so. This way, we put a limit on the amount of code that gets wasted if a roll-back must be performed. A consequence of this is that you work on only a few classes and test cases at a time. Any test failures are therefore likely to happen in these test cases.

In addition to coming up with techniques of ourselves, we had to run some techniques developed by others too, in order to compare our techniques with existing ones. Our primary choice fell on *Additional Method Coverage* (Section 4.4.2), developed by Rothermel et al. Other alternatives were techniques analyzing statement coverage and block coverage (Section 3.1.2). Although Rothermel et al. argues that the more low-level techniques (e.g. those operating on the statement level) perform better than high-level ones (e.g. those on the method level), we decided to implement a method coverage technique, as a low-level code analysis require more implemented code than a high-level one, and we operate with a limited time budget. Therefore, we implemented *Additional Method Coverage*. Implementation-wise, this technique shares a lot with *Total Method Coverage*; as a result, we also implemented this one.

We also needed some control techniques to compare our techniques with during the experiment (Section 4.6), so we implemented the techniques used by Rothermel et al. in their studies. These are: untreated (or original) order, random order and optimal order.

- Optimal ordering was implemented because it gives us an upper limit on how good a technique can possibly perform on a given test run.
- As untreated ordering shows us how a test run would look like without any

prioritization, we used this one as well.

- Random ordering was selected because it gives us a lower bound for how good our techniques should be able to perform. Much like untreated ordering.

There are probably many more techniques we could have included in the experiment—something that will be discussed shortly, in Section 6.4.3—but we had to set a boundary somewhere due to time limitations.

6.4.3 Potential Prioritization Techniques

Considering the prioritization techniques we implemented, new hybrid techniques can be designed based on the existing ones, including some completely different. For example:

- *Finding correlations between changed code and failing tests.* One possibility would be to assign a weight to each *test case–code change* relationship. A test failure that happens right after a code change would consequently increase the weight. Test cases would then be prioritized by the weights they have with the changed code. E.g. if we have three classes (Figure 6.1): A, B and C; each with their own test case. As class C is modified, the test cases will be executed in the following order: test case C, with a weight of 0.9; test case A, with a weight of 0.8; test case B, with a weight of 0.6. Weights could be assigned by giving points to the relevant class–test case association when a code change correlates with a test failure. For example, if class A has just been changed, and then test case C fails, the weight between that class and that test case is increased. By normalizing the weights we avoid large values. The challenge with a technique like this, is to develop a fitting formula for adjusting the weights.

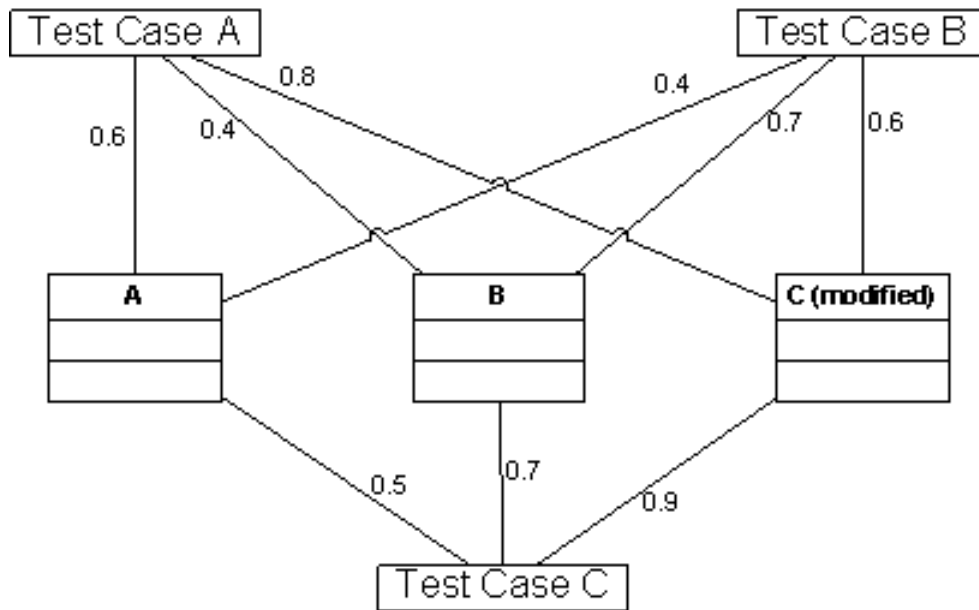


Figure 6.1: Weighted relationships between code changes and test failures.

- *Using code dependencies with information about local code changes.* Related to the *Local Code Changes* technique (Section 4.4.1), but also uses code dependencies to prioritize test cases not directly affected by a change, that have dependencies on changed code. E.g.: if we have a development project with four classes: A, B, C and D (Figure 6.2). A depends on B, and B depends on C and D. If B then gets changed, this could potentially affect both A and B; therefore, A's and B's test cases should be executed first. The set of test cases affected by a change could also be prioritized internally. For example, by their distance to the change. The closer a test case is to the changed class, the higher is its priority. Given the example in Figure 6.2, B would have a higher priority than A, which in turn would have a higher priority than the rest.

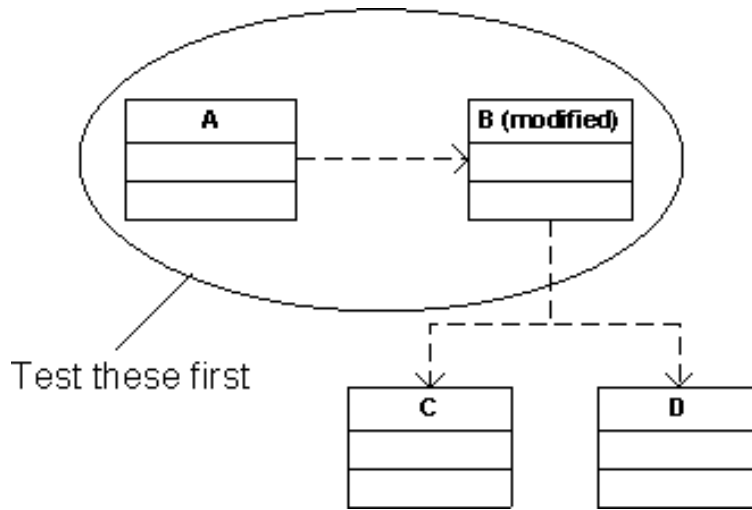


Figure 6.2: Local code changes and dependencies.

6.5 The Choice of Evaluation Metric

To address our problem definition and compare the effectiveness of several test case prioritization techniques, we needed a measure of how rapidly a prioritized test suite detects faults. We chose to use a weighted average of the percentage of faults detected (APFD) during execution of the suite [7].

This measure does not incorporate the cost of performing the prioritizations, but our goal was only to evaluate the prioritization of test cases according to their likelihood of failing. When concerning time cost of the techniques, we compensated by evaluating this in Section 4.7, using big-O analysis as described in Section 3.3.2.

Other measurement techniques that were up for evaluation was interpolated precision [31] (Section 3.1.4). However, since this metric does not provide a single value, rather one value for each 10% of the test suite being run (0%, 10%,...,100%), we decided to go with the APFD metric.

6.6 Abandoning Technique Counting Failing Tests the Last Three Days

In the specialization project [1] we designed and implemented a version of technique *Counting Failing Tests* with a three day cut-off. This technique prioritized test cases by counting the test cases history of failing the last three days, and ordering them descendingly. The technique is still present in the Pritest source code, and can be configured, but it was not further improved or altered this semester, and it was not included in our experiments.

This technique did not perform very well due to its simple binary cut-off, and we decided to not focus on it this semester. However, in retrospect we can think of several areas for possible improvements to this technique. E.g., if the cut-off value were a parameter that could be set in a configuration file, and not just hard coded as three days, the technique would be much more useful. In addition if not calendar days were the cut-off object, but rather the number of test runs since the last failure would probably be better. The reason for this is that days when there is no activity in the code base also are included in the cut-off in the present solution, with the same priority as days when the code is altered a lot. One could also think of a solution where some kind of weighting were involved. E.g., if a weight in inverse ratio were assigned to test cases, so that a test case gets a lower weight if it failed a long time ago.

6.7 Benchmark - Custom JUnit Runner

Last semester we developed a custom JUnit runner for our Pritest project. This semester, we decided to improve it, and ended up building a brand new one. We performed a benchmark for comparing the performance of the new custom JUnit runner against the old one. The goal for this benchmarking is to evaluate the *test scheduling performance* of the runners, and not their ability to detect failing tests or the quality of prioritization lists. Hence, we simply created a test suite of 7000 test cases, each containing five tests.

The test cases were auto-generated, and were of various complexities (e.g., some tests were simple mathematical addition functions, and others were mathematical power calculations). The test cases were generated using a bash script (Listing 6.1), which manipulated a set of existing test cases with new numerations of the same class with the same test cases inside. The script also alters the source code by editing the class signature header to comply with the new generated test class name.

```
1 #!/bin/bash
2 COUNTER=1
3 if [ -e "$1Test.java" ]; then
4     while [ $COUNTER -le $2 ]; do
5         echo "Copying $1.java to $1${COUNTER}Test.java"
6         cp $1Test.java $1${COUNTER}Test.java
7         echo -e "Replacing $1Test with $1${COUNTER}Test in $1${COUNTER}Test.
8             java\n"
9         sed -i "s/$1Test/$1${COUNTER}Test/g" $1${COUNTER}Test.java
10        let COUNTER=COUNTER+1
11    done
12 else
13     echo "Javafile $1Test.java does not exist"
14 fi
```

Listing 6.1: Bash script generating multiple test cases.

The script requires two arguments (\$1 and \$2), respectively the file to alter, and the number of test cases to generate. Figure 6.3 visualizes the command that runs the script, with two arguments `FactorialTest` and `3`.

```
[~/dev/calculator/src/test/java/no/ntnu]$ ./testclassgenerator.sh FactorialTest 3
Copying FactorialTest.java to FactorialTest1.java
Replacing FactorialTest with FactorialTest1 in FactorialTest1.java

Copying FactorialTest.java to FactorialTest2.java
Replacing FactorialTest with FactorialTest2 in FactorialTest2.java

Copying FactorialTest.java to FactorialTest3.java
Replacing FactorialTest with FactorialTest3 in FactorialTest3.java

[~/dev/calculator/src/test/java/no/ntnu]$ █
```

Figure 6.3: Bash script run.

The benchmarks were carried out in a controlled environment, using the same computer, hardware and software for both the new and the old runner. Table 8.2 displays a summary of the results from the benchmark.

	New custom JUnit runner	Old custom JUnit runner
Source lines of code (SLOC)	294(*)	1337
Total test run time	31 seconds	53 seconds
Tests pr. second	71.9	42.1

Table 6.1: Custom JUnit runner benchmark.

* *Source lines code of the actual runner, excluded the implementation of local prioritization techniques located in this module.*

This shows that our new custom JUnit runner improves the scheduling of test cases, and would certainly improve the efficiency when running large test suites.

6.8 Continuous Integration

Continuous Integration (CI) is a software development practice where developers integrate their work frequently, and each integration is validated by an automated build in order to detect errors as quickly as possible [36]. CI embraces the idea of running the entire test suite only after code check-ins on a shared server. This enables the individual developer to only run tests that covers the features being implemented at the time, and not the entire test suite of the project each time. This would to some extent lower the need for a system like Pritest.

However, CI is meant to be used as a safety net, and would *ideally* not find any errors when building [36], all though this is often hard to obtain. One way of preventing it is by running the entire test suite locally on the individual developers' machine in advance of committing code. Pritest would be of much use when doing so. When running the test suite locally, especially if it is a large test suite, it is useful to get the test results e.g. within three seconds rather than 30 seconds. If we additionally implemented a “failfast flag”, a mechanism for instantly exiting test runs when failures occur, the usability would be further improved (discussed in Section 7.2).

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this project we have researched the possibilities of prioritizing test cases within a test suite, so that test cases with a high probability of failing will be run first in the test execution. We implemented prioritization techniques in a tool called Pritest. The assignment was given by BEKK Consulting AS, and the motivation behind it was the increased popularity of using automated testing throughout the development lifecycle. Prioritizing test cases according to their likelihood of failing, would lower the feedback time for developers writing a lot of unit tests.

We started our work by researching previous and related work on the topic, and found that Rothermel et al. [4, 9, 12, 13, 14] have developed techniques with a similar purpose. In addition we conducted an industrial survey to gain a better understanding of the problem domain. The results show that slow test runs are in fact a challenge in the Norwegian software industry, and that there is a market for

a product like Pritest. We also did a technical prestudy to improve our skills and knowledge in the technologies we were about to use.

We designed four techniques for prioritizing test cases. In addition, we implemented two techniques designed by Rothermel et al. for comparison, as well as three “control techniques” used only for experimental purposes. The design of the techniques we implemented ourselves was determined through discussions, consultation with our external supervisor, partially based on our own experience with software development and automated testing, as well as the results from the industrial survey. Our techniques were divided into three categories: *online*, *local* and *hybrid techniques*, where online techniques make use of a service using a REST interface for sending test case priority lists to requesters and receiving reports after test runs. Local techniques prioritize test cases based on information available locally, and hybrids consist of two or more existing techniques.

After we finished implementing the tool, we ran an experiment to investigate if any of the prioritization techniques are better than the others. This was done by having two developers at BEKK use the tool in their everyday work for three days. The techniques are listed descendingly, by performance:

1. Counting Failing Tests.
2. Additional Method Coverage.
3. Total Method Coverage.
4. Local Code Changes with Failure Counting.
5. Local Code Changes.

We ran the secondary experiment using our tool after introducing errors in the code, and collecting the results. The techniques are listed descendingly, by performance:

1. Local Code Changes with Failure Counting.
2. Local Code Changes.
3. Counting Failing Tests.
4. Additional Method Coverage.
5. Total Method Coverage.

The results from both experiments were significant, with p-values of 0.000. The null hypotheses could therefore be rejected, i.e. there are differences between the techniques.

In the introduction we stated our goals as researching and implementing a set of techniques for test case prioritization, compare them in experiments, and evaluate them against existing techniques. This solution should be implemented as a tool designed for software qualities like efficiency, adaptability and maintainability. Regarding the second part of our goal—the tool—we achieved the desired software qualities by implementing several design patterns and complying with best practices. Our tool was compared to JUnit Max, the only tool we found similar to Pritest. Pritest is not IDE-dependent like Max is, and provides better support for customizing prioritization techniques with respect to what is needed in the actual development project.

Due to the increased popularity in agile software methodologies and practices like TDD, where automated testing is widely used, the problem of slow running test suites

becomes a larger problem. At the same time, tools and technologies for mitigating such a problem emerge—Pritest being one of them.

7.2 Future Work

Although we are quite pleased with the results, we still see some areas of improvement.

1. When selecting what technique to use when configuring the plugin, it would improve usability and readability to name the techniques instead of just using numbers as we do now. It could even be possible to use several techniques at the same time, or define certain “fall-back” techniques if the primary technique fails to execute, e.g., if a online technique is used, and there is no internet connection. This could probably look something like this:

```
1 <primaryTechnique>countingFailingTests </primaryTechnique>
2 <fallbackTechnique>codeChanges</fallbackTechnique>
```

Listing 7.1: Possible implementation of primary and fall-back techniques configuration.

```
1 <techniques>
2   <countingFailingTests>true</countingFailingTests>
3   <codeChanges>>false</codeChanges>
4   <localCodeChanges>true</localCodeChanges>
5   ...
6   <totalMethodCoverage>true</totalMethodCoverage>
7 </techniques>
```

Listing 7.2: Possible implementation of using several techniques at once configuration.

2. Pritest is today not able to handle multiple projects at the same time. When developing different solutions simultaneously, the database that comes with Pritest must be cleaned and restored for each project. This was sufficient for our experimental development in this thesis, but if the service should be used commercially, multiple projects must be supported.
3. It would improve maintainability to gather all the techniques in one module, and having every technique implementing a common interface. This interface could e.g. have a method `getPriorityList()` that every technique implementation should override. The techniques should probably be located in our *pritest-core* module.
4. Our APFD values are now written to a file in the file system. This could be improved by writing the values to a log instead. The tool should also be able to turn off logging and APFD recording through logging configuration.
5. It should be simple to expand the system with new techniques as the need for new heuristics emerge, and software development trends change. Suggestions for new techniques are discussed in Section 6.4.
6. To reduce the threats to validity we encountered in this experiment, we can see a few more experiments that could have been run. E.g. it would be interesting to explore whether or not different techniques are suitable for different types of projects (small vs. big projects, build server vs. developers pc, old vs. new code, when in the development cycle Pritest is applied).

7. Support custom JUnit runners used in the project where Pritest is plugged in. The Pritest plugin should detect test cases with the `@RunWith`-annotation present, and use the custom runner defined in this annotation¹ to run these test cases. Today, Pritest avoids running test cases that should be run with custom JUnit runner.

8. It could improve the usability of Pritest to implement a so-called *failfast flag*. This is a method for instantly terminating an application without throwing any exceptions [62]. The test run could be terminated at the first failed test case found, so that the developer can get right back to correcting that failure without having to manually stop the test run when failing tests occur.

¹A typical `RunWith` annotation looks like this: `@RunWith(MyCustomRunner.class)`, and tells the test case which JUnit Runner it should be run with. This annotation is placed before the class construct of the test case.

Chapter 8

Appendices

8.1 Survey Free Text Replies

- You should take a look at JUnit Max, which is the latest work from Kent Beck (who is behind Test Driven Development and JUnit):
<http://www.threeriversinstitute.org/junitmax/subscribe.html>
- We usually solve the problem of slow running tests by separating unit tests and integration tests in the build process. Unit tests are fast and runs frequently, integration tests are slower and run less frequently
- Awesome topic for a master's thesis! Good luck!
- I only run the nearby tests as i develop.

Before checkin i run all unit tests. At my current solution this is 4200 tests in 63 sec.

I usually let the build server run the integration tests when i know i havent made any big changes.

- Ved å benytte continuous integration og verktøy for dette (f.eks teamcity) og staged checkin så slipper man å kjøre testene lokalt og dermed vente på at disse skal kjøre ferdig før man utvikler videre.
- I run all test for a feature all the time and all unit tests before each checkin.
- Integration tests slower cause they have to be run so often (much more often than system tests). And when you first run your system tests you know you have to wait for them.

There are also many more integration tests than system tests

- Automated testing is a must.
- TDD on web is usually the slowest parts. Even large integration tests tend to take less time than running through a large suite of Web tests.
- The more experience I get with testing, the more meaningless and harmful the

classification of "unit test", "integration test" and "system test" becomes.

- We don't wait for the tests to run. They Are run asynch on the server by Team City before each check-in.
- TDD is very useful when done right, but it's time consuming and is therefore often not prioritized
- Related to: "How much do you agree to the following statement: "The test I am interested in is often run last, or late, in the test suite"; I usually select the test I am interested in to run independent of the test suite.
- The problem of long-running tests are to a certain degree mitigated by having a Continuous Integration server. We do this by running unit tests only in the module we are working on locally, while the CI runs the entire test suite (including time consuming integration/system tests) on checkin/push to source-control.
- The question "How many times a day do you approximately run through your unit tests?" seems ambiguous. Does it refer to "all tests" like a test suite, or "some tests/tests you believe are relevant to what you are working on"? I interpreted it as "all tests/test suite".
- When working with legacy applications, it is often difficult to work test driven. We often end up fixing the bug and writing the test afterwards.

- Svarene på frekvens kan være misvisende da dette er basert på erfaring fra smidig utvikling ved bruk av unittester, men ikke ren TDD.
- Test suite and a unit test for the specific code you are implementing should be differentiated in this quiz.
- "Waiting for slow test runs is a problem for me": Unit tests are usually fast. Integration and System tests are usually painful.
- The question "How many times a day do you approximately run through your unit tests?" is unclear to me; I run parts of the test suite lots of times a day, but I only run the entire suite a few times a day. The answer I gave is how often I run a part of the suite.
- "Michael Fogus: I've found many Scala and Haskell programmers who possess an extremely acute sense of humor. Is it static typing that attracts these minds, or does it create them?"

Martin Odersky: Maybe it's the creative pauses forced on them when they wait for the type-checker to finish ;-)" - <http://blog.fogus.me/2010/08/06/martinodersky-take5-tolist/>

Could the same be true for slow running tests?

- Du kan ikke gjøre TDD med tester som går så sakte som dere antyder i disse spørsmålene.
- During development, only the tests pertinent to the change in question is run. CruiseControl run the full set of tests, including integration tests. We do not have any automatic systems test.
- It's not enough: you need at least static code analysis and mutation testing too.
- If you use continuous integration slow tests are less of an issue

Any question just ask

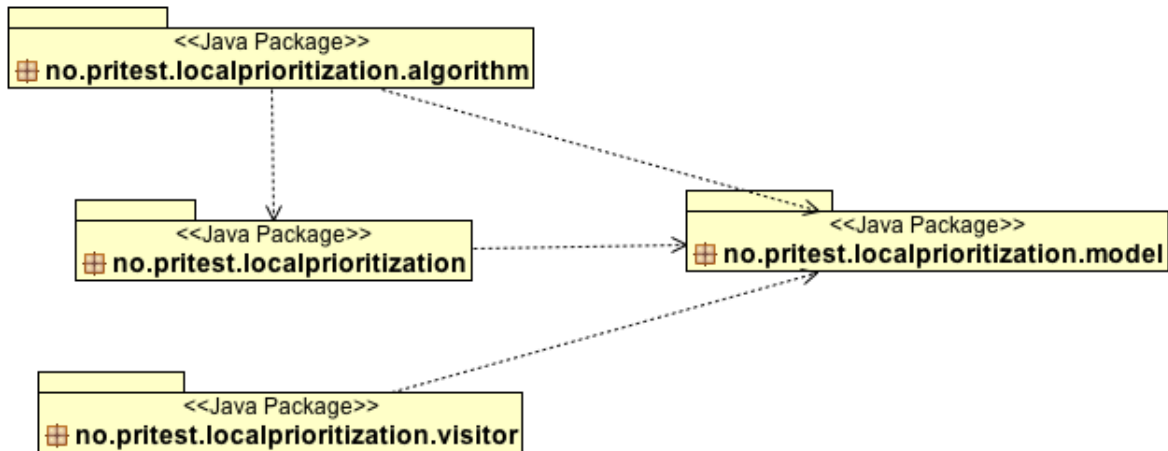
knutbo@ifi.uio.no

- Usually, I only run the unit tests that directly target the code I'm currently developing; I run the full test suite only a few times each hour. Waiting for the code to build (which takes 10+ seconds in large .NET projects, even if you have only changed a few lines and want to run a couple of tests) is usually a greater problem than waiting for the tests themselves to complete.
- We don't really differentiate between unit tests and other test, we put the slowest tests by them self. Can usually run over 1000 tests in less than 10 seconds. Also use Mighty Moose(<http://continoustests.com/>) to run only the affected tests.

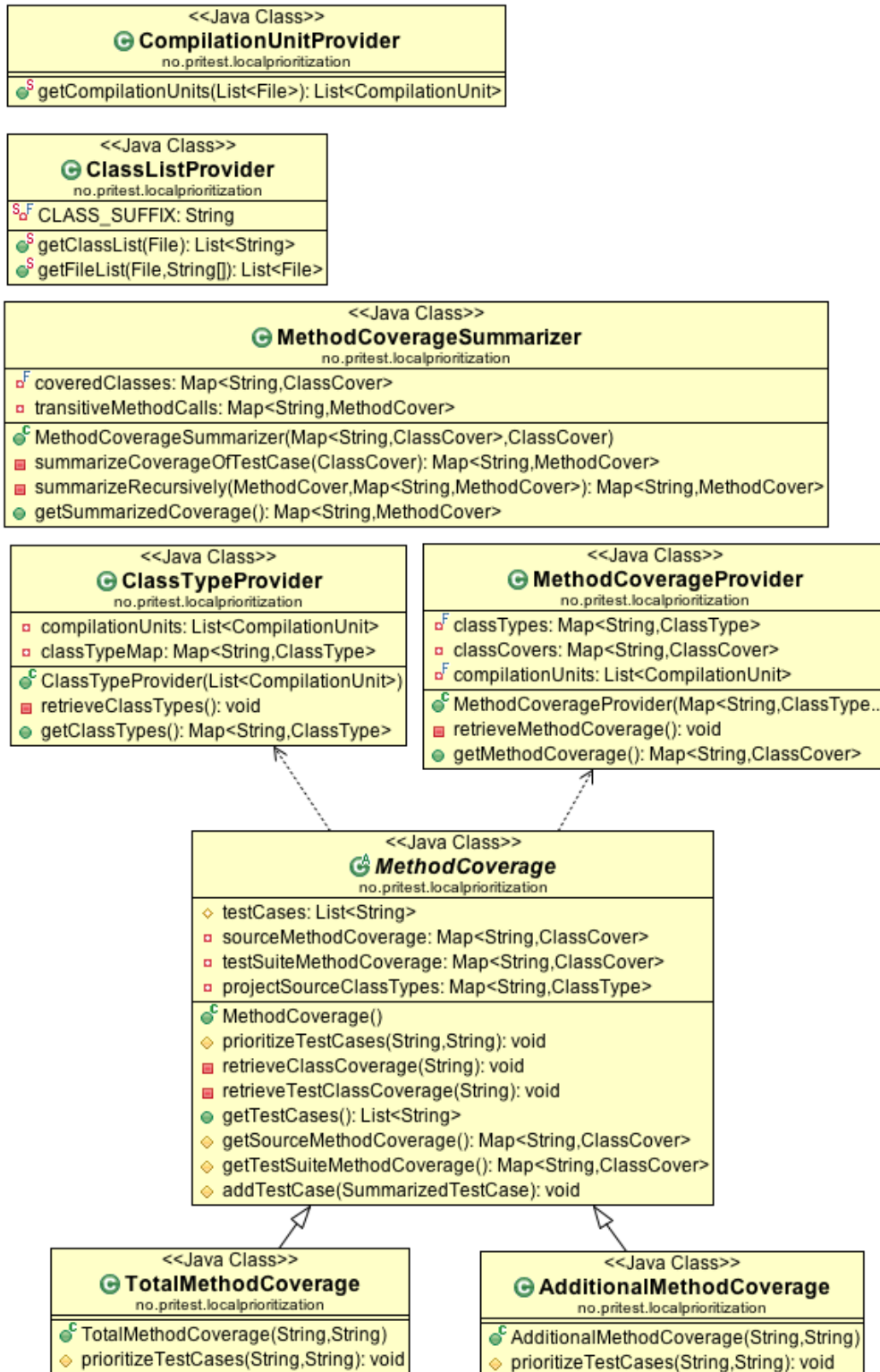
- TDD is the only way to do professional software development. If you do not use TDD, you should be sued for negligence when the software fails.
- For meg er det selenium som bruker tid da den skal laste nettsiden og utføre klikk på siden, vente på respons osv... Enhetstester går raskt.
- Web tests are not worth the effort (watir, selenium etc). Too rigid tests are worse than no tests, makes trivial changes complicated. Need something in between. If you need to touch the tests during refactoring, then the value of the tests are lost.
- The tests have very limited value once the feature is completed and checked in. I see the real value in speeding up my development

8.2 UML Diagrams for Pritest

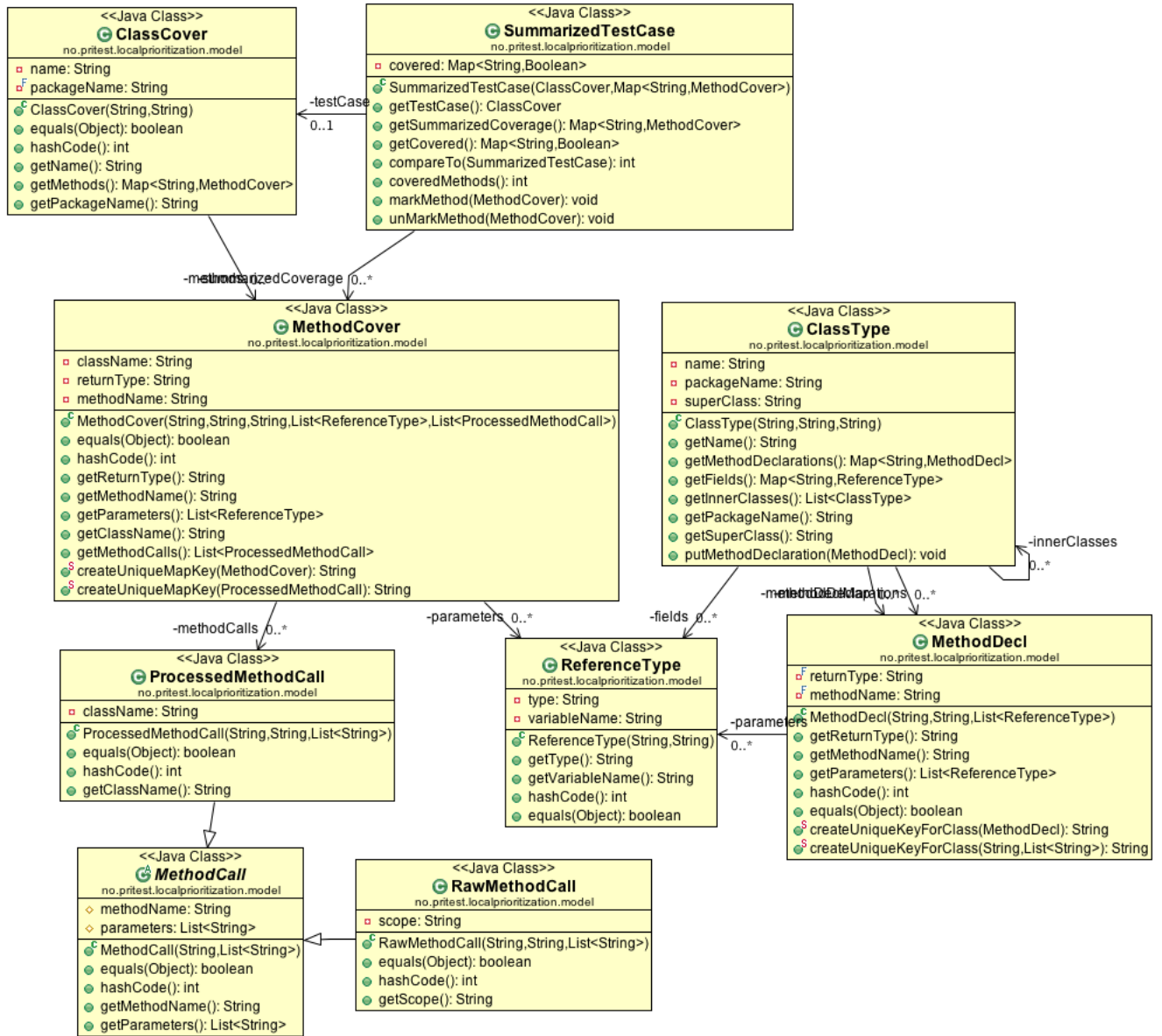
pritest-junit-runner : no.pritest


















pritest-junit-runner : no.pristest.localprioritization



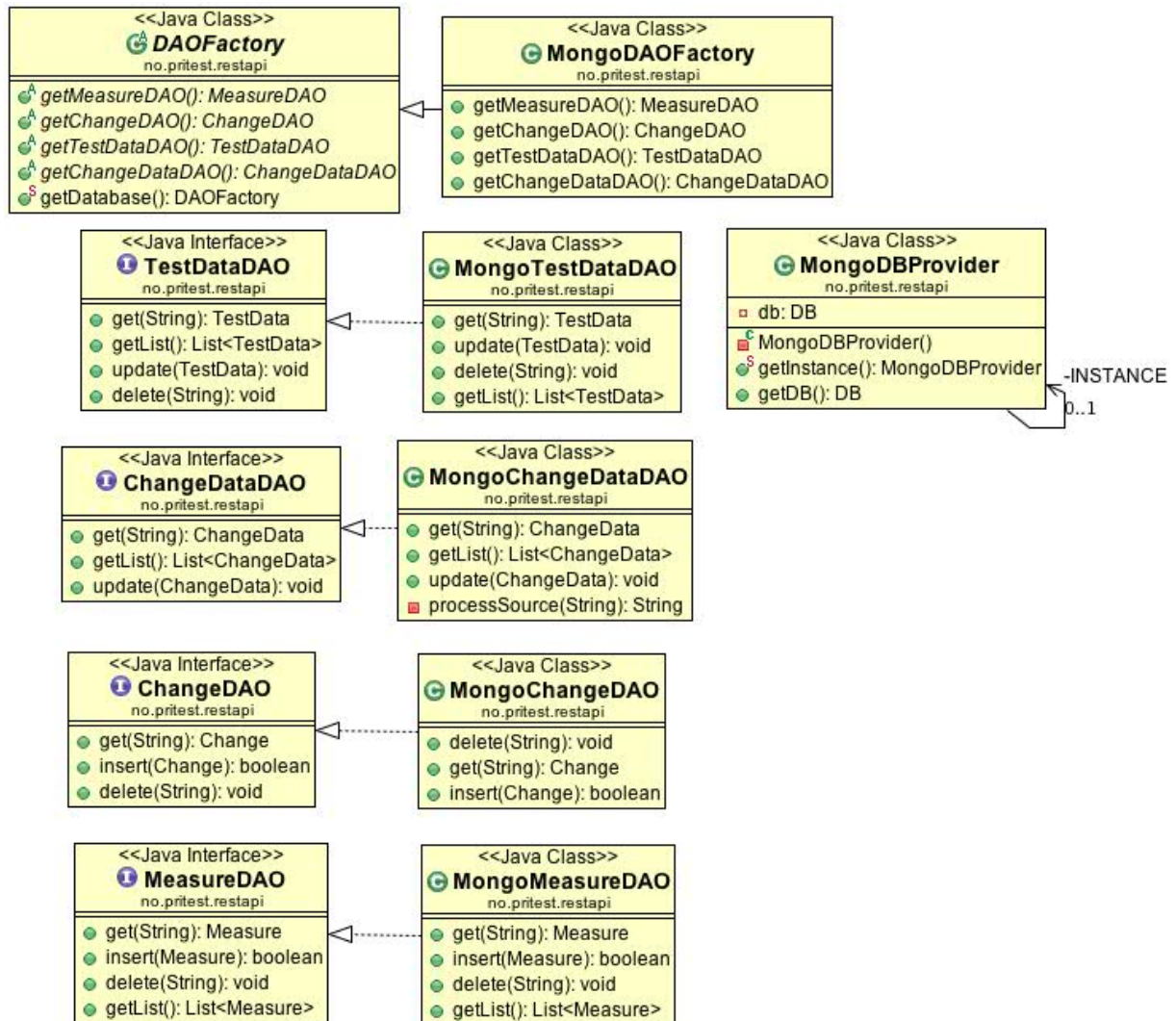
pritest-junit-runner : no.pritest.localprioritization.model



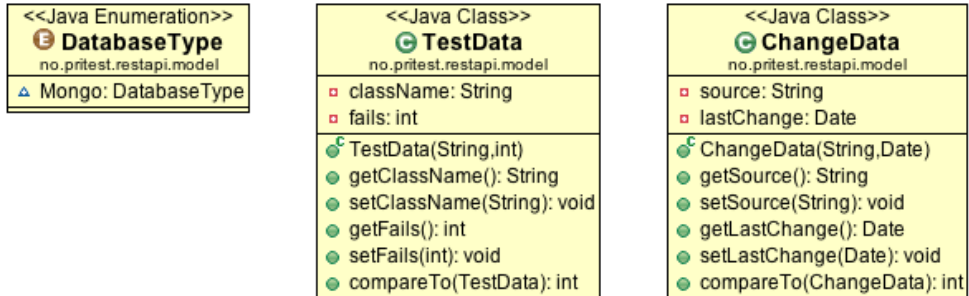
pritest-junit-runner : no.pritest.localprioritization.algorithm

<<Java Class>>	
 MethodCoverageAlgorithm	
no.pritest.localprioritization.algorithm	
	 <code>sortTestCasesByCoverage(Map<String,ClassCover>,Map<String,ClassCover>): List<SummarizedTestCase></code>
	 <code>totalMethodCoverage(Map<String,ClassCover>,Map<String,ClassCover>): List<SummarizedTestCase></code>
	 <code>additionalMethodCoverage(Map<String,ClassCover>,Map<String,ClassCover>): List<SummarizedTestCase></code>
	 <code>additionalMethodCoverageHelper(Map<String,ClassCover>,List<SummarizedTestCase>): List<SummarizedTestCase></code>
	 <code>markMethodAsCovered(List<SummarizedTestCase>,Map<String,MethodCover>): void</code>
	 <code>unMarkCoveredMethods(List<SummarizedTestCase>): void</code>
	 <code>coveredMethodsInSource(Map<String,ClassCover>): int</code>

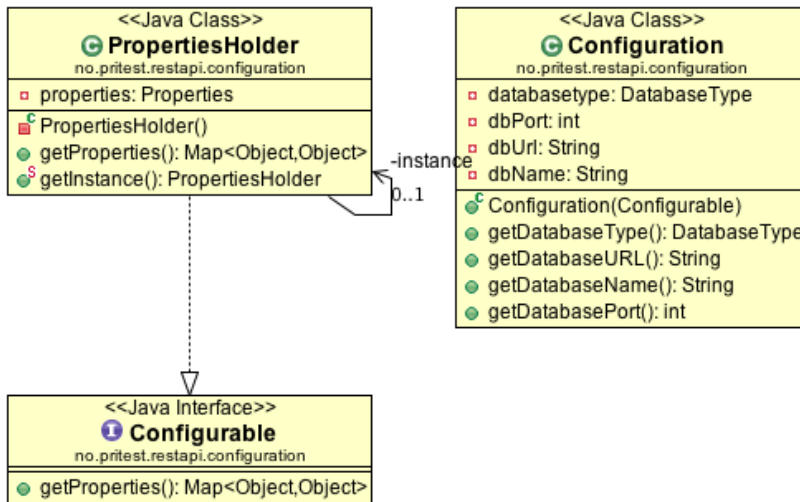
pritest-server: no.pritest.restapi



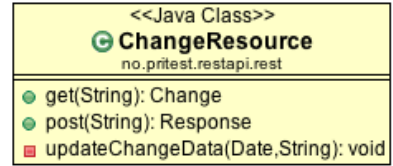
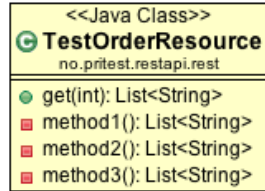
pritest-server: no.pritest.restapi.model



pritest-server: no.pritest.restapi.configuration



pritest-server: no.pritest.restapi.rest



8.3 Post-Experiment Interview Replies

Question: How did the setup and configuration of Pritest go?

Simple once the class loading issues were figured out. The plugin unfortunately only supports single-module maven builds at the moment, so we chose to manually configure Pritest for the webapp module, which is most heavily developed at the moment.

Question: How was it to use Pritest in your everyday work?

Unfortunately Pritest does not have support for tracking our JavaScript-unit tests (which are run by another maven plug-in), so according to test data sent from us it was not so representative of the development changes that had been done during the testing period (as most were web front end related changes).

Question: Compared to how you run your tests today (surefire), were there any differences using Pritest?

Surefire supports multi-module maven builds a bit better. Thus we can run that from the root project, while Pritest needs to be run inside single sub-modules.

Due to some custom JUnit runners used by some integration-tests, we do not run Pritest for integration tests as we do with surefire.

Question: Have you used JUnit Max?

Yes.

Question: If "Yes", how would you compare Pritest to JUnit Max?

Pritest is a good start, and the architecture resembles JUnit Max. However JUnit Max is a lot further in development and stability (having been developed for over three years that would be natural). I like that Pritest is open source, thus the whole community can contribute to it.

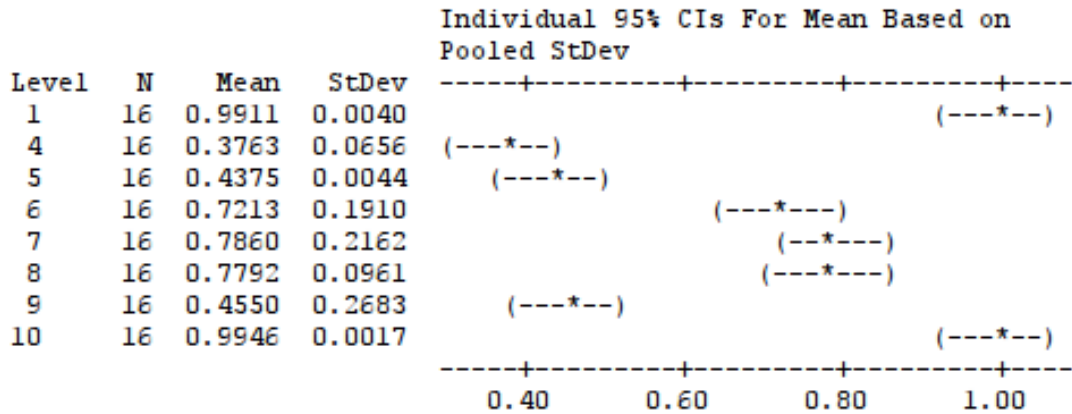
8.4 Experiment Results

8.4.1 Minitab Hypothesis Testing

One-way ANOVA: APFD value versus Technique number

Source	DF	SS	MS	F	P
Technique number	7	6.7022	0.9575	45.37	0.000
Error	120	2.5324	0.0211		
Total	127	9.2346			

S = 0.1453 R-Sq = 72.58% R-Sq(adj) = 70.98%



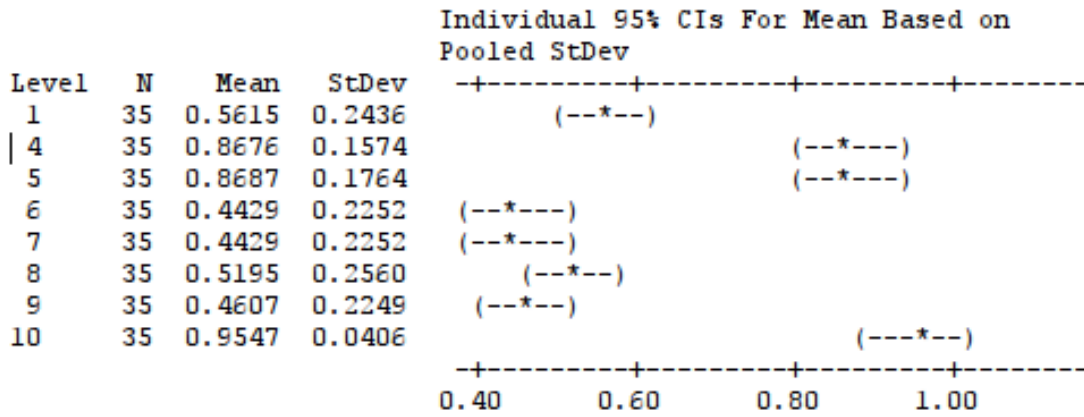
Pooled StDev = 0.1453

Figure 8.1: Minitab output - primary experiment.

One-way ANOVA: APFD value versus Technique number

Source	DF	SS	MS	F	P
Technique number	7	11.6796	1.6685	39.92	0.000
Error	272	11.3679	0.0418		
Total	279	23.0476			

S = 0.2044 R-Sq = 50.68% R-Sq(adj) = 49.41%



Pooled StDev = 0.2044

Figure 8.2: Minitab Output - secondary experiment.

8.4.2 APFD Values

Each line in the following tables correspond to a single test run and gives an APFD value for each technique.

Techniques							
1	4	5	6	7	8	9	10
0.9925595238	0.3913690476	0.4360119048	0.7723214286	0.8169642857	0.8020833333	0.3229166667	0.9925595238
0.9955357143	0.3913690476	0.4389880952	0.7723214286	0.8169642857	0.8020833333	0.4836309524	0.9925595238
0.9866071429	0.3973214286	0.4330357143	0.7544642857	0.8348214286	0.7991071429	0.7276785714	0.9955357143
0.9866071429	0.3883928571	0.4330357143	0.7723214286	0.8348214286	0.8080357143	0.7366071429	0.9955357143
0.9955357143	0.3883928571	0.4330357143	0.7544642857	0.8348214286	0.7991071429	0.7098214286	0.9955357143
0.9866071429	0.3973214286	0.4419642857	0.7544642857	0.8526785714	0.8080357143	0.4508928571	0.9955357143
0.9895833333	0.3883928571	0.4419642857	0.0937500000	0.0937500000	0.7991071429	0.0491071429	0.9955357143
0.9955357143	0.3938053097	0.4419642857	0.8008849558	0.9062500000	0.8080357143	0.5491071429	0.9955752212
0.9910714286	0.3928571429	0.4375000000	0.8080357143	0.8451327434	0.8080357143	0.0223214286	0.9955357143
0.9910714286	0.3973214286	0.4330357143	0.8080357143	0.9062500000	0.7991071429	0.1902654867	0.9955357143
0.9955357143	0.3883928571	0.4375000000	0.8035714286	0.9062500000	0.8035714286	0.9866071429	0.9955357143
0.9866071429	0.3928571429	0.4369369369	0.4279279279	0.9062500000	0.8008849558	0.2276785714	0.9910714286
0.9954954955	0.3883928571	0.4330357143	0.8080357143	0.8750000000	0.8080357143	0.5178571429	0.9955357143
0.9866071429	0.3973214286	0.4330357143	0.8080357143	0.8750000000	0.8035714286	0.6294642857	0.9910714286
0.9866071429	0.1306306306	0.4419642857	0.7991071429	0.4279279279	0.7991071429	0.4285714286	0.9955357143
0.9955752212	0.3973214286	0.4469026549	0.8035714286	0.8437500000	0.4189189189	0.2477477477	0.9954954955

Table 8.1: APFD result from the primary experiment.

CHAPTER 8. APPENDICES

Techniques							
1	4	5	6	7	8	9	10
0.5844444444	0.9629629630	0.9629629630	0.0555555556	0.0555555556	0.5740740741	0.1666666667	0.9814814815
0.2407407407	0.7169312169	0.9567901235	0.4135802469	0.4135802469	0.2345679012	0.4506172840	0.9444444444
0.2407407407	0.9814814815	0.9629629630	0.3148148148	0.5370370370	0.6604938272	0.5493827160	0.9629629630
0.6569664903	0.9629629630	0.9148148148	0.4111111111	0.3148148148	0.5370370370	0.3518518519	0.9691358025
0.8333333333	0.9567901235	0.9567901235	0.3888888889	0.4111111111	0.7666666667	0.2901234568	0.9469135802
0.5592592593	0.8888888889	0.9444444444	0.1296296296	0.3888888889	0.3271604938	0.3148148148	0.9814814815
0.7074074074	0.8148148148	0.9814814815	0.5370370370	0.5370370370	0.8148148148	0.2222222222	0.9629629630
0.7222222222	0.9814814815	0.9814814815	0.3703703704	0.3703703704	0.8333333333	0.0925925926	0.9814814815
0.8518518519	0.8851851852	0.7777777778	0.5740740741	0.1296296296	0.3333333333	0.2037037037	0.9814814815
0.3148148148	0.7962962963	0.8777777778	0.6407407407	0.0185185185	0.2777777778	0.7148148148	0.9592592593
0.7962962963	0.8597883598	0.9814814815	0.4703703704	0.5740740741	0.4259259259	0.0925925926	0.9814814815
0.6851851852	0.8777777778	0.9518518519	0.5740740741	0.6407407407	0.9074074074	0.5355555556	0.8007407407
0.1203703704	0.9629629630	0.9629629630	0.4629629630	0.4703703704	0.3148148148	0.2037037037	0.9814814815
0.3600823045	0.9814814815	0.8068783069	0.2936507937	0.5740740741	0.7148148148	0.7777777778	0.9629629630
0.4259259259	0.9222222222	0.6481481481	0.5972222222	0.4629629630	0.9074074074	0.4074074074	0.9567901235
0.1913580247	0.7074074074	0.6944444444	0.8827160494	0.2901234568	0.5370370370	0.6172839506	0.9382716049
0.7469135802	0.7037037037	0.9320987654	0.2901234568	0.2936507937	0.5987654321	0.3703703704	0.9629629630
0.5138888889	0.9320987654	0.8456790123	0.5370370370	0.5972222222	0.6640211640	0.3994708995	0.9338624339
0.5864197531	0.8703703704	0.9320987654	0.0555555556	0.8827160494	0.6898148148	0.7592592593	0.9814814815
0.2777777778	0.8950617284	0.9814814815	0.3888888889	0.3888888889	0.6481481481	0.3818342152	0.9629629630
0.6111111111	0.9814814815	0.3148148148	0.3888888889	0.3888888889	0.3077601411	0.2222222222	0.9518518519
0.6851851852	0.6111111111	0.9814814815	0.5592592593	0.6851851852	0.4814814815	0.4259259259	0.9814814815
0.6481481481	0.9814814815	0.6622574956	0.2716049383	0.5592592593	0.3395061728	0.5000000000	0.9814814815
0.2962962963	0.3130511464	0.9419753086	0.5740740741	0.2716049383	0.1203703704	0.8549382716	0.9666666667
0.9814814815	0.9814814815	0.9228395062	0.4037037037	0.5740740741	0.9074074074	0.5370370370	0.9444444444
0.6666666667	0.8876543210	0.6362433862	0.5158730159	0.4037037037	0.9000000000	0.8777777778	0.9518518519
0.4444444444	0.9228395062	0.8259259259	0.6037037037	0.5158730159	0.6494708995	0.3444444444	0.9398148148
0.8777777778	0.8991769547	0.9814814815	0.0555555556	0.6037037037	0.3888888889	0.3518518519	0.9814814815
0.2037037037	0.9814814815	0.9814814815	0.5987654321	0.0555555556	0.0555555556	0.5026455026	0.9691358025
0.5833333333	0.9814814815	0.9814814815	0.7962962963	0.5987654321	0.7469135802	0.6555555556	0.9660493827
0.1296296296	0.9814814815	0.9814814815	0.1666666667	0.7962962963	0.1666666667	0.7592592593	0.9003527337
0.9814814815	0.9629629630	0.9629629630	0.6851851852	0.0555555556	0.0555555556	0.1666666667	0.9814814815
0.7592592593	0.4259259259	0.9156378601	0.5658436214	0.1666666667	0.6111111111	0.7962962963	0.8187830688
0.7000000000	0.8111111111	0.2777777778	0.0185185185	0.5658436214	0.4794238683	0.6481481481	0.9629629630
0.6666666667	0.9814814815	0.9814814815	0.9074074074	0.9074074074	0.2037037037	0.5781893004	0.9814814815

Table 8.2: APFD result from the secondary experiment.

Introduction to Bibliography

This section presents our complete list of references. The bibliography consists of resources from articles, books and the internet, and is divided in that order throughout this chapter. The bibliography includes a rather large amount of web references. The reason for this is that this area is not that well researched in the past, and several technologies and concepts are not yet written in literature. This research area is viewed as “cutting-edge”.

Bibliography

- [1] S. Dalatun, S. I. Remøy, T. K. R. Seth and Ø. Voldsund, Decreasing Response Time of Failing Automated Tests Using Heuristic Functions, *Unpublished, Specialization Project Computer Science - NTNU*, <http://www.norsk-web.com/>, Trondheim, Norway, December 2010.
- [2] J. M. Kim and A. Porter, A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments, *Proceedings of the 24th International Conference on Software Engineering*, New York, USA , May 2002, pages 119-129.
- [3] S. Yoo and M. Harman, Pareto Efficient Multi-Objective Test Case Selection, *Proceedings of the 2007 international symposium on Software testing and analysis*, New York, USA, 2007, pages 140-150.
- [4] G. Rothermel, R. H. Untch, C. Chu and M. J. Harrold, Test Case Prioritization: An Empirical Study, *Proceedings of the International Conference on Software Maintenance*, Oxford, UK, September 1999, pages 179-188.
- [5] B. P. Bailey, J. A. Konstan and J. V. Carlis, The Effects of Interruptions on Task Performance, Annoyance, and Anxiety in the User Interface, *Proceedings of INTERACT, Vol. 2*, Nebraska, USA, 2001, pages 757-762.

- [6] S. B. Jenkins, Concerning Interruptions, *COMPUTER-IEEE COMPUTER SOCIETY*, Vol. 39, No. 39, Ontario, Canada, 2006, pages 1-5.
- [7] G., Roland, H. Untch, C. Chu and M. J. Harrold, Prioritizing Test Cases For Regression Testing, *IEEE Transactions on Software Engineering*, Vol. 27, No. 10, Washington, USA, October 2001, pages 929-948.
- [8] T. Mende and Rainer Koschke, Effort-Aware Defect Prediction Models, *14th European Conference on Software Maintenance and Reengineering*, Madrid, Spain, March 2010, pages 107-116.
- [9] S. Elbaum, G. Rothermel, S. Kanduri and A. G. Malishevsky, Selecting a Cost-Effective Test Case Prioritization Technique, *Software Quality Journal*, Vol. 12, No. 3, September 2004, pages 185-210.
- [10] M. Sherriff, M. Lake and L. Williams, Prioritization of Regression Tests using Singular Value Decomposition with Empirical Change Records, *The 18th IEEE International Symposium on Software Reliability*, Trollhättan, Sweeden, November 2007, pages 81-90.
- [11] H. Srikanth, L. Williams and J. Osborne, System Test Case Prioritization of New and Regression Test Cases, *2005 International Symposium on Empirical Software Engineering*, Noosa Heads QLD, Australia, November 2005, page 10.
- [12] S. Elbaum, A. Malishevsky and G. Rothermel, Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization, *Proceedings of the 23rd International Conference on Software Engineering*, Washington, USA, May 2001, pages 329-338.

- [13] S. Elbaum, A. Malishevsky and G. Rothermel, Test Case Prioritization: A Family of Empirical Studies, *IEEE Transactions on Software Engineering Vol. 28, No. 2*, Nebraska University, Lincoln, USA, February 2002, pages 159-182.
- [14] H. Do, G. Rothermel and A. Kinneer, Empirical Studies of Test Case Prioritization in a JUnit Testing Environment, *Proceedings of the International Symposium on Software Reliability Engineering*, Saint-Malo, Bretagne, France, November 2004, pages 113-124.
- [15] Dr. W. W. Royce, Managing the development of large software systems, *Proceedings of IEEE WESCON*, California Institute of Technology, San Francisco, USA, 1970, pages 328-338.
- [16] A. Clauset, C. R. Shalizi and M. E. J. Newman, Power-Law Distributions in Empirical Data, *Unpublished, Cornell University Library*, Cornell University, New York, USA, June 2007.
- [17] ISO/IEC 9126-1: Software engineering – Product quality – Part 1: Quality model, 2001-06-21, International Organization for Standardization, Geneva, Switzerland.
- [18] J. Bloch, Effective Java, *Prentice Hall, Second Edition*, May 2008, ISBN: 978-0321356680.
- [19] W. McAllister, Data Structures and Algorithms using Java, *Jones and Bartlett Publishers, First Edition*, December 2008, ISBN: 978-0-7637-5756-4.
- [20] L. Bass, P. Clements and R. Kazman, Software Architecture in Practice, *Addison-Wesley Professional, Second Edition*, April 2003, ISBN: 0-321-15495-9.

- [21] S. L. Pfleeger, *Software Engineering: Theory and Practice*, *Prentice Hall, Fourth Edition*, February 2009, ISBN: 978-0136061694.
- [22] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell and A. Wesslén, *Experimentation in Software Engineering — An Introduction*, *Springer, Kluwer Academic Publishers, First Edition*, December 1999, ISBN: 0-7923-8682-5.
- [23] E. Babbie, *Survey Research Methods*, *Wadsworth Publishing, Second Edition*, February 1990, ISBN: 0-524-12672-3.
- [24] T. Koomen, L. van der Aalst, B. Broekman and M. Vroon, *TMap Next— for result-driven testing*, *UTN Publishers, First Edition*, December 2006, ISBN: 9072194802.
- [25] D. Chelimsky, D. Astels, Z. Dennis, A. Hellesøy, B. Helmkamp and D. North, *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends*, *Pragmatic Bookshelf, First Edition*, December 2010, ISBN: 978-1-93435-637-1.
- [26] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, *Prentice Hall, First Edition*, August 2008, ISBN: 978-0132350884.
- [27] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, *Refactoring: Improving the Design of Existing Code*, *Addison-Wesley Professional, First Edition*, July 1999, ISBN: 978-0201485677.
- [28] R. Pressman, *Software Engineering: A Practitioner’s Approach*, *McGraw-Hill Science/Engineering/Math, Seventh Edition*, January 2009, ISBN: 978-0073019338.

BIBLIOGRAPHY

- [29] W. Trochim, The Research Methods Knowledge Base, *Atomic Dog, Third Edition*, December 2006, ISBN: 978-1592602919.
- [30] A. Hunt, The Pragmatic Programmer: From Journeyman to Master, *Addison-Wesley Professional, First Edition*, October 1999, ISBN: 978-0201616224.
- [31] R. Baeza-Yates and B. Ribeiro-Neto, Modern Information Retrieval, *Addison-Wesley, First Edition*, May 1999, ISBN: 978-0201398298.
- [32] K. Beck, Test Driven Development: By Example, *Addison-Wesley Professional, First Edition*, November 2002, ISBN: 978-0321146533.
- [33] JUnit Max,
<http://www.junitmax.org/>, Retrieved February 9, 2011.
- [34] Eclipse Foundation IDE,
<http://www.eclipse.org/>, Retrieved February 9, 2011.
- [35] JUnit,
<http://www.junit.org/>, Retrieved February 9, 2011.
- [36] Continuous Integration,
<http://martinfowler.com/articles/continuousIntegration.html>, Retrieved March 28, 2011.
- [37] Github,
<https://github.com/>, Retrieved March 7, 2011.
- [38] Maven,
<http://maven.apache.org>, Retrieved March 7, 2011.

BIBLIOGRAPHY

- [39] Java SDK,
<http://www.oracle.com/technetwork/java/index.html>, Retrieved March 7, 2011.
- [40] Hudson,
<http://java.net/projects/hudson/>, Retrieved March 7, 2011.
- [41] Jersey,
<http://jersey.java.net/>, Retrieved March 7, 2011.
- [42] JAXB,
<http://jaxb.java.net/>, Retrieved March 7, 2011.
- [43] Sonar,
<http://www.sonarsource.org/>, Retrieved March 7, 2011.
- [44] Git,
<http://git-scm.com/>, Retrieved March 7, 2011.
- [45] JUnit,
<http://www.junit.org/>, Retrieved March 7, 2011.
- [46] Cucumber,
<http://cukes.info/>, Retrieved March 7, 2011.
- [47] MongoDB,
<http://www.mongodb.org/>, Retrieved March 7, 2011.
- [48] Surefire Maven Plugin,
<http://maven.apache.org/plugins/maven-surefire-plugin/>, Retrieved March 7, 2011.

BIBLIOGRAPHY

- [49] Twitter - About us,
<http://twitter.com/about>, Retrieved March 21, 2011.
- [50] Byte Code Engineering Library,
<http://jakarta.apache.org/bcel/index.html>, Retrieved March 25, 2011.
- [51] Eclipse ASTParser,
<http://help.eclipse.org/helios/nftopic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/do>
Retrieved March 25, 2011.
- [52] javaparser,
<http://code.google.com/p/javaparser/>, Retrieved March 28, 2011.
- [53] JGit,
<http://www.eclipse.org/jgit/>, Retrieved March 29, 2011.
- [54] Factory Pattern,
<http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/factory.html>,
Retrieved April 4, 2011.
- [55] The Importance of Maintainable Software,
<http://www.basilv.com/psd/blog/2006/the-importance-of-maintainable-software>, Retrieved April 26, 2011.
- [56] How to Create Maintainable Software,
<http://www.basilv.com/psd/blog/2006/how-to-create-maintainable-software>,
Retrieved April 26, 2011.

BIBLIOGRAPHY

- [57] Smells to Refactorings,
<http://industriallogic.com/papers/smellstorefactorings.pdf>, Retrieved May 5, 2011.
- [58] Collections JavaDoc,
[http://download.oracle.com/javase/1.4.2/docs/api/java/util/Collections.html#sort\(java.util.List\)](http://download.oracle.com/javase/1.4.2/docs/api/java/util/Collections.html#sort(java.util.List)), Retrieved May 10, 2011.
- [59] Oracle Sun Developer Network,
<http://developers.sun.com/mobility/midp/ttips/HTTPPost/>, Retrieved May 18, 2011.
- [60] W3C,
<http://www.w3.org/XML/>, Retrieved May 18, 2011.
- [61] R. Riley - The Four Pillars of Maintainable Software,
<http://www.codeproject.com/KB/architecture/maintainablesw.aspx>, Retrieved May 19, 2011.
- [62] Stack Overflow,
<http://stackoverflow.com/questions/564581/what-is-environment-failfast>, Retrieved May 20, 2011.
- [63] Oracle Sun Developer Network,
<http://www.oracle.com/technetwork/articles/javase/index-137171.html>, Retrieved May 23, 2011.
- [64] Rail Spikes Blog,
<http://railspikes.com/2009/3/10/slow-tests-are-a-bug>, Retrieved May 23, 2011.

BIBLIOGRAPHY

- [65] Wire Researchers,
<http://wire.rutgers.edu>, Retrieved May 23, 2011.

- [66] SonarSource,
<https://github.com/SonarSource/sonar>, Retrieved June 1, 2011.

- [67] Mockito,
<http://mockito.org/>, Retrieved June 5, 2011.