SWC Software Construction

RWTH AACHEN UNIVERSITY

The present work was submitted to the Research Group Software Construction

of the Faculty of Mathematics, Computer Science, and Natural Sciences

Master Thesis

# A Framework for Regression Test Prioritization and Selection

## Ein Framework für die Priorisierung und Selektion von Regressionstests

presented by

**Christian Plewnia**

Aachen, December 4, 2015

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Andrej Dyck, M.Sc.

# Statutory Declaration in Lieu of an Oath

The present translation is for your convenience only.
Only the German version is legally binding.

I hereby declare in lieu of an oath that I have completed the present Master Thesis's entitled

A Framework for Regression Test Prioritization and Selection

independently and without illegitimate assistance from third parties. I have use no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

**Official Notification**

**Para. 156 StGB (German Criminal Code): False Statutory Declarations**
Whosoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.

**Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence**
(1) If a person commits one of the offences listed in sections 154 to 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

I have read and understood the above official notification.

# Eidesstattliche Versicherung

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Master Thesis mit dem Titel

A Framework for Regression Test Prioritization and Selection

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Aachen, December 4, 2015 (Christian Plewnia)

**Belehrung**

**§ 156 StGB: Falsche Versicherung an Eides Statt**
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicher ung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

**§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt**
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen.

Aachen, December 4, 2015 (Christian Plewnia)

# Acknowledgment

I am very grateful for the assistance given by my supervisor Andrej Dyck. Through him I discovered this great field of research in the first place. Thanks to him willingly sharing his knowledge and experience, I learned a lot. Further, he always found time for discussions, which were the origin of many valuable ideas. Last but not least, he contributed to my thesis by giving me numerous feedback.

I also like to thank Prof. Dr. rer. nat. Horst Lichter for giving me the opportunity to do my thesis at the Research Group Software Construction on this interesting field. Additionally, I am grateful for the acceptance of Prof. Dr. rer. nat. Bernhard Rumpe to review my thesis.

Furthermore, I wish to thank Andreas Steffens for joining some of the discussions between Andrej and me, the people at the Chair for the great atmosphere, my friends who did a great job in proofreading, and my family for their support.

*Christian Plewnia*

# Abstract

Regression testing ensures that modifications and improvements of new software versions do not introduce bugs to existing functionality. Automation of regression tests not only improves efficiency, but is an absolute necessity with shorter release cycles of modern software. However, the test suites are growing along with the software; thereby, making the execution of all tests in time impossible. Hence, further optimization of the regression test's efficiency is needed. To this end, research yielded numerous contributions describing *regression test optimization* (RTO) techniques that, for example, automatically prioritize tests or select only those tests that are relevant to the changes.

Yet, these techniques did not make their way into practice. They either do not scale to real projects or their integration into existing projects poses a problem. Furthermore, the initial effort to research RTO techniques is high. It involves more than the already difficult development of a prioritization or a selection algorithm: The techniques need to be integrated in the testing process, required information must be gathered, and the optimized test suite must be executed. It is not surprising that there are hardly any RTO tools available. Hence, a platform for an easy implementation of RTO techniques, their integration into existing projects, and their evaluation is needed.

This thesis establishes the foundations for an RTO platform. To this extent, the state-of-the-art of RTO is studied. With this in mind, a concept for an RTO framework that eases the implementation and integration of regression test prioritization and selection techniques is introduced. Further, the thesis presents a functional prototype along with exemplary extensions that demonstrate the implementation of RTO techniques and the framework's integration. Finally, suggestions for future research are made.

# Contents


**1 Introduction** **1**

**2 Regression Test Optimization** **5**
2.1 Background . . . 5
2.2 State of Research . . . 9
2.3 Applying Regression Test Optimization . . . 13
2.4 Need for a Platform . . . 16

**3 Concept** **17**
3.1 Background . . . 17
3.2 Objectives and Requirements . . . 22
3.3 Architecture of the Framework . . . 28

**4 Realization** **35**
4.1 Background . . . 35
4.2 Usage API . . . 45
4.3 Extension API . . . 48
4.4 Lazzer Framework . . . 54
4.5 Exemplary Framework Extensions . . . 54

**5 Evaluation of Lazzer** **65**
5.1 Software Quality Analysis . . . 65
5.2 Static Code Analysis . . . 72

**6 Conclusion** **75**
6.1 Summary . . . 75
6.2 Future Work . . . 76

**Bibliography** **79**

# List of Tables

2.1 Overview of regression test optimization tools . . . . . . . . . . . . . . . . . 13

3.1 Requirements for the RTO framework from the user's perspective . . . . . 25
3.2 Requirements for the RTO framework from the developer's perspective . . 27

# List of Figures

2.1 Illustration of safety and precision of regression test selection . . . . . . . 7
2.2 Illustration of efficiency of regression test selection . . . . . . . . . . . . . 8
2.3 Illustration of the advantage of an improved test order . . . . . . . . . . . 8
2.4 Screenshots of Infinitest . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3.1 Inversion of control of frameworks compared to libraries . . . . . . . . . . 18
3.2 UML class diagram of template method pattern . . . . . . . . . . . . . . . 19
3.3 UML class diagram of framework extension by interface implementation . 19
3.4 Example of a framework configuration . . . . . . . . . . . . . . . . . . . . 20
3.5 UML class diagram of xUnit's basic architecture . . . . . . . . . . . . . . 22
3.6 UML use case diagram of the framework's usage . . . . . . . . . . . . . . 23
3.7 UML use case diagram for an extension developer . . . . . . . . . . . . . . 25
3.8 UML component diagram of the framework's coarse-grained architecture . 28
3.9 UML component diagram of the framework's detailed architecture . . . . 30
3.10 RTO pipeline . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
3.11 UML sequence diagram of the RTO pipeline . . . . . . . . . . . . . . . . . 33
3.12 UML component diagram illustrating dependency loading . . . . . . . . . 33

4.1 Maven lifecycle . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
4.2 Screnshots of compiling a project with Maven . . . . . . . . . . . . . . . . 41
4.3 lllustration of the FreeMarker template engine . . . . . . . . . . . . . . . . 42
4.4 Illustration of an exemplary object-relational mapping . . . . . . . . . . . 44
4.5 UML class diagram of Lazzer's configuration . . . . . . . . . . . . . . . . 45
4.6 UML class diagram of Lazzer's dependency loading assistant . . . . . . . 47
4.7 UML class diagram of Lazzer's instation mechanism . . . . . . . . . . . . 48
4.8 UML class diagram of the data store interface . . . . . . . . . . . . . . . . 49
4.9 UML class diagram of the optimization strategy interface . . . . . . . . . 50
4.10 UML class diagram of the test framework adapter interface . . . . . . . . 51
4.11 UML class diagram of Lazzer's test data structure . . . . . . . . . . . . . 52
4.12 UML class diagram of the framework's internal architecture . . . . . . . . 53
4.13 UML class diagram of Lazzer's Maven plug-in . . . . . . . . . . . . . . . . 55
4.14 UML class diagram of Lazzer's command line interface . . . . . . . . . . . 57
4.15 UML class diagram of the exemplary data stores . . . . . . . . . . . . . . 59
4.16 UML class diagram of the exemplary optimization strategies . . . . . . . . 60
4.17 UML class diagram of the JUnit 4 test framework adapter . . . . . . . . . 63

5.1 Categories of ISO/IEC 25010. . . . . . . . . . . . . . . . . . . . . . . . . . 66
5.2 Lazzer run time evaluation: scatter plot . . . . . . . . . . . . . . . . . . . 67

5.3 Lazzer run time evaluation: box-and-whiske plot . . . . . . . . . . . . . . . 67

# List of Source Codes


4.1 Example of a unit test written with JUnit . . . . . . . . . . . . . . . . . . . 37
4.2 Example of a Maven pom.xml . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.3 Example of manual dependency injection . . . . . . . . . . . . . . . . . . . 41
4.4 Example of dependency injection with Guice . . . . . . . . . . . . . . . . . 41
4.5 Example of a FreeMarker template . . . . . . . . . . . . . . . . . . . . . . 43
4.6 Example of setting up a FreeMarker model in Java . . . . . . . . . . . . . 43
4.7 Instantiation of Lazzer's settings . . . . . . . . . . . . . . . . . . . . . . . 46
4.8 Usage of Lazzer's dependency loading assistant . . . . . . . . . . . . . . . 46
4.9 Minimal working example of running Lazzer . . . . . . . . . . . . . . . . . 49
4.10 Minimal working example of the integration of Lazzer into Maven . . . . . 56
4.11 Lazzer command line interface usage exampke . . . . . . . . . . . . . . . . 58
4.12 Alphabetic prioritization strategy . . . . . . . . . . . . . . . . . . . . . . . 61

5.1 Maven repository for Lazzer . . . . . . . . . . . . . . . . . . . . . . . . . . 68

# 1 Introduction

The primary concern of software testing is to uncover defects and to track down their cause and correct them. Test execution and verification can be automated by exercising the system under test and comparing the actual to the expected outcome. Automation yields a great benefit to *regression testing* that aims for continuously revalidating a program in order to recognize when modifications introduce bugs to existing functionality.

With the evolution of a software, new tests are continuously added to the test suite, inevitably increasing the time required to conduct test runs. This is contrasted by the trend of continuous delivery that leads to shorter release cycles. The time constraint is even more intensified when security fixes need to be released. Thus, regression testing eventually becomes an unfeasible slow-down to the development process [SLS06, p. .69].

*Regression test optimization* (RTO) aims to tackle this challenge. For instance, one way to reduce the time of a test run is to not rerun the complete test suite every time, but only a selection of its test cases that covers the impacts of the modifications. This requires finding a balance between safety, i.e., no faults are missed by the selection, and precision, i.e., no tests are considered that cannot reveal a fault. *Regression test selection* (RTS) techniques attempt to do this automatically. Sometimes, finding an apropriate selection is difficult or skipping tests is not permitted. Then, it is still possible to perform an optimization by running more important tests earlier in order to be able to react to their test results sooner. Determining such an order automatically is the concern of *regression test prioritization* (RTP).

RTP and RTS have been addressed by researchers for a long time; in this regard, the literature review by Singh et al. provides a good overview [Sin12]. Promising RTP techniques have been presented; for example, Rothermel et al. proposed a family of methods that base on test coverage information [RUCH99] and Srivastava et al. described an approach that analyzes the changes of program binaries. Similarly, there are many contributions for RTS; e.g., Rothermel et al. presented a technique that relies on control flow graphs [RH97] and Kim et al. proposed a technique for exploiting the test history [KP02].

Although, manual RTO is widely performed by developers [GNLM14], automatic techniques do not play a great role in practice. Either the techniques do not scale to real projects or it is their integration into existing that causes problems. The later would not be surprising, considering that only few tools exist [Gli, Inf, SSMS]. However, a few big companies, such as Google and Microsoft, recognized a necessity for automated RTO techniques, since manual approaches do not scale to their needs; consequently, they have developed their internal RTO solutions [GIP11, VSMM15, ST02]. Similarly, researchers are affected by the integration problem: it requires a high initial effort before they can

study RTO techniques.

The challenge of the integration of RTO comprises not only the already difficult implementation of optimization algorithms: The techniques need to be integrated into the testing process, i.e., it must be possible to invoke the optimized regression test run like a usual unit test run and to reuse existing test suites. Further, the optimization algorithms require information that must be gathered, e.g., the history of test results. Then, the optimized test suite must be executed and the test results must be reported. Therefore, a platform for an easy implementation of RTO techniques as well as their integration into existing projects, and their evaluation is needed.

### Contributions

The goal of this thesis is to develop a concept and prototype for a framework for RTS and RTP that paves the way for a platform for RTO. Subsequently, the contributions of this thesis are outlined:

1. The current state-of-the-art in RTO is examined. To this end, results of the research on RTO are recapitulated, comprising brief presentations of optimization techniques and empirical studies. Further, the practice of RTO is studied: An overview on the current landscape of RTO tools is provided and the application of RTO in industry is examined.

2. A concept for an RTO framework is presented that enables the easy implementation of RTP and RTS techniques and their integration into existing projects. To this end, the framework's objectives and requirements are specified. Moreover, the architecture of the extension mechanism and the RTO process is elaborated in detail. This concept is evaluated by implementing a software prototype named Lazzer.

3. Moreover, this thesis gives instructions how to extend the system, provides code skeletons for those extensions, and shows example implementations of: two simple, but useful prioritization techniques; two data gathering components; a command line interface and a Maven plug-in; and an adapter for the execution of optimized JUnit 4 test suites.

4. The thesis concludes with recommendations for future work on the prototype and topics for further research.

### Structure

Chapter 2 introduces regression testing as well as RTO, describes the state of the art of RTO in research and practice, and reasons the need for an RTO platform. The first step towards such a platform is the RTO framework presented in this thesis. Its objectives, requirements, and conceptual architecture is established in chapter 3. Chapter 4 describes a prototype of the RTO framework as well as some exemplary

extensions. In chapter 5, the prototype is evaluated. Finally, chapter 6 summarizes the contributions of this thesis and discuses possible topics for future work.

# 2 Regression Test Optimization

Contents


2.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.1.1 Regression Testing . . . . . . . . . . . . . . . . . . . . . . . . . 5
2.1.2 Regression Test Optimization . . . . . . . . . . . . . . . . . . . . 6
2.2 State of Research . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.2.1 Optimization Techniques . . . . . . . . . . . . . . . . . . . . . . 10
2.2.2 Evaluations of Techniques . . . . . . . . . . . . . . . . . . . . . . 12
2.3 Applying Regression Test Optimization . . . . . . . . . . . . . . . . . 13
2.3.1 Tools . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.3.2 In Practice . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15
2.4 Need for a Platform . . . . . . . . . . . . . . . . . . . . . . . . . . 16


In this chapter, the necessary background on regression testing and automatic RTO is provided in section 2.1. Then, in section 2.2, the current state of research on this topic is briefly summarized. Available tools for RTO and the application in industry are described in section 2.3. All this forges the basis for this chapter's conclusion, given in section 2.4: There is a need for a platform that allows for an easy development and evaluation of optimization techniques as well as an easy integration of RTO into existing projects.

## 2.1 Background

In the following, a brief introduction to regression testing as well as to automatic test prioritization and test selection, two ways to optimize regression testing, is given.

### 2.1.1 Regression Testing

The development of a software system usually does not end with its first release. Often, error corrections, improvements, and new features lead to continuous updates. With every new release, there is a risk of introducing faults breaking existing functionality; these bugs are referred to as *regressions*.

Regression testing is based on the assumption that the behavior of a program is extensively verified by an existing test suite, such that at least one test case will fail if the test suite is run for modifications of the program that introduced a regression [BRO13, p. 54]. Thus, revealing regressions is the aim of regression testing [SLS06][p. 68]. In order to detect regressions early, it is desirable to frequently perform regression testing.

Ideally, the tests are conducted after every change of each developer, allowing for an immediate identification of the modification that introduced the fault.

The repetitive nature of regression testing gives reason for automated tests [SLS06, p. 69]. Fortunately, it is already a common practice to develop automated tests alongside software systems to some extend, in order to verify the software system [HSVV12, MB15]. In the case of *test-driven development* (TDD), automated tests are even created before the software system. These tests may not only be used once to check that the corresponding functionality was implemented correctly, but they can be run again upon every modification to ensure that the functionality is not broken. Thus, these tests usually become part of the regression test suite.

In fact, with many evolving software systems, "the importance of automatic regression testing strategies is unequivocally increasing", as Böhme et al. concluded [BRO13, p. 54]. They justify this statement with two studies, suggesting that 9% of all reported bugs are caused by bad modifications [GBHS10] and that 14.8 to 24.4% of the security patches released by Microsoft in between 2000 and 2010 were faulty [YYZ+11].

### 2.1.2 Regression Test Optimization

Though frequent regression testing is desirable, it also introduces challenges. One is the continuous growth of the test suite. It causes the test runs to consume more and more time, eventually becoming an unfeasible slow-down to the development process [SLS06, p. 69]. One way to overcome this issue is to reduce the size of the test suite by removing irrelevant test cases; however, this is a tedious manual task, where the person doing this often does not know which tests are irrelevant. Running the tests in parallel is another option to reduce the test execution time; however, test designs do not always allow for concurrent execution.

Yet, another approach is not to rerun the complete test suite every time, but to run only a subset that covers the impact of the program's modification. In practice, this is something often done manually by developers or testing experts [GNLM14]. A less tedious way to obtain such a subset is by automation, which is referred to as RTS. However, sometimes finding a suitable subset is not possible or skipping parts of the test suite is not permitted; then, reducing the test execution time is next to impossible. Yet, it can be an improvement to get results of more important tests earlier in order to react sooner. This can be achieved by ordering the test cases accordingly prior to each test run. While, again, this can be done manually, automatic methods, referred to as RTP, are more desirable.

#### Regression Test Selection (RTS)

One of the presented approaches to save test execution time is RTS. Before every test run, selection techniques must decide on which tests to keep and which to exclude from the test suite for this test run. In the best case, only fault-revealing tests are selected. A formal definition of this problem is given in definition 2.1: The inputs are $p$ and $p'$, the program before and after a modification, and $T$, the test suite. The problem

| **Test Case Selection Problem** | |
|---|---|
| Let: | $t(P,T)$ be the function that determines the result of the test run for a program $P$ and a test suite $T$. |
| Given: | $P$, the original program,<br>$P'$, a modified version of $P$, and<br>$T$, a test suite. |
| Problem: | Find minimal $T' \subseteq T$ such that $t(P',T') = t(P,T') \Rightarrow t(P',T) = t(P,T)$ |

Definition 2.1: The *test case selection problem.* Definition introduced by Agrawal et al., originally referred to as *incremental regression testing problem* [AHKL93].

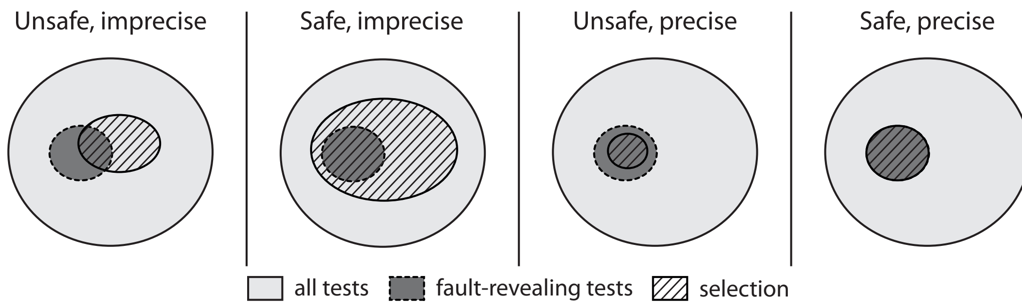



Figure 2.1: From left to right: an unsafe, imprecise selection; a safe, imprecise selection; an unsafe, precise selection; and an safe, precise selection of test cases.

is, to find a minimal subset $T'$ of $T$, that revalidates the behavior of $p$ inherited by $p'$ [AHKL93]. To the best of our knowledge, no proof was provided, but Lin et. al argue that this problem is *NP-complete* [LH09]. Thus, efficient RTS techniques attempt to find heuristic solutions.

Finding a good selection is difficult: For instance, there is the danger of excluding fault-revealing tests. The degree to which all fault-revealing tests are selected is referred to as *inclusiveness*. An inclusiveness of 100% is referred to as a *safe* selection while everything below is called *unsafe*, cf. figure 2.1. Furthermore, it is undesirable to include tests that are not fault-revealing. A technique is *precise*, if it does not select any non-fault-revealing tests; e.g., only the two selections depicted on the right in figure 2.1 are precise. Finally, if techniques need more time to find and run a selection than to just run the original test suite, they lack in *efficiency*, as depicted in figure 2.2.

### Regression Test Prioritization (RTP)

Regression testing is often done by running tests in an order that is fixed and was once, more or less consciously, specified by the developer. The problem with such a fixed order is that on average the execution of fault-revealing tests is spread over the whole test run and, in the worst case, they are executed last. Yet, it would be desirable, if these

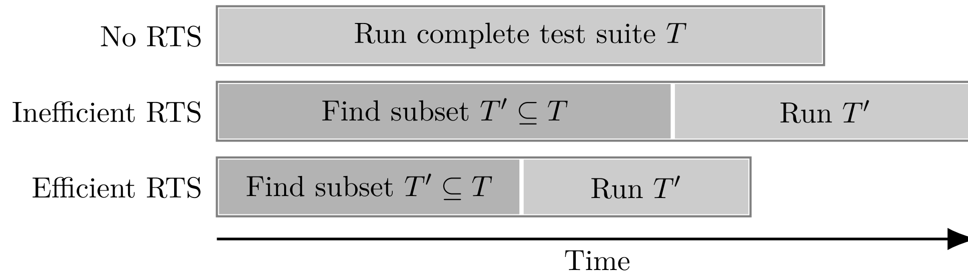



Figure 2.2: Comparison of no, an inefficient, and an efficient application of RTS. The inefficient application requires more time than the execution of the original test suite, while the efficient application saves some time. This figure is based on an illustration presented by Orso et al. [OSH04].

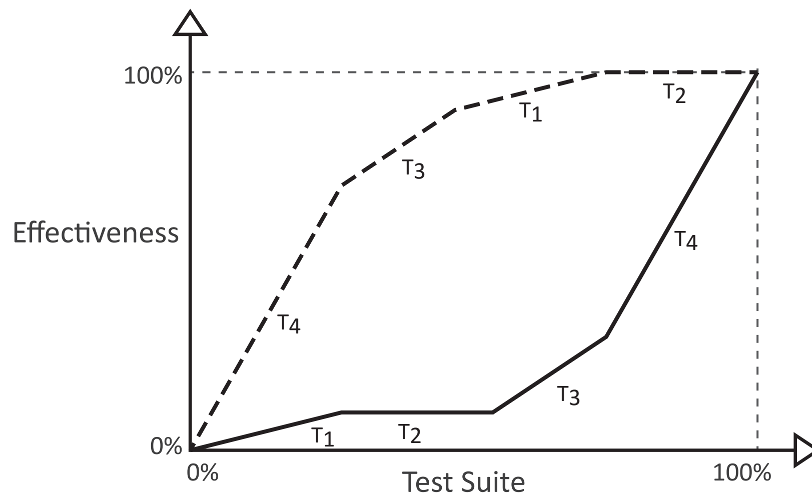



Figure 2.3: Illustration of how a different ordering of tests can optimize the test run.

tests are run first, since this yields two advantages: first, the probability to find faults increases when limited time forces test runs to be stopped prematurely; and, second, an early fault detection allows developers to start investigating the cause sooner.

Figure 2.3 illustrates the fault exposure of two different orderings of the same test suite with four test cases: The solid line represents an undesirable ordering, since the most fault-revealing tests are run last. The dashed line stands for an optimal ordering, since tests are run in descending order of their fault-revealing potential.

| **Test Case Prioritization Problem** | |
|---|---|
| Given: | $T$, a test suite, and |
| | $f$, a function assigning every permutation $\sigma(T)$ a real number value. |
| Problem: | Find $\sigma(T)$ such that $\forall \sigma'(T)$: $f(\sigma(T)) \geq f(\sigma'(T))$ |

Definition 2.2: The *test case prioritization problem* based on the definition by Rothermel et al. [RUH01].

Finding an optimal ordering is described by the *test case prioritization problem.* A formal definition by [RUH01] is given in definition 2.2. There, $T$ is the input test suite, containing all test cases that should be considered, $PT$ represents all possible orderings of the test suite $T$, and $f$ determines the value of each ordering. The goal is to find $T'$, an ordering with the highest value. Although, due to time limitations of this thesis, no proof is provided, this problem is a typical combinatorial problem that is similar to the *job scheduling problem* (with $M = 1$) [AC91], and thus, is *NP-hard*.

A desirable value function would be one that determines the likelihood to reveal a fault. This way, it would be possible to obtain the ordering with the best chances to find faults. Although, so far there is no general method for determining this likelihood, research yielded many prioritizations based on different criteria that seem to be related to it. For example, one criterion is a test's method coverage, a metric measuring whether a method was called during test execution. A prioritization based on this criterion prefers tests with a high method coverage value.

## 2.2 State of Research

There are numerous publications related to RTO. Fortunately, there are papers providing an overview [YH10, BMSS11, Sin12]. They reveal that research goes back to at least 1988: Harrold and Soffa described an "incremental testing system" that "determines which test cases must be rerun, which may be eliminated and whether the test cases are sufficient for the changed program" [HS88], which can nowadays be referred to as a selection technique. Since then, there were many more publications, i.a., describing a variety of RTO methods.

### 2.2.1 Optimization Techniques

The systematic literature review paper from 2012 by Singh et al. [Sin12] provides a comprehensive overview as well as a classification of RTO techniques. The classes mostly base on the input data the techniques rely on; such as test coverage information, modifications to the program, the tests' fault exposing probability, the software's requirements, and the test history. There is also a category about genetic algorithms and category for other approaches, not fitting into this schema. The following sections are based on this overview paper and are structured according to its classification, providing a brief summary of both, the class and related publications.

#### Test Coverage-based Techniques

When a program is tested, parts of it are executed which can be recorded. This allows for a variety of measures; for example, *statement coverage* is the number of executed statements divided by the total number of statements and, in the same way, *branch coverage* is the number of executed branches in relation to all branches [Lig02]. Each of these measures can be used to compute the corresponding accumulated coverage of all tests in the suite, referred to as *test coverage*.

Approaches assigned to this category build upon the hypothesis, that the higher the test coverage is, the higher the probability is to reveal a fault. This allows for selection techniques, e.g., by choosing a set of tests such that the desired test coverage is achieved, as well as prioritizations, e.g., by ordering the test suite such that a high test coverage is achieved as early as possible.

Wong et al. proposed a hybrid approach combining RTO selection and prioritization [WHLA97]. Rothermel et al. proposed and compared four prioritization techniques based on statement and branch coverage [RUCH99]. Elbaum et al. did a study of twelve approaches [EMR00]; they proposed several of these, including two based on function coverage. In 2002 Srivastava et al. presented a prioritization based on the test coverage of a program's binary code [ST02]; it was developed at Microsoft and implemented in a tool named *Echelon*, that was "being integrated into the Microsoft software development process" and "has been tested on large Microsoft product binaries". Do et al. studied the improvement of the fault detection rate of multiple proposed techniques for programs written in Java and using the testing framework JUnit [DRK04]; the techniques include some based on the coverage of blocks, which is an aggregation of multiple statements, and textural differences of two versions of a method. Bryce et al. developed a technique for testing event-driven software, that prioritizes on the basis of interaction coverage [BM07]. A graph-model based approach was presented by Belli et al. that uses graph-models [BEG07]; they emphasize the feature, that "[c]ontrary to other approaches, no prior information is needed about the tests carried out before". Jiang et al. propose a set of methods they refer to as *adaptive random testing* (ART) [JZCT09] and say that one of them "is consistently comparable to some of the best coverage-based prioritization techniques [...] and yet involves much less time cost". A prioritization based on the reactive GRASP (Greedy Randomized Adaptive Search

Procedures) metaheuristic was proposed by Maia et al. [MdCdF+10]. Mei et al. studied prioritization in the context of business web applications [MZCT09]; they proposed a model and some techniques that do not only consider program code, but, for example, artifacts from orchestration languages. In another paper, Mei et al. propose a "strategy for black-box service-oriented testing" including corresponding prioritization techniques [MCTM10]. Bryce et al. introduced a model that unites some regression testing aspects of web and graphical user interface applications and defined some prioritization criteria on the basis of this model [BSM11].

### Modification-based Techniques

Regression testing is done when changes are introduced to a software system. These can be tracked, e.g., by monitoring modifications of the system's source files. On this basis, selection and prioritization techniques can be defined. For example, it seems reasonable, if some source files have changed since the last version, to prioritize or select tests related to these files.

The hybrid technique by Wong et al., which has already been mentioned in the previous section, combines modifications and coverage to select and prioritize tests [WHLA97]. Korel et al. proposed prioritization techniques that order tests based on differences of models of state-based software systems [KTH05, KKT07]. Another model-based approach was presented by Filho et al. [FBH+10]; it combines selection and prioritization of test cases on the basis of modifications to *Unified Modeling Language* (UML) diagrams.

### Fault-based Techniques

Faults revealed in the past by a test, can indicate the test's likelihood to reveal faults in the future. This assumption is the foundation for fault-based techniques. Here, information on previous test runs need to be collected and analyzed. An example for such a technique is a prioritization that runs recently failed tests first.

Rothermel et al. proposed a metric referred to as *weighted average of the percentage of faults detected* (APFD) and introduced techniques based on it [RUCH99]. Another series of methods was presented by Elbaum et al. [EMR00, EMR02].

### Requirement-based Techniques

Many software systems are developed according to a requirements specification. Requirements can differ in their value to the stakeholders; e.g., the realization of some requirements can be more important than others. Although tests are often derived from requirements, most RTO techniques do not consider the requirements' priority when selecting or prioritizing corresponding tests. Requirement-based techniques intend to incorporate this information in their optimization.

Srikanth et al. presented a prioritization approach that "prioritizes system test cases based upon four factors: requirements volatility, customer priority, implementation complexity, and fault proneness" [SWO05, SW05]. Another prioritization technique

relying on some requirements related factors was proposed by Krishnamoorthi et al. [KS08, KS09]. Hou et al. researched a prioritization technique for web services considering request quotas, which, for example, can be a maximum number of requests a client can send to a web service in a certain time [HZXS08].

#### History-based Techniques

The frequent repetition of regression testing allows for the collection of information about a test's behavior over time; e.g, how many times tests succeeded and failed, the causes of failures, execution times, and more factors. This information is exploited by history-based techniques. A simple prioritization can be, e.g., to order the tests by the execution time of previous runs.

Kim et al. proposed a general technique for exploiting the test history. The technique allows to define a criterion that is used to prioritize and select tests, e.g., the preference of tests not run recently. Their technique enables the test run to meet certain resource constraints, like, for example, a limited time for testing [KP02]. A contribution by Park et al. presents an approach that uses historical information in order to estimate "the current cost and fault severity for [a] cost-cognizant test case prioritization" [PRB08]. Another technique that considers time and resource constraints was described by Fazlalizadeh et al. [FKAP09].

#### Other Techniques

There are many more techniques that do not fit into the formerly mentioned categories: Some utilize the concept of genetic algorithms, which imitate the natural selection mechanism that can be observed in nature; this can be used to realize prioritization and selection of tests [WSKR06, CRK10, SSS11, JYC11, MSGB14]. Others combine concepts from two or more of the categories; e.g., the coverage- and modification-based technique by Wong et al. mentioned earlier [WHLA97]. Still others are based on entirely different ideas, like, e.g., on inter-component behavior, graphs, or search algorithms. Due to the extend of the list of these techniques, no summary is given here; however, a comprehensive listing can be found in the overview paper by Singh et al. [Sin12].

### 2.2.2 Evaluations of Techniques

Many of the formerly mentioned publications provided a brief analytical or empirical evaluation of the presented techniques. Usually, the extent of these evaluations was limited to a comparison to random approaches, i.e., random prioritization or random selection, rarely to a few related techniques. More elaborated empirical studies were presented by Rothermel et al. [RUCH99] and Do et al. [DMTR08].

Besides these two studies, there is also at least one focusing on the comparison of different techniques: Graves et al. presented a preliminary empirical study comparing five techniques [GHK+01]. They observed differences and trade-offs among the reviewed techniques and discovered that the results were not only sensitive to the techniques but

| Name | Available | Open-source | Technique |
|---|---|---|---|
| Infinitest | yes [Inf] | yes | selection |
| Test Load Balancer | yes [SSMS] | yes | prioritization |
| Ekstazi | yes [Gli] | planned | selection |
| DejaVOO † | on request [Ors14] | no | selection |
| ATAC † | no, | - | selection |
| SPYDER † | no | - | selection |
| Aristotle Analysis System † | no | - | not applicable |

† discontinued

Table 2.1: List of tools referred to by some of the literature about RTO. No claim is made, that this list covers all tools. The availability of a tool refers to whether it can be obtained. A tool is marked to be open-source, if it is possible to access the source repository. The kind of RTO technique implemented by a tool is given where applicable.

also “to the programs, the characteristics of the changes, and the composition of the test suites”; one of their conclusions for future experiments is, that “[i]t will also be important to examine a broader range of subject programs”.

Altogether, these evaluations and studies provide reasons to assume that the application of RTO has the potential to yield an improvement to regression testing. However, the small number of comparative studies, their partly preliminary nature, and the question they pose call for more research. In order to give well-founded recommendations for applying RTO, the techniques’ trade-offs and the circumstances when to prefer one technique over others need to be known.

## 2.3 Applying Regression Test Optimization

In this section, the current landscape of tools for RTO and the application of optimization in practice are discussed.

### 2.3.1 Tools

A list of tools that either realize an automatic RTO technique or were used in the context of such an implementation is given in Table 2.1. Most of the tools and implementations mentioned in literature on RTO are no longer available or were not made public. Agrawal et al. described the tools SPYDER and ATAC [AHKL93]; the former is no longer available [Spa03]; the latter became part of a commercial software suite [Cle], but is, according to the company reselling it, discontinued. Furthermore, the Aristotle Analysis System was used by Rothermel et al. [RH97] as well as Elbaum et al. [EMR02], but is,

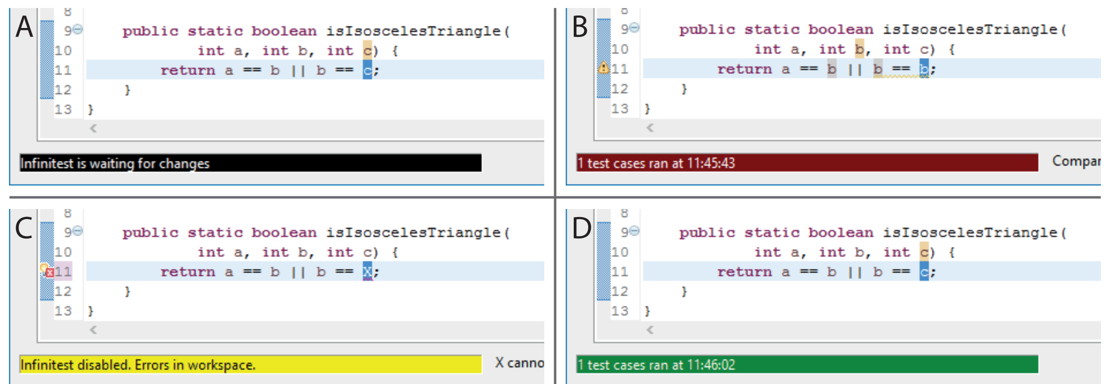



Figure 2.4: Infinitest can be integrated into Eclipse's status bar showing the latest testing status. Initially, Infinitest waits for changes, indicated by a black bar (A); after a fault was introduced causing a test to fail, the bar turns red (B); in case of syntax errors Infinitest cannot be run, indicated by a yellow bar (C); once all issues are fixed, the bar appears in green (D).

according to a former member of the corresponding research group, no longer available. DejaVOO was described by Orso et al. and implemented a RTS technique [OSH04]; the software is discontinued and available on request only [Ors14].

Fortunately, there are a few optimization tools available. These are: Infinitest, Ekstazi, and *Test Load Balancer* (TLB). They all have in common that they implement a specific approach to RTO. Despite that, they address different usage scenarios and only Ekstazi aims to implement a selection technique as its main concern. Subsequently, the three tools are briefly described.

Continuous testing while developing is a concept implemented by the open-source tool Infinitest [Inf]. It can be integrated into the *integrated development environments* (IDE) Eclipse and IntelliJ IDEA and selectively runs tests each time changes to the source code are made. Its test selection is a modification-based method that determines which tests are affected by the changed source files. The user is informed about Infinitest's status and results in the IDE's status bar: as depicted in figure 2.4, differently colored bars tell the user whether the performed modification caused tests to fail or succeed.

The integration of RTS into the build process is possible with Ekstazi, a tool presented recently in publications by Gligoric et al. [GNLM14, GEM15]. Similar to Infinitest it determines the tests affected by a modification and only reruns these. The tool can be integrated with Maven and Ant. It is being used by some projects, such as Apache Camel and Apache Commons Math.

TLB is a tool that primarily addresses automatic partitioning of test suites such that they can be run in parallel [SSMS]. In addition to that, TLB implements a simple test prioritization by reordering the tests such that failed ones are run first.

### 2.3.2 In Practice

An empirical study by Gligoric et al. suggests, that manual RTO is practiced by "almost all developers" [GNLM14]. Automatic optimization, however, is rarely done or even known, as research for this thesis revealed. Only big companies, such as Siemens, Microsoft, or Google, seem to address this topic out of the necessity that manual optimization is impractical for huge software systems.

A publication by Hutchins et al. from Siemens Corporate Research in 1994 shows that Siemens, a major technology company, did research on automatic RTO [HFGO94]. Yet, there is no evidence that their research resulted in a practical application. In fact, a case study on a product called Siemens PLM by the SQS Software Quality Systems AG, a consulting company for software testing and quality management, from 2011 indicates the opposite [SQS11]. According to this study, Siemens' "testing [still] relie[d] on selective testing by subject matter experts", although they reported that "the number of subject matter experts is never sufficient, and their availability is not guaranteed".

Efforts by Microsoft resulted in a tool, which Srivastava et al. referred to as *Echelon* in a publication from 2002 [ST02]. It was developed at Microsoft Research with the intention to optimize their regression testing. They concluded that it is "possible to effectively prioritize tests in large[-]scale software development environments" and they said that "Echelon [was] being integrated into the Microsoft development process". In 2012, Jean Hartmann, an engineer at Microsoft, presented a chronology of regression testing at Microsoft; he reports the successful application of Echelon, which was then called Scout [Har12]. In 2015, Herzig et al. presented a test selection strategy that was verified in simulations based on historical data and metrics collected about several of Microsoft's products [HGCM15].

Engineers at Google also dealt with RTO. A post in the Google Engineering Tools blog by Gupta et al. in 2011 described an automatic selection technique based on dependencies [GIP11]; they "maintain[ed] an in-memory graph of coarse-grained dependencies between various tests and build rules across the entire codebase"; this allowed them to select only the tests affected by a modification. Vakilian et al. from Google presented an approach that intends to avoid unnecessary builds and tests. They identified targets "with files not needed by some of its dependents" as a problem and referred to them as *underutilized* [VSMM15]. They developed and implemented an algorithm that automatically decomposes underutilized targets into smaller ones and showed its efficient and effective application in Google's build environment.

Other companies are struggling with long test execution times as well: For example, in the panel discussion of the RELENG 2014 workshop [ABB+14], which employees from companies like, e.g., Facebook, Google, and Netflix attended, it was asked for automated test selection. In addition to that, industrial partners of the Software Construction Group from the insurance and energy sector stated, that they are also interested in the optimization of their test execution time.

## 2.4 Need for a Platform

The research on the state of RTO revealed a long list of promising contributions to this topic, but also a lack of comparative studies of the presented techniques (cf. section 2.2). Furthermore, it showed that there are only very few tools and implementations available, and that, despite the necessity for optimization, tedious manual optimization is practiced more than automatic optimization (see section 2.3).

The application of RTO is difficult: Not only is it very time-consuming to implement and evaluate the techniques themselves, but also a lot of effort is necessary for building a complex infrastructure around them. This includes a system that runs the selected and/or prioritized tests; systems that collect, store, and provide the information required by the techniques; and an integration of the techniques into existing projects.

This gives reason for a gap that remains to be filled: There is a need for a platform that allows for an easy implementation of automatic RTO techniques and a quick integration of these into existing projects. Furthermore, on the basis of this platform, techniques can be evaluated and comparative studies of different approaches can be conducted. This thesis presents a first step towards such a platform by introducing an RTO framework.

# 3 Concept

Contents

3.1 Background . . . 17
3.1.1 Frameworks . . . 17
3.1.2 XUnit . . . 20
3.2 Objectives and Requirements . . . 22
3.2.1 Objectives . . . 22
3.2.2 Requirements . . . 23
3.3 Architecture of the Framework . . . 28
3.3.1 Coarse-grained Architecture . . . 28
3.3.2 Details of the Architecture . . . 29

This chapter tackles the jump-start problems when developing RTO techniques, i.e., their integration into existing projects, gathering information required by the techniques, and executing the optimized test suite. First, an introduction to frameworks and xUnit is given in section 3.1. Then, the objectives and requirements of a framework for regression test optimization are elaborated in section 3.2. Finally, section 3.3 presents a conceptual architecture for such a framework.

## 3.1 Background

This section, first, provides an understanding of frameworks and how they can be used to develop applications and, second, introduces the family of xUnit test frameworks.

### 3.1.1 Frameworks

Separation of concerns is an important design principle in software engineering [Dij82]. It describes that unrelated concepts should be located in different sections of a software implementation. One approach to realize separation of concerns is to divide a software project into independent parts that encapsulate a specific functionality or concern. These parts are referred to as software components.

A component is made up of its *interface*, which defines what functionality this component provides, and an *implementation*, the realization of the interface. The advantage of this separation is the exchangeability of the implementation without affecting users of the component as long as the interface does not change.

Building a project with software components results in a highly reusable code; since each component is independent and encapsulates exactly one concern, it can easily be

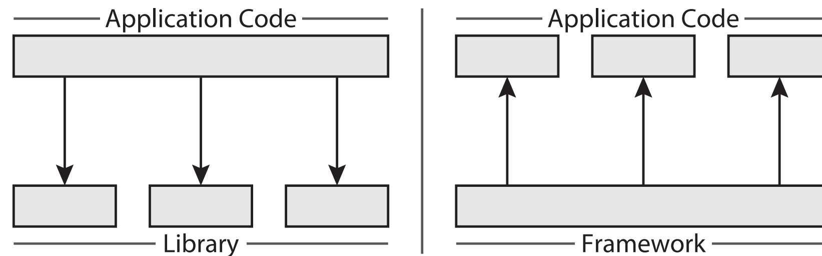



Figure 3.1: The left-hand side depicts the interaction of an application with a library; the library's functions are called from the application code whenever required. The opposite is the case when using a framework, as depicted on the right-hand side: the framework is in control and makes calls to the application code, giving the application the possibility to customize the behavior.

used in other software projects. While this requires a well engineered component to work flawlessly, reuse has some advantages: One is a reduced development time for projects, another is, reusing an often used component minimizes the risks of introducing software flaws with an own implementation that is less elaborated. In addition to that, it is easier to introduce fixes and improvements to a single component, compared to dealing with numerous individual solutions.

In general, there are two approaches of realizing reuse of software components: *libraries* and *frameworks*. Libraries are made up of software components with the intention, to offer functionality that can be used in an application; that way, the application is in control of using the functionality whenever it needs to. In contrast, frameworks define the application's architecture in a varying level of detail and leave blanks to be filled in with application specific adaptions. Thereby, frameworks embody an *inversion of control* [FSJ99, p. 5]. The converse behavior of both approaches is depicted in figure 3.1.

Consequently, a framework can be defined as "a reusable, semi-complete application, that can be specialized to produce custom applications" in the framework's domain [FSJ99, p. 4]. In the context of object-oriented programming, a framework can be defined as "a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes." [JF88, p. 2].

### Hot spots

Application-specific adaptions to the framework's behavior, can only be made at predefined points, referred to as *hot spots* [Pre97] or *variation points* [FPR02]. Correspondingly, *frozen spots* are the fixed parts of a framework that cannot be influenced by the developer. Hot spots can be realized in different ways enabling a classification: *white-box*, *black-box*, and *gray-box* frameworks [JF88, Pre97, FSJ99]. From here on, only object-oriented frameworks are considered.

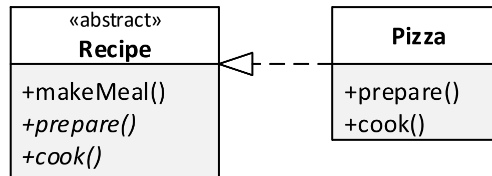



Figure 3.2: UML class diagram depicting an example for the *template method pattern* [GHJV95]: The concern of the abstract class `Recipe` is to prepare a meal. The general procedure of doing so is implemented in the *template method* `makeMeal`, that calls the abstract *hook methods* `prepare` and `cook`. The class `Pizza` represents the preparation of a concrete meal, i.e., `prepare` yields the pizza and `cook` corresponds to putting the pizza into the oven.

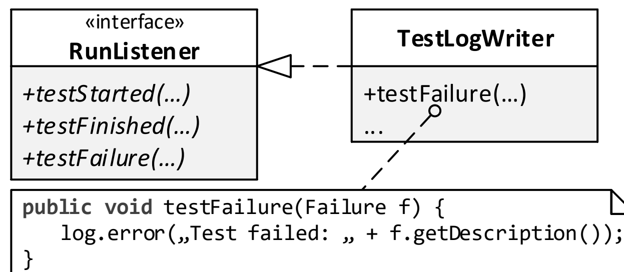



Figure 3.3: UML class diagram depicting an example for framework extension by interface implementation: The interface `RunListener` provided by the JUnit testing framework allows for observing a test run. Here, an implementation logs test failures. Note that the developer needs detailed knowledge about the interface and the parameter `Failure f`.

**White-box Frameworks** Many frameworks build on *generalization*, in order to allow adaptions by the developer. To this end, several approaches exist: A framework can specify an interface and require the developer to implement its methods. Similarly, abstract classes can be used, allowing the framework to specify some of the class' implementation. The later is utilized in a specific way by the *template method pattern* [GHJV95] that enables the implementation of an algorithm such that some of its steps are kept variable (cf. figure 3.2). To this extent, the abstract class declares several methods, one being the *template method* and the others being abstract *hook methods*; the template method realizes the algorithm by calling for the abstract methods, where a variable step is needed. Here, developers needs only to implement the abstract methods in a subclass.

Adaption through generalization requires the interior of the framework to lay open, e.g., by providing a documentation of the framework's *application program interface* (API). Developers needs to incorporate this knowledge before they are able to implement applications using the framework. This motivates the name *white-box frameworks*.

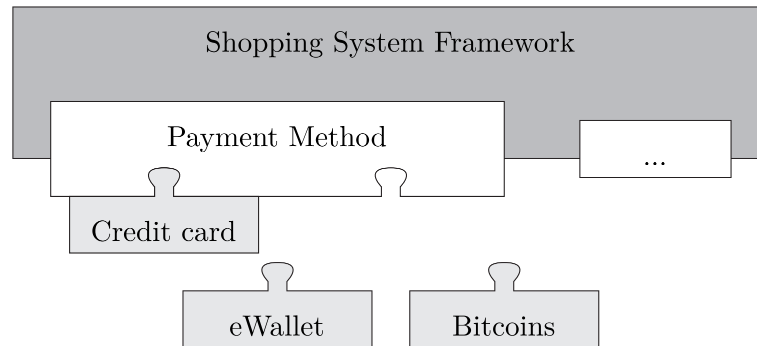



Figure 3.4: Example for framework configuration: The presented shopping framework allows for different *payment methods*: *credit card*, *eWallet*, and *Bitcoins*. Instead of providing an implementation, the developer specifies which of the framework's predefined payment methods will be used.

An example for a white-box extension mechanism can be found in JUnit, a framework that allows for testing Java programs. When calling JUnit, the developer may specify a `RunListener` that allows for notification of events that occur during the test run. To this extent, the methods, specified by the interface, need to be implemented. Figure 3.3 depicts an UML diagram of the `RunListener` interface and an implementation that writes JUnit events to a log.

**Black-box Frameworks** Instead of requiring the developer to provide an implementation for a hot spot, frameworks can provide predefined implementations from which the developer can choose. Here, the implementation is limited to composing predefined components. With an appropriate user interface, even end users could configure such a framework [Pre97, p. 37]; for example, someone who operates an online shop could select a payment method from a set of predefined methods through the shops administration interface (cf. figure 3.4).

**Gray-box Frameworks** In practice, most frameworks are neither pure white-box, nor pure black-box frameworks, but range between both, reasoning the name *gray-box frameworks*. Initially, frameworks tend to be more white-box than black-box frameworks, but in the course of their continued development, frameworks often shift towards black-box frameworks [Pre97, p. 21]. Gray-box frameworks share the strengths of both sides: They have the flexibility and extendibility of white-box frameworks and "the ability to hide unnecessary information" from black-box frameworks [FSJ99, p. 10].

### 3.1.2 XUnit

In 1989, Beck described patterns for unit testing in the object-oriented programming language Smalltalk and outlined a corresponding test framework [Bec89]. His ideas were the basis for test frameworks for other object-oriented programming languages; i.a.,

JUnit for the Java programming language. The family of frameworks that share this concept is referred to as *xUnit* [Fow06, Mes07].

### Test Definition

Tests are implemented as methods in *test case* classes, often also referred to as *test classes*. *Test methods* realize a process with four phases: the *setup*, the *exercise*, the *verification*, and the *teardown* phase [Mes07, p. 77].

The setup phase is responsible for the construction of the *test fixture*, i.e., everything needed to run the *system under test* (SUT). Meaning, at least the tested classes need to be instantiated and, for example, setting up a database that is required by the SUT. In the second step, the SUT is exercised; here, the tests interact with the system. Then, in the third phase, it is verified that the effects of the interaction match the expected outcome. To this end, assertion methods are used; they take a boolean expression as input that evaluates to true in case of a successful test; otherwise, the test fails and the test framework is informed about it, e.g., by throwing an exception. The purpose of the final phase, the teardown phase, is to leave the test infrastructure in a clean state after the test ended; for example, this may require the proper closing of a previously opened database connection.

Setup and teardown can both be conducted inside the test methods. Yet, xUnit frameworks usually allow to implement special methods for a *local* setup and teardown that is shared among all methods of a test class. To this end, these methods are automatically called by the framework before and after each test case, respectively. Further, *global* setup and teardown methods are run before and after each test class; i.e., the setup is executed before the first and the teardown after the last test case of the class.

### Test Execution

The basic architecture of xUnit test frameworks is depicted in the UML class diagram in figure 3.5. It shows the definition of tests via test case classes, their composition to *test suites*, and the *test runner* that triggers the test run.

Before the test run can be started, all the tests need to be gathered to a test suite. This is either done by *test enummeration*, i.e., a manual specification of the tests that should be considered, or by letting an automatic *test discovery* find the relevant tests [Mes07, p. 78].

Internally, the *composite pattern* is applied to represent test suites [Mes07, p. 81]. To this end, there is a *test interface* (the component) that specifies a *run method*. This method is implemented by the *test suite object* class (the composite) and the *test case object* class (the leaf). The first represents a test suite and implements the run method by invoking the run methods of all its children that themselves implement the test interface. The later represents single test methods; it implements the run method by executing the test, i.e., calling the setup-, the test-, and the teardown-methods of the respective test case class. The benefit of applying the composite pattern is that running a single test

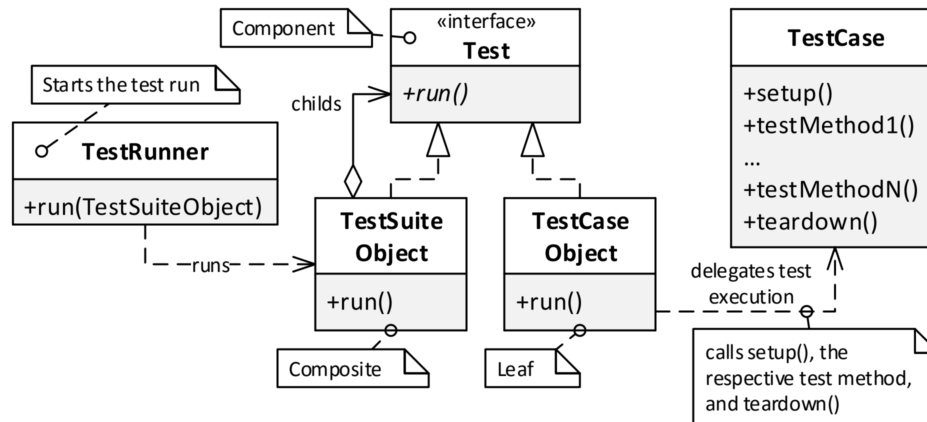



Figure 3.5: UML class diagram of xUnit's basic architecture. The definition of tests is done in the `TestCase` class. The test framework creates a `TestCaseObject` for each test method, which can then be composed to test suites. The test execution is started by the `TestRunner` that calls the test suite's `run` method.

method is handled similarly to running every test of a test case class or even all tests of the application.

The execution of a test suite is triggered by the test framework user through a test runner. Various kinds of test runners exist for different usage scenarios; for example, a test runner embedded into an IDE can provide the developers with a graphical user interface for starting the test run and analyzing its results, while a command-line test runner can allow for integrating the test execution into a build process.

## 3.2 Objectives and Requirements

A concept of an RTO framework is presented in this section. To this end, the objectives and the thereof derived requirements are specified first. Then, the big picture of the architecture is outlined before getting into the details of individual parts.

### 3.2.1 Objectives

For the successful application of RTO, some difficulties need to be overcome, as section 2.4 described. A brief recapitulation: it is expensive to implement and evaluate RTO techniques, since a complex infrastructure has to be setup first. Further, it has to allow for running the selected and/or prioritized tests and must collect, store, and provide the information required by the techniques. Additional effort is required to integrate the infrastructure into existing projects.

Tackling these difficulties is the overall objective of the presented RTO framework. With this in mind, first, the framework must support an easy implementation of RTO

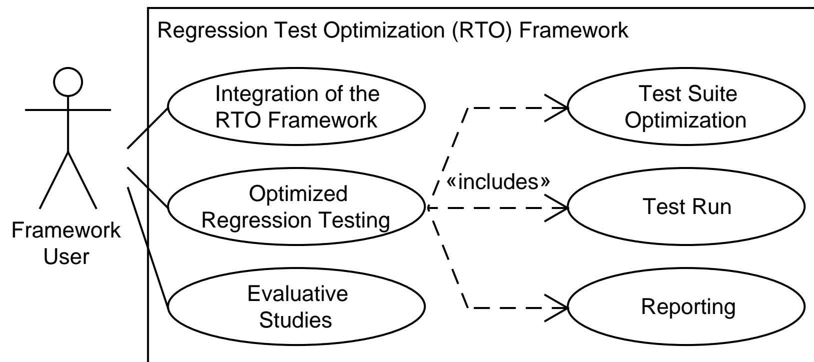



Figure 3.6: UML use case diagram depicting the usage scenarios of a framework user.

techniques; i.e., everything not directly related to the technique itself has to be handled by the framework; for example, finding out the list of tests used as input for the techniques. Second, the integration of the framework into existing projects must be as simple as possible. Finally, the framework must support conducting evaluative studies, e.g., by comparing the performance of different techniques; this requires the ability to run different RTO techniques under the same conditions.

### 3.2.2 Requirements

The interests of two user groups have to be considered by the RTO framework: On the one hand, there are users who want to integrate the RTO techniques into their projects. On the other hand, there are developers aiming for extending the framework, e.g., by implementing new techniques or applying existing ones to new test frameworks. The following specification of the requirements is split up accordingly.

#### User's perspective

Multiple use cases have been identified for the framework user: the *integration* of the RTO framework, its usage for *optimized regression testing*, and the realization of *evaluative studies* of RTO techniques. These use cases and the therefrom derived requirements are described in this section. An UML use case diagram depicted in figure 3.6 supports the description and the requirements are enumerated for later reference in table 3.1.

The RTO framework must provide an API for the integration of the framework (cf. F1.10). In order to keep this simple, the user may not be required to make more than three API calls for a running example (cf. N1.10). Further, the integration must be limited to modifications of the project or build configuration; the test suite may not be changed in any way (cf. N1.20). Note that the former requirements only apply to an integration done with the framework's API. There may be extensions for the framework

that allow for an integration that does not require any programming, e.g., a ready-to-use plug-in for an IDE. These are not part of the framework and, thus, do not necessarily comply with the previous requirements.

The RTO framework must allow for optimized regression testing, which includes the optimization of the test suite, the test run, and the reporting of the test results. First, in order to fulfill these tasks, the framework needs some information: This includes the specification of the system under test, the test suite, and the RTO techniques that should be applied (cf. F1.20). Then, the framework must use the specified RTO techniques to compute the *optimized test suite* from the given test suite (cf. F1.30). The tests from this optimized test suite have to be run by the framework (cf. F1.40). Finally, the framework must inform the user about the results of the test run (cf. F1.50).

The purpose of regression testing would be undermined if the optimized test run takes more time than the plain execution of all tests. Thus, the framework is required to be efficient (cf. N1.30), i.e., the framework itself may, in average, not need more than one percent of the time needed by an unoptimized test run.

| No. | Description |
|---|---|
| F1.10 | The RTO framework must provide an API that enables the user to integrate the RTO framework in his software build and test environment. |
| F1.20 | The RTO framework must allow the user to set required parameters for the invocation of the RTO framework by the user. These parameters are:<br>a the system under test,<br>b the test suite, containing the tests for the given software system, and<br>c an ordered list of one or more RTO techniques, implementing RTO techniques, along with their configuration. |
| F1.30 | The RTO framework must apply the RTO technique, specified by the user, in order to optimize the test suite and to deliver the *optimized test suite* as result. |
| F1.40 | After the RTO framework applied all RTO techniques, it must run the tests of the optimized test suite. |
| F1.50 | After the RTO framework run the tests of the optimized test suite, it must report the results to the user. To this extend,<br>a it must state whether the test run was successful, which is the case if, and only if, all tests of the optimized test suite run successfully, or<br>b it must list all failed tests. |
| N1.10 | The integration of the RTO framework, as described in requirement F1.10, may not require the user to make more than three API calls for a running example. This must not include additional optional configuration or settings of extensions. |

Continued

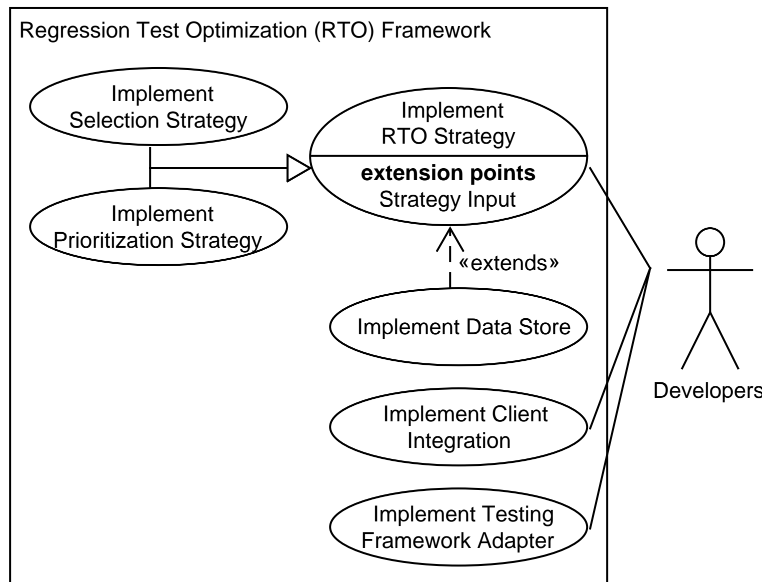



Figure 3.7: UML use case diagram depicting the usage scenarios of a developer aiming for extending the framework.

| No. | Description |
|---|---|
| N1.20 | The RTO framework must enable the use of existing test suites without requiring any framework-specific modifications to the test suites. |
| N1.30 | The RTO framework must be efficient, i.e., the average time needed by the framework itself may not exceed one percent of the time needed for the plain run of the unoptimized test suite. Note that the optimization and the run of the optimized test suite are not considered to be part of the framework itself. |

Table 3.1: List of functional (F) and non-functional (N) requirements for the framework from the user's perspective.

### Developer's perspective

The use cases associated with the developer are all related to different framework extensions: the collection of information required by the RTO techniques, the implementation of new RTO techniques, *client adapters* allowing users to integrate the framework in new environments, and *testing framework adapters*, enabling the handling of test suites realized with these testing frameworks. Again, the following use case descriptions are supported by an UML diagram depicted in figure 3.7 and the requirements are listed in table 3.2.

One of the user's requirements is the integration of the RTO framework (cf. F1.10). To this end, the user can call the framework programmatically via the provided API. However, for common integration targets, such as IDEs or build systems, it is not reasonable that each user develops his own integration. For this reason, the framework must allow a developer to implement an adapter for new integration targets. To this extent, the API must allow the configuration and the invocation of the framework, and it must provide access to the test run's result (cf. F2.10).

A variety of information is needed for the implementation of different RTO techniques, as discussed in section 2.2. It is infeasible for the framework to provide every technique with the required input right out of the box. A simple solution is, to assign the responsibility to collect the required information to the RTO technique itself. However, this has two drawbacks: First, collecting the needed information and performing the optimization by the RTO technique violates the separation of concerns principle. Second, multiple RTO techniques having some input data in common cannot easily and efficiently share the information collection mechanism. Thus, a developer must be enabled to extend the framework by components, referred to as *data stores*, that allow the framework to collect and store possibly any kind of information as well as make them available to the RTO techniques (cf. F2.20).

Reducing the complexity of realizing RTO techniques by providing some common functionality is one of the frameworks objectives. Therefor, the framework must provide an API that enables the developer to implement RTO techniques (cf. F2.30). To this extent, the framework must provide RTO techniques with a list of the regression tests as well as access to the data stores that provide other input required by the optimization algorithm (cf. F2.30a). For performing the optimization, the developer of the RTO technique must be able to prioritize and select tests from the given test suite, in order to obtain the *optimized test suite* (cf. F2.30b). This test suite must then be used for the test run (cf. F2.30c).

Many existing projects already have test suites that were implemented using different testing frameworks, like, e.g., JUnit for Java projects. Since one of the framework's objectives is the easy and inexpensive integration into such projects, requiring modifications to their test suites does not seem reasonable. Thus, the RTO framework must be able to adapt to these testing frameworks and corresponding test suites. Thus, the developer must be able to extend the RTO framework, such that it can deal with possibly any testing framework (cf. F2.40).

| No. | Description |
|---|---|
| F2.10 | The RTO framework must provide an API that allows<br>a the RTO framework's configuration (cf. F2.11),<br>b to invoke the regression test run, and<br>c to obtain the results of that test run. |
| Continued | |

| No. | Description |
|---|---|
| F2.11 | The RTO framework must allow a configuration of<br>a the RTO techniques and their order,<br>b the data stores, i.e., the input for the RTO techniques,<br>c the test framework, and<br>d the tests to be run. |
| F2.20 | The RTO framework must provide an API that allows developers to implement their own data stores, i.e., components that can collect, process, and store information of possibly any kind as well as make this information accessible by RTO techniques. To this end, the framework must enable the data store to perform its collection, processing, and storage<br>a before the RTO is performed, providing it with a list of the regression tests, and<br>b after the framework completed the test run, providing it with the test results. |
| F2.30 | The framework must provide an API that allows developers to realize an RTO technique as. To this end,<br>a the framework must provide RTO techniques with their required *input*, being<br>(1) a list of the regression tests, i.e., a list of names that allow for the test's identification, and<br>(2) access to the data stores the RTO technique depends on;<br>b the framework must enable RTO techniques to construct an *optimized test suite* as *output* by defining<br>(1) an order on the tests and/or<br>(2) a selection of the tests;<br>c the framework must use the optimized test suite for the test run. |
| F2.40 | The framework must allow the developer to implement extensions integrating testing frameworks into RTO framework, such that regression test suites developed with these testing frameworks can be considered by the RTO framework. |

Table 3.2: List of functional requirements for the framework from the developer's perspective, whose task it is to extend the framework.

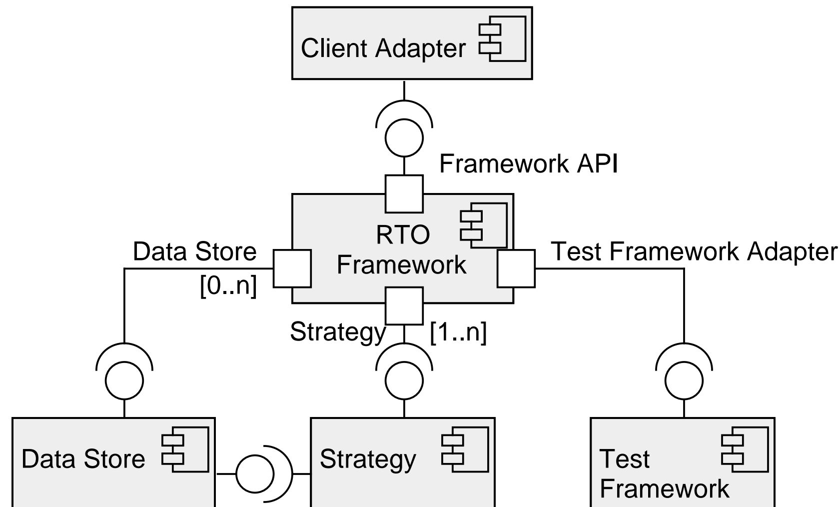



Figure 3.8: UML component diagram of the coarse-grained architecture of the RTO framework's. There are four components around the framework's core: The client adapter uses the API provided by the framework in order to integrate the framework. For information collection and provision, the data store interface is realized. Components implementing an RTO technique realize the strategy interface. The test framework is used by implementing the test framework adapter interface. Note that multiple data stores and strategies can be plugged into the framework.

## 3.3 Architecture of the Framework

In this section, a conceptual architecture of an RTO framework is presented. First, an overview is provided that presents the framework within its environment. Then, the framework's architecture is discussed in detail.

### 3.3.1 Coarse-grained Architecture

The architecture is established around a component representing the framework's core, as depicted in the UML component diagram in figure 3.8. The core component's concern is the abstraction of the RTO process from particular data collection and provision mechanisms, RTO techniques, and test frameworks. To realize this, the process itself is kept general and concrete steps are delegated via three required interfaces to external components; each of these interfaces representing one of the framework's *hot spots*. Furthermore, the framework provides an interface that allows its usage.

The *data store* interface can be used to realize custom data collection and provision mechanisms that may be plugged into the RTO process, as it is required by F2.20. Note

that the framework may use multiple data stores simultaneously or none at all. The later can make sense if no information is gathered by the RTO technique; for example, this is the case with an alphabetic test case prioritization.

RTO techniques are realized and integrated with the framework using the *strategy* interface. This interface corresponds to requirement F2.30. It takes an ordered list of techniques, which allows for a sequential combination of multiple techniques; for example a selection technique can be applied first and then the selected tests are ordered by using a prioritization method.

The only provided interface is the framework's API that may be used by anyone to programmatically use the framework. Since these components build a bridge to a client who is using the framework, they are named *client adapters*. This API aims for realizing requirements F2.10, F2.11, and F2.40.

Finally, the test framework that was used to implement the test suite needs to be integrated, as demanded in requirement F2.11; accordingly, the interface is named the *test framework adapter* interface. Its purpose is twofold: One the one hand, it should help the framework to find the all tests in the given test suite; this is necessary since tests are implemented differently depending on the used test framework. On the other hand, the framework delegates the details of the test run to the test framework. Note that the test framework adapter unites two concerns which appears to be in conflict with the separations of concerns principle. The reason not to have two separate interfaces for both concerns is the fact that this interface's intention is to encapsulate the test framework and not individual tasks that are delegated to it.

The RTO framework is meant to be a gray-box framework. Developers should always be allowed to implement custom hot spot extensions, but on a long term perspective the framework aims for providing a set of predefined components that can cover the most common usage scenarios.

### 3.3.2 Details of the Architecture

The conceptual structure of the RTO framework's core is presented in this section. The architecture is illustrated in the UML component diagram in figure 3.9. It is divided into three parts: The *RTO pipeline* and its stages that represent the RTO process, the *dependency loader* that helps providing strategies with the required data stores, and the *API* that makes the framework's functionality available to users. Subsequently, the concern of each part is described in more detail.

#### RTO Pipeline

Essentially, the RTO process is a plain sequence of tasks. Each task takes some input and produces an output that serves as input of the following task. This principle has been around in Unix operating systems for a long time. There, a pipe operator can be used to interconnect independent programs to solve a complex task. Accordingly, the component realizing the RTO process is named *RTO pipeline*.

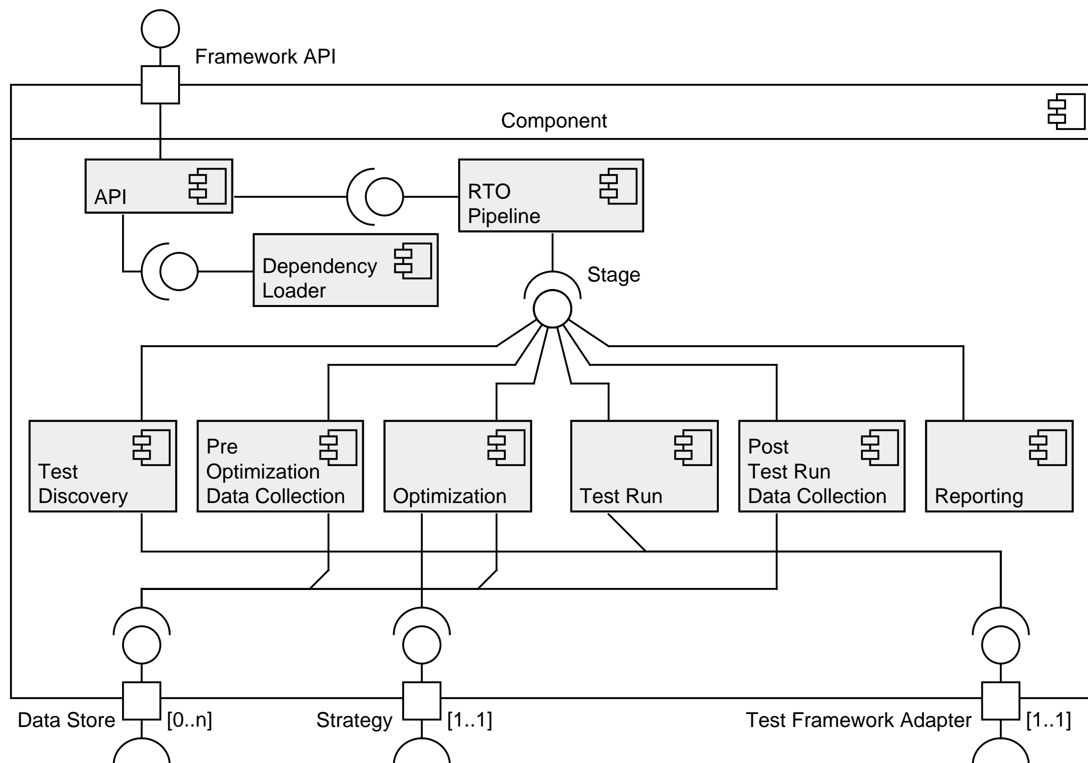



Figure 3.9: An UML component diagram of the framework's architecture. The RTO process is represented by the RTO pipeline and its six stages: the *test discovery*, the *pre-optimization data collection*, the *optimization*, the *test execution*, the *post-test-run data collection*, and the *reporting* stage. The Dependency Loader helps providing strategies with an access to required data stores. The API provides an interface to framework users.

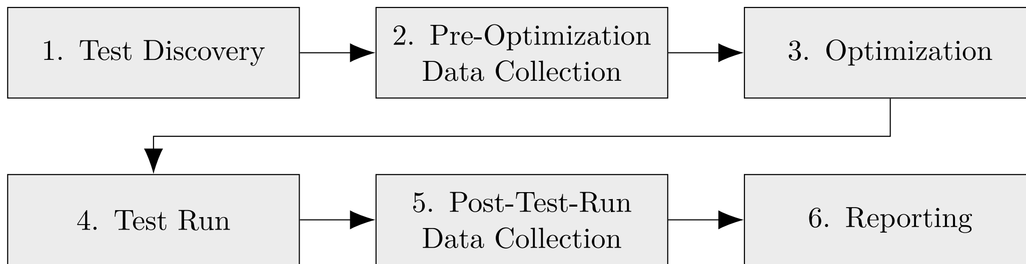



Figure 3.10: The sequence of tasks that has to be performed by the RTO framework.

The six stages of the RTO pipeline are depicted in figure 3.10: the *test discovery*, the *pre-optimization data collection*, the *optimization*, the *test execution*, the *post-test-run data collection*, and the *reporting*. In the following each stage is briefly presented.

In the test discovery stage, the RTO framework attempts to find all tests in the specified test suite. The location of the test suite is given by a path to a file or directory. The framework then needs to search this path for test classes and scan each class for methods that correspond to a test case. Since the implementation of tests differs for each test framework, there is no general approach in deciding whether a method implements a test case or something else. Thus, this decision is delegated to the adapter for the test framework.

The pre-optimization data collection stage enables data stores to perform a collection of information before the optimization started. To this end, the data stores are called one after another. Thus, this stage does only perform a simple delegation.

The actual optimization is performed in the third stage. All strategies are called in their configured order: The first strategy is given the previously discovered test suite as input. Then, it applies its optimization and, thereby, yields an optimized test suite. If more than one strategy is specified, the next strategy receives the intermediate result of its predecessor as input. The test suite delivered by the last strategy is the final optimized test suite.

Once the strategies delivered the optimized test suite, the test run can be started. This is done by invoking the test framework with the optimized test suite through the adapter. Further, the adapter needs to retrieve the test results and translate them into the framework's internal data structures.

In the fifth stage, the data stores are once again allowed to collection information. This time they can access the test's results, which can, for example, be used to build a test history database. Apart from that, this stage is identical to the pre-optimization data collection stage.

Finally, a report of the test run's result needs to be created. This can includes the succeeded and failed tests, as well as further information, e.g., execution times and failure reasons. The reporting mechanism should allow for different target formats, for example, a console log for developers studying the test results or an XML file that allows machines to parse and process the information.

The UML component diagram in figure 3.9 depicts only the static architecture of the

RTO pipeline. The picture is completed by an UML sequence diagram in figure 3.11 that illustrates the previously described dynamic aspects of the pipeline.

### API

The framework can be used through a provided API. To this extent, the interface allows for *configuring the framework*, *starting a test run*, and *receiving the results*. The mandatory configuration comprises the data stores, strategies, and the test framework. Note that these might need some settings as well; for example, a data store may use a database to store the test history and, thus, needs the settings that enable the access to the database. Furthermore, the framework has optional settings, such as the forced inclusion and exclusion, i.e., lists of tests that should always be included or excluded no matter what the results of the strategies are, or the report output format, e.g., write results to the log or an XML file. Once the configuration is done, the API enables the start of the test run. Its results are reported back to the caller when the test run is finished.

### Dependency Loader

Strategies may require access to one or more data stores to perform their optimization. There are two ways to realize this: First, the framework's configuration knows which data stores and strategies should be used and the framework takes care of their setup including the interconnection of data stores and strategies. Second, the framework's configuration gets ready-to-use data stores and strategies that are already wired appropriately; this shifts the responsibility for the setup onto the framework user. The second approach is pursued by the framework. Since this alone would break with the framework's objective to allow for a simple integration, the framework provides a tool that helps with the setup: the *dependency loader*.

This tool is provided via the API and takes care of connecting strategies with the respective data stores, as depicted in figure 3.12. To this end, the dependency loader utilizes the concept of *dependency injection*. It describes that all dependencies of a component should be injected from the outside rather than creating new instances of them internally [Fow04]. This can, for example, be realized by delivering the dependencies when constructing the component.

The dependency loader expects the data stores and strategies as input and the strategies must specify which dependency they expect on construction. Then, the dependency loader constructs the data stores and injects them, where applicable, when constructing the strategies. Finally, the output, i.e., the setup data stores and strategies, can be provided to the framework configuration.

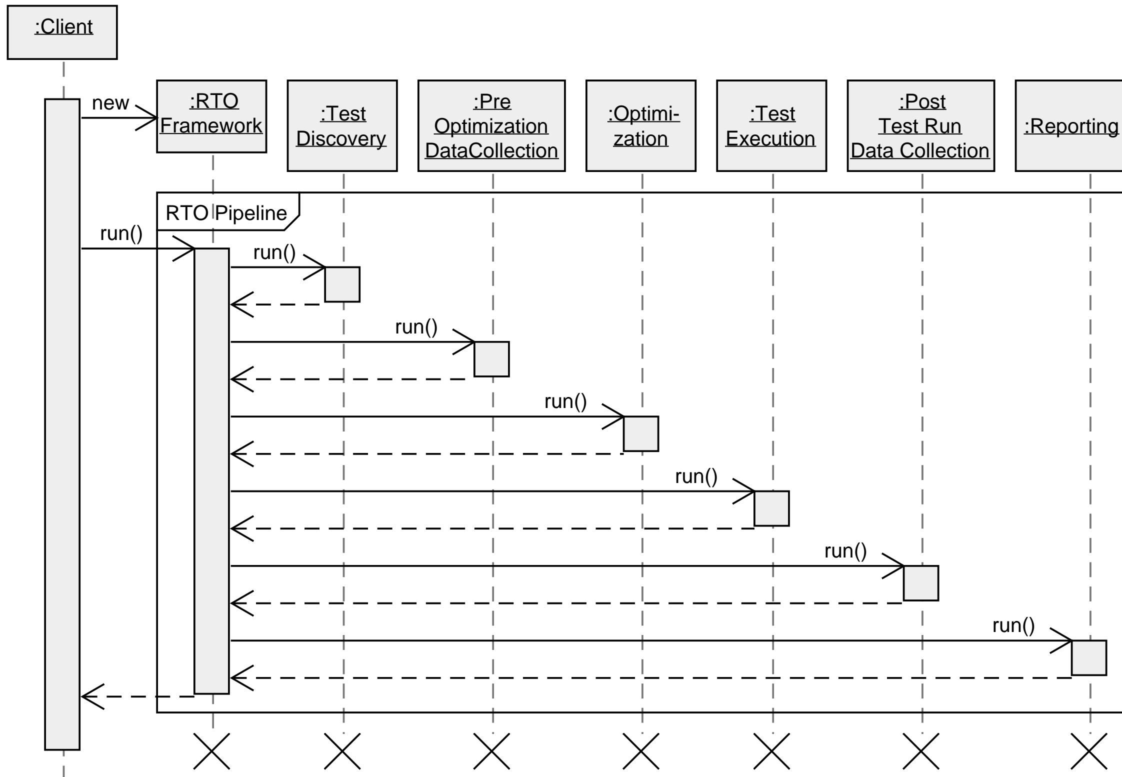



Figure 3.11: UML sequence diagram depicting the framework's RTO pipeline. When the framework's `run()` method is invoked, the pipeline, consisting of six stage, is processed sequentially; the stages are: the test discovery-, the pre-optimization data collection-, the optimization-, the test execution-, the post testing data collection-, and the reporting stage.

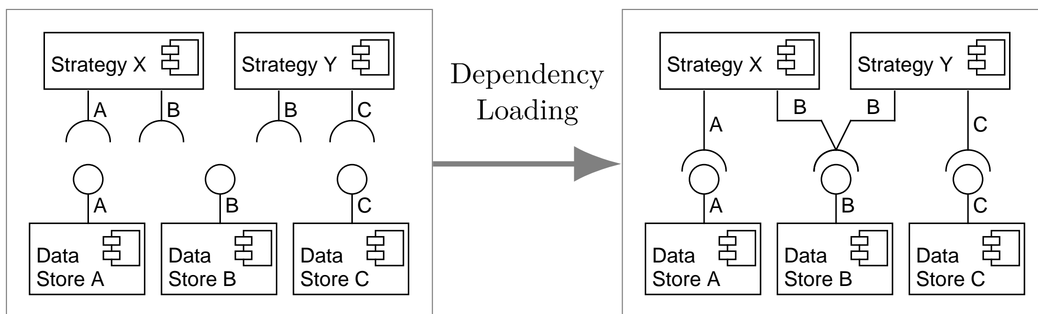



Figure 3.12: The dependency loader takes a given set of data stores and strategies and connects their interfaces appropriately.

# 4 Realization

Contents


4.1 Background . . . 35
4.1.1 Java Programming Language . . . 36
4.1.2 JUnit Testing Framework . . . 36
4.1.3 Maven . . . 38
4.1.4 Guice . . . 42
4.1.5 FreeMarker . . . 42
4.1.6 Logging with slf4j and logback . . . 43
4.1.7 Hibernate . . . 44
4.1.8 Liquibase . . . 44
4.2 Usage API . . . 45
4.2.1 Framework Settings . . . 45
4.2.2 Running Lazzer . . . 47
4.3 Extension API . . . 48
4.4 Lazzer Framework . . . 54
4.5 Exemplary Framework Extensions . . . 54
4.5.1 Client Adapters . . . 54
4.5.2 Data Stores . . . 58
4.5.3 Strategies . . . 60
4.5.4 JUnit Test Framework Adapter . . . 62


This chapter presents Lazzer, a prototype implementation of the RTO framework described in chapter 3. The technology used to build Lazzer is introduced first. Then, the details of the prototype's architecture are described. This chapter concludes with the some exemplary framework extensions.

## 4.1 Background

The implementation of Lazzer depends on several technologies that are introduced in this section.

The framework is written in the *Java* programming language and for the test execution the popular Java test framework *JUnit* is utilized.

The build automation tool *Maven* is used, on the one hand, for building Lazzer and, on the other hand, for a demonstration of how the framework can be integrated into a project

The dependency injection framework *Guice* is used to aid users with the setup of the framework; *FreeMarker* gives the framework flexibility regarding the format of the

results report; the logging facade *slf4j* and the logging framework *logback* are used to log the test run and its results; the tool *Liquibase* helps with setup of the database required by presented example for a data store and the library *Hibernate* helps the same data store to access the information in that database.

### 4.1.1 Java Programming Language

For the implementation of Lazzer, the Java programming language was chosen. It is a modern, actively developed object-oriented programming language [Orab]. According to the monthly updated TIOBE index, Java is currently the most popular programming language (state as of October 2015). The index is created by considering criteria like "the number of skilled engineers world-wide, courses[,] and third party vendors" as well as on "[p]opular search engines such as Google, Bing, Yahoo!" [TIO15]. Moreover, multiple modern xUnit test frameworks are available for Java, such as JUnit [JUnb] and TestNG [Tes]. Last but not least, there are well elaborated Java build tools, like, e.g., Maven [Mava].

### 4.1.2 JUnit Testing Framework

For Java projects, the open-source test framework JUnit [JUnb] is a popular choice and is considered as the xUnit framework (cf. section 3.1.2) for Java. The subsequent sections describe, how tests are defined and executed with JUnit.

#### Test Definition

In accordance with the xUnit architecture, tests are implemented as methods in test classes. Since JUnit 4, tests are identified via the Java annotation `@Test` [JUna, org.junit.Test] (cf. listing 4.1). Methods annotated with `@Before` and `@After` enable the implementation of a setup and teardown that is executed before and after each test is started [JUna, org.junit.Before, org.junit.After]. The annotations `@BeforeClass` and `@AfterClass` allow for a setup and teardown that is run once per test class [JUna, org.junit.BeforeClass, org.junit.AfterClass].

For the verification of the SUT, JUnit provides a set of assertion methods [JUna, org.junit.Assert], like, e.g., `assertEquals(int a, int b)` that checks for equality of a and b. In case that an assertion is violated, the assertion method throws a Java `AssertionError` that is caught by JUnit; the test is then recorded as failed. If the test finishes without raising such an exception, the test succeeded.

Listing 4.1 presents a brief code snippet of a test written with JUnit. The test `addCustomer` aims for the validating that new customers can be added to a shopping system. Therefor, it adds a customer to the system and then checks for his existence. The test's setup is done in the method `initialise`.

```java
public class CustomerTest {
    private ShopSystem shop;

    @Before // this method performs some setup
    public void initialise() {
        shop = new ShopSystem();
    }

    @Test // this method is a test method
    public void addCustomer() {
        shop.addCustomer("Bob");                   // Execute
        assertTrue(shop.existsCustomer("Bob")); // Verification
    }
}
```

Listing 4.1: A Java code snippet showing an exemplary unit test written with JUnit. The method annotated with `@Before` sets up a shop system which is then used in the test method that is recognized by the annotation `@Test`. The test executes the SUT by adding a customer and then verifies the outcome by checking for his existence.

### Test Execution

There are multiple ways to start a test run: one is, to execute JUnit as a stand-alone command line application that takes the test classes' names as input; another way is, to use the inbuilt support of IDEs, such as Eclipse and IntelliJ IDEA, that usually allows to select the tests and start the test run with a few mouse clicks; yet another way is, to use JUnit's API directly [JUna].

Independent of how JUnit is run, it expects the test suite as input, e.g., by providing a list of test classes. For discovering the tests in theses classes, JUnit utilizes Java's reflection mechanism that allows a program to analyze its structure, such as its classes, methods, and annotations. This allows JUnit to find the test, setup, and teardown methods and to construct a test runner that is responsible for the test execution.

Once the test run finished, JUnit reports back the result, i.e., information on whether the run was successful and, if not, a list of failed tests. For each failed test, the test's class and method is provided along with a message that describes the failure. The JUnit command line application prints the result to the output stream. IDEs usually provide a graphical interface that allows the developer to inspect the test result.

### Test Prioritization and Selection

Since the RTO framework prioritizes and selects test cases, it is noteworthy that JUnit has a mechanism for ordering and filtering tests [JUna, org.junit.runner.Request]. To this end, a `Request` instance has to be provided to JUnit.

A request allows the specification of a test order by implementing Java's `Comparator`

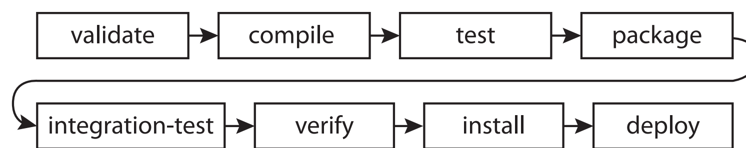



Figure 4.1: The Maven lifecycle

interface; it requires the definition of a method that decides upon the order of two test classes or methods. When JUnit processes the request, the method is repeatedly called to determine the order of all test classes and, for each class, the order of its test methods. Note that the methods of a test class are always executed in sequence; thus, methods of one test class cannot be arranged between two methods of another test class, i.e., a sequence like `A.x(), B.x(), A.y()` is not possible.

In order to filter test cases, an implementation of the abstract class `Filter` must be provided to the `Request` [JUna, org.junit.r.m.Filter]. To this end, the filter must decide for each test class and method, whether it should be executed. Similarly to the order mechanism, JUnit repeatedly calls this method for each test class and method before execution to filter the test suite.

### 4.1.3 Maven

Building Java projects often is a tedious job, that involves managing the project's dependencies, performing regression tests, and building a version of the software that can be released. Usually, a software is not only build and released once, but changes are made frequently, requiring this process to be repeated many times, making it error-prone if done manually.

Maven is an open-source tool that allows to automate the build process [Mava]. It enables developers to define build processes as well as project dependencies. With its flexible plug-in architecture it allows for adapting or extending the build process in many ways. Examples are *Surefire*, that automates test execution [Mavk], *checkstyle*, that checks whether the code meets the project's code style [Mavf], or the *reports* plug-in, that allows to create reports on the project's status [Mavg].

The tool is written in the Java programming language and is primarily designed for automating the build of Java projects; similar projects support other programming languages, too [Mavl, Mavh]. Maven requires only a *Java Development Kit* (JDK) installation, e.g., Maven 3.3 requires a JDK of at least version 1.7 [Mavb], and thus, is available for many platforms.

#### Build Lifecyle

The build process of Maven bases on a defined build lifecycle [Mavi]. This allows anyone knowing the basics of Maven to build Maven project's without knowing any details of the project specific build process; these details are hidden in the configuration. As depicted

in figure 4.1, the lifecycle is divided into phases, e.g., the *compile*, *test*, and *package* phase. Each phase has a distinct purpose: in the compile phase, the source code is compiled, in the test phase, tests are run, and in the deploy phase, the software system is made available to the users. The phases are executed sequentially, e.g., the compile phase is followed by the test phase, which is in turn succeeded by the package phase; when such a phase is executed, Maven automatically runs all preceding phases.

### Project configuration

The configuration of a Maven project is done via *pom.xml* files. These *extensible markup language* (XML) files must conform with Maven's *Project Object Model* (POM) [Mavj]. A simple project configuration includes a unique identification, a specification of the project's dependencies, and the plug-ins used during the build process. The project's identification, referred to as "fully qualified artifact name" [Mavd], is made up of three tags: the `groupId`, the `artifactId`, and the `version`. By convention, the `groupId` should be a reverse domain name of a domain you own, thereby ensuring a unique identification. The `artifactId` can be chosen freely and for the `version` a notation using numbers separated by dots is suggested, e.g., 3.0.1. In the same manner, dependencies and plug-ins are referenced by providing their `groupId`, `artifactId`, and `version`.

An example of a pom.xml is given in listing 4.2. In lines five to seven, the project identification is defined: according to the `groupId` the project belongs to the owner of the domain `swc.rwth-aachen.de`, the name of the project is `sample`, and the version is `1.0.1`. The lines 10 to 17 give the only dependency of the project: version 3.4 of the Apache Commons Java library `commons-lang3`. Finally, in lines 20 to 29 the build process is enhanced by specifying one plug-in: The `maven-javadoc-plugin` in the group `org.apache.maven.plugins`, that helps generating the source code documentation.

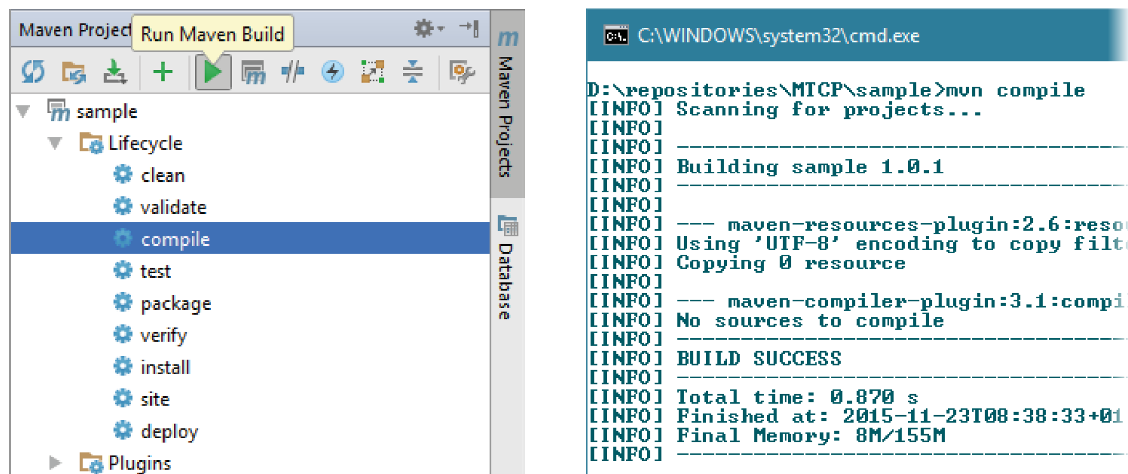
Projects can be composed of subprojects by building a hierarchy of POMs. To this extent, the tag *modules* is used in a POM to specify child projects, and, the POMs of child projects include a reference to the parent's POM using the *parent* tag. The POM also implements a concept of inheritance [Mavm], allowing child projects to inherit the configuration of their parent POM, e.g., dependencies and build plug-ins.

### Usage

Maven can be run and used in different ways. For developers the most convenient way is to use Maven from within their IDE, e.g., via the Eclipse Maven plug-in [Mave] or via IntelliJ IDEA's inbuilt Maven controls [Mavc] (cf. figure 4.2a). It is also possible to run Maven from command line (cf. figure 4.2b); this is often used to integrate Maven projects into *continuous integration* (CI) environments.

```
<?xml version="1.0" encoding="UTF-8"?>
<project <!-- ... -->>
    <modelVersion>4.0.0</modelVersion>

    <!-- Project identification -->
    <groupId>de.rwth-aachen.swc</groupId>
    <artifactId>sample</artifactId>
    <version>1.0.1</version>

    <!-- Project depends on the following artifacts -->
    <dependencies>
        <!-- Add dependency to Apache Commons library -->
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.4</version>
        </dependency>
    </dependencies>

    <!-- Build process relies on the following plugins -->
    <build>
        <plugins>
            <!-- Use javadoc to create source code documentation -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-javadoc-plugin</artifactId>
                <version>2.10.3</version>
            </plugin>
        </plugins>
    </build>

</project>
```

Listing 4.2: A minimal example of a Maven pom.xml for a fictitious project named *sample* that has one dependency to the Apache Commons library and uses the javadoc plug-in to build the project's documentation.




(a) In IntelliJ IDEA 15, Maven's phases are listed and can be selected for execution.

(b) Running `mvn compile` on command line compiles the Maven project.

Figure 4.2: Screenshots demonstrating how a Maven project can be compiled via IntelliJ IDEA 15 and via the command line.

```java
1 // Dependencies
2 PaymentService pay = new CreditCardService();
3 ShippingMethod ship = new ExpressDelivery();
4
5 // Injection
6 Order o = new Order(pay, ship);
```

Listing 4.3: Java code snippet illustrating manual dependency injection.

```java
1 // Construct guice injector
2 Injector injector = Guice.createInjector();
3
4 // Get instance
5 Order o = injector.getInstance(Order.class)
```

Listing 4.4: Java code snippet depicting dependency injection with Guice.

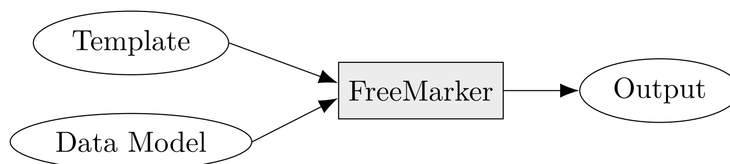



Figure 4.3: FreeMarker is a template engine, i.e., it takes a template and a model to produce an output of a format defined in the template.

### 4.1.4 Guice

Dependency injection can be performed manually or automatically with a dependency injector. For the manual injection, the developer first has to create instances of the dependencies and is then able to inject them; for example, in the Java code snippets in listing 4.3 an instance of order is created that depends on `PaymentService` and `ShippingMethod` that need to be instantiated before.

Google's Guice is an open-source Java library that provides a dependency injector that can take care of the injection [Goo]; the code snippet in listing 4.4 depicts how the order can be constructed using Guice; note that the instantiation of the dependencies is now encapsulated by the injector.

### 4.1.5 FreeMarker

For programs that produce an output, it is disadvantageous to define the output's format, e.g., plain text or XML, in the program code; whenever a different format is needed, the code needs to be changed. Thus, it is preferable to separate the output format definition from the program's source code. This can be done with *template engines*. As figure 4.3 depicts, such engines generate the output from a template and a data model. The data model is built in the program and the template can be read from an external source, such as a file. The template engine provides a simple scripting language that allows to build the output format and to access the contents of the data model via template variables. Since the template determines the format and it is stored in a file, the output format can be adapted easily.

FreeMarker is an open-source Java template engine [Fre]. Listing 4.5 presents an example of a template for HTML output. The example template can be used to render a website that greets a user and states his last login time. These information are stored in the data model that is setup in the program code, as depicted in listing 4.6. According to this model, the user is named Gordon and logged in last on 11th November 2015. Correspondingly, the rendered website will state “Welcome back Gordon. Your last login was on 2015-11-20.” FreeMarker's template engine is very feature-rich; for example, it allows the usage of collections, it provides a variety of string manipulation functions, and it allows for calling Java methods from the template.

```
1 <html>
2   <body>
3     Welcome back ${user}. Your last login was on ${lastLoginTime}.
4   </body>
5 </html>
```

Listing 4.5: An example of a FreeMarker template that defines a simple HTML output. Template variables such as `${user}` are replaced with their respective value from the data model.

```
1 // Create a map that serves as model
2 Map<String, Object> model = new HashMap<>();
3 // Define the model's contents
4 model.put("user", "Gordon");
5 model.put("lastLoginTime", "2015-11-20");
```

Listing 4.6: A Java code snippet that demonstrates the setup of a FreeMarker model. Here, a model with two template variables is defined: `user` will store the name "Gordon" and the `lastLoginTime` is the 20th of November 2015.

### 4.1.6 Logging with slf4j and logback

Logging is an essential part for profiling and tracking a program's execution. *Logging* is a mechanism that allows to do this by writing text messages to output streams or log files. The provided information can vary in their importance, e.g., some messages are only meant to give developer's a clue what part of a program is being executed while others inform about severe failures. To this end, messages are usually associated with a logging level; for example, some common levels are *debug*, *information*, and *error*. Note that logging levels are ordered, e.g., debug is a lower level than information and error, which allows for filtering by specifying a minimum level.

Logging frameworks aim to give the developer a uniform interface for publishing log messages. To this extent, frameworks usually specify a set methods that are named after log levels; each taking a message as input that is then published accordingly. Some frameworks provide additional features, like the possibility to filter out message that are below specific log level.

The open-source Java library slf4j provides a facade for various logging frameworks [QOSb]. This yields the advantage of being able to choose a logging framework when the application is deployed by simply attaching the corresponding dependency. If no logging framework is specified, logging is disabled. One of the logging frameworks that support slf4j is logback [QOSa]. Noteworthy, is its configuration that can be done via XML files. Among other things, it allows for filtering and formatting the log messages. The logging settings can be modified while the application is running, which allows to obtain more detailed information when required.

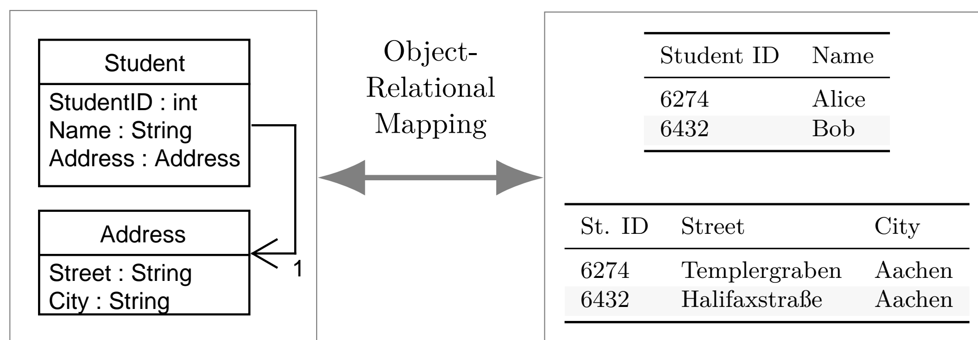



Figure 4.4: The object-relational mapping is responsible for translating data between the object-oriented and the relational world.

### 4.1.7 Hibernate

Relational databases are well established for storing and retrieving information. However, their concept of data representation is different to the one of object-orientation: In relational databases, information is represented as a set of tuples with scalar values, i.e., no complex data types. In contrast, object-oriented software structures data by classes that can have fields of scalar as well as complex types. Moreover, these structures can utilize associations to other classes as well as inheritance. Their objects correspond to respective sets of data. Transforming information between both worlds is a challenging task, since object-orientation features concepts that are unknown to relational database, such as inheritance. The solution to this problem is the definition of an *object-relational mapping* (ORM) that deals with such a transformation, as illustrated by figure 4.4.

Another challenge when using databases is to always keep the state of the information in the software synchronized with the database. Inconsistencies can arise when data is added, modified, or deleted, but the database is not updated accordingly. Taking care of this manually is tedious and error-prone motivating the usage of *persistence* libraries that take care of this.

Hibernate is a Java library that aids the developer by providing a persistence API and a high level description mechanism for the ORM. To this end, the developer must specify what information has to be considered by the mapping and how it should be mapped by choosing from predefined mapping options. This settings can either be done in dedicated configuration files or by annotating the relevant fields of respective classes.

### 4.1.8 Liquibase

When a program uses a relational database to store and retrieve information, this database needs to be set up first. Doing this manually is tedious and error-prone, which is why its automation is desirable. This can be achieved by writing scripts that, when executed, perform the setup by issuing a batch of queries to the database. Another beneficial effect of using scripts is that they can be put under version control along with the application's source code.

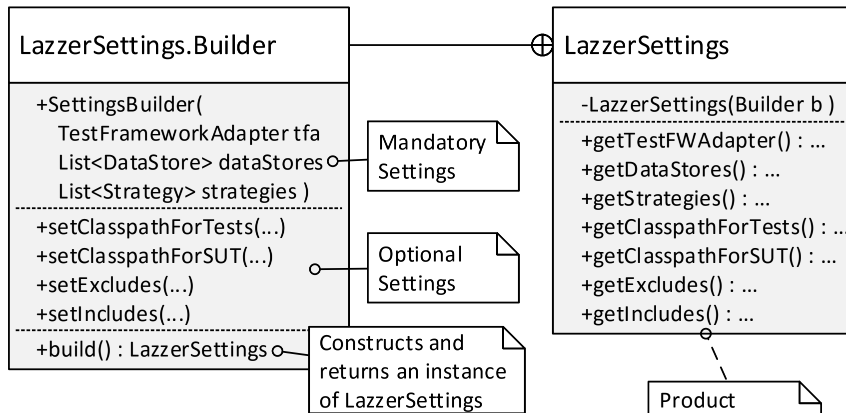



Figure 4.5: UML class diagram depicting the classes for Lazzer's configuration. The class `LazzerSettings` provides read-only access to every setting. The builder pattern is used to instantiate a settings object.

The evolution of a software is likely to repeatedly cause changes to the database schema. Thus, developers are faced many times with the challenge of migrating existing data sets. The complexity of scripting a migration process motivates the usage of *database refactoring tools* that apply the respective modification and migration automatically.

Liquibase is an open-source Java database refactoring tool [Liq]. It enables developers to define the initial database schema and subsequent changes in a configuration file, which is then used by Liquibase to setup the database and to migrate existing data sets.

## 4.2 Usage API

This section presents the API that makes Lazzer's functionality available for usage. This includes a description of the settings and an explanation of how Lazzer is executed.

### 4.2.1 Framework Settings

Before Lazzer can be used, it needs to be configured. Mandatory is the specification of the *test framework adapter*, a list of *RTO strategies*, and a possibly empty list of *data stores* (cf. requirements F1.20 and F2.11). The optional settings are *classpaths* and *forced exclusion and inclusion* of tests. In Java, classpaths are used to specify where a program's class files can be found. Lazzer allows to set classpaths for the SUT as well as the tests. The forced inclusion and exclusion of tests enables a framework user to remove test methods or whole test classes from the specified test suite; giving the user additional control on what to consider in a test run.

The configuration is an instance of `LazzerSettings`, depicted in the UML class diagram in figure 4.5. The instances are immutable, i.e., a configuration is read-only

```
1 LazzerSettings settings = new LazzerSettings.Builder(
2     testFrameworkAdapter,
3     aListOfDataStores,
4     aListOfStrategies
5 ).setTestsClasspath(testCp).build()
```

Listing 4.7: Java code snippet that depicts the instantiation of Lazzer's settings

```
1  TestFrameworkAdapter = assistant.getTestFrameworkAdapterLoader()
2      .getInstanceByName(new ClassName("de.rwth.swc.lazzer." +
3          "testrunner.junit.JUnit4Adapter"));
4
5  List<DataStore> dataStores = assistant.getDataStoreLoader()
6      .getInstancesByName(ClassNames.fromStrings(Arrays.asList(
7          "de.rwth.swc.lazzer.datastores." +
8          "testhistory.TestHistoryDataStore")
9  ));
10
11 List<DataStore> strategies =  assistant.getStrategyLoader()
12     .getInstancesByName(ClassNames.fromStrings(Arrays.asList(
13         "de.rwth.swc.lazzer.strategies." +
14         "failedfirstprioritization.FailedFirstPrioritization"
15 )));
```

Listing 4.8: Java code snippet that shows the usage of the dependency loading assistant.

and needs to be specified completely when it is created. Using constructors for the instantiation is tedious since Java does not have a mechanism for optional constructor arguments. One way to mimic optional arguments is to apply the *telescoping constructor pattern*; i.e., a class provides one constructor that only takes the mandatory arguments and overloaded constructors that additionally require the optional arguments. This is feasible for a few optional parameters, but it is expected that Lazzer will have a growing set of optional settings in the future. To live up with these expectations, a variant of the *builder pattern* is applied [Blo08].

The instantiation of `LazzerSettings` is done via the `Builder` class (cf. listing 4.7). Its constructor expects the specification of all mandatory settings. Once an instance of the builder is created, its setters allow for specifying the optional settings. When the configuration is complete, the builder's `build` method can be called to obtain an instance of `LazzerSettings`. Note that new optional settings can be introduced easily by adding a setter to the builder and a corresponding getter to the settings.

### Dependency Loader

Lazzer aids the user with an optional assistant for the construction of the data stores, strategies, and the test framework adapter. As depicted in figure 4.6, the

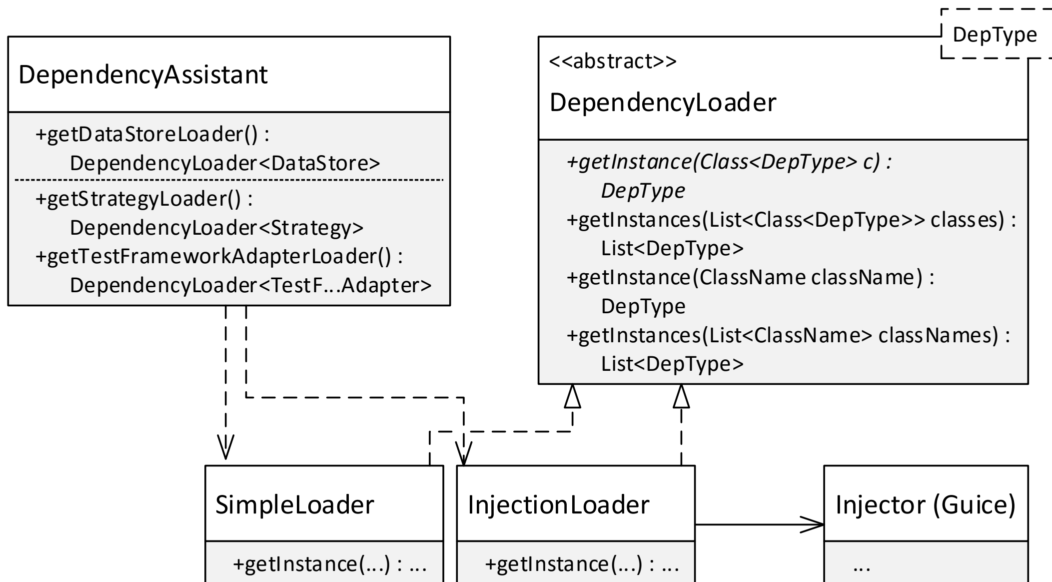



Figure 4.6: UML class diagram depicting the assistance mechanism for instantiating data stores, strategies, and the test framework adapter.

`DependencyAssistent` provides a `DependencyLoader` for each of the three classes. The user can ask the loaders to get instances of these classes by calling `getInstance` with the respective Java class that implements the data store, strategy, or test framework adapter (cf. listing 4.8). In case only class names are known, the convenience method `getInstanceByName` can be used; it attempts to find and load, and instantiate the respective Java classes. Analogously, the method `getInstances` and `getInstancesByName` allow the instantiation of multiple classes at once.

The concrete instantiation is done by two classes realizing the abstract class `DependencyLoader`: The `InjectionLoader` is used to construct data stores and strategies. It takes case of injecting all dependencies by delegating the construction to the dependency injector of the Guice library. It tries to find, instantiate, and inject the dependencies. The test framework adapter should not have dependencies, and thus, does not require injections. Hence, the `SimpleLoader` is used; it calls `newInstance()` on the provided class to perform the instantiation.

### 4.2.2 Running Lazzer

Once the user has finished the settings, Lazzer can be instantiated. To this extent, a variation of the *abstract factory* is applied (see figure 4.7). The construction is the responsibility of the `create` method in the `LazzerFactory` class; it takes the settings as input and returns an instance of the interface `Lazzer` (the abstract product). The application of this pattern allows for easily exchanging the implementation of Lazzer

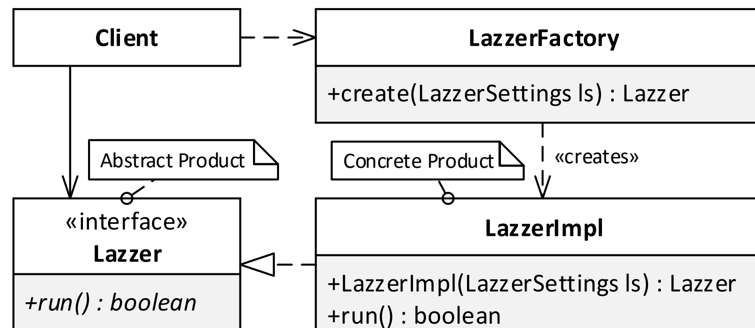



Figure 4.7: UML class diagram depicting a variation of the *abstract factory* pattern, which is used for the instantiation of Lazzer.

without requiring modifications on the framework user's side.

The `run` method of this interface allows for Lazzer's execution (cf. requirements F1.10 and F2.10); it returns true in case of a successful test run and false otherwise. The actual implementation of this is done in the `LazzerImpl` class (the concrete product).

A minimal working example of the configuration and execution of Lazzer is given by the code listing 4.9. The first step, performed in lines four to twelve, is the specification of the settings. Here, JUnit is used as test framework, an alphabetic prioritization is used as optimization, and, since this strategy does not need additional information, no data stores are used. Then, in line 15, these settings are used to instantiate Lazzer. Finally, in line 18 Lazzer is executed.

## 4.3 Extension API

Lazzer has four hot spots allowing the developers to extend the framework in multiple ways: by new data stores, new prioritization and selection strategies, and new test framework adapters. This section describes the respective extension mechanisms in more detail. Further, it explains how Lazzer handles test classes and methods, which is relevant for the development of strategies and test framework adapters.

### Data Stores

For the implementation of data stores, the developer either needs to implement the `DataStore` interface or the abstract class `AbstractDataStore` that are both shown in the UML class diagram in figure 4.8 (cf. requirement F2.20). The interface specifies two methods: The first, `preOptimizationDataCollection`, is called by the framework prior to the optimization phase, giving a data store the chance to perform an operation before strategies access it. For example, a data store that gathers version control information updates its database prior to the optimization step. The second method, `postTestRunDataCollection`, is invoked after the test run provided with the report

```java
public class RunLazzer {
    public static void main(String[] args) {
        // Configuration
        LazzerSettings lazzerSettings =
            new LazzerSettings.Builder(
                // test framework adapter
                new JUnit4Adapter(),
                // strategies
                Arrays.asList(new AlphabeticPrioritisation()),
                // data stores
                Collections.emptyList()
            ).build();

        // Instantiate lazzer
        Lazzer lazzer = LazzerFactory.createLazzer(lazzerSettings);

        // Execute test optimization & test run
        lazzer.run();
    }
}
```

Listing 4.9: Java code snippet that depicts a minimal working example of setting up and executing Lazzer. Lines four to twelve contain the configuration. This is followed by the instantiation of Lazzer in line 15. Then, in line 18 Lazzer is executed.

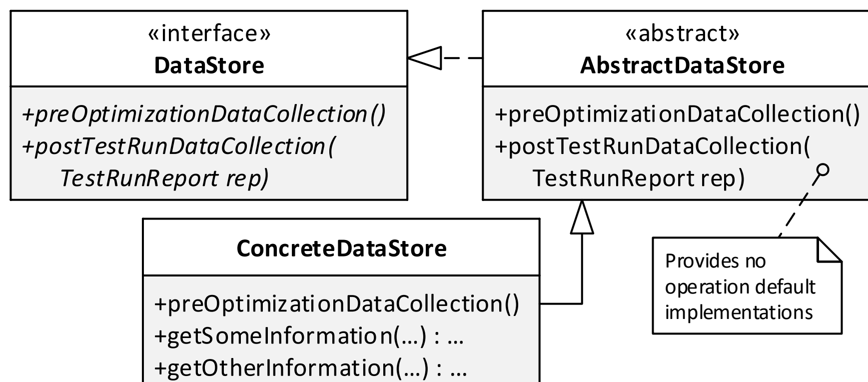



Figure 4.8: UML class diagram depicting the interface of data stores, a abstract class providing no-operation implementations of the interface's methods for convenience, and an exemplary data store implementation.

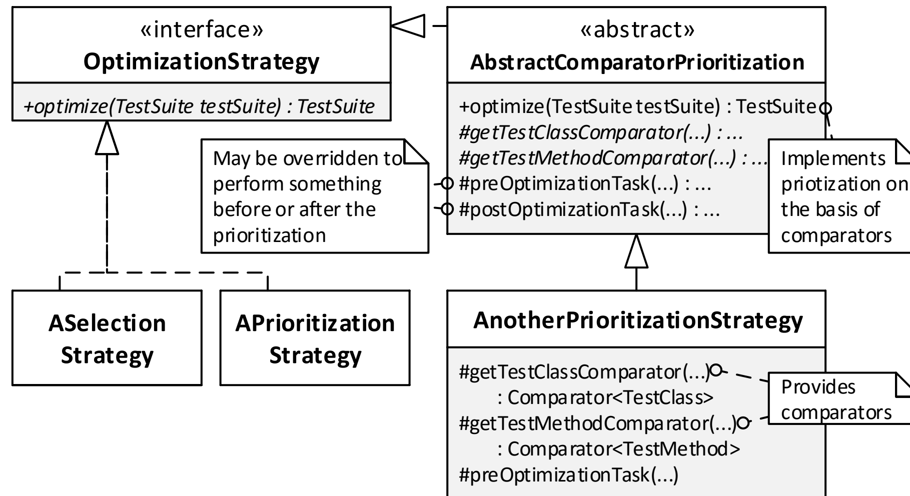



Figure 4.9: UML class diagram depicting the interface of optimization strategies.

of the test results, and can, for example, be used to build up a history of test runs. Not all data stores have a need for one or both of these methods. In this case, it is preferable to subclass `AbstractDataStore` that provides default implementations for the methods that do nothing. In order to provide strategies with information, concrete data stores have to define additional methods, as depicted in the UML diagram.

### Strategies

New strategies can be made available to the framework by implementing the `OptimizationStrategy` interface as presented in the UML class diagram in figure 4.9 (cf. requirement F2.30). The interface requires the implementation for the method `optimize`. It is invoked by the framework in the optimization phase with the test suite as input and expects an optimized test suite as a result. To this end, the test suite data structure allows for reordering and removing tests cases, which enables the implementation of prioritization, selection, and hybrid approaches.

### Test Framework Adapters

Test test framework is adapted through the `TestFrameworkAdapter` interface, which is shown in the UML class diagram in figure 4.10 (cf. requirement F2.30). It requires the implementation of the methods `provideTestDiscovery` and `runTests`.

The method `provideTestDiscovery` aims for identifying the test cases and test suites. This is necessary, because tests are implemented in a way specific to the test framework, and thus, no general solution exists. Hence, the test framework adapter provides an implementation of the `TestDiscovery` interface. Its method `discoverTests` is called by the framework in the test discovery stage and yields the test

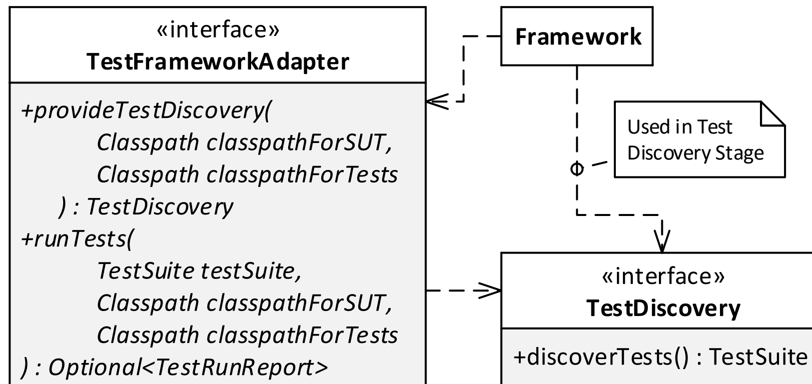



Figure 4.10: UML class diagram depicting the interfaces for the test framework adapter and the test discovery.

suite as result. Note that in order to locate and load the test classes, the test framework adapter is provided with the classpaths of the tests and the system under test.

The second method, `runTests`, is called by the framework in the test run phase and is responsible for the execution of the test run. To this end, it is provided with the optimized test suite as well as the classpaths for the tests and the system under test. The method returns a report of the test results. In case the test run could not be started, no result is expected; thus, `Optional.empty()` is returned.

### Test Suite

The test suite provided by the test framework adapter's discovery must conform to a structure predefined by the framework. As depicted by figure 4.11, it consists of three classes: the `TestSuite`, the `TestClass`, and the `TestMethod`.

The test suite is a container for test classes. It implements Java's `List` interface that allows developers to manipulate and iterate over the test classes in a way they are familiar with. Additionally, it provides a method for retrieving a test class by its class name.

Test classes are represented by `TestClass`. It is made up of at least one test method. Like the test suite, it implements Java's `List` interface. Besides that it provides methods for getting the class' name and the respective Java class. For convenience, it enables accessing test methods by their names.

The structure corresponding to a test method is `TestMethod`. For now, it can only provide its name. However, having a designated class for test methods allows for future enhancements; for example, the incorporation of the method's signature.

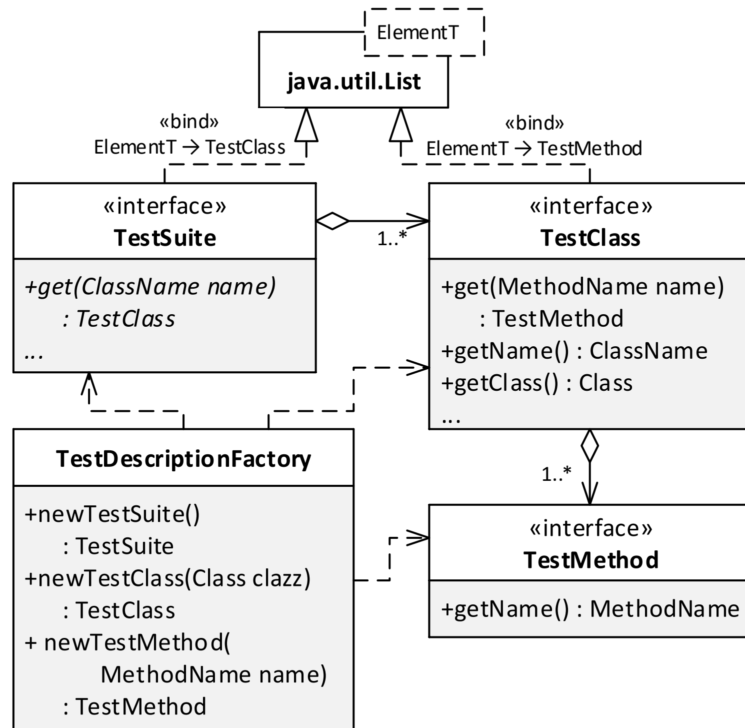



Figure 4.11: UML class diagram depicting the classes `TestSuite`, `TestClass`, and `TestMethod` as well as their factory.

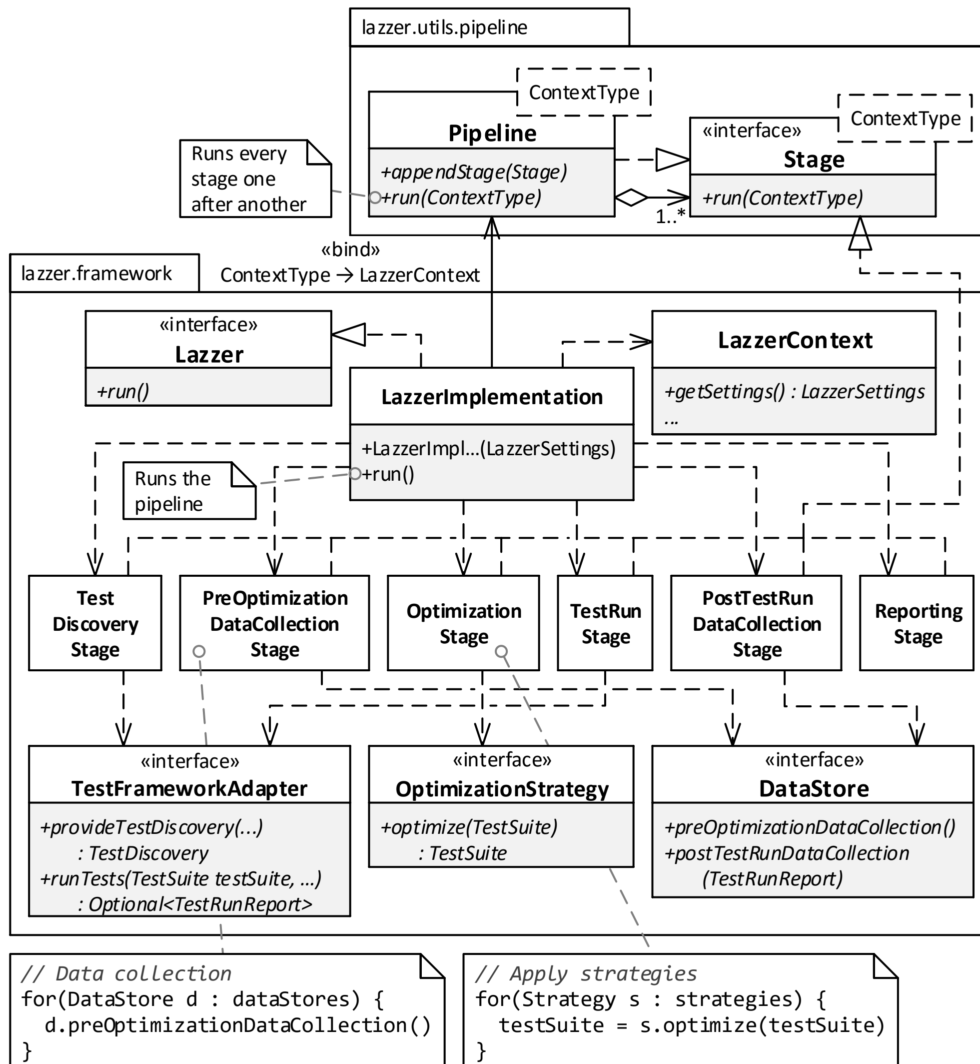



Figure 4.12: UML class diagram showing the internal architecture of the Lazzer framework.

## 4.4 Lazzer Framework

The Lazzer framework can be accessed via the `Lazzer` interface. The corresponding implementation is located in the class `LazzerImplementation`, as depicted in 4.12. It implements only the `run` method that is required by the interface. The method's concern is to realize the RTO pipeline that is introduced in section 3.3.2. For this purpose, the a variant of the *pipeline pattern* is used.

The pipeline pattern has two key elements: One is the interface `Stage` that only declares the method `run` that takes as input a context that may be used to share data along the pipeline's stages. The other key element is the class `Pipeline`; it consists of a list of stages and is itself a stage; the pipeline implements the inherited `run` method by sequentially calling the `run` method of its stages.

The `LazzerImplementation` internally creates a pipeline and adds instances of six classes, each realizing a stage of the previously described RTO pipeline: the *test discovery*, the *pre-optimization data collection*, the *optimization*, the *test run*, the *post-test-run data collection*, and the *reporting* stage. The concern of the test discovery stage is to obtain the test suite, which is delegated to the `TestDiscovery` interface provided by the test framework adapter. The pre-optimization data collection stage iteratively calls the data stores's method `preOptimizationDataCollection`. In the optimization stage, the `optimize` method of each strategy is called one after another (cf. requirement F1.30); the first strategy is provided with the discovered test suite while the following are given the outcome of the preceding `optimize` call. In the test run stage, the thereby obtained optimized test suite is passed to the test framework adapter for execution (cf. requirement F1.40). The post-test-run data collection again invokes each data store, this time providing them with the test result. Finally, the reporting stage is responsible for outputting the test results (cf. requirement F1.50); to this extent, the FreeMarker library is used to load a template file and to generate an output that is then logged using the logging facade slf4j and its implementation logback.

## 4.5 Exemplary Framework Extensions

The Lazzer framework's extension points are the *client adapters*, the *data stores*, the *optimization strategies*, and the *test framework adapter*. In the course of this thesis, exemplary implementations were developed for each extension point, which are described subsequently.

### 4.5.1 Client Adapters

The prototype implementation features two client adapters. The first is a Maven plug-in that enables the integration of Lazzer into a build process of Maven projects. The second is a command line interface that allows for using Lazzer as a stand-alone application. Hereafter, the two components are described in more detail.

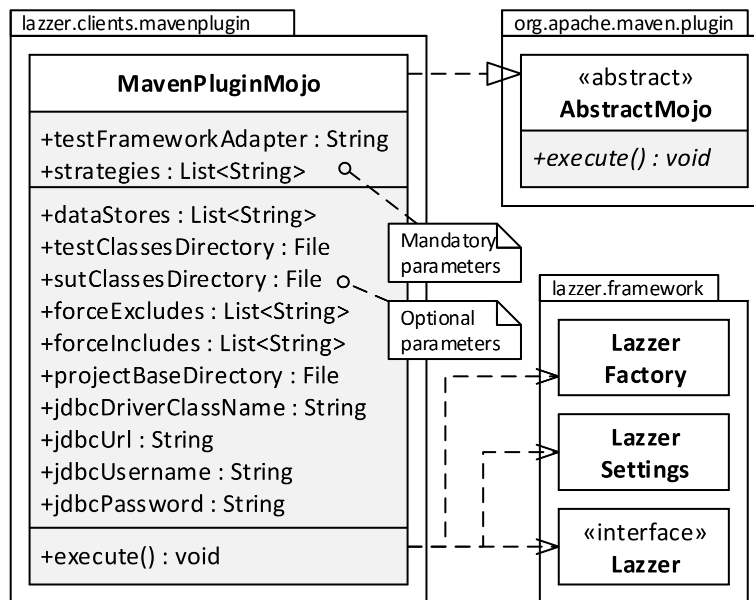



Figure 4.13: UML class diagram of the framework's Maven plug-in.

### Maven Plug-in

Maven plug-ins can be implemented by realizing the abstract class `AbstractMojo`, and thus, implementing the method `execute`. Mandatory and optional configuration can be defined for the plug-in via fields annotated with `@Parameter`; thereby, enforcing or allowing to set parameters in the pom.xml (cf. listing 4.10). Here, Maven takes care of parsing the settings in the pom.xml and assigning them to the corresponding fields. Further, the implementing class can be annotated with `@Mojo`, allowing to specify, i.a., a default phase in which the plug-in should be called.

Lazzer's Maven plug-in is realized by the class `MavenPluginMojo`. This class and its fields, i.e., the configuration parameters, are depicted in the UML class diagram in figure 4.13. Here, the specification of the test framework adapter's class name and a list of class names of the strategies is mandatory. The remaining fields represent optional settings. Namely, the data stores – Maven interprets no specification as an empty list, which is a valid input for Lazzer. The paths to the test classes and the SUT, needed to set up the respective classpaths; by default, both parameters obtain the respective paths from the Maven variables `${project.build.testOutputDirectory}` and `${project.build.outputDirectory}`. The parameters for forced exclusions and inclusions allow for a list of regular expression patterns that exclude or include a test class or case, if it matches their name.

The remaining parameters, namely, the project base directory as well as the four parameters with the prefix "jdbc", do not belong to the framework's configuration, but are needed by some data stores that are introduced in section 4.5.2. The first tells the

```
<project><build><plugins>
    <!-- Lazzer Maven plugin -->
    <plugin>
        <groupId>de.rwth.swc.lazzer.clients</groupId>
        <artifactId>lazzer-maven-plugin</artifactId>
        <version>1.0</version>
        <!-- Framework configuration -->
        <configuration>
            <testFrameworkAdapter>
                de.rwth.swc.lazzer.testrunner.junit.JUnit4Adapter
            </testFrameworkAdapter>
            <strategies>
                <strategy>
                    de.rwth.swc.lazzer.strategies
                        .alphabeticprioritization
                        .AlphabeticPrioritisation
                </strategy>
            </strategies>
        </configuration>
        <!-- Activate lazzer's "test"-goal -->
        <executions>
            <execution>
                <goals>
                    <goal>test</goal>
                </goals>
            </execution>
        </executions>
        <!-- Define dependencies (resolved by Maven) -->
        <dependencies>
            <!-- Test Framework Adapter -->
            <dependency>
                <groupId>
                    de.rwth.swc.lazzer.test-framework-adapters
                </groupId>
                <artifactId>junit4-adapter</artifactId>
                <version>1.0</version>
            </dependency>
            <!-- Strategies -->
            <dependency>
                <groupId>de.rwth.swc.lazzer.strategies</groupId>
                <artifactId>alphabetic-prioritization</artifactId>
                <version>1.0</version>
            </dependency>
        </dependencies>
    </plugin>
</plugins></build></project>
```

Listing 4.10: A snippet of a Maven configuration file depicting the integration of the Lazzer framework.

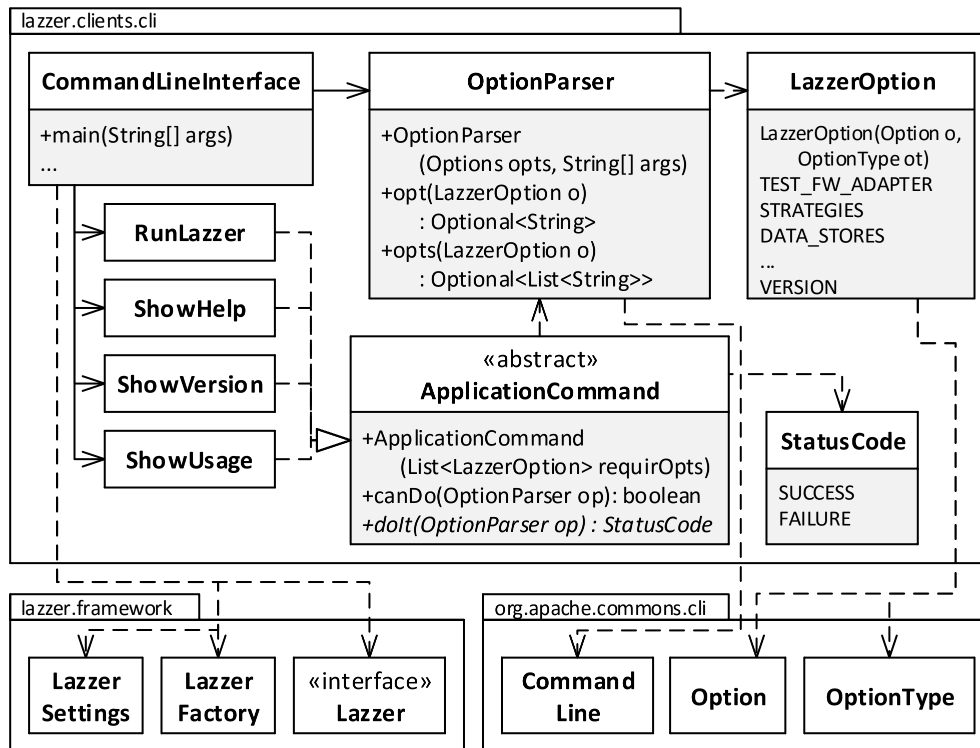



Figure 4.14: UML class diagram depicting the command line interface allowing for the framework's stand-alone usage.

plug-in where the root directory of this Maven project is located; by default, it is replaced with the value of the variable `${project.basedir}`. The later are needed to establish a database connection via the *Java Database Connectivity* (JDBC) API. Surely, a better solution would be to have a mechanism that allows for individual configurations for each data store. However, it was not possible to realize such a mechanism within the scope of this thesis, which reasons this compromise.

### Command Line Interface

Using Lazzer as a stand-alone application is made possible by the command line interface. To this end, Java requires the implementation of a `main` method. It is located in the class `CommandLineInterface` that is shown in the UML class diagram in figure 4.14.

The configuration of Lazzer is done via command line arguments (cf. listing 4.11). The provided arguments are parsed by the `OptionParser` class. The allowed options are defined by constants of the enum `LazzerOption`. The constants specify the parameters, e.g., `--strategies`, and whether they expect arguments. The *command pattern* is used to implement the execution of Lazzer as well as the display of the version, the help, and the usage information. The commands provide an additional method `canDo` that

```
1 lazzer
2     --runner=de.rwth.swc.lazzer.testrunner.junit.JUnit4Adapter
3     --strategies=de.rwth.swc.lazzer.strategies
4         .alphabeticprioritization.AlphabeticPrioritisation
5     --testClasspath=... --sutClasspath=...
```

Listing 4.11: An example for how to use Lazzer from the command line.

determines whether all the options required to execute it are provided; for example, for Lazzer to run a test framework adapter and at least one strategy must be provide.

### 4.5.2 Data Stores

In order to demonstrate the implementation of data stores, two examples are presented: The *Git data store*, allowing strategies to obtain information about the project's Git repository, e.g., the commit history and changes that were introduced. The other, the *test history data store*, collects information from test result reports and allows strategies to access the history of test runs. Both are depicted in the UML class diagram in figure 4.15.

#### Git

The Git data store extends the abstract class `AbstractDataStore`. Since it does not perform any task before the optimization or after the test run, the default no-operation implementation is not overridden. Strategies can get information about the repository via the method `getGitRepository` that returns a `Repository` object, which is part of the library jGit that allows for accessing git repositories. In order to setup an instance of `Repository`, Git's root directory is required. This is located by scanning the project's path obtained from the `ProjectDescriptor` that is injected upon construction of the Git data store.

#### Test History

This example illustrates a more complex data store, comprising multiple classes and interfaces. The `TestHistoryDataStore` extends the abstract class `AbstractDataStore` and overrides its method `postTestRunDataCollection` to store the test results of a test run, represented by the classes `TestRunReport` and `TestResult`, in a database. The access to the database is abstracted with the interface `TestHistoryDAO`. It defines methods for storing test results and retrieving the test history.

For the realization of the `TestHistoryDAO`, the Hibernate library is used. In order to connect to the database and process the test results, i.e., the `TestRunReport` and the `TestResult`, it needs a JDBC configuration and an ORM, respectively. The former is injected when the data store is constructed. The later is implemented in the classes

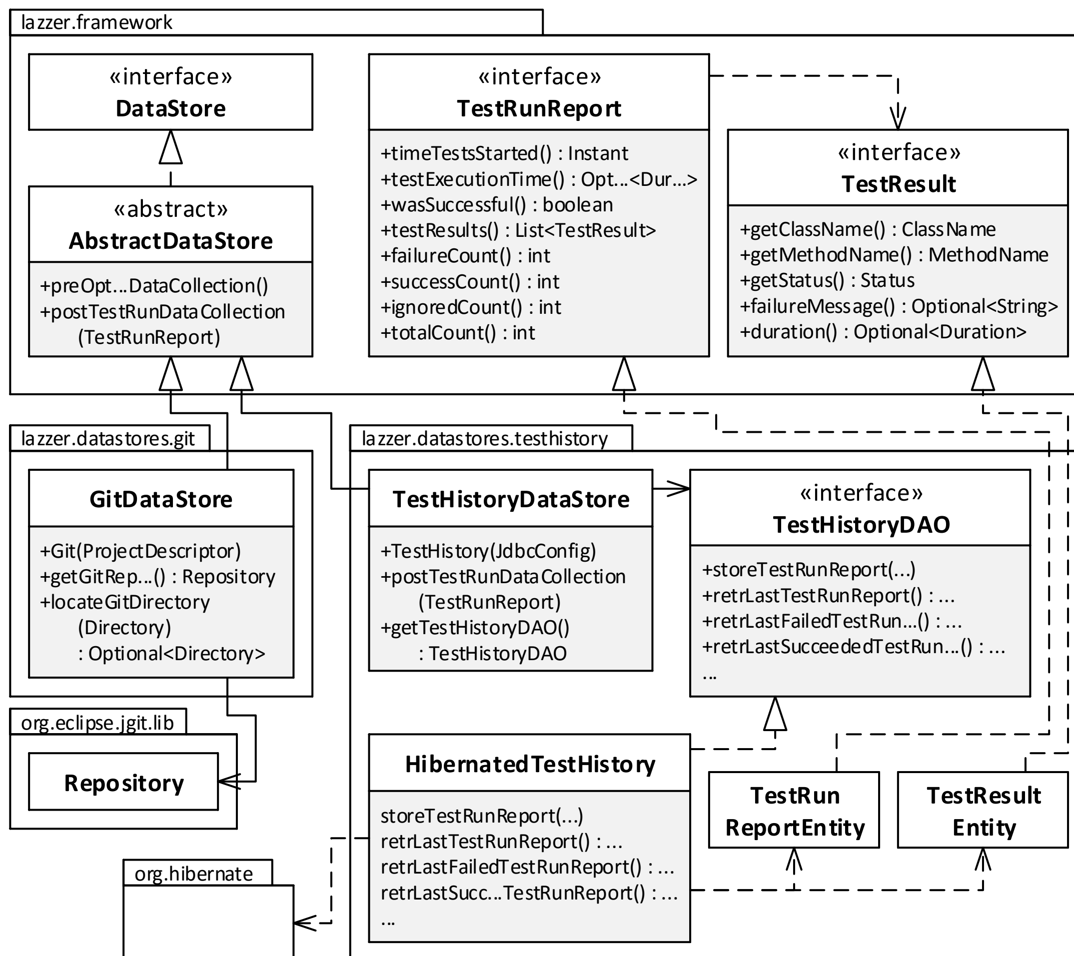


Figure 4.15: UML class diagram depicting the exemplary data stores.

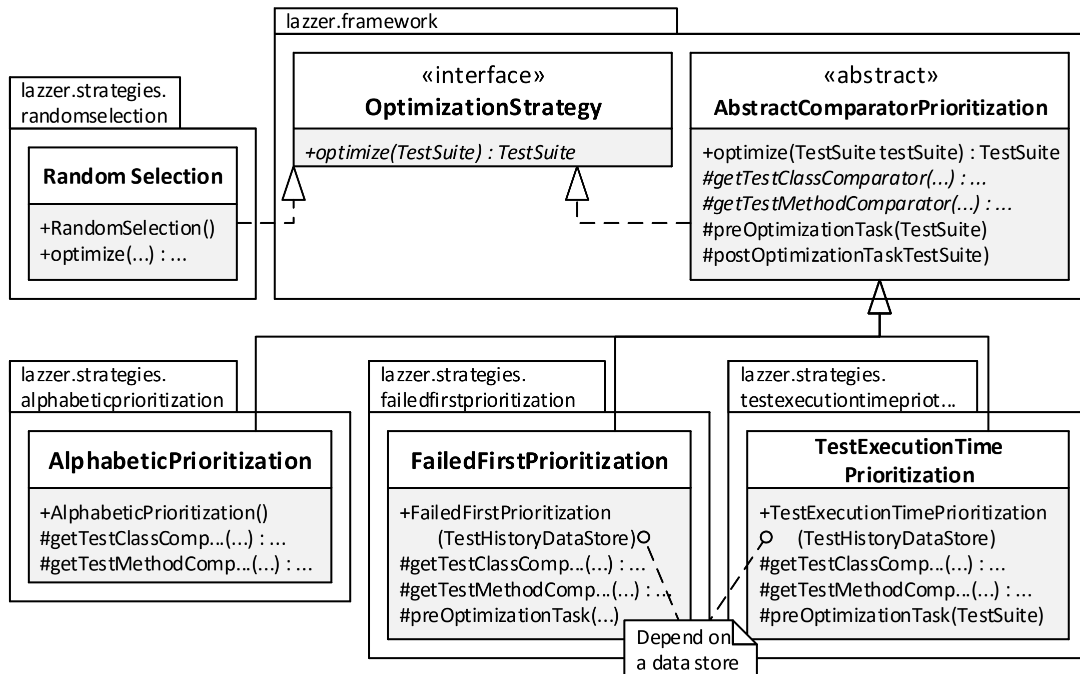



Figure 4.16: UML class diagram providing an overview of the exemplary optimization strategies. Random selection directly implements the interface `OptimizationStrategy`, whereas the other strategies extend the abstract class `AbstractComparatorPrioritization`. Note that failed-first prioritization and test-execution-time prioritization both depend on the `TestHistoryDataStore` that has to be injected upon their instantiation.

`TestRunReportEntity` and `TestResultEntity`. The required database setup and possible migrations are handled by Liquibase; the respective configuration file is provided along with the data store's sources.

Strategies can use the functionality provided by the interface `TestHistoryDAO` via the method `getTestHistoryDAO` in the class `TestHistoryDataStore`. This gives strategies access to the test history database.

### 4.5.3 Strategies

Four exemplary optimization strategies have been developed in the course of this thesis: *random selection*, *alphabetic prioritization*, *failed-first prioritization*, and *test-execution-time prioritization*. The first two are meant to illustrate how strategies are implemented and do not provide an optimization. They are depicted in the UML class diagram in figure 4.16 and each of them is briefly described in the following.

```
1  public final class AlphabeticPrioritisation
2      extends AbstractComparatorPrioritization {
3
4      @Override
5      public Comparator<TestClass> getTestClassComparator
6          (TestSuite testSuite)
7      {
8          return (tcA, tcB) -> tcA.getName().getCanonicalName()
9              .compareTo(tcB.getName().getCanonicalName());
10     }
11
12     @Override
13     public Comparator<TestMethod> getTestMethodComparator
14         (TestSuite testSuite)
15     {
16         return (tmA, tmB) -> tmA.getName().compareTo(tmB.getName());
17     }
18 }
```

Listing 4.12: Java code of the alphabetic prioritization strategy. The comparators are defined in lines eight to nine and 16, respectively.

#### Random Selection

The random removal of test methods from a test suite is implemented by the random selection strategy. This does not provide an optimization but helps to understand the `OptimizationStrategy` interface. The strategy iterates over all test methods and decides for each whether to keep it or not. This example uses a fixed probability of 50% for the removal. In the case that all test methods of one test class are removed, the class is excluded as well. The corresponding implementation is located within the class `RandomSelection`.

#### Alphabetic Prioritization

Ordering the test classes and methods in alphabetical order is the strategy of alphabetic prioritization. Though it is arguable whether this really is an optimization, it is an easy to understand example for an implementation of a prioritization technique.

The strategy is implemented by the class `AlphabeticPrioritization` as depicted in the code snippet in listing 4.12. It extends the abstract class `AbstractComparatorPrioritization` that implements the prioritization by utilizing Java's `Comparator` interface. Therefor, the `AlphabeticPrioritization` class only needs to provide two comparators – one for the test classes and one for the test methods – by implementing the corresponding abstract methods. Here, the returned comparators perform a string comparison on pairs of names of either test classes or methods.

#### Failed-first Prioritization

An example for a strategy that depends on a data store is given by failed-first prioritization. It prioritizes those test methods that failed in the previous run and sorts the test classes by the number of failed test methods in descending order.

This optimization strategy is realized by the class `FailedFirstPriotization`. Similar to the alphabetic prioritization, it extends the abstract class `AbstractComparatorPrioritization` and provides two comparators. The first one orders classes by their number of recently failed test methods and the second one orders the test methods within a class, such that recently failed test methods are prioritized. To this extent, the test history database is accessed via the injected data store.

#### Test-Execution-Time Prioritization

The time it takes to execute a test method is the basis for test-execution-time prioritization. It orders the test methods in a class by the time that was measured in the last run, such that those consuming less time are preferred. Test classes are sorted in ascending order by the accumulated execution times of their test methods.

The strategy's implementation is located in the class `TestExecutionTimePrioritization`. Like the other two prioritization strategies, it extends the abstract class `AbstractComparatorPrioritization`. Further, it utilizes the test history data store to order the test classes and test methods.

### 4.5.4 JUnit Test Framework Adapter

The here presented test framework adapter enables Lazzer to handle tests written with JUnit 4, as depicted in the UML class diagram in figure 4.17. The respective adapter is implemented by the class `JUnit4Adapter` that realizes the `TestFrameworkAdapter` interface.

The method `provideTestDiscovery` assists the framework in finding the test suite. To this end, it uses the predefined `JavaClassTestDiscovery` that searches for all Java classes on the provided test classpath. Then, for each class it queries an injected `JUnit4TestClassFilter` whether the respective class contains JUnit tests; if this is the case, it constructs and returns a corresponding `TestClass`. Finally, all JUnit test classes are returned.

The test execution is invoked through the `runTests` method. The process is realized as a sequential pipeline with five stages. First, `CollectTestClassesStage` creates a list of Java classes that contain tests from the provided test suite. Next, `JUnitCoreSetupStage` instantiates the `JUnitCore` along with listeners that observe the test run, e.g., to the test progress. Then, `JUnitRequestSetupStage` prepares a `Request` that comprises information about what test to run and in what order. With it, `JUnitRunStage` can start the test run. Finally, `TestResultConversionStage` converts the JUnit test run results, which are provided as object of `Result`, to the corresponding framework's representation.

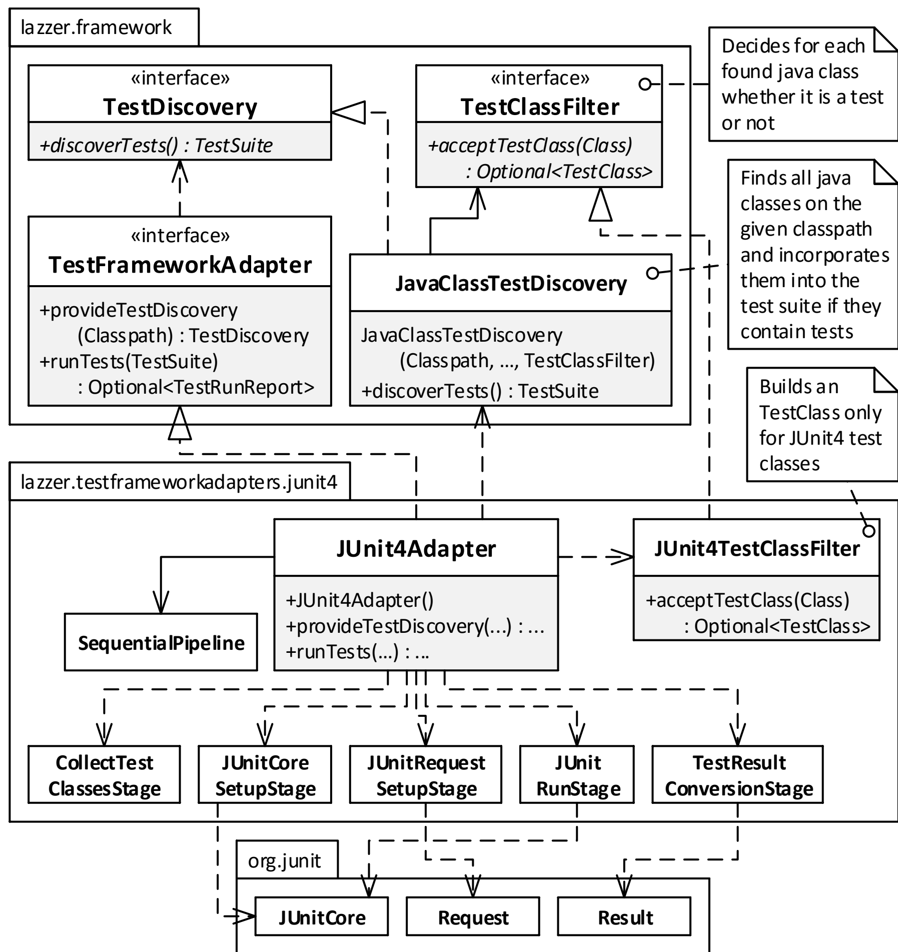



Figure 4.17: UML class diagram depicting the JUnit 4 test framework adapter.

# 5 Evaluation of Lazzer

Contents

5.1 Software Quality Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . 65
5.1.1 Performance Efficiency . . . . . . . . . . . . . . . . . . . . . . . . 65
5.1.2 Usability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
5.1.3 Functionality . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
5.1.4 Maintainability . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
5.1.5 Reliability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
5.1.6 Portability . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
5.1.7 Compatibility . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
5.1.8 Security . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.2 Static Code Analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

Chapter 3 established the concept of an RTO framework and chapter 4 introduced the implementation of the prototype Lazzer. This chapter evaluates Lazzer by providing a software quality analysis in section 5.1 and a static code analysis in section 5.2.

## 5.1 Software Quality Analysis

The software quality analysis bases on the terminology as it is established by the *International Standard Organization* (ISO) and the *International Electrotechnical Commission* (IEC) in the standard ISO/IEC 25010 [ISO11]; the successor of ISO/IEC 9126 [ISO01]. As depicted in figure 5.1, this standard defines eight categories for assessing software quality: *performance efficiency*, *functionality*, *compatibility*, *usability*, *reliability*, *security*, *maintainability*, and *portability*. In the following, Lazzer is evaluated according to these categories.

### 5.1.1 Performance Efficiency

Since regression test prioritization and selection are not efficient, if the analysis takes a long time, the Lazzer framework must not be a slow-down (cf. figure 2.2 on page 8). In contrast to simply running tests without any regression test optimization, Lazzer has to handle some additional tasks. In particular, performing the optimization, loading the corresponding optimization strategies, the required data stores, and the test framework adapter as well as gathering data before the optimization and after the test run.

For a meaningful performance evaluation of the framework, the results must be reproducible and may not be affected by the strategies, the data stores, or the test framework adapter. Therefor, no data stores are used and the strategies as well as the

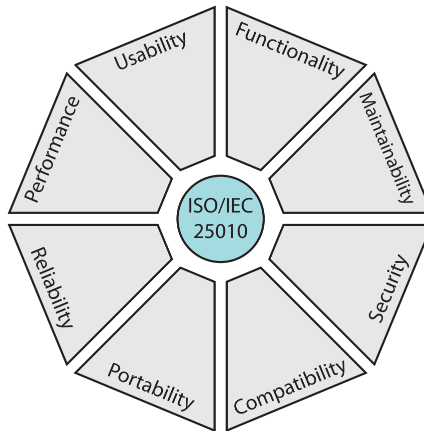



Figure 5.1: Categories of ISO/IEC 25010.

test framework adapter are replaced with corresponding mock objects that realize the their interface but mimic only the minimum of their behavior that is required to run Lazzer.

Avoiding environmental influences, like, e.g., load caused by other processes running on the test machine, is harder. This would require an elaborate preparation of the test machine, and thus, due to time limitations of this thesis, a more simple approach is applied: the evaluation is repeated numerous times; thereby, the impact of occasional environmental influences is reduced.

The evaluation is implemented as a JUnit test. First, mock objects are constructed using the Mockito mocking framework. Then, Lazzer is invoked repeatedly using the framework's API, each time measuring the duration between directly before and after the invocation. The result is written to a file in the *comma separated balues* (CSV) format that allows for an easy processing of the data.

The test was executed on a computer using the IntelliJ IDEA IDE 15, the JDK version 1.8.60, and Windows 10. The machine had an Intel Core i7 4510U 2.60 GHz processor and 8 GB of memory.

For the performance evaluation Lazzer was executed with the mock objects 1000 times. The result is depicted in figure 5.2 as a scatter plot and in figure 5.3 as a clipped box-whisker plot. Cleared from a single outlier, the maximum run time was 121 milliseconds and the minimum was 1 ms. On average, it took 6.0 ms to execute the framework. Half of the runs needed less than or equal to 4 ms and three quarter of all measurements are below or equal to 5 ms. Note that the maximum run time was measured for the first invocation only, representing an extreme outlier of 989 ms, a value being 492.0 interquartile ranges away from the third quartile. Presumably, there are multiple reasons for this; for example, the loading of class files from the hard drive to the memory and the optimizations performed by the Java virtual machine during the first run.

On average, Lazzer performs well considering that test suites targeted by optimization

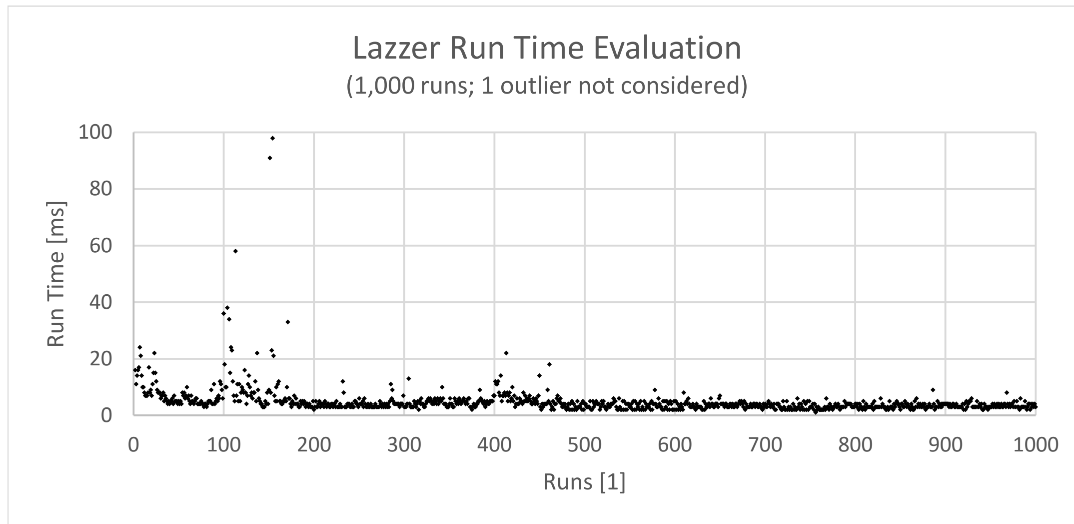



Figure 5.2: Scatter plot showing the run times of 1000 executions of Lazzer. The summary statistics are: a maximum of 989 ms (outlier; not shown), a minimum of 1 ms, an average of 6.0 ms, a lower quartile of 3 ms, a median of 4 ms, and an upper quartile of 5.

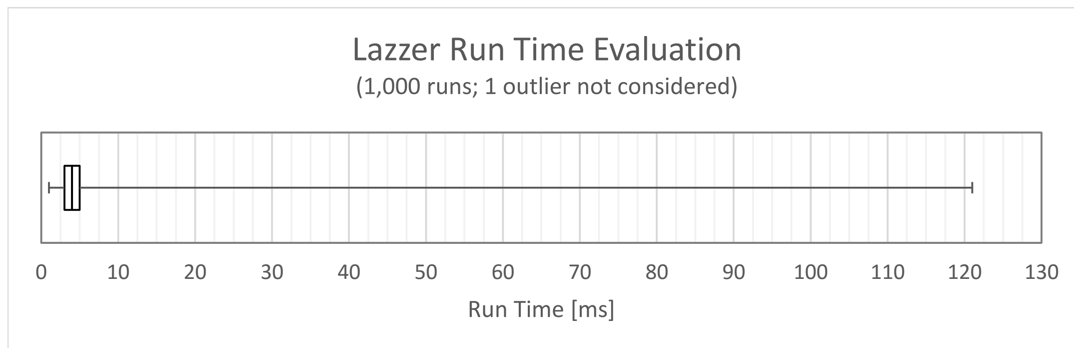



Figure 5.3: The results presented in figure 5.2 depicted as a box-and-whisker plot. The minimum, the first quartile, the median, the third quartile, and the maximum are shown. The data set was adjusted by removing the outlier.

```
1  <project>
2      <!-- ... -->
3
4      <!-- First use Default Maven Repo -->
5      <repository>
6          <id>swc-nexus</id>
7          <name>SWC Nexus for Releases</name>
8          <url>
9              http://buildaddon.swc.rwth-aachen.de/nexus/content
10                 /repositories/plugins/
11         </url>
12 </repository>
13 </project>
```

Listing 5.1: A snippet of a Maven configuration file showing the repository that needs to be added in order to obtain Lazzer.

take minutes, hours, or even days to run. This is also true for the maximum run time measured for the first invocation, which is 989 ms. Thus, the time needed by Lazzer can be considered to be negligible.

### 5.1.2 Usability

Assessing the usability of the Lazzer framework is an important, yet difficult undertaking due to the largely subjective nature of usability. Assumably, one of the best approaches would be to conduct an empirical study among real users. Unfortunately, it would not be easy to gather a representatively sized user group for a not yet publicly available software product like Lazzer. Thus, this thesis will rather pursue an argumentative approach for the usability evaluation.

For now, the framework can be used either as a stand-lone application or as a Maven plug-in. Regardless of which way is used, there are three situations in which the user interacts with Lazzer that have to be considered by the usability evaluation: the framework's integration, its execution, and the inspection of the provided test results.

The integration of the command line interface requires the installation of Lazzer. To this end, the necessary Java archives need be deployed. For the integration of the Maven plug-in, the project's pom.xml needs to be extended. It requires the incorporation of Lazzer's Nexus repository that provides the framework artifacts (see listing 5.1) as well as the addition of the plug-in and its configuration (see listing 4.10 on page 56).

Starting Lazzer from the command line requires to run the respective command and provide the necessary parameters (cf. listing 4.11 on page 58). For the Maven plug-in, the test phase must be executed. Both can be considered to be straightforward; yet, the usage of the Maven provides more comfort than the command line interface.

The inspection of the test results is the same for both, the command line interace and the Maven plug-in. The user is provided with a logged report, similar to the output

generated by common test frameworks, such as JUnit. Thus, it should be fairly easy for the user understand it.

### 5.1.3 Functionality

The prototype of the Lazzer framework realizes all the requirements described in section 3.2. Yet, during the evaluation of the prototype some limitations were discovered and ideas for additional features or improvements came up. The following sections describe these limitations and ideas.

#### Configuration of Data Stores and Strategies

The data stores for git and the test history presented in section 4.5 both required a configuration; the git data store needed the project's directory and the test history data store required access to a database. Similarly, the optimization strategies might need some configuration; for example, it seems reasonable to let the framework user specify the number of recent test runs that should be considered by the failed-first prioritization. So far the framework does not have a mechanism allowing for the individual configuration of data stores and strategies. For the exemplary data stores introduced in section 4.5, their settings were incorporated into the framework's configuration. This solution was chosen due to the limitations of this thesis, but a proper mechanism should be realized in the future.

#### Old JUnit Tests

Lazzer supports JUnit 4 tests only. During the evaluation, Lazzer was applied to some open-source projects, like, e.g., the Apache Commons IO library. When using Lazzer to test the current snapshot of the next release (version 2.5), only a fraction of the tests were executed. An investigation revealed that the cause of this was that the majority of tests was not compliant with JUnit 4 but used the test design mechanism of older versions, e.g., JUnit 3.

This shows that current projects still use old versions of JUnit test implementations. Thus, it seems reasonable that Lazzer should support those as well. To this end, the test framework adapter for JUnit should be enhanced, such that it can handle all kinds of JUnit tests.

#### Parameterized Tests

Sometimes, it can make sense to test a functionality with lots of different input. Since a manual implementation can be tedious and inflexible, some test frameworks, such as JUnit, provide a mechanism named *parameterized tests* that allows for the definition of an input data list and a repeated execution of tests for each element of the list.

Test runs for a software project that contain parameterized tests showed that Lazzer cannot handle this kind of tests. In contrast to regular tests, parameterized tests are

defined in a different way, and thus, are not recognized by the test discovery of the JUnit test framework adapter.

#### Extension API for Processing Test Reports

Processing the test result report created by Lazzer once the test run ended is difficult. For now, the results can only be reviewed through the framework's log. This makes it difficult for other applications to parse the test results. Thus, a future improvement could be to enable any program that starts Lazzer to get the report of the test run after is has finished via the framework's API. Additionally, Lazzer could realize a hot spot, allowing developers to attach one or more reporting components.

#### Prioritization of Methods

Lazzer does allow for an ordering of tests, but in the current state this has a limitation: Tests are ordered first by test classes and second by test methods. Thus, it is not possible to run a test method from one class in between two tests of another class, e.g., `A.x(), B.x(), A.y()`. A future improvement is to overcome this limitation. To this end, two adaptions are required. First, the framework's test suite data structure must be revised such that it supports the definition of an order on method level. Second, the test framework itself must support ordering on method level. For JUnit, this is currently not the case. Alternatively, the test framework adapter could mimic this behavior: for example, by splitting the test class so that new test classes are constructed that contain only test cases that can be executed sequentially; e.g., `A1.x(), B.x(), A2.y()`. Note that the setup and teardown methods need to be delegated as well, thus, requiring the global setup and teardown to be run multiple times.

### 5.1.4 Maintainability

The extent to which a software can be effectively and efficiently improved or adapted to new requirements is described by the term maintainability. This comprises whether a software is designed in a modular and reusable way, whether impacts of changes and causes of deficiencies can easily be understood, whether modification can be made without introducing regressions, and the degree to which a software can be tested.

For the development of Lazzer, modularity and reusability were primary concerns. Thus, Maven modules are used to configure and link components, such as the framework, the data stores, and the optimization strategies. Thereby, adding, replacing or removing a component requires only the modification of a single line in a pom.xml file.

The components interact via interfaces defining the component's API and are separated from their implementations. Further, due to the use of dependency injection, components only depend on interfaces and not their implementations.

The RTO process is implemented as a pipeline, enabling an easy modification of the stages. Last but not least, by using a template engine to generate the test result report, its output format is separated from the source code.

### 5.1.5 Reliability

Lazzer is a first research prototype of an RTO framework that should not be used in production environments. With this in mind, more effort has to be put in testing the framework. However, early adopters might find it reasonable to evaluate benefits of RTO using the Lazzer along with their current test framework.

### 5.1.6 Portability

The evaluation of the portability comprises three aspects: the *adaptability*, the *installability*, and the *replaceability*. Subsequently, each of these is discussed briefly.

The adaptability describes the ability to adapt to new environments, such as new software and hardware. In this regard, Lazzer has no special requirements. It is written in the Java programming language version 8. Therefore, the framework should be able to run on every system having a fully featured Java virtual machine; as of November 2015, this applies to all common computers running an operating system that is supported by the Java standard edition, such as Windows, Mac OS X, and Linux [Oraa]. Note that the adaptability of data stores and strategies can differ, since they may require special software or hardware.

Installability is related to the question, how well a software system can be installed. For now, there are two ways to set up Lazzer: The stand-alone application requires only the deployment of the respective Java archives. However, the installation of data stores and optimization strategies can require additional effort, such as setting up a database or other software. The Maven plug-in can be installed by extending the project's configuration file by a couple of lines. Maven will then download the respective artifacts from a Nexus repository and deploy them. Note that the data stores and optimization strategies again may require additional effort. Considering the target audience of Lazzer, i.e., software developers, it can be concluded that the installability of the framework itself is reasonable.

A software product performs well regarding replaceability, if it can be substituted by a product serving the same purpose. The reason for the development of Lazzer is the lack of a framework for RTO. Thus, there is no direct substitute. However, a user can always do the regression testing without the optimization by replacing Lazzer with any other test framework.

### 5.1.7 Compatibility

The ability to co-exist and interoperate with other systems is the concern of compatibility. Lazzer's design focused on realizing this as good as possible, which becomes apparent when looking at the framework's hot spots: Lazzer can be used with a broad variety of xUnit test frameworks and can be integrated into probably any environment by developing respective client adapters.

Room for improvement has the reporting of Lazzer. Although, the framework internally uses a template engine to generate the test result output, the substitution

of the template file is more cumbersome than necessary. Furthermore, Lazzer currently writes the output always to the log, which makes it difficult for automated processing. Therefor, corresponding configuration options that allow for the specification of a template and an output location would be desirable future improvements.

### 5.1.8 Security

For the development of Lazzer, the security was not a concern. First, the framework is a research prototype that is not designed for usage in production environments. Second, the framework is meant to be run as a local application, i.e., it neither requires nor provides any network services, and it does not need elevated rights for its execution. Thus, it is considered unlikely that Lazzer is used to compromise a system's integrity. Third, the framework does not deal with very sensitive information that must be protected from unauthorized access; it just processes given software tests to obtain a test result.

## 5.2 Static Code Analysis

SonarQube is a popular open-source platform for the assessment of software quality by using various methods of static code analysis. When integrated into the continuous integration process of a project's development snapshots, SonarQube automatically analyzes every new version and allows for constantly monitoring various metrics via SonarQube's web-based dashboard. The metrics cover multiple aspects of a program, such as its *architecture*, its *documentation*, its *unit testing status*, its *complexity*, and its *compliance with predefined rules*. Subsequently, Lazzer's latest results for these aspects are presented.

The architecture of Lazzer's core is comprised of 83 classes (not including the unit tests), 303 functions, and 2.201 *lines of code* (LOC). Thus, Lazzer is a slim framework. More than 76% of the public API is documented with javadoc, which suggests a fairly good documentation. The analysis of the JUnit test results yields a *code coverage* of 15.6%, which is a unfortunate as it means that tests are missing. The highest coverage (over 60%) is achieved for the classes related to the representation of tests.

SonarQube's complexity analysis is based on the *cyclomatic complexity* (CC) metric presented by McCabe [Mcc76]. It is a quantitative measure for the number of linearly independent paths through a program or module. The lowest value that can be achieved is one, i.e., there exists only a single linear path, and a higher values correspond to an increasing complexity. McCabe suggests modules with a CC of ten or higher need to be restructured [Mcc76, p. 314]. The functions of Lazzer's framework core have a CC of 1.4 and the classes have a CC of 5.1.

For the coding rules analysis, SonarQube's default profile "Sonar way" was used. It inspects the source code for a variety of flaws, such as violations of coding conventions and the usage of anti-patterns, and classifies them by severity. The result for Lazzer indicates no blocker or critical flaws but four major violations spread over three classes;

one is related to throwing a raw exception type and the others are caused by unused private fields or method parameters.

# 6 Conclusion

Contents

6.1 Summary . . . . . . . . . . 75
6.2 Future Work . . . . . . . . . . 76
6.2.1 Enhancing the Prototype . . . . . . . . . . 76
6.2.2 Future Research . . . . . . . . . . 77

In this last chapter, the contributions of this thesis are recapitulated in section 6.1. Moreover, an outlook on possible improvements of the prototype as well as suggestions for further research on regression test optimization are given in section 6.2.

## 6.1 Summary

This thesis made some contributions to the field of regression test optimization. This includes a literature study of the state of research, tools, and practice of RTO. Further, a concept and a prototype for an RTO framework as well as its evaluation were presented. Subsequently, each of these contributions is briefly recapped.

In chapter 2, an overview of the state of RTO was given. This comprises brief presentations of various optimization techniques and empirical studies that have been presented in research papers in the last decades. Then, the landscape of tools that implement RTO techniques was studied; finding that of the few tools that have been developed many are discontinued, leaving only three that are still available. Further, the application of RTO in industry was investigated. In short, companies, such as Microsoft, Google, and Siemens, identified a need for RTO in order to cope with their growing amount of regression tests. However, research results have not made their way to practice. The chapter concluded with the finding that implementing optimization algorithms, which is challenging by itself, involves a lot of technical difficulties. In particular, the difficulties are the integration of RTO techniques in existing testing processes, gathering the data required by the techniques to perform their optimization, and running the optimized test suite. This reasoned a need for a platform for regression test optimization that eases the integration and implementation of optimization techniques.

The framework concept presented in chapter 3, is a first step towards such an RTO platform. First, the framework's objectives were defined to tackle the mentioned difficulties, i.e., the complex infrastructure setup that makes the integration and implementation of optimization techniques so challenging. With this in mind, a list of requirements for the framework was established, pointing out the different needs of

framework users and developers. Thereafter, the coarse-grained framework's architecture with its hotspots was presented. Finally, the details of the API, the RTO process, and the framework's handling of dependencies on extensions were elaborated.

Lazzer, a prototype of the RTO framework, was introduced in chapter 4. To this extent, its API that allows for the framework's usage and extension was described and code skeletons were given to jump-start implementations of own extensions. Then, the implementation of the RTO process was outlined. Finally, exemplary implementations of the extensions were presented: some clients, optimization strategies, data stores, and a test framework adapter for JUnit. Thereby, making Lazzer a gray-box framework.

An evaluation of the prototype was given in chapter 5. To this end, the framework's software quality was assessed by analyzing it according to the ISO/IEC 25010 standard and by inspecting the results of a static code analysis. The findings revealed a couple of limitations as well as ideas for future work.

## 6.2 Future Work

The contributions of this thesis are a first step towards an RTO platform. In this section, suggestions for future work on the prototype and for more research are made.

### 6.2.1 Enhancing the Prototype

While the prototype features all functions demanded by the requirements, it leaves potential for further improvements. Some ideas for future enhancements of the prototype are outlined in the following.

#### Overcoming Current Limitations

The evaluation of the framework prototype in chapter 5 uncovered some functional limitations that should be addressed by future development. For instance, an individual configuration of each data store and optimization strategy should be supported. Further, the JUnit test framework adapter should be extended to discover older versions of JUnit test implementations as well as support parameterized tests. Moreover, a new hot-spot, allowing for one or more custom reporting components, should be introduced. Last but not least, prioritization should be enabled on method level.

#### Parallel Testing

The efficiency of a test run can further be improved by parallelizing it. There are test frameworks, including JUnit, that provide mechanisms for parallel test runs. However, this is a feature that is not yet supported by Lazzer. Changing this appears to be a more complex undertaking, since there is not a known best solution to transfer the concept of test prioritization from the sequential to the parallel world.

#### Enhanced Framework Configuration

Lazzer's current configuration options leave room for future enhancements. The subsequently proposed settings are inspired by parameters of the Maven plug-in Surefire that takes care of starting test runs in Maven builds [Sur]: The Java `assert` statement can be used in programs and tests to ensure that a given condition holds; however, those assertions are disabled by default due to performance reasons [Orac]. Thus, it is preferable to have an option to enable assertions for the test run. Further, environment variables and Java properties are sometimes used to pass information to tests; therefore, it seems reasonable to have options to set those variables and properties for the test run. Some test frameworks, such as JUnit, have a mechanism for assigning tests to a group. An option allowing for the inclusion or exclusion of tests that belong to specific groups gives the user more control on defining the test suite that is passed to Lazzer. Another option could specify whether a test run should be stopped or continued once a failure occurred; the later is currently the case.

#### Testing Lazzer

The static code analysis in section 5.2 showed that the existing unit tests for the framework's core achieve a test coverage of only 15.6% and that most of them focus on the data structure that represents test suites, classes, and methods. Therefore, a goal for the future must be, to increase the test effectiveness and to incorporate more integration tests.

### 6.2.2 Future Research

By providing a concept for an RTO framework as well as a corresponding prototype, this thesis laid the foundation for future contributions to the field of regression test optimization. Subsequently, a few suggestions for research topics are briefly described.

#### Implementation and Evaluation of Optimization Techniques

The RTO framework's objective is to provide a foundation for the implementation and integration of optimization techniques. Therefor, the next steps should be to realize some of the techniques presented in research papers (cf. section 2.2), to integrate the framework into the testing process of some real-world industry projects, and to conduct empirical studies on the optimizations' effectiveness.

#### Supporting other Programming Languages

The RTO framework concept presented in this thesis can be applied to a diversity of programming languages. It requires that the language's test framework complies with the xUnit architecture. Fortunately, such frameworks already exist for a lot of programming languages, like, e.g., C++, PHP, and JavaScript. Thus, only corresponding test framework adapters have to be implemented.

### The Problem of Choosing the Best Optimization Technique

The literature research presented in chapter 2 showed that there are numerous RTO techniques. Assuming that there will be implementations for some of them in the future, the choice of the optimal combination of one or more optimization techniques becomes a problem. This appears to be an optimization problem itself. Thus, having recommendations for common scenarios would be desirable.

### Recommendation for Test Suite Reduction

Applying prioritization and selection techniques is not the only way to deal with growing test suites. Moreover, it should be continuously questioned whether every test still has a right to exist. With evolving test suites, some of their tests can become obsolete, e.g., when tests are redundant or the corresponding features are removed. Consequently, the tests should be removed from the test suite in order to save resources.

When integrated in the testing process, the RTO framework can access and track a variety of information about the test suite and the system under test. It seems reasonable to investigate whether this can be used to identify obsolete tests and give recommendations for their removal.

### Test Optimization as a Service

In its current design, the RTO framework is an application that only becomes active when an optimized test run should be performed, e.g., as part of a build process. Yet, it also makes sense to run the framework on dedicated testing machines as a service. It could provide an API that allows clients to request an optimized regression test run for a given test suite and system under test, which is then executed on the testing machines. The potential benefit of such a service is that optimized regression testing on dedicated machines that outperform regular work stations can be made available to many developer or teams within an organization.

# Bibliography


[ABB+14] Bram Adams, Stephany Bellomo, Christian Bird, Foutse Khomh, and Kim Moir. Website of the 2nd International Workshop on Release Engineering 2014. http://releng.polymtl.ca/RELENG2014/html/; Online, last accessed 2015-10-29, 2014.

[AC91] David Applegate and William Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on computing*, 3(2):149–156, 1991.

[AHKL93] H. Agrawal, J.R. Horgan, E.W. Krauser, and S.a. London. Incremental regression testing. *1993 Conference on Software Maintenance*, 1993.

[Bec89] Kent Beck. Simple Smalltalk Testing: With Patterns, 1989.

[BEG07] Fevzi Belli, Mubariz Eminov, and Nida Gökçe. A Fuzzy Clustering Approach and Case Study 1 Introduction : Motivation and Related Work. In *Dependable Computing*, pages 95–110. 2007.

[Blo08] Joshua Bloch. *Effective Java,*. Addison-Wesley Professional, 2nd editio edition, 2008.

[BM07] Renée C. Bryce and Atif M. Memon. Test suite prioritization by interaction coverage. *Workshop on Domain specific approaches to software test automation in conjunction with the 6th ESEC/FSE joint meeting - DOSTA '07*, pages 1–7, 2007.

[BMSS11] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. Regression test selection techniques: A survey. *Informatica (Ljubljana)*, 35(3):289–321, 2011.

[BRO13] Marcel Böhme, Abhik Roychoudhury, and Bruno C D S Oliveira. Regression Testing of Evolving Programs. *Advances in Computers*, 89:53–88, 2013.

[BSM11] Renée C. Bryce, Sreedevi Sampath, and Atif M. Memon. Developing a single model and test prioritization strategies for event-driven software. *IEEE Transactions on Software Engineering*, 37(1):48–64, 2011.

[Cle] Cleanscape Software International. Cleanscape TestWise. http://stellar.cleanscape.net/products/testwise/index.html; Online, last accessed 2015-10-23.

[CRK10] Alexander P Conrad, Robert S Roos, and Gregory M. Kapfhammer. Empirically studying the role of selection operators duringsearch-based test suite prioritization. *Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10*, page 1373, 2010.

[Dij82] Edsger W Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982.

[DMTR08] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. *European Conference on Software Engineering / Foundations of Software Engineering*, page 71, 2008.

[DRK04] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Empirical Studies of Test Case Prioritization in a JUnit Testing Environment Empirical Studies of Test Case Prioritization in a JUnit Testing Environment. 2004.

[EMR00] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing Test Cases For Regression Testing. *International Symposium of Software Testing and Analysis*, pages 102–112, 2000.

[EMR02] Sebastian Elbaum, A.G. Malishevsky, and Gregg Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, 2002.

[FBH+10] Roberto S Silva Filho, Christof J. Budnik, William M. Hasling, Monica McKenna, and Rajesh Subramanyan. Supporting concern-based regression testing and prioritization in a model-driven environment. *Proceedings - International Computer Software and Applications Conference*, pages 323–328, 2010.

[FKAP09] Yalda Fazlalizadeh, Alireza Khalilian, Mohammad Abdollahi Azgomi, and Saeed Parsa. Incorporating Historical Test Case Performance Data and Resource Constraints into Test Case Prioritization. *Lecture Notes in Computer Science*, 5668:43–57, 2009.

[Fow04] Martin Fowler. Website of Martin Fowler: Article on Dependency Injection. `http://martinfowler.com/articles/injection.html#FormsOfDependencyInjection`; Online, last accessed 2015-11-18, 2004.

[Fow06] Martin Fowler. Webiste of Martin Fowler: Article on Xunit. `http://www.martinfowler.com/bliki/Xunit.html`; Online, last accessed 2015-11-16, 2006.

[FPR02] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. *The UML Profile for Framework Architecture.* 2002.

[Fre] Website of the FreeMarker Java Template Engine. `http://freemarker.incubator.apache.org/`; Online, last accessed 2015-12-01.

[FSJ99] Mohamed E Fayad, Douglas C Schmidt, and Ralph E Johnson. *Building Application Frameworks: Object-oriented Foundations of Framework Design*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[GBHS10] Zhongxian Gu, Earl T. Barr, David J. Hamilton, and Zhendong Su. Has the bug really been fixed? *2010 ACM/IEEE 32nd International Conference on Software Engineering*, 1:55–64, 2010.

[GEM15] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Ekstazi : Lightweight Test Selection. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 713–716, 2015.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GHK$^{+}$01] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porters, and Gregg Rothermel. An Empirical Study of Regression Test Selection Techniques. *ACM Transactions on Software Engineering and Methodology*, 10(2):184–208, 2001.

[GIP11] Pooja Gupta, Mark Ivey, and John Penix. Testing at the speed and scale of Google. `http://google-engtools.blogspot.de/2011/06/testing-at-speed-and-scale-of-google.html`, 2011.

[Gli] Milos Gligoric. Website of Ekstazi. `http://www.ekstazi.org/`; Online, last accessed 2015-10-26.

[GNLM14] Milos Gligoric, Stas Negara, Owolabi Legunsen, and Darko Marinov. An Empirical Evaluation and Comparison of Manual and Automated Test Selection. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, pages 361–372, 2014.

[Goo] Google. Website of the Guice Project on GitHub. `https://github.com/google/guice`; Online, last accessed 2015-12-01.

[Har12] Jean Hartmann. 30 Years of Regression Testing : Past, Present and Future. In *PNSQC 2012 Proceedings*, pages 1–8, 2012.

[HFGO94] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. *Proceedings of 16th International Conference on Software Engineering*, pages 191–200, 1994.

[HGCM15] Kim Herzig, Michaela Greiler, Jacek Czerwonka, and Brendan Murphy. The Art of Testing Less without Sacrificing Quality. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 483–493, 2015.

[HS88] Mary Jean Harrold and M.L. Souffa. An Incremental Approach to Unit Testing during Maintenance. *Proceedings. Conference on Software Maintenance, 1988.*, pages 362–367, 1988.

[HSVV12] Peter Haberl, Prof. Dr. Andreas Spillner, Prof. Dr. Karin Vosseberg, and Prof. Dr. Karin Vosseberg. Umfrage 2011: Softwaretest in der Praxis. Technical report, 2012.

[HZXS08] Shan-Shan Hou, Lu Zhang, Tao Xie, and Jia-Su Sun. Quota-constrained test-case prioritization for regression testing of service-centric systems. *2008 IEEE International Conference on Software Maintenance*, pages 257–266, 2008.

[Inf] Website of Infinitest. `https://infinitest.github.io/`; Online, last accessed 2015-10-23.

[ISO01] ISO. ISO/IEC 9126-1:2001, 2001.

[ISO11] ISO. ISO/IEC 25010:2011, 2011.

[JF88] Ralph E. Johnson and Brian Foote. Designing Reuseable Classes. *Journal of Object-Oriented Programming*, 1988.

[JUna] JUnit API Documentation. `http://junit.org/apidocs/`; Online, last accessed 2015-11-20.

[JUnb] Website of the Java Test Framework JUnit. `http://junit.org/`; Online, last accessed 2015-11-20.

[JYC11] Wang Jun, Zhuang Yan, and Jianyun Chen. Test Case Prioritization Technique Based on Genetic Algorithm. *2011 International Conference on Internet Computing and Information Services*, (2):173–175, 2011.

[JZCT09] Bo Jiang, Zhenyu Zhang, W. K. Chan, and T. H. Tse. Adaptive Random Test Case Prioritization. *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 233–244, 2009.

[KKT07] Bogdan Korel, George Koutsogiannakis, and Luay H. Tahat. Model-based test prioritization heuristic methods and their evaluation. *Proceedings of the 3rd international workshop on Advances in model-based testing - A-MOST '07*, pages 34–43, 2007.

[KP02] Jung-Min 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. ICSE 2002*, pages 119–129, 2002.

[KS08] Ramasamy Krishnamoorthi and Arul Mary Sahaaya. Incorporating varying requirement priorities and costs in test case prioritization for new and regression testing. *2008 International Conference on Computing, Communication and Networking*, (2):1–9, 2008.

[KS09] R Krishnamoorthi and S A Sahaaya Arul Mary. Factor Oriented Requirement Coverage Based System Test Case Prioritization of New and Regression Test Cases. *Inf. Softw. Technol.*, 51(4):799–808, apr 2009.

[KTH05] B. Korel, L.H. Tahat, and M. Harman. Test prioritization using system models. *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 559–568, 2005.

[LH09] Jun-Wei Lin and Chin-Yu Huang. Analysis of test suite reduction with enhanced tie-breaking techniques. *Information and Software Technology*, 51(4):679–690, 2009.

[Lig02] Peter Liggesmeyer. *Software-Qualität*. Spektrum Akademischer Verlag, Heidelberg, 2002.

[Liq] Liquibase. `http://www.liquibase.org/`; Online, last accessed 2015-11-20.

[Mava] Apache Maven Project. `https://maven.apache.org/`; Online, last accessed 2015-07-20.

[Mavb] Apache Maven System Requirements. `https://maven.apache.org/download.cgi#System_Requirements`; Online, last accessed 2015-07-24.

[Mavc] IntelliJ IDEA Maven Integration. `https://www.jetbrains.com/idea/help/maven.html`; Online, last accessed 2015-07-23.

[Mavd] Introduction to the POM. `https://maven.apache.org/guides/introduction/introduction-to-the-pom.html`; Online, last accessed 2015-07-23.

[Mave] M2Eclipse (Maven Eclipse Plug-in). `http://www.eclipse.org/m2e/`; Online, last accessed 2015-07-23.

[Mavf] Maven Checkstyle Plugin. `http://maven.apache.org/plugins-archives/maven-checkstyle-plugin-2.15/`; Online, last accessed 2015-07-23.

[Mavg] Maven: Configuring Reports. https://maven.apache.org/plugins/maven-site-plugin/examples/configuring-reports.html; Online, last accessed 2015-07-23.

[Mavh] Maven for PHP. http://www.php-maven.org/; Online, last accessed 2015-07-24.

[Mavi] Maven Lifecycle Introduction. https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html; Online, last accessed 2015-07-24.

[Mavj] Maven POM Reference. https://maven.apache.org/pom.html; Online, last accessed 2015-07-23.

[Mavk] Maven Surefire Plugin. http://maven.apache.org/surefire/maven-surefire-plugin/index.html; Online, last accessed 2015-07-23.

[Mavl] Native ARchive plugin for Maven. https://github.com/maven-nar/nar-maven-plugin; Online, last accessed 2015-07-23.

[Mavm] POM Reference: Inheritance. http://maven.apache.org/pom.html#Inheritance; Online, last accessed 2015-08-20.

[MB15] Joel Montvelisky and Lalitkumar Bhamare. State of Testing 2015. 2015.

[Mcc76] Thomas J Mccabe. A Complexi-ty. (4):308–320, 1976.

[MCTM10] Lijun Mei, W.K. Chan, T.H. Tse, and Robert G. Merkel. XML-manipulating test case prioritization for XML-manipulating services. *Journal of Systems and Software*, 84(4):603 – 619, 2010.

[MdCdF+10] Camila Loiola Brito Maia, Rafael Augusto Ferreira do Carmo, Fabrício Gomes de Freitas, Gustavo Augusto Lima de Campos, and Jerffeson Teixeira de Souza. Automated Test Case Prioritization with Reactive GRASP. *Advances in Software Engineering*, 2010:1–18, 2010.

[Mes07] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Pearson Education, 2007.

[MSGB14] Samaila Musa, Abu Bakar Md Sultan, Abdul Azim Bin Abd Ghani, and Salmi Baharom. A Regression Test Case Selection and Prioritization for Object- Oriented Programs using Dependency Graph and Genetic Algorithm. *International Journal of Engineering And Science*, 4(7):54–64, 2014.

[MZCT09] Lijun Mei, Zhenyu Zhang, W. K Chan, and T. H Tse. Test case prioritization for regression testing of service-oriented business applications. *Proceedings of the 18th international conference on World wide web - WWW '09*, page 901, 2009.

[Oraa] Oracle. Java Download Website. `http://www.oracle.com/technetwork/java/javase/downloads/index.html`; Online, last accessed 2015-11-29.

[Orab] Oracle. Oracle Technology Network Website on Java. `http://www.oracle.com/technetwork/java`; Online, last accessed 2015-12-01.

[Orac] Oracle. Website on Java Assertions. `http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html`; Online, last accessed 2015-11-29.

[Ors14] Alessandro Orso. Alex Orso - Software. `http://www.cc.gatech.edu/home/orso/software.html`; Online, last accessed 2015-10-23, 2014.

[OSH04] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. *ACM SIGSOFT Software Engineering Notes*, 29(6):241, 2004.

[PRB08] Hyuncheol Park, Hoyeon Ryu, and Jongmoon Baik. Historical Value-Based Approach for Cost-Cognizant Test Case Prioritization to Improve the Effectiveness of Regression Testing. *2008 Second International Conference on Secure System Integration and Reliability Improvement*, pages 39–46, 2008.

[Pre97] Wolfgang Pree. *Komponentenbasierte Softwareentwicklung mit Frameworks*. 1997.

[QOSa] QOS.ch. Website of the logback Java Logging Framework. `http://logback.qos.ch/`; Online, last accessed 2015-12-01.

[QOSb] QOS.ch. Website of the slf4j Java Logging Facade. `http://www.slf4j.org/`; Online, last accessed 2015-12-01.

[RH97] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, 1997.

[RUCH99] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test Case Prioritization: an Empirical Study. In *Proceedings of the IEEE International Conference on Software Maintenance*, page 179, Oxford, UK, 1999.

[RUH01] Gregg Rothermel, Roland H. Untch, and Mary Jean Harrold. Prioritizing Test Cases For Regression Testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001.

[Sin12] Yogesh Singh. Systematic Literature Review on Regression Test Prioritization Techniques Difference between Literature Review and Systematic Literature. 36:379–408, 2012.

[SLS06] Andreas Spillner, Tilo Linz, and Hans Schaefer. *Software Testing Foundations*. dpunkt.verlang GmbH, Heidelberg, Germany, 1s edition edition, 2006.

[Spa03] Gene Spafford. Website of Spyder. `http://spaf.cerias.purdue.edu/Students/spyder.html`; Online, last accessed 2015-10-23, 2003.

[SQS11] SQS Software Quality Systems AG; Cologne; Germany. Siemens PLM Software. Technical report, 2011.

[SSMS] Janmejay Singh, Pavan K Sudarshan, Rajesh M, and Sachin Sudheendra. Website of the Test Load Balancer. `http://test-load-balancer.github.io/`; Online, last accessed 2015-10-23.

[SSS11] Sangeeta Sabharwal, Ritu Sibal, and Chayanika Sharma. Applying Genetic Algorithm for Prioritization of Test Case Scenarios Derived from UML Diagrams. *International Journal of Computer Science Issues*, 8(2):433–444, 2011.

[ST02] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. *ACM SIGSOFT Software Engineering Notes*, 27(4):97, 2002.

[Sur] Website of Surefire Maven Plug-in. `http://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html`; Online, last accessed 2015-12-03.

[SW05] Hema Srikanth and Laurie Williams. On the economics of requirements-based test case prioritization. *ACM SIGSOFT Software Engineering Notes*, 30(4):1, 2005.

[SWO05] Hema Srikanth, Laune Williams, and Jason Osborne. System test case prioritization of new and regression test cases. *2005 International Symposium on Empirical Software Engineering, ISESE 2005*, 00(c):64–73, 2005.

[Tes] Website of the Java Test Framework TestNG. `http://testng.org`; Online, last accessed 2015-12-01.

[TIO15] TIOBE Software BV. TIOBE Index Octber 2014. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`; Online, last accessed 2015-10-11, 2015.

[VSMM15] Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, and Vahab Mirrokni. Automated Decomposition of Build Targets. *Proceedings of the 37th International Conference on Software Engineering*, (1):123–133, 2015.

[WHLA97] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. *Proceedings The Eighth International Symposium on Software Reliability Engineering*, pages 264–274, 1997.

[WSKR06] K R Walcott, Mary Lou Soffa, G M Kapfhammer, and R S Roos. Time-Aware test suite prioritization. *2006 International Symposium on Software Testing and Analysis, ISSTA 2006*, pages 1–12, 2006.

[YH10] S. Yoo and M. Harman. Regression Testing Minimisation, Selection and Prioritisation: A Survey. *Software Testing, Verification and Reliability*, pages 1–30, 2010.

[YYZ+11] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How Do Fixes Become Bugs? *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, pages 26–36, 2011.