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

Bachelor Thesis

Enhancing Lazzer for Metric-based RTO Strategies

Erweiterung von Lazzer um metrikbasierte RTO Strategien

presented by

Jan-Hendrik Telke

Aachen, September 13, 2016

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 Bachelor's thesis entitled

Enhancing Lazzer for Metric-based RTO Strategies

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 Bachelorarbeit mit dem Titel

Enhancing Lazzer for Metric-based RTO Strategies

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, September 13, 2016 (Jan-Hendrik Telke)

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, September 13, 2016 (Jan-Hendrik Telke)

Acknowledgment

I would like to thank Prof. Dr. rer. nat. Horst Lichter for the opportunity of doing my thesis at the Software Construction Research Group and Prof. Dr. rer. nat. Bernhard Rumpe for agreeing to review my thesis. Also, I would like express my deep gratitude towards Andrej Dyck for supervising this thesis. Through him and his willingness to share his domain knowledge and engage in discussions, many of the ideas reflected on this work arose, not to mention the valuable feedback that he constantly provided. Thanks to him I was immersed in this topic and I was able to strengthen my knowledge in this field. Lastly, I thank my family and friends for the constant support they have given me along my studies. Further, special thanks to my girlfriend for reviewing this thesis countless times.

Jan-Hendrik Telke

Abstract

Regression testing is common practice to reduce bugs in software projects. Even though test cases are often automated, test execution takes a significant amount of time. Increasingly so on huge software projects or if testing needs to be performed for different configurations with regard to architecture, processor type, or operating system. Long test executions can delay development due to a longer feedback loop for developers. Additionally, if tests are skipped due to time limitations, code faults may propagate and the cost to fix them increases over time.

Regression test optimization aims to optimize test execution. While test selection reduces execution time by finding a subset of change-relevant test cases, test prioritization orders test cases corresponding to a weight function, such that important tests are executed first. Especially test selection there involves risks, because faults may stay undetected and that time, the cost to fix it increases.

In literature, many algorithms have been presented. Recent research yield promising results with multi-objective, search-based, and self-adaptive optimization algorithms. For example, in 2015, researchers at Microsoft presented an approach, named THEO, for test selection using metric measurements. Based on the test history, THEO computes some metrics, e.g., the detection rate , and based on those cost values are derived for each test case.

This thesis presents the requirements to implement a metric based strategy, and proposes how to realise them. Furthermore, metric-based strategies are implemented, one that is very similar to THEO and another one that extends the idea by increased use of parameters. Also a first step toward runtime analysis is made.

Contents

List of Tables

List of Figures

List of Source Codes

1 Introduction

Contents

Software testing is increasingly used in today's development processes. Its main goal is to assure high software quality, specifically by exposing bugs and thus removing them before release. Regression testing is a testing approach, where a test suit is run on every change to detect changes in functionality, and consequently, increase confidence in the software. However, software testing is taking up an increasing amount of time, especially on huge software projects.

Test automation is famous technique to reduce testing effort by discovering, running and evaluation tests. This drastically reduces testing effort. Sophisticated testing framework exist to help developers with test automation. For instance, the JUnit framework for Java is widely known. Indeed, it has become so successful that it has been ported to many other languages. Those frameworks are called the xUnit family [Fow06].

Even though test execution can be automated, it takes a significant amount of time. One reason might be that, when code changes, new test cases are added, while the old remain in the test suite. Consequently the test suite keeps growing and takes longer to be executed [CRK10]. This effect even increases with respect to huge software projects and when testing needs to be performed for a variety of different configurations with regard to architecture, processor type, or operating system [Her+15]. Long test executions can delay development due to a longer feedback loop for developers. Additionally, if test execution is skipped, code faults may propagate and the costs to fix them increase over time [Ors+01].

Projects often are on a tight schedule for varying reasons, e.g., that a security relevant patch needs to be released quickly. In those cases it is not feasible to run a test suite, that may need multiple hours to finish. Developers often decide to run a manual selection of test cases. They select those, which are in their experience or intuition relevant for the code change they made [Gli]. Manual selection is prone to human error. Often test cases fail, which are not linked to the change at a first glance. In the worst case scenario the overseen bugs propagate and fixing them becomes a complex process.

Continuous delivery is another frequently used technique. It describes the process, that after a commit to repository is made, software is automatically built, compiled and deployed to the server. Usually test suites are run prior to deployment, Regression test optimisation (RTO) could allow a shorter time between a code change and deployment. Shortening the time between code change and deployment can also decrease the time to

discover problems [EMR02].

RTO aims to optimise test execution, e.g., by reducing execution time. Multiple different strategies of RTO can be found in the literature. Since the problem of determining test cases that are relevant for a code change is NP-hard, many strategies try to use an heuristic approach [AC91]. While some analyse coverage data or evaluate data gathered in previous test runs, others use self-learning structures like genetic algorithms [Jyo14; SSS11; BM07]. Even though there is a multitude of strategies, the common goal of most strategies is to reduce the time for test execution.

The following two approaches are used for that purpose. Test prioritisation on one hand, is the method of reordering the execution of test cases. This may not reduce the overall execution time, but it is advantageous to run more important test cases earlier, because if those fail, developers can react faster. On the other hand test case selection, which completely removes some test cases from the test run, and obviously reduces execution time by doing so [Jyo14].

THEO is a self-adaptive strategy. It was recently published by Kim Herzig et al [Her+15]. This strategy calculates a cost function to decide whether it is more cost efficient to run a test case or skip it. The goal of this thesis is to implement a metric-based strategy, which is based on the ideas of THEO.

The Lazzer framework, which was developed in 2015 at the RWTH Research Group Software Construction, will support the implementation [Ple15]. It provides basic functionality for regression test optimisation, including support for selection and prioritising test cases. Lazzer's architecture is modular, each of its core components can be exchanged. For that reason multiple programming languages, test frameworks and data storages can be supported.

1.1 Contributions

The goal of this thesis will be to conceptualise and implement a multi-objective regression test optimisation strategy. Thereby relying on the Lazzer framework as a framework for test suite manipulation.

1. **Separate data evaluation and storage:** Currently Lazzer does not separate data evaluation and storage. In the course of this thesis the data store is split up in analysis engines and data stores, to prepare the implementation of a self-adaptive strategy.

2. **Implement support for a graph database:** Relational databases show to be an ineffective solution to store test history data, hence support for a graph database is implemented.

3. **Strategy parameterisation:** THEO and other strategies need parameterisation of the strategy. By doing so the knowledge developers may have about their test suite can be used to improve optimisation and make it more project specific. Consequently parameterisation is requirement and hence implemented.

4. **Implement metric-based strategies:** Finally three metric-based optimisation strategies are implemented, to prove that the updated Lazzer is capable of running such a strategy.

1.2 Structure of this Thesis

Chapter 2 gives an introduction to Lazzer and THEO. It also briefly gives background knowledge about regression test optimisation. Next, Chapter 3 the requirements for a metric-based strategy are covered and introduces concepts for an updated Lazzer. The realisation is explained in chapter 4, giving detailed insight on the refactoring of Lazzer and implementation of the THEO strategy. Further, Chapter 5 evaluates the implemented strategies and gives insights on the performance. Finally, chapter 6 summarises this thesis and hints at improvements for the future.

2 Related Work

Contents

This chapter gives a introduction to related work. First a introduction on regression testing is given, then the THEO strategy is introduced as an example for a metric-based strategy. Further Lazzer as a important basis for this work is covered.

2.1 Regression Test Optimisation

Software regression testing is widely used technique to discover code faults introduced with a code change, such that those faults can be fixed. Since solely manual testing is not feasible, even on medium sized software projects, automatic testing has become the standard. Considering large and complex software systems, the need for extensive testing suites increases. The downside becomes obvious if one looks at execution times for those test suites. For instance, running all available tests of the Apache Spark project needs approximately 35 minutes [Spa]. While this project is rather small, execution times of multiple days a not unusual [Do+10]. Ideally tests should be run with every code change a developer performs. In practice this would massively delay development and hence is not feasible.

Software tests, depending on the project, may also have to be run on multiple configurations [Her+15], thus consuming even more computation time. All this has a measurable impact on the effectiveness of a development process. While on one hand software quality needs to be held high, software release cycles shorten, especially with practice like continuous integration. Additionally, security fixes are to be released as fast as possible. If fixing takes up to much time or even introduces new weak points, the results can be devastating.

The previous section states that regression testing is a task that has to be conducted with great care, but also suffers from an increasing pressure regarding time. Many, if not most, developers resort to manual test selection [Gli]. This term describes the common practice to run hand picked tests. Moreover, developers pick test cases which regarding their experience relate to the code change they recently made and run them. There is a

History-based	[Gra+01], THEO [Her+15]
Coverage-based	[GIP11]
Modification-based	[TTL89]
Requirement-based	[Gor+08]
Other approaches	[SSS11]

Table 2.1: Examples of RTO Strategies

lot of room for mistakes, because code which does not directly correlate to the changed code may also break with the change.

There are several approaches to optimise a test suite, one is test case prioritisation. It reorders test cases, often with the goal to increase the probability that a certain test case fails early in the process. This is beneficial because commonly if one test case fails during test execution, the remaining tests are not run, such that prioritisation can lead to a time reduction. Though if no test case fails, test prioritisation has no effect. A more radical approach is test selection, which is very effective to reduce execution time. Test selection modifies the test suite, such that a subset of test cases is run. The risk of not detecting certain bugs is higher with selection than with prioritisation, because some tests are not executed.

Solutions on how to optimise test cases automatically can be found in the literature. The first approach coming to mind would be to analyse code changes and calculate which code relates to the change. Unfortunately, solving this problem is NP-hard [AC91]. With this in mind, approaches, that gather data from previous test runs have been found.

Table 2.1 gives an overview on some algorithms which have been envisaged to find a heuristic solution. History-based approaches gather data on test runs, e.g., execution time or outcome. This history data is then evaluated for optimisation. Among others, THEO, which is introduced in section 2.2, is a history based strategy. While history-based approaches store test results, coverage-based approaches store the code that was executed when running a certain test case. The tested program is sliced into multiple parts, which are linked to a certain test case. If a change occurs test cases linked to that code are run. Modification-based algorithms take a similar approach. But instead of gathering coverage data they rely on models, e.g., object diagrams. Another approach takes a closer look at software requirements. Considering the fact that software is often build upon certain specifications, which are documented prior to development, requirement-based approaches optimise test suites such that test cases for more important requirements have a higher weight. Not every approach is covered here. Also there are some that do not fit the categories above, for example there are genetic algorithms which can be used for RTO purposes.

2.2 THEO

THEO is one strategy, which was presented in a paper by [Her+15], to optimise test suites. It is a test selection technique. The goal of their research was to find a strategy that not only reduces testing costs, but to find a trade off between optimising costs and software quality. THEO is self-adaptive, which means that it improves over time and can react on changes to the test suite. Since THEO relies heavily on this data it may need a training phase of about 50 executions upon initialisation. After that training phase THEO accesses four general categories of data: General Test Execution Information, Test Runtime, Test Results, Execution Context (e.g. processor type).

Cost of Execution: $cost_{exec} = cost_{machine} * (P_{FP} * cost_{inspect})$

Cost of Skipping: $cost_{skip} = P_{TP} * cost_{escaped} * Time_{delay} * \#engineers$

Detection Rate: $P_{TP}(t, c) = \frac{\#detecteddefects(tic)}{\#executions(tic)}$

False Alarm Rate: $P_{FP}(t, c) = \frac{\#falsealarms(t,c)}{\#executions}$

THEO is a metric-based strategy, it calculates the two cost functions depicted in the listing above. The $cost_{execution}$ formula describes the costs of running the test considering machine cost, false alarm and inspection cost. A false alarm is a failing test that is not caused by a code fault. Those can for example occur if the test case is outdated.

The $cost_{skip}$ formula describes the costs which are inferred by skipping a test. This formula considers that a certain bug escapes attention and needs to be fixed at a later point in development. This bug may block other engineers from committing their changes, which may lead to a time delay, $Time_{delay}$. Additionally, to fix that bug at a later point leads to higher costs. Those costs depend on a variety of factors. For example on the severity of the bug, but also on how long the bug stays undetected. In certain circumstances a undetected bug in a production release can infer huge damages.

False alarm and detection rate can be calculated by utilising the stored history values of previous test runs, other values have to be parameterised. An example is the machine cost per hour, or the costs for an escaped bug. Even though default values can be found, in practice those values are project specific.

To ensure software quality, the strategy enforces a branching schema, which is commonly used in software development. Quality guards are established at important points in the branching schema. Those quality guards ensure that at this point each test is run at least once. For instance, all tests should be executed at least once before the feature branch is merged into the main development branch. Merging erroneous code into the main development branch can lead to a multitude of follow up errors: Other developers may build upon faulty code and that code may need to be changed when the bug is finally resolved. Also if a bug is merged on the main branch, the developer who discovers the fault may not be the author of the faulty code, so he may need an increased amount of time to fix it.

2.3 Lazzer

The Lazzer framework was developed in 2015 at the RWTH Research Group Software Construction, [Ple15]. The framework can discover, select, prioritise test cases and then execute them. Discovery and execution is delegated to the test framework, e.g., JUnit. Lazzer offers an interface to modify test suites. Modifications can consists of selection and prioritisation. Additionally, Lazzer is able to store data, which is gathered before, during or after the test run, e.g., outcome and execution time of test cases. It was developed in a modular structure as figure 2.1 shows.

2.3.1 Architecture

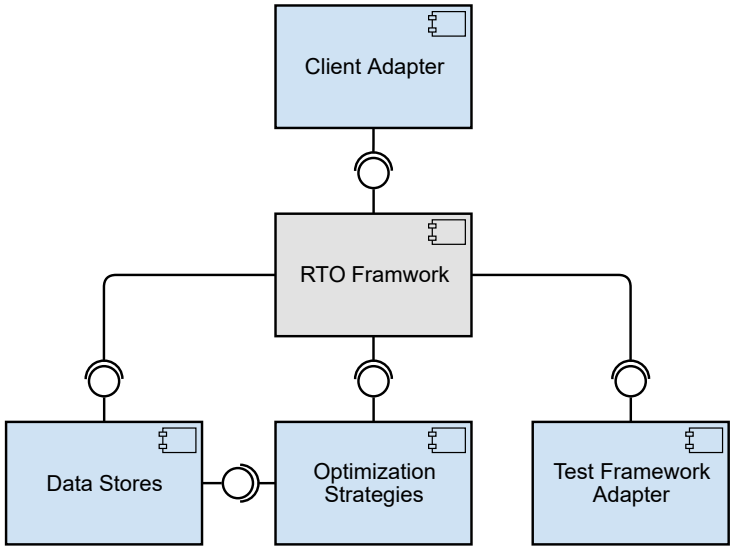

Figure 2.1: UML component diagram of the Lazzer Architecture.

Each of the main steps in the Lazzer framework are generalised by interfaces: storing test data, retrieving test data, and the optimisation strategy itself. Implementations of those interfaces are located in the corresponding modules. The main processing is executed in form of a pipeline. This part is called the RTO framework.

There a nine stages registered to the pipeline pattern. Each stage is executed separately in fixed order, depicted in figure 2.2 and executes certain tasks that can be modified by injecting different implementation of the interfaces mentioned before. These classes are pre-configured when using Lazzer and then loaded with the help of Google Guice.

Test Discovery Uses the testing framework's discovery mechanism to discover all test cases that are available for optimisation.

Pre-Optimisation Data Collection Takes care of instantiation of the data store and

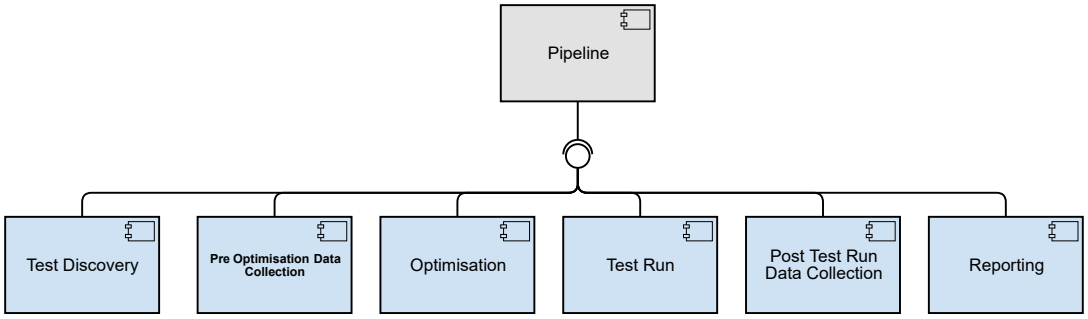

Figure 2.2: UML component diagram of the Lazzer pipeline stages.

runs data collection methods which are overwritten by the strategy, that has been selected.

Optimisation Strategy specific optimisation methods are executed.

Test Run The testing framework is used again, to run the modified test suite. Test results are collected here and processed in the next stage.

Post-Test Run Data Collection The data store is used to save test results.

Reporting In the reporting stage results of test execution and optimisation are shown.

Google Guice also takes care of injecting a singleton instance of the datastore into the strategy, the strategy can then access previously stored data on test runs. This makes Lazzer very configurable and new strategies or data stores can be implemented and used without changes to the RTO framework module.

2.3.2 Running Lazzer

Lazzer can be run either by using the command line runner or by using a Maven plugin which registers a Maven goal and executes test cases, similar to surefire. Surefire is a test runner for Maven [Sur].

Currently several simple strategies are implemented in Lazzer: Prioritise Execution Time, Prioritise/Select Failed Tests. Prioritise Execution Time accesses the datastore and retrieves the last execution time for a certain test case, then orders test cases starting with the lowest execution time. More tests can be run in a shorter time and under the exception that all test cases fail equally often, this increases the probability that a test case fails early and the test run finishes in a shorter period of time. Select Failed Tests avoids running succeeded tests if there is one or several that failed. The scenario is that a developer is currently fixing broken test cases and checks whether his changes have been successful. If no test has failed the previous test run, all tests need to be run.

There is also one example implementation of a data store which currently accesses an SQL database. That database is able to store time and outcome of previous test

runs and it can be accessed through the default data store interface. Tests are stored in the pre-optimisation data collection phase and results are gathered and added to the database in the post-test run data collection stage. In the current implementation only the canonical class name of those tests is used to store in the database. No additional attributes are evaluated in the process.

Furthermore, a test framework adapter implementation is necessary to run Lazzer, which is provided for the JUnit framework. Unit tests in Java projects are discovered in the discovery phase of Lazzer execution. The client adapter is also able to alter the test suite and run only the selected tests.

Lazzer can be run as a Maven Plugin, which is the most convenient way to run tests in a project, because tests can easily be included in a certain Maven goal, and for example bound to every build. The Maven plugin allows parameterisation, e.g., the data store implementation can be defined here.

3 Concept

Contents

Chapter 3 gives an introduction to graph databases at first. Then requirements of a metric-based strategy are elaborated. Finally, concepts on how to refactor Lazzer and implement the THEO strategy are given.

3.1 Graph Databases

Non-relational databases have been around since the late 1960, but while relational databases are the uncontroversial standard, non-relational databases regain popularity, examples are Googles BigTable and Facebook's Cassandra.[Vic+] The biggest difference to relational databases is the lack of a predefined schema. There are different kinds of non-relational databases, some of them are also called NoSQL databases. NoSQL

NoSQL databasses process data faster than relational databases [Lea10]. The downside is that data is handled with less precision. The lack of a schema can lead to programming errors, there is no security that a certain objects hold the correct data type.

Also most relational databases management system enforce atomicity, consistency, isolation and durability (ACID). Atomicity and consistency mean that transactions and updates are performed completely or not at all. Isolation means that multiple applications, that work on the database don't influence each other. Finally Durability means that a transaction which successful completes persist [Lea10]. NonSQL databases often relinquish to enforce those rules, which makes them more performant but also more prone to flaws.

There are Document databases consist of objects, which may have relations and

attributes. Also an object in a document database can nest another object. Also if there are frequent changes to the domain model, there is no schema that needs to be changed.

Graph databases consist of Directed Acyclic Graphs (DAG's). Those databases apply well if the domain model is similar to a graph. As a matter of fact it can become a complex and time consuming task to store and maintain a graph structure in a relational database. For this case graph databases offer a simpler and more seamless solution.

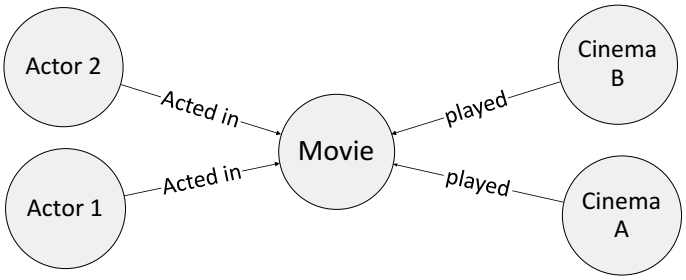

Figure 3.1: Illustration of a directed acyclic graph.

The paper by Mccoll et al. also suggests that graph databases have an performance edge when applying typical graph operations (e.g. depth first search). [MB] They suggest suggested that graph databases do not perform worse even on problems which seem more suitable for relational databases.

3.2 Requirements

This chapter covers the requirements of a metric-based optimisation strategy. Requirements already fulfilled by the Lazzer framework are covered shortly and then additional requirements are introduced.

3.2.1 Requirements for a Metric-Based Strategy

There are some general requirements for RTO, which this section summarises shortly, since Lazzer already supports simple optimisation strategies, and hence fulfils those requirements. A optimisation framework should be able to discover and execute tests.

Since testing frameworks exists the Lazzer should delegate those task to them, consequently Lazzer needs a connection to such a testing framework. Lazzer also should be able to model the discovered test suite, and modify it before finally delegating the test execution to the testing framework. Afterwards test results should be displayed and stored.

After giving an introduction to the requirements of RTO we will take a closer look at THEO, which has already been introduced earlier. The strategies developed in this thesis will be based on the idea of THEO and consequently have similar requirements. THEO is a self-adaptive strategy, that means it relies on gathering data from previous test runs. We call this data history data, to put it in other words THEO is a history-based strategy (cf. section 2.1).

Prior to selecting test cases, THEO calculates two cost functions $cost_{skip}$ and $cost_{exec}$, which are described in detail in section 2.2. Since those cost functions are aggregated metric values, the framework should be able to compute these metric values.

As a first step to calculate those metric values the test history data is accessed. After retrieving the required data some kind of calculation has to be performed. Additionally parameters like $cost_{inspection}$ are allocated to the calculation process. Finally the metric values have to be stored. The metric values used in the course of this thesis can be categorised as follows.

Constant parameters: Project/Test specific values which are constant for multiple test runs. For instance the cost for computation time.

Heuristic measures: Values, that are acquired by some kind of data analysis.

Constant parameters need be injected into the computation process by using some kind of configuration file. In contrast history values need be stored in the database section 3.2.2. A good example for a constant parameter would be the cost of a compute hour. Even though THEO does only use a fixed number of those values, the new Lazzer should provide a flexible solution to add fixed values.

Heuristic measures should be calculated in some kind of analysis engines. This analysis engines should have the single task of computing these values. Lazzer's modular structure should also be applied to those analysis engines. This will make it easy to add or exchange analysis engines, by using the frameworks hotspots. Additionally when looking at parts of the cost function, e.g. the detection rate, it occurs that metric values depend on each other. The detection rate as an example depends on the failure rate and the total number of test runs.

An implementation of a analysis engine should have a recursive element, such that more complex values can be constructed by combining multiple engines. Also to avoid duplicate calculations these engines should cache values, which have already been calculated, during a test optimisation. Additionally metric values should be stored for later usage. Conceivably those stored values could be used to calculate measures incrementally. The next section 3.2.2 will explain how those values are saved in detail.

3.2.2 Test History

THEO falls in the category of history-based strategies (cf. section 2.1). As a consequence there is the need for a way to comprehensible access result data, which was gathered in previous test runs. Figure 3.2 shows the structure of multiple test results in form of directed graph.

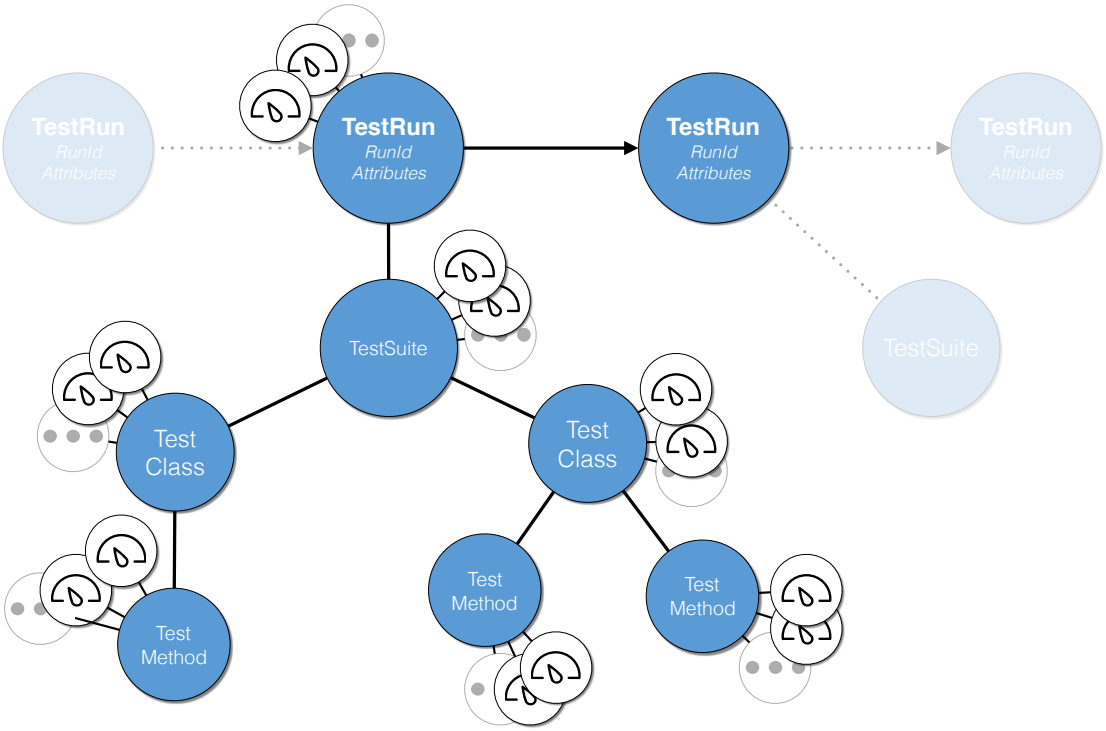

Figure 3.2: Illustration of the data, that will be stored in the testhistory data store.

This structure widely resembles the structure of Lazzer's data model (cf. section 2.3) of a test suite, which also is very similar to the xUnit architecture section 4.1.1. But since the structure should be able to represent multiple test runs with evolving test suites, there is an additional node TestRun. If a test run is completed it is appended as a successor to the most recent test run already stored in the database.

Metric values are appended as nodes, which are either connected to a test method, test class or a test run. One of such test objects can have a number of metric values, but each metric value has one target node. History values are essential and every node has them. In comparison to metric values it's also not necessary that history values are connected to other nodes. For instance metric values could be an aggregation of other metric values, while history values are rather simple. As a consequence history values are stored inside the node they belong to. This differentiation makes it easier to separate calculated metric values from values which are provided by the testing framework.

Lazzer currently uses a relational database and it would be possible to store this structure in that database. But as mentioned in section 3.1 there graph databases are a more efficient way of storing such a graph structure. Even though this is not a necessary requirement it is a reasonable extension to the Lazzer framework.

3.3 Refactoring and Extension of Lazzer

This section covers in detail how Lazzer is refactored and extended. As already stated in the previous section 3.2.2 some kind of analysis engine is needed. A new hotspot for analysis engines has to be added to the framework. Also parameterisation of a strategy is currently not implemented in Lazzer and it needs to be implemented. In the course of that access to some kind of configuration file and a way to directly assign parameter values to test cases needs to be created. A detailed concept on how parameterisation is done will be given in section 3.3.3. Finally history data storage needs to be reworked to store data in a graph database.

3.3.1 Coarse-Grained Architecture

Figure 3.3 depicts the extended architecture of Lazzer. A new hotspot for analysis engines has been added. Those analysis engines can recursively depend on each other. Strategies can access those analysis engines to get metric values. The benefits of this architecture are that multiple strategies can share analysis engines.

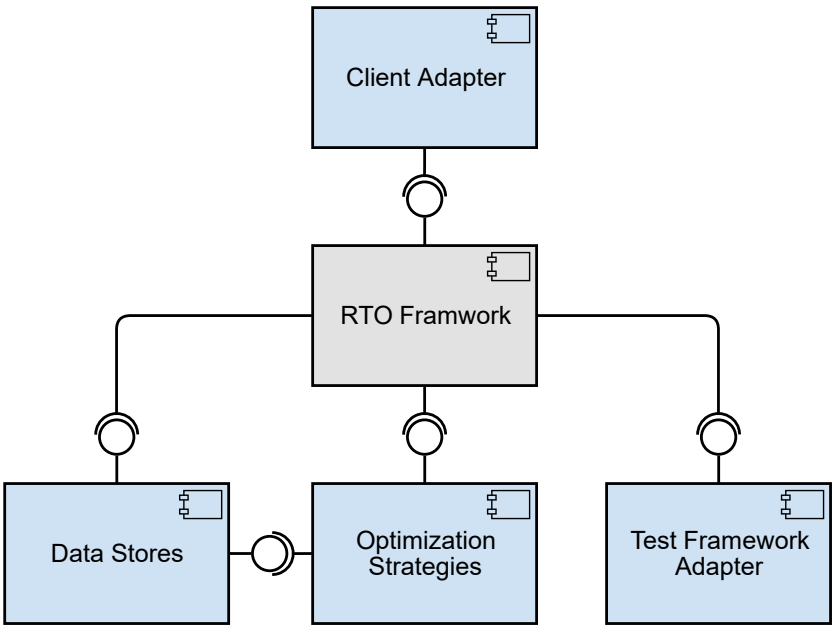

Figure 3.3: UML component diagram of the coarse grained architecture

Parameterisation will be implemented as one analysis engine, such that it can be reused by multiple strategies. Additionally changes to the RTO framework to support analysis engines are made. And the test framework needs to be extended to support test specific parameters. The following chapters will give a detailed concept of those changes.

To implement a data store, which supports graph databases no changes to the archi-

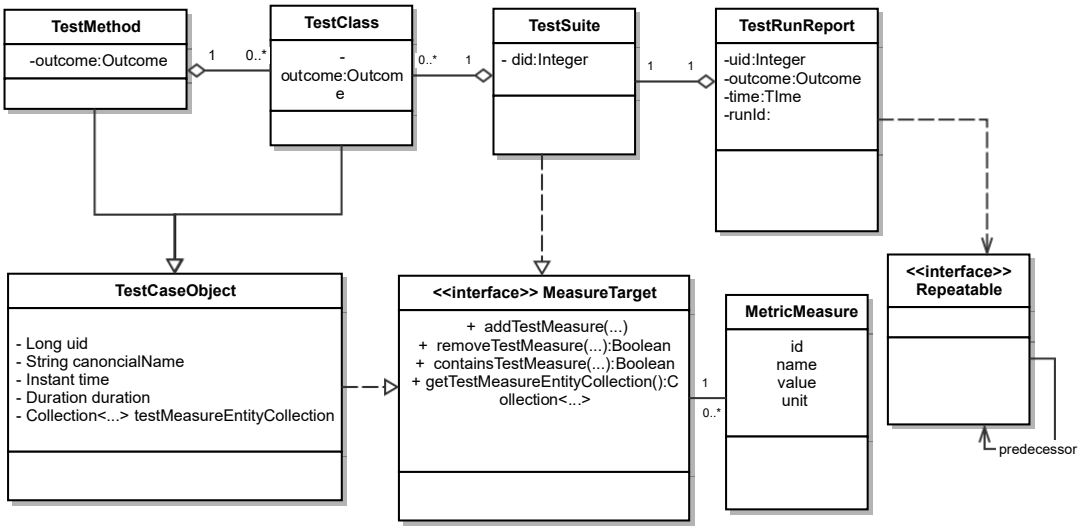

Figure 3.4: UML class diagram, which shows the meta model used to model history data in the updated Lazzer.

tecture are necessary. The new data store will make use of the test history data store hotspot. Even though most changes can be made using the framework's hotspots some changes to the core framework need to be made, those will be covered in the following.

3.3.2 Test History's Meta Model

As already stated before a graph database would be a promising way to store data that is gathered by Lazzer. This chapter will give a fine grained concept of how that data can be stored and which benefits result from using a graph database.

First of all, Lazzer should in the future support multiple different strategies. As stated in section 2.1, there a many strategies that take completely different approaches. Since some of them may need to store data it would be beneficial if the data store could easily be adapted. Relational databases have a schema which often needs to be updated if mayor changes occur to the domain model. Even though a schema that is flexible enough could be created, but the effort can be saved by using a non-relational database.

Also Lazzer should be able to process a reasonable amount of data fast enough to really save execution time. Another benefit could be that non-relational databases generally perform better. In particular graph databases perform very regarding typical graph algorithms like depth first search [MB].

The meta model depicted in figure 3.4 shows how the graph database will be stored. The interface `Repeatable` is used to model that each test run as a predecessor. Both test method and test class extend the class test case object. Lazzer differentiates between test classes and test methods while the xUnit framework does not, this is a compromise between both architectures.

The interface measure target is implemented by the `TestMethod`, `TestClass` and `TestSuite`, because metric values exists for each of them. Each `MeasureTarget` can have a number of test measures. A test measure in this modal is not strictly defined. Hence it can consist every bit of information that is related to a test suite or test case object.

3.3.3 Parameterisation

Another important requirement was the need of parameters to inject project specific values. Parameterisation was design to be flexible enough to be used with other strategies and not only THEO. The following Parameters could be divided in three categories: strategy parameters, project parameters and test parameters. The first two should be gathered in some kind of configurations file, while the test specific parameters could be added directly to the test they influence. This chapter will give a detailed explanation of this structure.

Strategy Parameters are used to define the basic behaviour of a strategy.

Project parameters are used to define constant values, e.g. compute cost per hour.

Test parameters are used to define test specific values, e.g. test priority.

Parameters consist of fixed values, which are not altered by the strategy. Strategy parameters are those which vary for every project. An example of such a parameter would be the compute cost per hour, as this is a value that is not influenced by test runs and hence is not changed by the strategy. There are multiple other values that can be found in the THEO strategy and are discussed more detailed in section 2.2.

Test parameters are assigned to test cases. The idea is that a test can have for example a priority. Considering there is a project that has some core functions which are reused in almost every component. Assuming that one of those failed, a failure can be overseen easily. The result would most likely be that the whole application fails to function within specifications. Those tests are more important than others. If this fact is known by developer working on that project, it can be used to improve optimisation. Test parameters should be added to the optimisation process to reflect this knowledge.

Other kinds of test parameters are possible. For instance, tests could be bound to a branch, such that those tests are always executed if ran on that branch. For simplicity we choose to implement a test priority parameter. There are five priority values, which are explained with detail in section 3.3.3. Test prioritisation is also influenced by priorities, while the usual prioritisation methods are considered first, if two test cases have the same value test priority influences the order.

3.4 Strategies

This section will give an introduction of the ideas of three example strategies which are implemented in the course of this thesis. The first will make usage of an analysis engine

Priority	Test Selection	Test Prioritisation
NEVER	never executes	execute last
AVOID_EXEC	avoid execution	execute later
DEFAULT	default priority	
AVOID_SKIP	avoid skipping	execute former
ALWAYS	never skips	execute first

Table 3.1: Testing Frameworks Release Dates

and order test methods regarding the values it delivers. The metric-based parameter strategy will show how a cost function can be parameterised and the last strategy implemented is the THEO strategy to show that the updated Lazzer really is compatible.

3.4.1 Detection Rate Prioritisation

Detection rate prioritisation is the first example strategy. It was mainly implemented to show how an analysis engine is built. There is no parameterisation involved in this strategy. Figure 3.5 depicts how the necessary analysis engines are built. The detection rate will be calculated like stated in section 2.2. Each engine implements the interface AnalysisEngine and has three methods. The first, named calculate, is called by the strategy to retrieve the value for a certain target, which might be a TestMethod or TestClass in this case.

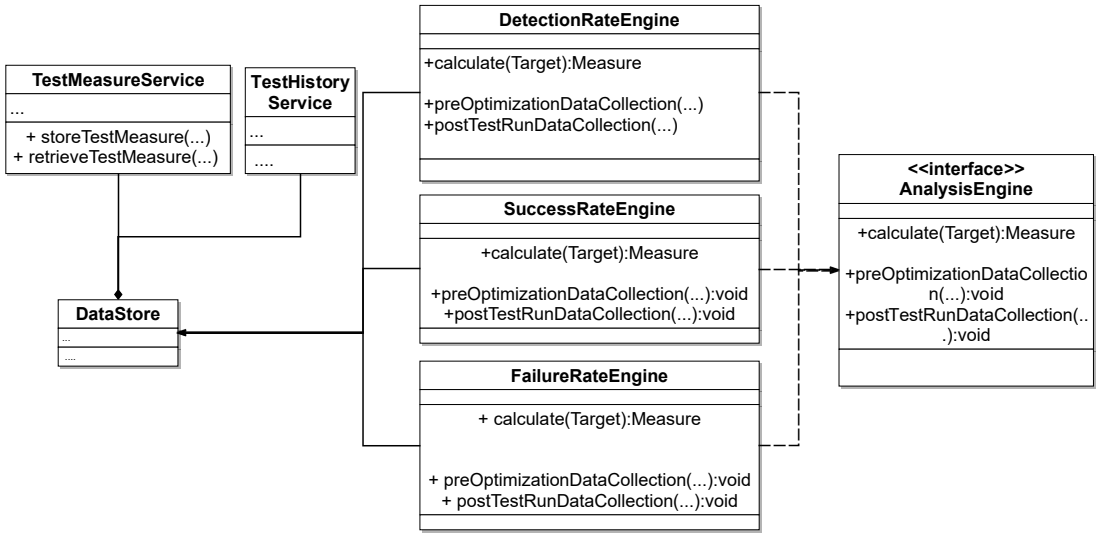

Figure 3.5: UML class diagram of the detection rate engine.

The preOptimizationDataCollection and postTestRunDataCollection are called in the equally named stages of the Lazzer pipeline and in this case save the calculated values to

the data store.

3.4.2 THEO strategy

In the previous chapter an example for a analysis engine was given, this chapter will concentrate on the strategy itself. To calculate the cost functions the following analysis engines will be needed:

- **CostEngine**:
  $cost_{exec} = cost_{machine} * (P_{FP} * cost_{inspect})$,
  $cost_{skip} = P_{TP} * cost_{escaped} * Time_{delay} * \#engineers$
- **MachineCostEngine:** $cost_{machine} = duration_lastexecution * parm_{machinecost}$
- **DetectionRateEngine:** $P_{TP}(t, c) = \frac{\#detecteddefects(tic)}{\#executions(tic)}$
- **FalseAlarmRateEngine:** $P_{FP}(t, c) = \frac{\#falsealarms(t,c)}{\#executions}$
- **SuccessCountEngine:** $\#executions$, $\#detecteddefects(tic)$
- **FailureCountEngine:** $\#executions$, $\#falsealarms(t, c)$
- **ParameterEngine:** $cost_{inspect}$, $cost_{escaped}$, $cost_{machine}$, $\#engineers$, $Time_{delay}$
- **GitEngine:** Branch and commit Information for quality guards (cf. section 2.2).

The false alarm rate engine is build similar to the detection rate engine, but it requires some more effort, because the inspection result is considered. The inspection results gives information whether the failure of a test cases was a bug or if it was a false alarm. An attribute inspection result was added to the test case object. The inspection result can have three values, unresolved, resolved and false alarm. When a test fails the inspection result is set unresolved. After it has been inspected by a developer it is either set to resolved if a bug was fixed or to false positive if no bug could be found.

The Git engine accesses information about the branch and once a certain branch is hit, e.g., the master branch. All tests are executed. This is a simplification in comparison to the original THEO structure, but suffices to show how quality guards can be implemented.

The machine cost engine uses the history data to determine the last execution duration of a specific test case. After that it utilises the parameter engine to calculate a value for the computation cost. Even though this is only an approximation, because execution times may fluctuate, it is better than a fixed value for every test.

Finally the Lazzer framework's hot spot for optimisation strategies is used to add the THEO optimisation strategy itself. This strategy uses the state analysis engines. The strategy does not perform any calculations itself, morover it utilises the cost engine and optimises based on the values provided.

3.4.3 Metric-Based Parameter Strategy

The new Lazzer allocates multiple new possibilities to build strategies. Having that in mind this strategy uses those possibilities and evaluates the idea of using a sum of costs for optimisation. The basic idea is similar to the THEO strategy. Two cost functions are defined. One cost functions to skip a test case and one to execute it. The difference is that those cost functions can be configured. The underlying thought is that a cost function may not fit every project. In figure 3.6 depicts how the optimisation process of such a strategy looks like.

Both cost functions are parameterised and injected into the optimisation process. That optimisation process also has access to test parameters, project parameters, and the original test suite. The combination of that information is used to calculate the cost functions and finally to optimise the test suite.

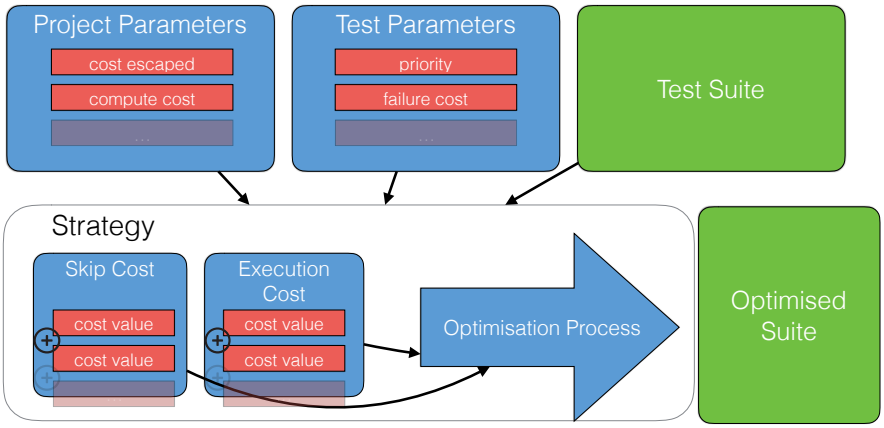

Figure 3.6: Illustration of the parameterised optimisation process.

This strategy will be similar to THEO. The relevant formulas are shown below, where fp and dr are the false alarm and detection rate as defined in the THEO strategy.

$$cost_{skip} = cost_{costbranch} + (dr * cost_{annotated})$$

$$cost_{execution} = cost_{machine} + (fp * cost_{inspection})$$

Each cost functions is again calculated by an analysis engine. Some of those can be reused, since they have already been part of the THEO strategy, others have to be implemented. The following engines have been added:

- CostBranchEngine
- StrategyParameterEngine
- MetricMeasureEngine

The annotated cost engine calculates the value of a certain test case, based on the annotated priority and a default cost. Additionally, test cases which are annotated with the priority NEVER are not executed. Test cases with a higher priority are also valued higher by the strategy. For instance tests that ensure a critical function of the program can be associated with a very high priority, this way external knowledge can be utilised in the strategy.

The cost branch engine is used to connect the cost of skipping a test to the branch the test is executed on. The idea is that skipping a test on the master branch may lead to much higher cost than skipping a test on some development branch. This is due to the fact that code faults on the master branch may be undetected when releasing the software to production. Even though this rule is not universal, we assume it for the sake of this example. This strategy could be configured to execute few of the most important tests on the development branch, but execute a more extensive test suite on the development branch. The cost of false detection calculates the formula $dr * cost_{annotated}$, which represents the cost of a false alarm depending on the annotated value of a test case.

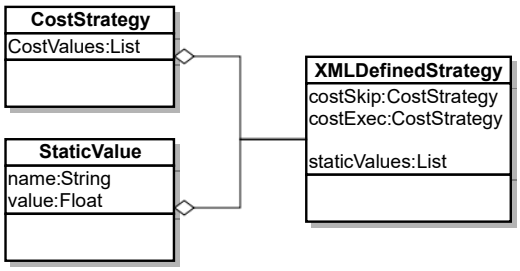

Figure 3.7: UML class diagram, which shows the meta model for strategy parameters.

Finally the strategy parameter engine takes care of reading the file, which specifies the cost functions and static parameters. Assuming such a analysis engine is implemented, no new implementation is needed to add or remove a cost value to the strategy. This allows for the strategy to be easily adapted for the individual projects. The meta model is depicted in figure 3.7.

4 Realisation

Contents

This chapter gives an introduction to the necessary background for implementation. Further, details on the implementation are given.

4.1 Background for Realisation

This section gives an introduction to all necessary libraries and frameworks that are utilised to realise the updated Lazzer framework and metric-based strategies. First unit testing is introduced, then Spring, Spring Data and Neo4J is covered.

4.1.1 Unit Testing

'Fixing a bug is usually pretty quick, but finding it is a nightmare. And then when you do fix a bug, there's always a chance that another one will appear and that you might not even notice it until much later. Then you spend ages finding that bug.', was stated by Martin Fowler in [Fow99].

Tests are used to find bugs, after a change has been made to the software. The citation above shows how much effort is connected to finding bugs and emphasises how important unit testing has become. Unit tests, in contrast to integration tests, do not test the complete program, but rather each test is designed to evaluate a isolated unit of software. The idea is to split up the code in small fragments and test if they work as specified in a controlled environment. Often other parts that are less relevant to a specific test are mocked (cf. [Ham]).

Test frameworks support implementation and automated execution of tests. Some of those are described by xUnit frameworks. The term xUnit is derived from JUnit which is the most common unit testing framework for the Java programming language. JUnit has been ported to a number of other programming languages. Those frameworks all have their basic architecture in common, which will be introduced in the following section.

xUnit

While xUnit may not be a huge project, regarding lines of code, the impact to a development still is respectable [Fow06]. The xUnit architecture, figure 4.1, is built around two objects: test methods and test classes. Since test classes are only a container to test methods, xUnit does not distinquish between those. Instead xUnit has a class called `TestCaseObject`, which represents cases.

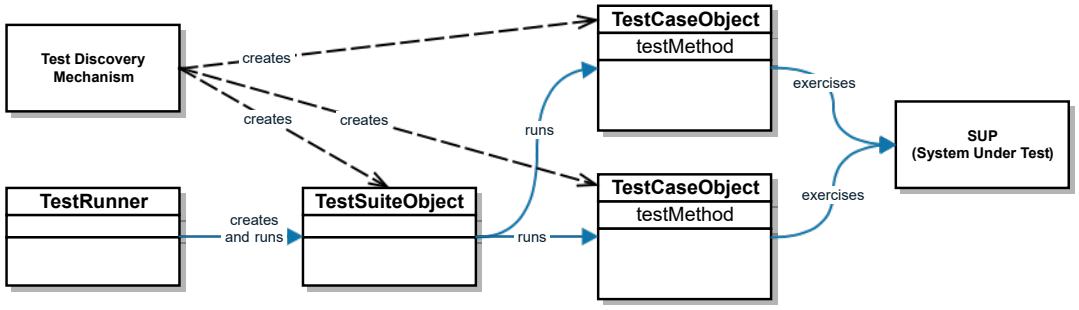

Figure 4.1: UML Diagram of the simplified xUnit architecture.

Test case objects are then gathered in a *TestSuiteObject*, which is generated in a *TestSuiteFactory*. Those test suite objects can contain test methods from multiple test classes. A test suite object can be run by a test runner. Consquently all test case objects in that suite are run.

The unit framework does not only run tests, it also has the capability to find tests that have been written before. Those tests can then be loaded into a test suite object and finally be run. Discovering test cases often is parameterised by a classpath or class.

When tests are about to be executed, the test runner delegates the run of single test methods to the test suites. Those build up the SUT (System Under Test) in four steps: setup, exercise, verify and teardown. Setup and teardown steps are taking care of the surrounding fixture, needed to exercise the tests. The exercise step executes the test itself and the verify step takes care of verifying the results.

4.1.2 Spring

The Spring dependency injection framework can be seen as an extension to Java EE, it is licensed under the Apache license. The Spring framework aims to simplify the creation of Backend-Application by favouring standards of configuration. While previous versions of Spring relied heavily on Xml configurations, the most current development relies heavily on annotations. Annotations are an integral part of Java since version 5.0.

Annotations, provided by Spring, are not only used for configuration but also to realise some default design patterns, e.g., the singleton pattern. In this thesis, we mainly used two Spring modules. On one hand, dependency injection, and on the other hand, Spring data is used within the data store to provide database access. Spring contains a variety of

other modules. Those, for example, provide support for easily implementing web services (Spring MVC framework). Castor is also part of the Spring core framework and is used in this project to map Xml files to Java objects.

Dependency Injection

The Spring dependency injection can be initialised in a quite small example. The following example shows how to initialise a Spring Boot project. Spring Boot is a collection of some Spring components, which allow to easily bootstrap an application. The most important aspect of a Spring Boot application is the entry point, where the dependency context is initialised, as can be seen in listing 4.1, listing 4.2, and listing 4.3.

```
1  package example.springboot;
2
3  import ...
4
5  @ComponentScan(basePackages = {"example.springboot"})
6  public class DependecyContextMain {
7
8      public static void main(String[] args) {
9          ApplicationContext applicationContext = new
               AnnotationConfigApplicationContext(Main.class);
10         ExampleService service =
               applicationContext.getBean("example.springboot.ExampleService")
11         System.out.println(service.getMessage());
12     }
13
14     @Bean
15     public ExampleBean getExampleBean(){
16         return ExampleBean("Hello World");
17     }
18 }
```

Source Code 4.1: DependecyContextMain.java

```
1  package example.springboot;
2
3  import ...
4
5  @Component
6  public class ExampleBean {
7      private String message;
8
9      public ExampleBean(String message){
10         this.message = message;
11     }
12
13     public String getMessage(){
```

```
14          return this.message
15      }
16  }
```

Source Code 4.2: ExampleBean.java

```
1  package example.springboot;
2
3  import ...
4
5  @Service
6  public class ExampleService {
7      @Ressource
8      ExampleBean exampleBean;
9
10     public String getMessage{
11         return exampleBean.getMessage();
12     }
13 }
```

Source Code 4.3: ExampleService.java

This code example will execute and return 'Hello World'. When the Spring context is initialised, Spring automatically searches for classes, which are annotated with `@Component` and `@Service`. Custom annotations which extend the `@Component` annotation are also supported. Then those classes are injected where `@Resource` is utilised. In this example the `ExampleBean` class is initialised in its `@Bean` configuration and then injected into the example service. Applications that are built with this pattern can be easily configured and save a lot of code. Also in comparison to more elaborate xml configurations code can be saved significantly. For instance if a bean has the default constructor without parameters, no `@Bean` configuration is necessary.

Spring Data

Spring Data is relatively new component of Spring which supports data from different origins. Among others, there are:

JPA The Java Persistence API can be used to access a relational database.

MongoDB is a popular NoSQL database.

Redis is a in memory data structure storage.

Key-Values Makes it easy to access custom data stores that store key value pairs.

REST Allows to access custom web services that use the REST protocol to provide data.

Neo4J Access the graph database management system Neo4J.

Every implementation of Spring Data follows the same main structure, which is depicted in figure 4.2. The domain model consists of entities and relationships, those are modelled with Java classes and then annotated either with `@NodeEntity` or `@RelationshipEntity`, to map those classes to the graph database. Repositories are used to access those entities on the database. Those repositories access an entity, that is specified with a generic attribute. A collection of methods to access and store that entity is then provided via inheritance. If those default methods are not sufficient, there are multiple ways of adding custom methods. Some of those will be covered in the following section.

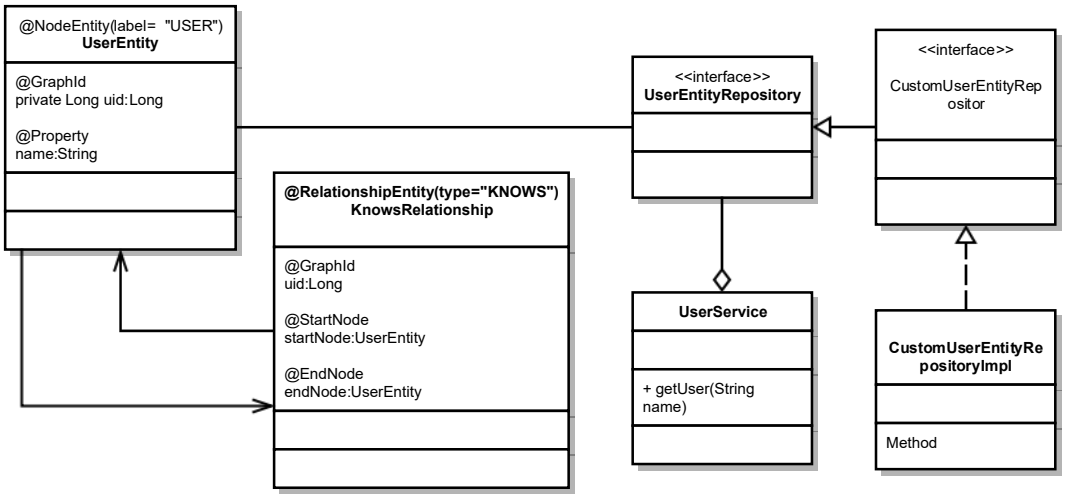

Figure 4.2: UML Model of a spring implementation.

```
1    @Query("MATCH n:Actor WHERE n.name= {name} RETURN n")
2    public Actor getMovieActors(@Param("name") String name);
```

Source Code 4.4: QueryAnnotation.java

One way to add custom methods is the `@Query` annotation. It can be used as depicted in listing 4.4. This statement queries for nodes with the label 'Actor' and the attribute name, that equals the passed parameter. Spring is also able to generate queries from method names. Spring detects some common keywords ,e.g., 'findBy' and if a methods starts with such a keyword, the following word is taken as an attribute name. An example is given in listing 4.5.

```
1 //Method for named query
2 private findByName(String name);
3 //Corresponding cypher query
4 MATCH n:Actor WHERE n.name=#{name} RETURN n
```

Source Code 4.5: NamedQuery.java

Repositories can be extended using the schema depicted in figure 4.2. For instance, if data can not be queried with a single query, a custom repository can be instantiated. First an interface is created, which is then implemented. Spring searches for custom repositories. For Spring to be able to find that class it has to be named exactly considering the rule: Custom + RepositoryName. And if found, it searches for implementations of that interface, which have to end with 'Impl'.

Castor

Castor is one possibility to read Xml files provided by the Spring framework. It consists of a marshaller and an unmarshaller, those can be used like depicted in listing 4.6

```
1 public class Person {
2     String name;
3 }
4
5 public class Group{
6     List<Person> persons;
7 }
```

Source Code 4.6: XMLMetaModell.java

```
1 <group>
2     <person>
3         <name>John</name>
4     <person>
5
6     <person>
7         <name>Mike</name>
8     <person>
9 </group>
```

Source Code 4.7: Group.xml

```
1     public void convertFromObjectToXML(Object object, String
          filepath) throws IOException {
2
3         FileOutputStream os = null;
4         try {
5             os = new FileOutputStream(filepath);
6             getMarshaller().marshal(object, new StreamResult(os));
7         } finally {
8             if (os != null) {
9                 os.close();
10            }
11        }
12    }
13
```

```
14      public Object convertFromXMLToObject(String xmlfile) throws
            IOException {
15
16          FileInputStream is = null;
17          try {
18              is = new FileInputStream(xmlfile);
19              return getUnmarshaller().unmarshal(new
                    StreamSource(is));
20          } finally {
21              if (is != null) {
22                  is.close();
23              }
24          }
25      }
```

Source Code 4.8: ConvertXML.java

Castor can convert XML objects to Java objects and the other way around, without any further configuration.

4.1.3 Neo4J

Neo4J is a graph database management system, which is licensed under the GNU license [Neo]. All values are stored either in a node an edge or an attribute. An example can be seen in figure 4.3. Each node has a type, here street, town and country. The attribute name can be stored in a node. Each element in the graph database has an automatically generated id attribute. This attribute is recommended for internal use only, since it is reused if an entity is deleted. In this example all edges have the same type, which is 'isLocatedIn'.

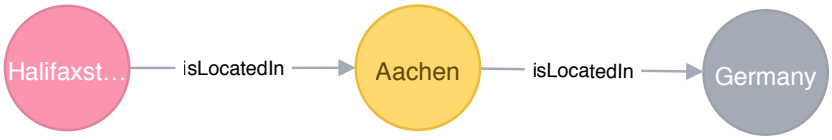

Figure 4.3: Neo4J Graph Example

Graph databases are not compatible to SQL-Queries, so Neo4J brings its own query language, which is called cypher. In Listing listing 4.9 three examples of cypher are shown that create the example figure 4.3, then match all nodes in the example, and afterwards delete the country node of the example.

```
1
2     CREATE (n:Street)-[e:isLocatedIn]->(b:City{name:'Aachen'})
3                     -[f:isLocatedIn]->(c:Country{name:'Germany'});
4
5     MATCH (n) return n;
6
7     MATCH (n:Country) where n.name='Germany' DELETE n;
```

Source Code 4.9: CypherExample1.java

```
1     CREATE CONSTRAINT ON (n:Country) ASSERT n.name IS UNIQUE;
```

Source Code 4.10: CypherExample2.java

Cypher has a variety of other features, which are not important regarding the topic of this thesis, and therefore are not presented here. By design there is no need for schema, but since version 3.0 of Neo4J schemas can be implemented. Having a schema can have some advantages for data integrity. A technique similar to using schemas is the creation of constraints, those can be defined in cypher.Listing 4.9 shows how to assure that each country's name is unique. There are some additional possibilities to add constraint to Neo4J databases, which are not covered here.

Figure 4.4: Illustration of Neo4J graph

Another useful feature of Neo4J is indexing, which allows for faster and simpler queries. When using an early version of Neo4J the only way to find an entry point was to manually define it. Before Neo4J version 1.0 was released, manual indexing was the only option to indexing.

```
1   START n=node:Person(index='abc') RETURN n
```

Source Code 4.11: CypherQueryIndex

The cypher statement, listing 4.11, shows how to access the manual created index. Later in version 2.0 of Neo4J automatic indexing was introduced. Automatic indexing basically is manual indexing with a fixed name and can be utilised as follows in listing 4.12.

```
1     START n=node:node_auto_index(index='abc') RETURN n;
```

Source Code 4.12: CypherQueryIndex2

To model a chain of nodes, a start node had to be defined. This start node may have a special type or an attribute identifying it uniquely. This way of indexing has one

pitfall, if both a person and a city are index with the attribute name index, both will be returned as a result of the query for 'abc'. Those manual and automatic indexes are also called legacy indexes and practically should not be used in favour of schema indexing. Schema indexing allows to use a way similar to creating constraints to create an index on a certain type of node or relationship.

```
1    CREATE INDEX ON :Person(index);
2
3    MATCH (p:Person {index: 'abc'}) RETURN p
```

Source Code 4.13: CypherExample3.java

Also there is no extra code necessary to improve the performance of a query. Neo4J automatically utilises schema indexes without the change of queries; in listing 4.13, such a match statement is shown.

4.2 Realisation

This section will cover the refactoring of Lazzer and how the metric-based strategy is implemented.

4.2.1 Test History Data Store

A database is needed, since THEO is a history-based strategy. Neo4J, section 4.1.3, will be used as a database management system. There are multiple reasons that make Neo4J a suitable choice. One of them is that it stores graphs natively, which leads to a better performance. Neo4J also offers an extensive documentation and has an active community, keeping the project up to date [Neo].

Neo4J can be used with Java in multiple ways. One is to use Java's JPA adapter with hibernate, another would be to use the Neo4J Java driver, which was especially developed for Neo4J. Spring data seemed best, because other parts of Spring are also used in this project. Spring Data uses annotations to map the domain model to database objects. Figure 4.5 depicts the meta model of the test history datastore.

For each accessed entity a repository is needed. Each repository inherits a number of methods by default. Those can be used to access the database. Each repository provides access to one entity. The Spring framework needs a main class where it is initialised. Since Lazzer has implemented its own way of dependency injection and we did not want to lose Lazzer's modular structure, the Spring context is limited to the package of the test history data store, line 7 in listing 4.14.

```
1 package de.rwth.swc.lazzer.datastores.testhistory;
2
3 import ...
4
5 @Configuration
6 @EnableNeo4jRepositories
```

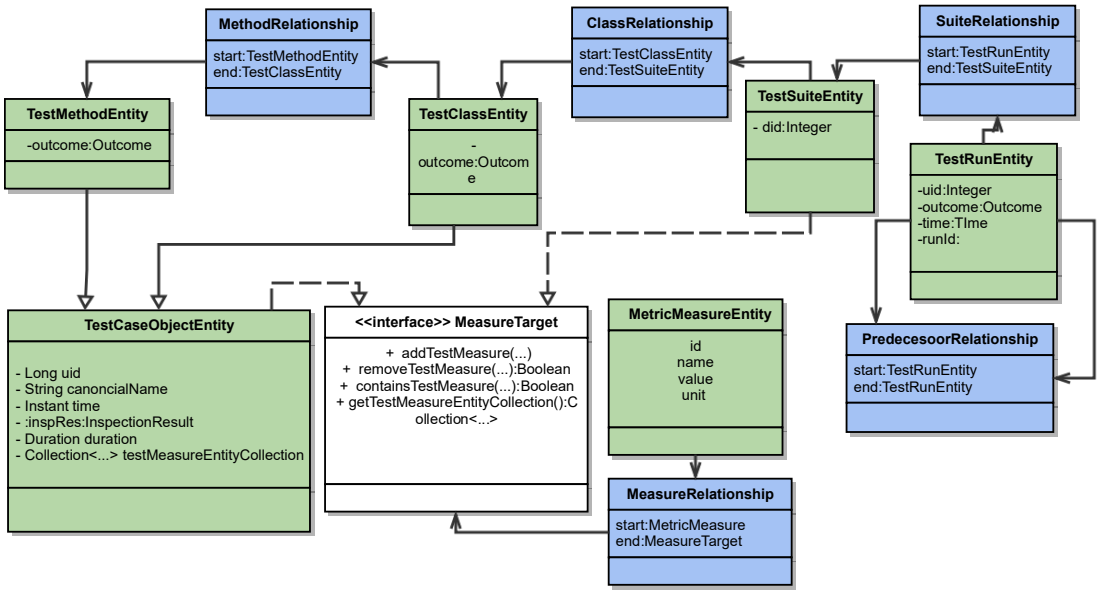

Figure 4.5: UML class diagram showing the domain module of the test history database.

```
7  @ComponentScan(basePackages =
       {"de.rwth.swc.lazzer.datastores.testhistory"})
8  public class DatabaseContext extends Neo4jConfiguration {
9
10     public static final String URL =
11         System.getenv("NEO4J_URL") != null ?
12         System.getenv("NEO4J_URL") :
13         "http://neo4j:passwort@localhost:7474";
14
15     public static ApplicationContext initContext() {
16         return new
17         AnnotationConfigApplicationContext(DatabaseContext.class);
18     }
19
20     @Bean
21     public org.neo4j.ogm.config.Configuration getConfiguration() {
22         org.neo4j.ogm.config.Configuration config = new
               org.neo4j.ogm.config.Configuration();
23         config
24         .driverConfiguration()
25         .setDriverClassName(
26           "org.neo4j.ogm.drivers.http.driver.HttpDriver"
27           )
28         .setURI(URL);
29         return config;
30     }
```

```
31
32      @Override
33      public SessionFactory getSessionFactory() {
34          return new SessionFactory(getConfiguration(),
35          "de.rwth.swc.lazzer.datastores.testhistory.entities");
36      }
37  }
```

Source Code 4.14: DatabaseContext.java

Listing 4.14 shows the initialisation of the context. The database location and credentials are configured by creating a bean of the class `Configuration`. This example can also be seen as a minimum working example, because no further configuration is necessary to use Spring Data.

4.2.2 Analysis Engines

This section introduces analysis engines and show how they are implemented. Figure 4.6 gives an overview of those engines and their dependencies on each other. Each engines implements the `AnalysisEngine` interface, and hence inherit the `preOptimization-DataCollection()` and `postTestRunDataCollection()` methods, these methods are currently used to store results in the database, but may be used otherwise in the future.

The analysis engine also has multiple calculation methods, which have different parameters. For example there is a `calculate()` method for test classes. Those calculation methods return objects of a type that implements the interface `Measures`.

The specific type is set using the generic argument of the `AnylsisEngine` interface. Measures have a name attribute, which is used to store them in the database. Additionally, those measure objects have a value attribute. The type of that attribute can be set using a generic parameter. In the `Measure` class, there also is a method to convert the value to a string that can be stored in the database.

It was mentioned before as a requirement (cf. section 3.2), that the engines should cache already calculated values to avoid multiple calculations. The solution is to use a decorator pattern, which is depicted in figure 4.7. The class `CachedAnalysisEngine` stores calculated values in a map and if the value has been calculated before, it returns the value from that map rather then to calculate it. The benefit of using this pattern is that the engine can be used either way. The `CachedAnalysisEngine` decorator can be used to cache values or the engine can be accessed directly to avoid caching values.

Another requirement was that analysis engines work recursively. Furthermore, it turned out that each cache analysis engine should be accessed as a singleton. This avoids multiple calculations if one engine is reused multiple times. Fortunately, Spring offers an annotation offering just that. The `@Service` annotation ensures that only one instance of a class is injected. An example of the @Service annotation is also given in listing 4.3 (page 26). Additionally the annotation `@Qualifier` is used to uniquely name analysis engines. This could in the future also be utilised to name multiple configurations of analysis engines. Listing 4.15 shows how injection works.

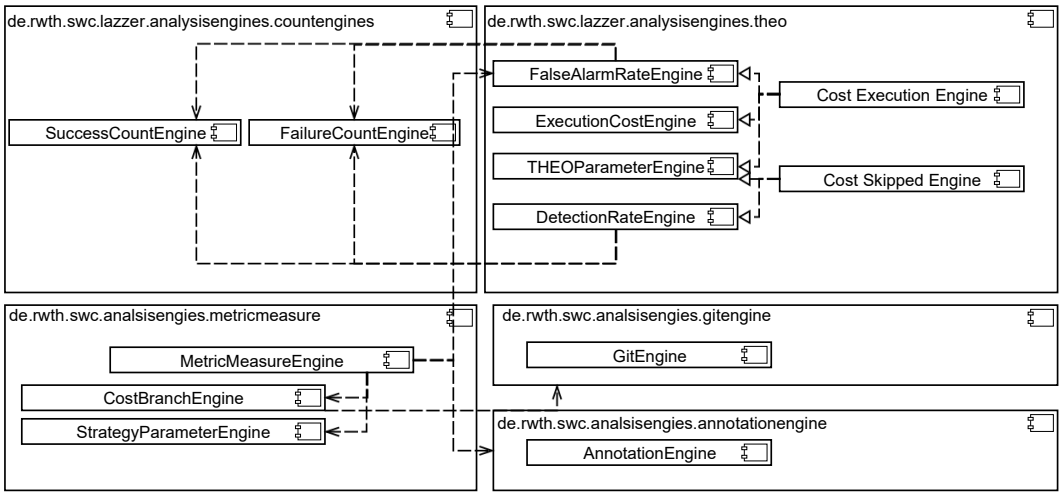

Figure 4.6: UML component diagram of analysis engines.

```
...
    @Autowired
    @Qualifier("de.rwth.swc.lazzer.analysisengines"
                +".countengines.success.SuccessCountEngineCached")
    private SuccessCountEngineCached successCountEngine;
    @Autowired
    @Qualifier("de.rwth.swc.lazzer.analysisengines"+
                ".countengines.failure.FailureCountEngineCached")
    private FailureCountEngineCached failureCountEngine;
...
```

Source Code 4.15: AutowireDetectionRate.java

Success/Failure Count Engines

The success and failure count engine access method can be calculated for three targets: test methods, test class, and test suites. In either case the database is accessed and the the number of test cases with the respecting property is calcualted. Both engines do not access any other analysis engines, such that the only dependency is the test history datastore. The `TestHistoryDataStore` is injected using the `@Autowire`, which is shown in listing 4.16

```
...
    @Autowired
    TestHistoryDataStore testHistoryDataStore;
...
```

Source Code 4.16: SuccessCountEngine.java

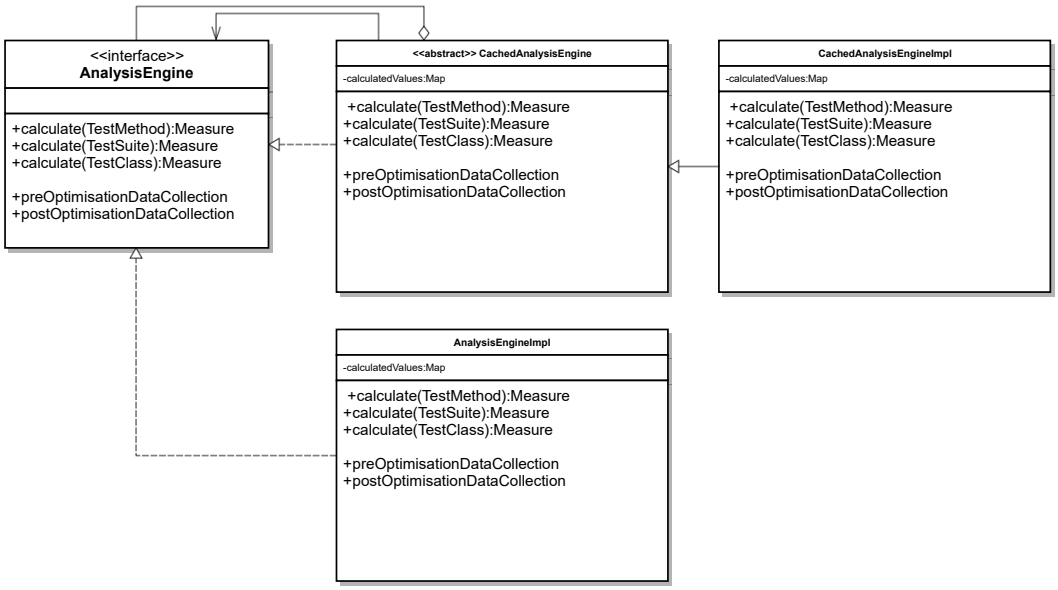

Figure 4.7: UML class diagram of cached analysis engine

```
1 ...
2 @Service("TestHistoryService")
3 public class TestHistoryService {
4 ...
```

Source Code 4.17: TestHistoryService.java

Detection Rate Engine

The detection rate engine overrides the calculation methods of `AnalysisEngine` interface for test classes and test methods. Only those methods, that can be applied for the measure, are overwritten by the analysis engine. This is possible because within the interface a default method is defined. This method returns an empty `Optional` to show that the calculation can not be applied.

False Alarm Rate Engine

There is an attribute `InspectionResult` on each test case node in the database, this value can either be `UNRESOLVED`, `FIXED` or `FALSE_POSITIVE`. Unresolved is set after the test has been run, the developer then has to inspect the test result and decide whether there is a bug or the test case itself is flawed. If a bug is discovered and fixed, the new status becomes `FIXED`, otherwise the new status is false positive. Since the paper on THEO does not give any information on how this should be handled, we decided to treat unresolved test cases as true positives, meaning they are treated the same like fixed bugs.

The alternative to treat them as false positvies would lead to a higher cost for execution and it would be less likely for the test to be executed. The query to retrieve all previous test executions for a test method with a specified output is shown in listing 4.18.

```
Match
    (a:TestRunEntity)-[b:RUN]->(c:TestSuiteEntity)-[d]
                        ->(e:testClassEntity)-[f:METHOD]
                        ->(g:TestMethodEntity)
WHERE
    e.canoncialName='de.rwth.swc.'+'
            'examples.triangletester.tests.TestTriangle' AND
        g.outcome='SUCCESS' AND
        g.canoncialName='WhenTriangleIsScalene'
RETURN g;
```

Source Code 4.18: Query for False Positives

THEO Xml Parameter Engine

The THEO Xml parameter engines allocates constant parameters for the THEO strategy. The listing 4.19 gives an example of a configuration. In chapter chapter 3 a introduction on constant parameter values is given. Opposed to the annotation parameters these are note bound to a certain test case.

```
<theo>
    <hourly_cost>0.027</hourly_cost>
    <cost_escaped>50</cost_escaped>
    <cost_inspection>25</cost_inspection>
    <time_delay>10</time_delay>
    <number_of_engineers>10</number_of_engineers>
</theo>
```

Source Code 4.19: ExampleXMLStrategy.xml

To read the xml file we used Castor which usage is described in section 4.1.2. The domain model to map this file consists of one file called TheoParameter.

Cost Engine

The cost engine finally calculates the aggregated values $cost_{skip}$ and $cost_{execution}$, therefore using the necessary analysis engines. The calculated measures are returned as a a object of type `Measure` in the case of the cost engine this measure is of the type `CostMeasure`.

This cost measure holds a `CostValue` in the value attribute. A `CostValue` consist of two float values, the skip cost and the execution cost. These values are accessed in the optimisation strategy.

Annotation Parameter Engine

To implement the annotation parameter engine, the Lazzer test suite description needed to be updated. As a reminder, Lazzer uses the JUnit framework to discover test cases. The resulting model is described in chapter section 4.1.1. This model already contains annotations to classes and methods. They need to be extracted to the Lazzer model. The main task of the analysis engine is to filter out Lazzer annotations and get their values. Annotations are defined like depicted in listing 4.20.

```
1  package de.rwth.swc.lazzer.framework.api.annotations. ...;
2
3  import ...
4
5  @Retention(RetentionPolicy.SOURCE)
6  @Target(ElementType.METHOD) //on method level
7  public @interface LazzerCostSkipAnnotation {
8
9      String value() default "";
10     String tags() default "";
11 }
```

Source Code 4.20: LazzerCostSkipAnnotation.java

CostBranchEngine

The `CostMasterBranchEngine` uses the Git engine to check which branch is currently used. The engine is depended on the branching schema that is used by a development team. A popular branching schema is Git flow. Part of its idea is that the there is a master branch that holds code ready for release. Development is made on the development branch. THEO requires a similar branching schema (cf. [Her+15].

The master branch holds code that is ready for release and if a test fails on that branch, a bug almost reached production use. Consequently, less test cases should be skipped on the master branch. The cost branch engine calculates an additionally value that is higher when a test is about to be skipped on the master branch. Consequently less tests are skipped on the master branch in comparison to the development branch.

Strategy Parameter Engine

The strategy parameter engine allows that two cost function in addition to fixed values can be configured. Listing 4.21 gives an example of how such a configuration file may look like. The static project values hold fixed values, e.g., the machine cost per hour. Furthermore, cost of skip and of execution are separately defined. Each cost function can hold some cost values which are static values. Further, analysis engines can be added, those have a class as parameter. The class name is used to instantiate a engine which is then used to return a numeric value. Each of those is then summed to obtain the cost value.

```
<strategy>

    <static_project_values>
        <static_project_value>
            <name>{value_name}</name>
            <value>{value_amount}</value>
        </static_project_value>
        ...
    </static_project_values>

    <cost_skipped>
        <cost_values>
            <cost_value>
                {cost_name}
            </cost_value>
            ...
        </cost_values>

        <analysis_engines>
            <analysis_engine>
                {class_name}
            </analysis_engine>
            ...
        </analysis_engines>

    </cost_skipped>

    <cost_execution>
        <cost_values>
            <cost_execution>
                {value_name}
            </cost_execution>
            ...
        </cost_values>

        <analysis_engines>
            <analysis_engine>
                {class_name}
            </analysis_engine>
            ...
        </analysis_engines>
    </cost_execution>
</strategy>
```

Source Code 4.21: ExampleXMLStrategyTemplate.xml

Metric-Measure Engine

The metric-measure engine uses the strategy parameter engine to get a strategy configuration. First static parameters are read, and then the analysis engines defined in each cost function are instantiated, using the dependency context. Finally, the calculate methods of those analysis engine are called for the parameter test method. Afterwards the resulting cost values are returned and can be used in a strategy.

4.2.3 Strategy Implementation

This chapter will cover the realisation of three strategies, which have been implemented.

Detection Rate Prioritisation

The prioritise detection rate implements a `AbstractComparatorPrioritization` and uses the detection rate engine to calculate values to prioritise test cases with a high detection rate. This strategy is mainly used to create a minimum working example of how analysis engines can be used. A drawback of this strategy is that test cases that are flawed also have a high detection rate, since they always failed, which can not be differentiated from a false positive at this point. Currently the only possibility is to calculate the detection rate for classes, since that suffices for a minimum example.

THEO

The THEO strategy is implemented as an implementation of the `Optimisation -Strategy` interface. The method `optimise()` is called in Lazzers optmisation stage. Lazzer then accesses the cost engine to calculate the cost values. Finally, the selection of test cases within the `THEOStrategy` class, by comparing the $cost_{execution}$ to the $cost_{skipped}$.

The Git engine is used to determine the development branch and if that branch is called 'Master' all test cases are run. This is similar to the quality guards of the original THEO strategy.

Metric Based Parameter Strategy

This strategy is developed similar to the THEO implementation. The metric-measure engine is used to calculate two cost values. Those are then compared to decide whether a test case is run or not. Additionally, methods are prioritised using their gain $g_{execution} = c_{execution} - c_{skip}$, which also has been defined in section 3.4.3.

Special about this strategy is that it also uses annotation parameters. It accesses the annotation engine for a priority, which is used for prioritisation when two methods have the same gain. Additionally, classes can also be skipped completely with the LazzerSkipAnnotation.

5 Evaluation

Contents

This chapter will cover the evaluation of the Lazzer software. First, an analysis of the requirements is made. Then, a short introduction to the example projects is given. Finally, an analysis of the stability and performance is evaluated for one of the example projects

5.1 Fulfilment of Requirements

Generally, it can be said that the requirements, as stated in section 3.2, have been fulfilled, since a metric-based strategy was successfully implemented. Nevertheless, there are certainly some improvements that can be done, but could not be implemented, because of the limited time given.

An user interface to display the test history would be a nice feature, telling the user about reasons why a strategy performed a certain prioritisation or selection. Even a very basic but individual logging for each strategy would help usability. Also, currently, there is no convenient possibility to insert the inspection result. For testing purposes a script has been implemented, that randomly sets the inspection result. Most certainly developers will not resort to manual database queries to input these values. An integration to a commonly used bug tracking system, e.g., Bugzilla, would be the perfect solution.

Furthermore, performance of graph databases can be optimised by indexing certain nodes. Indexing is described in this thesis but has not bee implemented. Anyone how wants to use this implementation should add these indexes to their database. Another point regarding data storage may be that currently each database stores the data of one project, it would be useful if one database could be used for multiple projects.

5.2 Example Projects

At the time this thesis was written, two example projects were available for the Lazzer framework. One is a minimum example of how Lazzer can be run using its API. The other one tests how Lazzer can be run using the Lazzer Maven plugin. There are several test cases implemented for both projects.

Those test projects are fine as long as to test whether a test method executes. But since there are no code changes, the initial test result will not change, unless code is changed manually. As a consequence test cases never fail. Since those projects seemed insufficient to test a metric-based strategy, because of those limitations, a new test project was implemented.

The main requirement for that test project was that it delivers a somehow realistic test history. That is to say that test cases realistically fail and succeed in a alternating fashion. Also it would be nice to see how the optimisation strategies after a certain amount of time gives some test cases a higher priority.

The solution was to use random numbers to let test cases fail and succeed regularly. Some test cases succeed on a smaller range of numbers than others, such that different heuristic values are found. Even though these results do not have the same significance, as test on a real project would, these can still be used to test if the strategy works.

5.3 Performance and Stability

To evaluate whether the updated Lazzer performs stable the analysis engines are modified in a way that they calculate all possible values for the test suite. That would not be necessary in a real scenario, because most of the time not all values are needed. Then the THEO strategy was executed 500 times on a single database. The strategy was run on the random example project. The result is shown in figure 5.1.

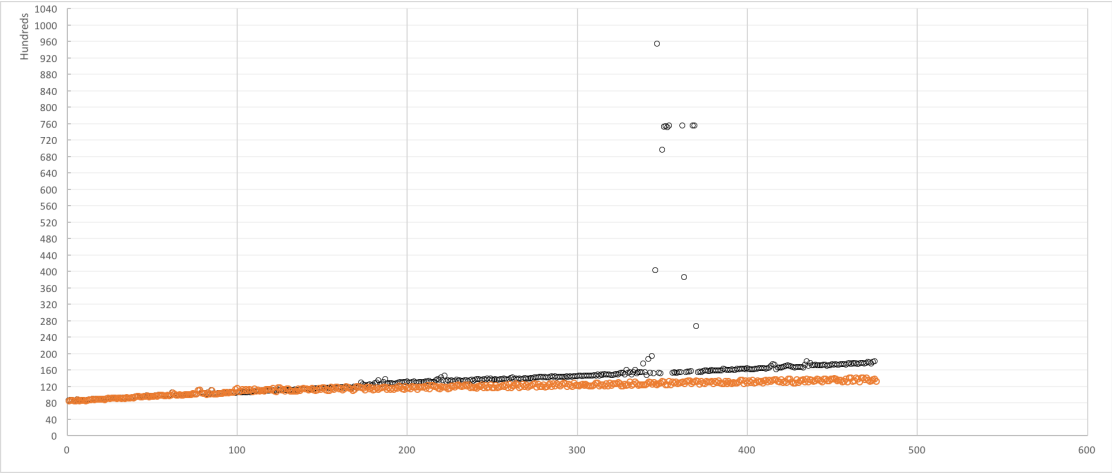

Figure 5.1: Graph that illustrates Lazzer runtimes.

Black dots represent the runtime of Lazzer, having no limit to how many test runs in the past are considered, while the orange dots represent a limit of 100. The results show that runtime increases linear without a limit. It is expected that the runtime increases the more history values are considered, but a linear increase seems satisfactorily. Also ignoring some outliers the runtime has a low variance. To limit the runtime it suffices to decrease the number of test cases considered.

Some test suite have a execution time of multiple days. Considering a linear increase of runtime, such a big test suite could still be optimised in the matter of several minutes. Overall the performance is adequate, because a selection process most probably will save up a multitude of that time.

6 Conclusion

Contents

This chapter, first, gives a summary of the contributions of this work and then hints at possible topics for future work, especially regarding further improvements.

6.1 Summary

This work took the Lazzer framework as a starting point and performed a refactoring and extension with the purpose of implementing metric-based strategies. Those strategies are mainly founded on the research about the THEO strategy, by [**Herzig2005**] et al.

The main contributions of this thesis are: separation of data access and evaluation, support for a graph database, parameterisation for strategies and the implementation of strategies. Recursive analysis engines are realised and used to calculate metric values efficiently. Furthermore they realise the separation of data access and evaluation.

A graph database adapter was implemented to enhance how test results are stored. Further an adapter was implemented to store metric values along with the testhistory. Parameterisation has been provided using two ways: An analysis engine has been written that consumes xml files. Additionally, the annotation engine has been written that is able to return values that are linked to test cases by annotations in the test code.

Finally, three strategies have been implemented. One of those utilises the detection rate engine to prioritise test cases with a high detection rate. The other closely resembles THEO and a third one which is similar to THEO, but provides more possibilities for parameterisation, including the definition of individual cost functions.

6.2 Future Work

This thesis prepared the Lazzer framework for metric-based strategies. Consequently, some future work can be done on implementing other metric-based strategies. Furthermore, the implementation of a graph database may offer other possibilities.

1. **Real world evaluation**
   The implemented strategy was not tested on a real project. The demo projects may deliver a hint at the performance, but that does not suffice make a reasoned statement about the benefit of a implemented strategy.

2. **Other metric-based strategies**
   Other metric-based strategies could be implemented using Lazzer. There is a variety of papers that have been published on such strategies. If multiple strategies are implemented for Lazzer, an empirical performance evaluation would be interesting.

3. **Depndency graph strategy**
   A strategy that uses a genetic algorithm on dependency graphs was proposed by [Mus+14]. The strategy could utilise the database adapter for a graph database to store dependency graphs.

Bibliography

[AC91] D. Applegate and W. Cook. “A computational study of the job-shop scheduling problem”. In: *ORSA Journal on computing* 3.2 (1991), pp. 149–156 (cited on pages 2, 6).

[BM07] R. C. Bryce and A. M. Memon. “Test suite prioritization by interaction coverage”. In: *Workshop on Domain specific approaches to software test automation in conjunction with the 6th ESEC/FSE joint meeting - DOSTA '07* (2007), pp. 1–7. ISSN: 0975-6809. DOI: 10.1145/1294921.1294922. URL: http://portal.acm.org/citation.cfm?doid=1294921.1294922 (cited on page 2).

[CRK10] A. P. Conrad, R. S. Roos, and G. M. Kapfhammer. “Empirically studying the role of selection operators duringsearch-based test suite prioritization”. In: *Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10* (2010), p. 1373. DOI: 10.1145/1830483.1830735. URL: http://portal.acm.org/citation.cfm?doid=1830483.1830735 (cited on page 1).

[Do+10] H. Do et al. “The Effects of Time Constraints on Test Case Prioritization: A Series of Controlled Experiments Software Engineering”. In: *IEEE Transactions on Software Engineering* 36.5 (2010), pp. 593–617 (cited on page 5).

[EMR02] S. Elbaum, A. Malishevsky, and G. Rothermel. “Test case prioritization: a family of empirical studies”. In: *IEEE Transactions on Software Engineering* 28.2 (2002), pp. 159–182. ISSN: 00985589. DOI: 10.1109/32.988497. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=988497 (cited on page 2).

[Fow06] M. Fowler. *Webiste of Martin Fowler: Article on Xunit*. Online, last accessed 2015-11-16. 2006. URL: http://www.martinfowler.com/bliki/Xunit.html (visited on 11/13/2015) (cited on pages 1, 24).

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0201485672 (cited on page 23).

[GIP11] P. Gupta, M. Ivey, and J. Penix. *Testing at the speed and scale of Google*. \url{http://google-engtools.blogspot.de/2011/06/testing-at-speed-and-scale-of-google.html}. 2011. URL: http://google-engtools.blogspot.de/2011/06/testing-at-speed-and-scale-of-google.html (visited on 10/26/2015) (cited on page 6).

[Gli] M. Gligoric. *Website of Ekstazi.* \url{http://www.ekstazi.org/}; Online, last accessed 2015-10-26. URL: http://www.ekstazi.org/ (visited on 10/26/2015) (cited on pages 1, 5).

[Gor+08] R. P. Gorthi et al. "Specification-based approach to select regression test suite to validate changed software". In: *Neonatal, Paediatric and Child Health Nursing.* 2008, pp. 153–160 (cited on page 6).

[Gra+01] T. L. Graves et al. "An Empirical Study of Regression Test Selection Techniques". In: *ACM Transactions on Software Engineering and Methodology* 10.2 (2001), pp. 184–208. ISSN: 1049-331X. DOI: 10.1145/367008.367020. URL: http://portal.acm.org/citation.cfm?doid=367008.367020 (cited on page 6).

[Ham] *Hamcrest.* URL: https://github.com/hamcrest/JavaHamcrest (cited on page 23).

[Her+15] K. Herzig et al. "The Art of Testing Less without Sacrificing Quality". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering.* 2015, pp. 483–493. ISBN: 9781479919345. DOI: 10.1109/ICSE.2015.66 (cited on pages 1, 2, 5–7, 37).

[Jyo14] K. S. Jyoti. "A Comparative Study of Five Regression Testing Techniques : A Survey". In: 3.8 (2014), pp. 76–80 (cited on page 2).

[Lea10] N. Leavitt. "Will NoSQL Databases Live Up to Their Promise?" In: (2010), pp. 12–14 (cited on page 11).

[MB] R. Mccoll and D. A. Bader. "A Performance Evaluation of Open Source Graph Databases". In: (), pp. 11–17 (cited on pages 12, 16).

[Mus+14] S. Musa et al. "A Regression Test Case Selection and Prioritization for Object-Oriented Programs using Dependency Graph and Genetic Algorithm". In: *International Journal of Engineering And Science* 4.7 (2014), pp. 54–64 (cited on page 46).

[Neo] *Neo4J Developers Guide.* URL: https://neo4j.com/developer/ (visited on 01/01/2016) (cited on pages 29, 31).

[Ors+01] A. Orso et al. "Using component metacontent to support the regression testing of component-based software". In: *IEEE International Conference on Software Maintenance, ICSM.* 2001, pp. 716–725. ISBN: 0-7695-1189-9. DOI: 10.1109/ICSM.2001.972790 (cited on page 1).

[Ple15] C. Plewnia. "A Framework for Regression Test Priorization and Selection". Master Thesis. Germany: RWTH Aachen, 2015 (cited on pages 2, 8).

[Spa] *Spark.* URL: http://spark.apache.org (cited on page 5).

[SSS11] S. Sabharwal, R. Sibal, and C. Sharma. "Applying Genetic Algorithm for Prioritization of Test Case Scenarios Derived from UML Diagrams". In: *International Journal of Computer Science Issues* 8.2 (2011), pp. 433–444 (cited on pages 2, 6).

[Sur] *Website of Surefire Maven Plug-in.* \url{http://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html}; Online, last accessed 2015-12-03. URL: http://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html (visited on 12/03/2015) (cited on page 9).

[TTL89] A.-B. Taha, S. M. Thebaut, and S.-S. Liu. "An Approach to Software Fault Localization and Revalidation Based on Incremental Data Flow Analysis". In: *COMPSAC 1989: The 13th Annual International Computer Software and Applications Conference, Vol I, Proceedings* (1989), pp. 527–534. DOI: 10.1109/CMPSAC.1989.65142 (cited on page 6).

[Vic+] C. Vicknair et al. "A Comparison of a Graph Database and a Relational Database A Data Provenance Perspective". In: () (cited on page 11).