JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Technisch-Naturwissenschaftliche
Fakultät

Unit Tests From Runtime Observations

BACHELORARBEIT

(Projektpraktikum)

zur Erlangung des akademischen Grades

Bachelor of Science

im Bachelorstudium

INFORMATIK

Eingereicht von:
Benedikt Aumayr, 0855608

Angefertigt am:
Institute for Systems Engineering and Automation

Beurteilung:
Univ.-Prof. Dr. Alexander Egyed M.Sc.

Linz, November 2012

Abstract

Automatically creating unit tests from runtime observations is a practically-relevant technique for making software testing more efficient. The primary application for unit testing is regression testing where we want to ensure that software does not change its behavior during its evolution - or does change, depending on intention.

This thesis implements an approach for generating unit tests by observing program executions such as system tests. The major challenge for making such runtime observations is minimizing the amount of state information that needs to be maintained to ensure a correct functioning of individual unit tests. Maintaining state is important because unit tests are typically randomly executed, which is contrary to the specific order of unit execution during system tests. So, for example, if during a system test a given stack contained a specific value during retrieval then this specific value (aka state) must be restored for unit testing prior to retrieval to ensure that the unit test corresponds to the system test. The goal of this work is reducing the overhead of capturing and maintaining such state. As such, for example, not the entire stack state much be restored prior to retrieval.

Several approaches have already been proposed that help in the capture and maintenance of state for unit testing. However, the majority of them rely on instrumentation techniques. We analyze the feasibility of gathering test data for unit tests by observing system tests with a high-level debugging technology, the Java Debug Interface, and compare it to the existing approaches. A prototypical implementation of our approach is used to evaluate this choice of technology.

Contents

1 Motivation **2**

1.1 Definitions 2

1.1.1 Levels of Software Testing 2

1.1.2 Regression Testing 3

1.1.3 Differential Testing 4

1.1.4 Continuous Testing 4

1.1.5 Random Testing 4

1.1.6 Symbolic Execution 4

1.1.7 Capture and Replay 5

1.2 Goal 6

1.3 Contributions 8

2 Taxonomy of Unit Tests **9**

2.1 Dependency on State 9

2.2 Evolutionary Aspects 10

3 Related Work **12**

3.1 Elbaum *et al.* 12

3.2 Orso *et al.* 13

3.3 Saff *et al.* 14

3.4 Other Approaches 14

4 Approach **16**

4.1 Principles 16

4.1.1 Capture 16

4.1.2 Replay 18

4.2 Implementation 19

4.2.1 Architecture 19

4.2.2 Data Model 23

4.2.3 GUI 26

4.3 Evaluation 27

5 Conclusion **30**

Bibliography **31**

List of Figures **34**

List of Tables **35**

Chapter 1

Motivation

1.1 Definitions

In this section, several fundamental terms in the field of software testing are explained in order to define the scope of this work and to outline (certainly in a very shortened and not complete way) the state-of-the-art in software testing.

1.1.1 Levels of Software Testing

Unit Testing

Unit testing is the verification of a subset of a software, which can be tested in isolation of the remaining parts (see figure 1.1). It normally is performed with access to the source code (white box testing) and using debugging tools [1]. Compared to other testing levels such as *integration testing* and *system testing*, unit testing can be considered as a rather low-level technique since it typically concentrates on small parts of the system.

Developers commonly use tools like JUnit to automate the process of unit testing. Unit tests are typically written using a framework of such a tool and consist of three parts [2]:

1. Generating the necessary input values
2. Calling the unit under test
3. Finally defining the expected results/outputs and checking them

Unit tests are also a fundamental technology of several development methods like *Extreme Programming* or *Test-driven Development* [3].

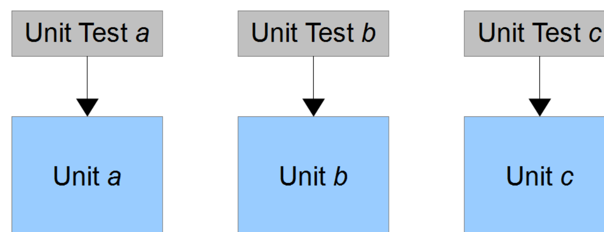

Figure 1.1: Unit Testing

Integration Testing

After verifying the functioning of individual components during unit testing it is necessary to check whether they collaborate and interact as desired (as shown in figure 1.2). There are different

approaches for conducting the integration of the components and testing their interaction such as top-down, bottom-up or "big bang" and it heavily depends on the architecture of the software which one to use. [1]

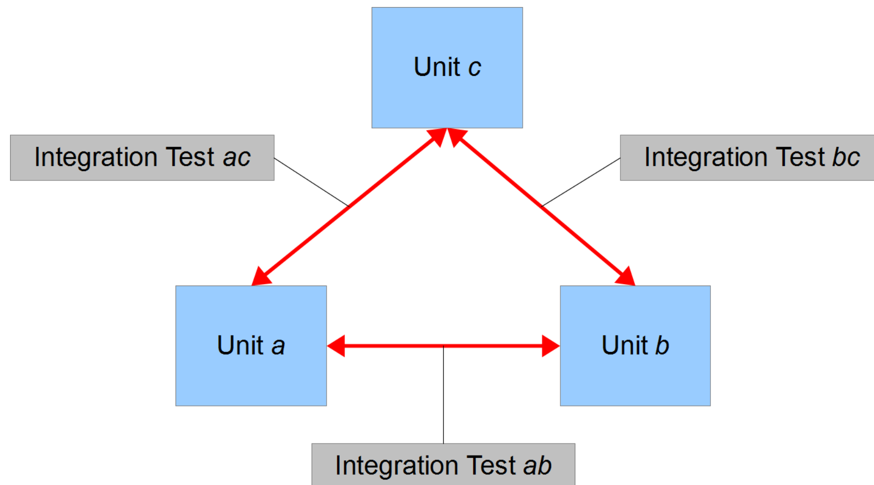

Figure 1.2: Integration Testing

System Testing

The aim of *system testing* is to examine the behavior of a software system as a whole (as shown in figure 1.3). Based on the assumption that major functional flaws have already been detected during unit and integration testing, system testing is also appropriate for verifying non-functional requirements such as reliability, security or performance. Interfaces to external applications or the operating system are included in this stage of testing as well. [1]

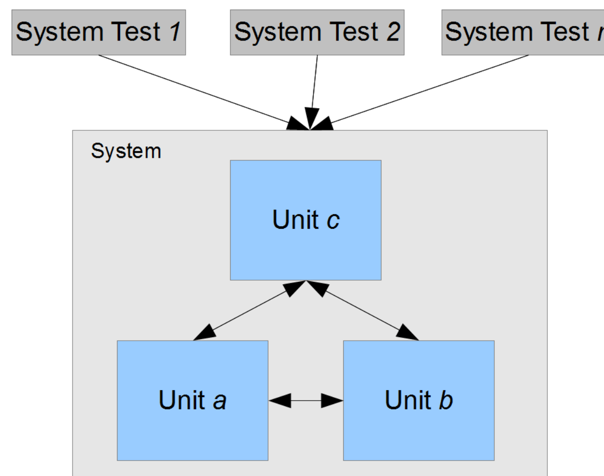

Figure 1.3: System Testing

1.1.2 Regression Testing

Regression testing has its purpose in verifying that modifications of a software have not caused unintentional side-effects by retesting certain parts of the system. This is done by repeating tests which passed on a previous version on the modified version in order to show that behavior of the system remained unchanged except for parts where changes are expected. [1]

1.1.3 Differential Testing

If different implementations of a software exist, it is essential to know whether they behave in the same way. In general, these implementations could actually be developed independently from each other or (in a simpler case) just be different versions of the same implementation. Anyway, differential testing aims at running as many tests as possible on all copies of the software in order to point out possible differences in behavior. If all exemplars of the software are relatively usable and stable, only few differences are supposed to be found. Therefore, a major requirement is to reduce the effort of manual test evaluation by just identifying the actual differences. In case of multiple versions of one implementation this method is obviously quite applicable for regression testing. [4]

1.1.4 Continuous Testing

Modern integrated development environments (IDEs) immediately notify a programmer of syntax errors or other compilation problems. This usually happens already while editing or at least after saving. Similar to that, continuous testing is a method for frequently running specific regression tests while editing source code [5]. Those tests can provide quick feedback to the programmer whether any regression bugs have been introduced. It is obvious that this is only practical and beneficial with fast and focused unit tests.

1.1.5 Random Testing

A relatively simple method to automatically generate a large number of test cases is *random testing*. It randomly chooses input values for a software under test from the *input domain* (all possible input values). Amongst its advantages are the facts that it is cheap, does not even need to know about the specification (how the software is supposed to behave) and that is a good and easy way to simulate “chaos”, which also occurs during real use in the field. Therefore, it is widely used in software industry. [6]

The idea of *Adaptive Random Testing* [7] is to use input values from the input domain which are more evenly spread than purely random chosen values. For non-clustered failure patterns this increases the *fault detection effectiveness* and because adaptive random testing is not really more complex than random testing it is an practical replacement for it [6].

Another interesting technique based on random testing and symbolic execution is *Directed Automated Random Testing (DART)* by Godefroid *et al.* [8]. It performs random tests while simultaneously observing the program (dynamic analysis) in order to create new test inputs, which cover additional execution paths of the program code. Somewhat related to DART is *Feedback-directed Random Test Generation* [9] which uses the response of the tested program to input values (like them being invalid) for pruning the search space of input values, hence making the testing more efficient.

1.1.6 Symbolic Execution

Symbolic Execution of programs [10] is a practice that somehow lies in between software testing and proving (but is still closer to testing). Instead of executing a program with actual input values the input space is grouped into classes by applying symbolic expressions (formulas over the input symbols) in order to cover all possible execution paths of the program. The possibly exponential number of paths causes scalability issues which require sophisticated approaches.

During symbolic execution, variables are kept track of in form of symbolic expressions and a path constraint is built. When conditional statements occur, symbolic execution is able to proceed with both branches and just stops at paths which become unreachable. The number of paths could become infinite due to loops or recursions in the program. Therefore, measures to limit the search are necessary. [11]

Sen [12] proposed a technique that combines concrete (by means of concrete input values) and symbolic execution calling it *Concolic Testing*. It uses both random testing and symbolic execution to overcome shortcomings of either methods.

1.1.7 Capture and Replay

Capture and Replay has proven to be one of the most important techniques to increase test efficiency (especially for creating test inputs) [2]. An early attempt to use it for testing in general was proposed in [13]. On an abstract level, there are basically two different approaches for capturing and replaying program executions [14]:

- "Content-based" capture and replay:

  All "data" used by a program (or just a single thread of it) is stored in a trace file. It is obvious that this is very expensive and generates very large trace files. However, the advantages are that it makes it possible to replay just parts of the program (possibly out of order) or for example just one specific thread.

- "Order-based" capture and replay:

  This method only stores the order in which the program uses data (i.e. accesses memory). If the necessary initial input and order of processing (for example in the form of method calls) can be replayed then this should lead to an equivalent program execution but needs less trace information. However, replaying is limited to the exact sequence as it occurred during the capture.

Capture and replay can be used for test generation as follows: During the *capture* phase, all interactions between a defined *unit under test* (see chapter 2) and its environment (normally meaning the remaining parts of the system) while performing a system execution (e.g. a system test) are recorded. Based on the observed interaction it is possible to create unit tests for the monitored unit under test. This is done by connecting the input values for method calls on the unit under test and the corresponding return values using them as test oracles. The *replay* phase more or less simply consists of rerunning the unit tests using the input values for method invocations and verifying the test oracles. [2]

This technique offers two major advantages: First of all, running the created unit tests is much faster than automatically executing a whole system test because it only focuses on specific units under test. Compared to manually executing system tests, it is much faster of course because it requires no manual effort. Secondly, it makes an automatic generation of unit tests possible contrary to writing them manually. It is especially suitable for regression testing because you can rerun unit tests which were created based on an old version of a software on a new version of it. However, there is also a downside related to regression testing: you cannot be sure that the previously captured behavior was actually correct since the unit might have been broken. [2]

Another widespread application of Capture and Replay is to use it for the purpose of debugging i.e. to find respectively reproduce faults in software like in [15, 16]. Xu *et al.* have also used it for checkpointing at language level providing an efficient way to restore an application after faults and supporting debugging them [17].

Georges *et al.* [14] presented a system called *JaRec* that aims at debugging of problems which are hard to reproduce due to non-determinism caused by multi-threaded implementation respectively execution. In order to minimize the induced overhead it only records areas of synchronization between threads. JaRec is implemented by means of instrumentation at runtime based on the *Java Virtual Machine Profiler Interface* which was part of previous versions of the Java Development Kit but was marked as deprecated and finally removed.

1.2 Goal

Developer testing using manually written and maintained unit tests is cost-intensive, tedious and often insufficient in covering all parts of a system [2]. Modern software systems tend to be increasingly configurable and universal, making quality assurance even more difficult [18]. Therefore, every possible automation in developer testing is desirable.

The goal of this thesis is to automatically create small, focused and fast unit tests gathered from (possibly) longer system tests (either manually or automatically executed) and use them for regression testing. Figure 1.4 provides a graphical sketch of these objectives. Capturing such executions tends to be easier than writing unit test cases [19]. Primary technique to get the necessary test inputs and oracles is a dynamic observation and not any kind of static analysis of source code. Random testing or symbolic execution are also not applied in this approach.

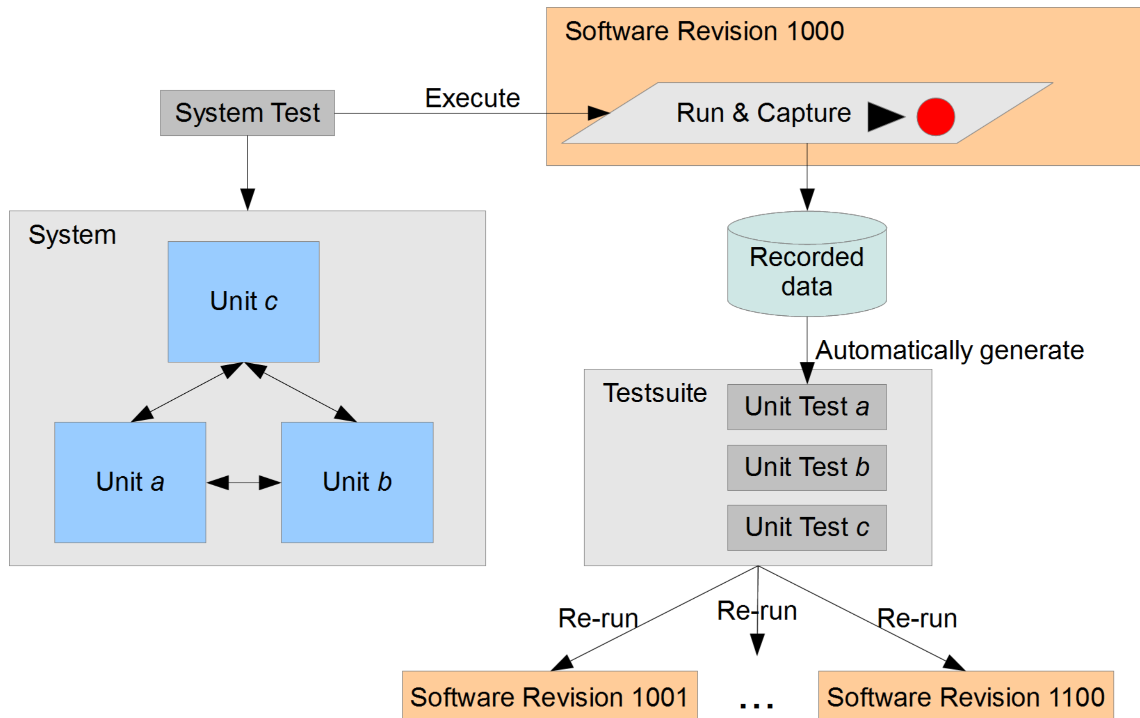

Figure 1.4: Generating Unit Tests from Executing System Tests

The major challenges and objectives for such an approach are:

- System test execution overhead

  The additional effort of time and resources induced by the observation of system tests should be as little as possible. This means that a system execution that is observed should not take much longer or use a lot of more memory than an execution that is not observed.

- Amount of captured data

  Only data which is absolutely required to ensure a correct replay in form of a unit test should be recorded. The amount of data created and used during program executions can be extremely high and hence it is very important to find out which data is needed and to limit the recording accordingly. As the code example in figure 1.5 shows, it is likely that certain executions (e.g. single method calls) only require a subset of the overall state of the program and therefore can be replayed just by restoring this reduced subset. In the example, that part of the program state is highlighted by yellow framing and in some cases (e.g. the collection objects) even only parts of these objects are necessary. Interesting work regarding this matter was also published by Lee *et al.* [20].

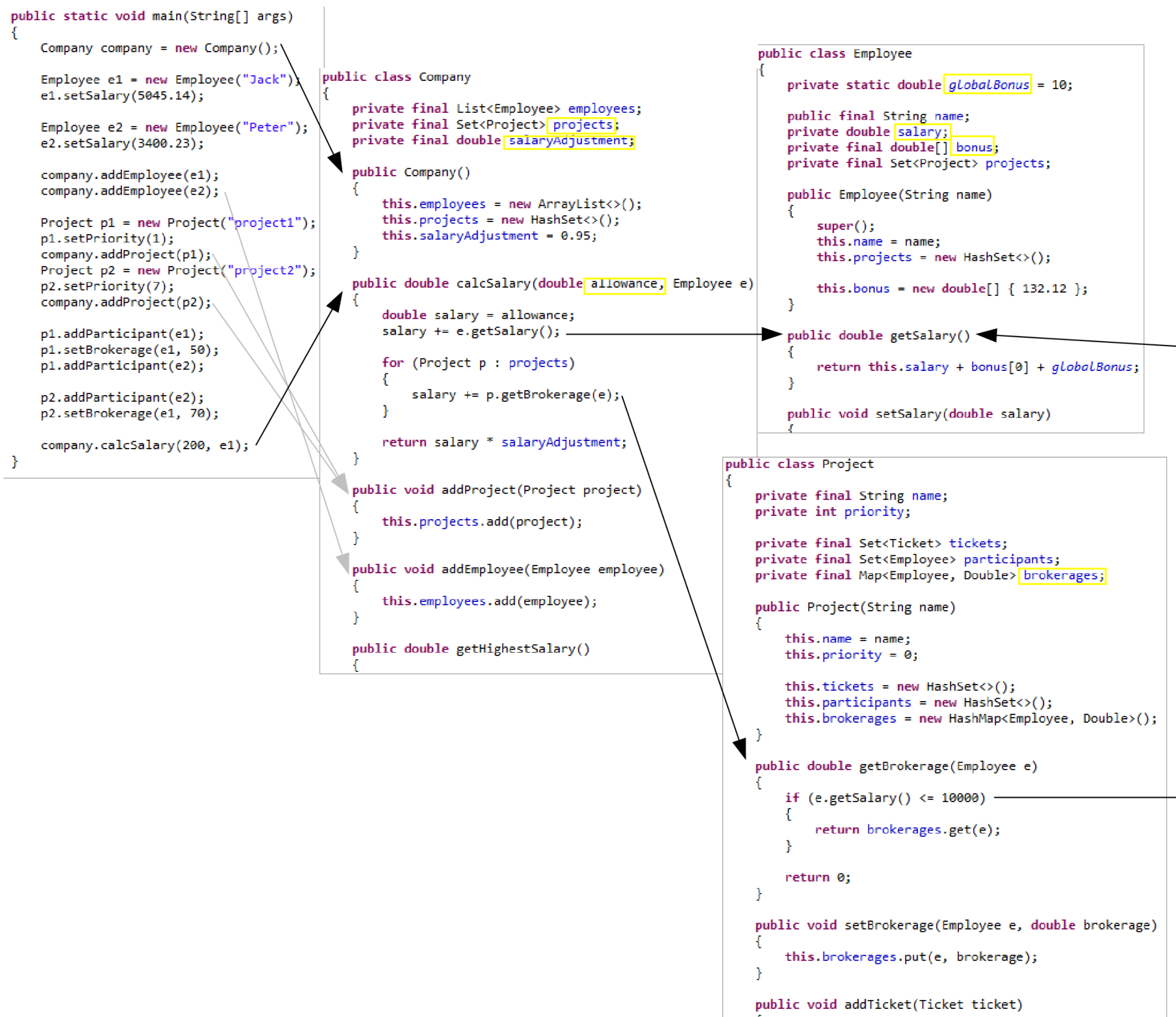

Figure 1.5: Code example showing amount of required state

- Durability (resistance to changes of the software)

  The created unit tests should be runnable on as many newer, modified software versions as possible. During evolution of software, interfaces may change meaning that methods are removed or their arguments are changed (arguments are added or removed). Therefore it is not always possible to replay previously recorded calls to changed methods.

- Lenience towards internal implementation changes

  A new version of a software could feature minor changes regarding the internal implementation which do not affect the behavior as it is seen from the outside (hence the interface is kept stable). For example, internal fields could be added, removed or changed in type. In this case, unit tests are supposed to lead to same results on both the old and the new version. However, the replay mechanism is prone to check too much of internal details if it for example compares the captured and the replayed version regarding the state of the program after execution. Therefore, it can be helpful to ignore minor differences in the expected resulting state.

1.3 Contributions

This thesis presents the following contributions:

1. It surveys the state-of-the-art in software testing regarding the creation of unit test cases from system executions.
2. The feasibility of performing such an automated creation using the high-level technology *Java Debug Interface* (see 4.2.1) is evaluated.
3. A corresponding prototypical implementation of the developed approach is provided.

Chapter 2

Taxonomy of Unit Tests

In this chapter, a classification of unit tests regarding their dependency on other units or usage of state information is presented and the implications of the different cases to the capture and replay processes are discussed.

2.1 Dependency on State

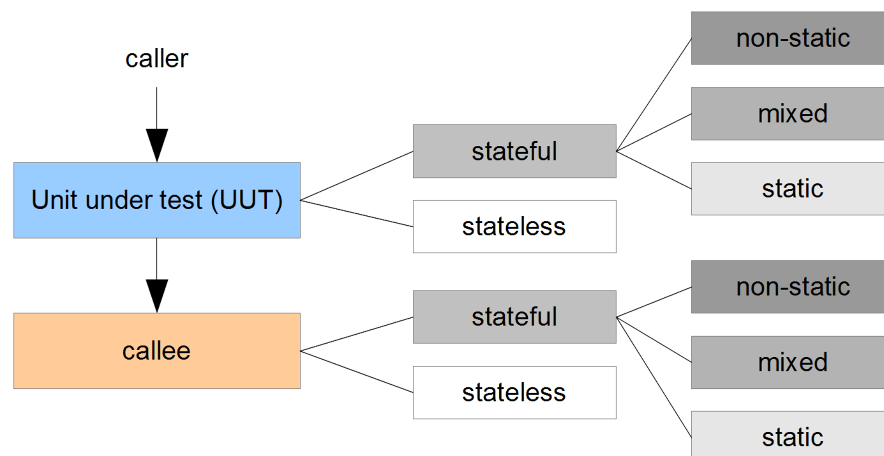

Figure 2.1: Classification of unit tests regarding their usage of state

As shown in figure 2.1, we have basically three scopes to consider while writing respectively executing or observing unit tests. First of all, there is of course the *unit under test* which is the current area of code that is tested by a certain unit test. In object-oriented programming this is typically one class but could also be a module or just a distinct method. Sometimes this is also referred to as *testee*. Moreover, there is the *caller* which calls methods of the unit under test. Another term for this is *test driver*. This may be an actual unit test specifying a certain sequence of test calls or a call sequence that happens during the execution of a system test or real use of the software. Hence, during capturing we have to look at the calls which were invoked on an object of the unit under test.

The ideal case is that the unit under test is independent from other units, which means that it does not call or use other parts of the system. Consequently, for replaying a certain unit test you only need the data which was directly accessed by the unit under test. In terms of Java (or other object-oriented languages) this *state* of the unit is manifested in the values of the fields of a class. However, in many cases the unit under test will depend on other units or an *external world* in general because there can be long chains of dependencies in the external scope. An implication of this is that during the execution of a specific test the unit under test will use state

from an external scope and it is necessary to capture all the required state in order to be able to restore it while replaying. The state of the unit under test or an external unit can be static or non-static in the sense of object-oriented programming languages. Static state is valid for all objects of a class whereas non-static state only applies to a distinct object. Of course a mixed combination is also possible.

Regardless whether a unit under test depends on external units or not, a unit itself can also be completely stateless. If there is neither internal nor external state accessed (read or written) during the execution of a unit test then there is no need to restore anything before replaying. It is only required to pass the same arguments as they were used in the captured calls. However, if any usage of state is observed during capturing, a method of the UUT can be considered stateful. In other words, a unit under test is stateful if it accesses internal state itself or external state is accessed during the usage of dependencies.

As a consequence of that, you can treat each stateless method call on the unit under test completely isolated and out of context if you want to replay it. In contrast to that you have two possibilities to correctly replay stateful method calls on the unit under test: either replay the entire life of an object (all calls that had been made on it in the exact same sequence which should implicitly restore the required state) or explicitly restore the required pre-state before replaying a call.

2.2 Evolutionary Aspects

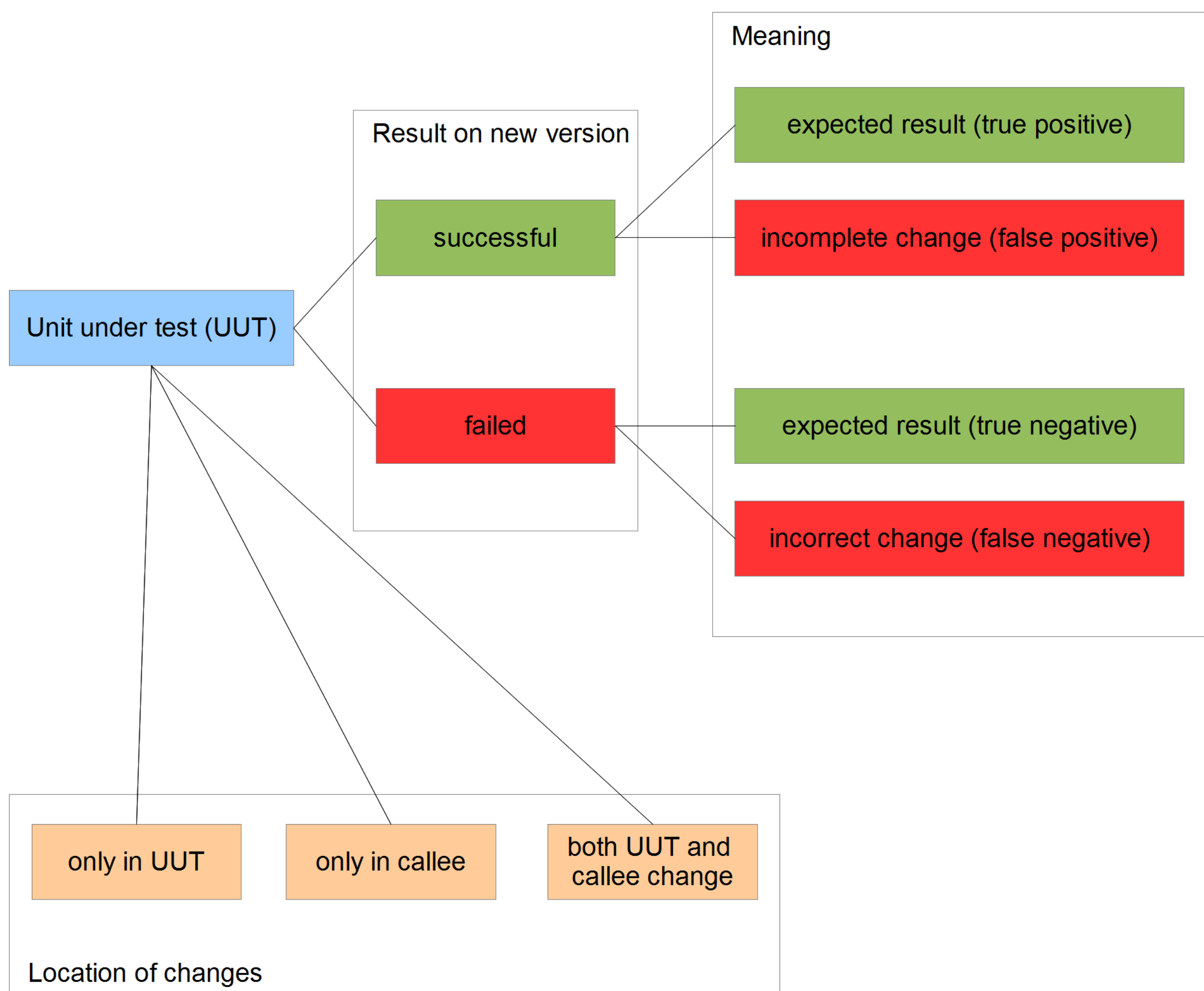

Figure 2.2: Classification of unit tests regarding evolutionary changes

Software is modified during its evolution (both in initial development or maintenance phases) in order to add functionality, fix bugs or just change its behavior. Developers want to ensure that the changes made in a new version of the software do not have unexpected impacts on the unchanged parts of the code. This is typically verified using a regression test suite that is rerun on the new version of the software. However, the ability to find regression faults heavily depends on the quality and coverage of those test cases [21].

Related to our scopes, if we compare a new version of a software to an older one, we can distinguish three cases where there was a change made: In the first case, only the unit under test changed. Second case is that there was a change in a callee the unit under test depends on. Additionally, it is possible that both the UUT and one or more callees were changed.

If the result of a unit test replayed on a new version differs from the original result (regardless whether successful or not) and the unit under test has not been changed but a callee has been modified then this indicates an unintentional side-effect (regression fault) of a change. However, if the unit under test itself has been changed and the replay results are different from the original ones, it may just show that the unit test has to be rewritten or in our case recaptured as well.

When running an existing test suite on a new version of a software we can also classify according to the expected result as shown in figure 2.2. If a test succeeds or fails that you expect to do so, everything is fine (true positives or negatives). However, if a test still succeeds although it should now fail due to a change, it means that the change is incomplete. On the other hand, if a test fails although it should still complete with success, the change was incorrect. These conclusions of course only hold assuming that the tests are correct.

Chapter 3

Related Work

In the following sections, the most closely related existing literature is discussed. Additionally, there is an overview on other sophisticated approaches in software testing given.

3.1 Elbaum *et al.*

In [22, 3] an approach was presented, which is very close to what we want to achieve. It also already focuses primarily on evolutionary aspects as it defines the term *differential unit tests* standing for unit tests for different version of a unit during software evolution. They call their technique *carving* and replaying, already indicating that only a minimum amount of state information is recorded.

Elbaum *et al.* introduce *state projections* in order to achieve two concerns: *test case reduction* and *test case filtering*. Test case reduction means that the state information that is stored is kept at as little as possible. This is done by only processing limited number of object instances which are reachable from the current context. Simplistic projections (like k-bounded reachable) are prone to omit objects which had potentially unmasked a bug. Several possible state projections are described:

- *k-bounded reachable*: In short, this projection simply reduces state by only traversing reference chains of a certain length k starting from the root object reference.
- *Touched-carving*: This projection uses dynamic information for determining which fields of object instances are read or written and is basically somehow what we want to achieve. Initially, it is based on the k-bounded reachable projection and introduces additional instrumentation in order to know which fields are actually referenced because that is of course unknown prior to execution.
- *May-reference*: Using static code analysis before execution, this method tries to estimate whether certain objects will be referenced during execution. Since we do not want to use static code analysis, this is not relevant for us.
- *Clustering*: They also describe a technique for identifying helper methods which are still public and therefore are part of the interface to test. Such a helper method could probably be invoked very often when it is used in an iteration of another method. Hence, a lot of test cases would be generated. However, by limiting to a certain maximum number of calls, such a method could be marked for *indirect* replay meaning that it will be tested by the replay of another method anyway.
- *Normalizing transient data*: This aims at normalizing data which is actually transient and therefore not relevant for serialization (e.g. the seed in `java.util.Random`).

In addition to that, test case filtering removes redundant test cases identified by the fact that they use the same pre-state and input values.

Regarding replay, the term of *differencing functions* is issued. Those differencing functions are projections of the post-state and the return values in order to use them as pseudo-oracle for verifying the success of replay. The key characteristic of such a differencing function is the ability to still detect discrepancies (fault detection capability) while ignoring implementation details. First of all, the projections described above can be applied. Secondly, for reducing the storage size of post-states (for checking the correct result on replay) they mention the idea of only storing hashes of serializations instead of storing the complete serializations. Although this is very efficient, it is also quite sensitive to internal implementation changes.

As a strategy for dealing with replay errors due to anomalies like modified internal data structures between software versions, they suggest to *recarve* a part of the original system test which provides a sufficient amount of new state information. In the worst case, this can be the whole system test of course.

Like other approaches (e.g. [23]), they use the Apache Byte Code Engineering Library[1] for instrumenting in order to capture respectively replay. For serializing objects they use the same library as we do (XStream).

In their case study they measured the size of the captured state information for the following state projections: 1-bounded reachable, 5-bounded reachable, ∞-bounded reachable (equals full state information), may-reference (needs additional runtime due to prior static analysis) and touched. While the size of 1-bounded reachable was a little less than the size of 5-bounded projection, there was no difference between $k=5$ and $k=\infty$ (this means that there were no reference-chains with $k>5$). May-reference and touched projection were basically in between $k=1$ and $k=\infty$, with touched requiring less amount than may-reference.

3.2 Orso *et al.*

SCARPE is an approach illustrated in [23] (based on previous work which was presented in [24, 25]), which is designed to capture and replay of deployed software in the field and makes it possible to automatically get test cases from end users. However, they also mention *user-based regression testing* as a possible (future) application, which is quite similar to the goal of this thesis. Additionally, an earlier approach which also already aims at using field data for regression testing has been shown in [26].

The technique captures only the state which affects the execution and captures at the boundaries of the observed classes. It needs preparation before capturing where it takes a user-provided list of classes to observe. Their technique keeps track of certain events (calls from observed to unobserved code and vice versa and the corresponding returns, write accesses from observed code to fields of unobserved code and vice versa) in an event log. Calls that do not cross the boundaries are ignored. The bottom line is that only scalar get actually captured but object values are only recorded by an unique identifier number and that only values which are used for exactly the sequence of calls are stored.

This is all done by inserting probes which write to the event log on capturing respectively read from the log on replay using the Apache Byte Code Engineering Library.

One difference to our approach is again that is uses explicit instrumentation (even for replay). On the other hand, it is sequence-orientated and therefore replaying is limited to the exact order of calls like it appeared on capturing. This makes it possible to minimize the required state to record because it will implicitly restored by just executing the same sequence of calls again. Our approach, however, is kind of item-orientated (where an item could be a class or method) which makes it possible to replay calls out of order.

[1] https://commons.apache.org/bcel

3.3 Saff *et al.*

In [27] a technique for *automatic test factoring for Java* using capture and replay is presented. The term *test factoring* was previously issued in [28], describing it as a method for creating fast and focused unit tests from slow system tests (which is the same what we want). The biggest motivation is to reduce the time effort for testing and consequently making it probably even fast enough for *continuous testing*. The major goal of their approach is to use capturing for the creation of mock objects because they remove expensive generations like database calls. In order to achieve this, the system is partitioned into the *code under test* and the *environment*. Hence, it keeps track of calls from the environment on objects from the code under test and vice versa. Only calls at the defined boundary are considered. In their implementation they use a rather complex instrumentation mechanism, which even requires a modification of the Java Runtime library. This is necessary to handle all kinds of features of Java like classloaders, native system calls and so on. This differs from our approach of course because the observation using JDI does not need explicit modification of neither the code under test nor the environment (including the Java class library).

The biggest problem of the approach considering our goal and regarding software evolution is that it only makes regression testing of changed code under test possible if the environment remained unchanged. However, when the environment has been changed as well, the previously captured mocks might not be valid anymore (as they represent an old version of the software) which could lead to unnecessary errors on replay. On the other hand, it is very efficient of course because the mocks replace a lot of (potentially expensive) calls to the environment.

3.4 Other Approaches

Object Capture based Automated Testing (OCAT) is an approach presented in [19]. Its purpose is to capture object instances from system testing or real use in order to use them as input for test generation tools which are based on *random testing*. It utilizes object capturing, objection generation (based on *feedback-directed random testing* for generating valid method sequences) and object mutation for increasing branch coverage of the code under test. This is more effective than just random testing because the search space for the state of object instances can be huge and desirable object instances are likely to be close to captured instances [19]. Like other approaches, it uses byte code instrumentation and object serialization in the capturing phase.

Automated behavioral regression testing (BERT) by Jin *et al.* [21] is a technique for automatically disclosing behavioral differences between two versions of a software using dynamic analysis. First of all, it determines the area where source code changes were made between the given software versions. Outcome of this is e.g. a set of classes. After that, a *test generator* is used to generate test inputs for these classes (for example by random testing). In the next phase, BERT runs the generated tests on both the old and the new version of the software and logs the state of the instances of the classes under test as well as return values and other possible output. The *raw behavioral differences* are identified by comparing the logs of both runs. This is concluded by a *different behavioral analysis* which reduces redundancies, removes unnecessary raw information and assesses the likelihood of a difference to be a regression bug.

Xie *et al.* show a framework for *differential unit testing* in [29], which is not based on capture and replay though. Based on two versions of a software, they annotate the newer version by means of JML (Java Modeling Language). After compiling the newly generated sources, program executions are performed (e.g. using test generation tools). Like in our case, usage of system tests would be also possible. During test execution, the behavior of the older/reference version is verified by the JML annotations which were added to the newer/changed version.

Observing program executions is also used for determining valid method-sequences suitable to create object instances which are then used as input for random testing. In subsequent work [30],

Zhang *et al.* proposed a combination of static and dynamic automated test generation. After getting method-sequences by observing program executions, they continue with a static analysis for increasing coverage by searching for relations between methods because testing related methods is more likely to discover unknown program states.

Chapter 4

Approach

4.1 Principles

First of all, the differing underlying algorithms for both capture and replay are explained to show the basics of how our approach works.

4.1.1 Capture

The development of the prototype led to two different methods of capturing the data during an observation. Because we first tried to show the basic feasibility of doing the capture by means of JDI, we have implemented an operating mode of the prototype in which it captures all data that is accessible for a certain method call. We call this *full capture* because nothing (regardless of being useful or not) is omitted. First of all, this means that all arguments to methods and all fields of the called instance (or class if they are static) are captured. In case of primitive values like integers, this might not be that drastic. However, in case of reference values (arguments or fields that are objects), it means that further references in the referenced objects have to be traversed as well until you finally end up with primitive values. Obviously, this might lead to quite large amounts of state to capture. Whereas primitive values can be captured directly as they are and stored to a relational database, objects have to be serialized to a character stream so that it is possible to reconstruct their structure.

It is not unlikely that most of the objects of a current program state are somehow linked (meaning that it is possible to create a reference chain between most of the objects). Thus, you might end up with almost the whole program state all the time. In practice, this method is probably not quite useful but it was interesting to implement and to use it as a baseline for the following, more economic method.

After finishing this method which is kind of a brute force approach, we aimed at reducing amount of values to capture to a minimum. In order to achieve this we use the filter mechanism of JDI (see 4.2.1) to only getting notified of a very limited set of events. Therefore, we call it *filtered capture* and this is essentially what Elbaum *et al.* described as *touched-carving* [22]. The underlying idea is that we do not capture anything at all from the start. Instead of that, only primitive arguments are immediately captured but reference (i.e. non-primitive) arguments to a method are *watched* when a call to that method occurs. Additionally, public fields of all classes (that are loaded) and all fields (also non-public) of classes of the code under test (which represent the state of called instance) are *watched*. When a reference respectively object is *watched*, it means first of all that again all fields of this specific object instance are also under observation. In other words, any access to the fields of a watched object instance is observed. Consequently, all watched object references are being kept track of. However, there is nothing essentially captured until a primitive field of an object instance is read. If that is the case, the object hierarchy is created from the bottom up to all references which are able to reach this

object instance. In the end, a representation of the objects which just contains the primitive fields that have been actually read and all necessary object references are stored.

Both the *full* and the *filtered* capture work without any previous analysis or processing of the classes under test. Hence, we kind of create a knowledge base about the program under test and its classes and their contents on-the-fly while executing it. Since we do not create mock objects at certain boundaries to separate the code under test from external code or libraries, we are able to replay it later on new versions of classes of both the code under test as well as "external code".

Example

Now we take a second look at the code example in figure 1.5. We define the class `Company` as our *unit under test* and therefore observe all calls to the public methods of this class.

The first method call we get notified of is the call to the constructor. It initially creates the object, thus there is of course no previous state we have to care about and in this case there are also no arguments to the constructor. Since it is a constructor, there is neither a return value to capture. The only thing we can actually capture is the state after exiting the constructor which we call *post-state*. If a *full capture* is performed, it means that the fields `employees`, `projects` and `salaryAdjustment` of this object instance are stored. Whereas there is only a numeric value to save for `salaryAdjustment`, the other two fields are object references themselves and hence all their contents and their succeeding references have to be traversed as well. A *filtered capture* is not really less expensive here because all fields are internally initialized (eventually leading to primitive fields being written) and no other references are involved.

The following calls are not of our concern as they are mostly not invoked on the unit under test. There are of course the calls to `addEmployee` and `addProject` but those are omitted here because the just build up the state of the `Company` instance and are not that interesting. The really interesting part is the call of the method `calcSalary` since it represents a possible unit test case for this method. This time, the object instance has a certain state before the call, which we call *pre-state*. On *full capture* that previous state is stored just in the beginning by traversing the object instance (i.e. its fields). If *filtered capture* is used, the required pre-state is saved when the exit of the method is observed because at that time it is known which parts of the state have actually been read. As indicated in the figure, only the fields `salaryAdjustment` and `projects` are really read. Filtering the capture also applies to the external scope like the class library. Therefore, only those parts of the `java.util.Set`, which have in fact been read, are processed. For example, if the loop exited after half of the entries, only the first half would be captured. Besides, the example shows that also just a small part of the `Project` objects is really used: The calls on them within the loop provoke that the field `brokerages` which holds a double value for a `Employee` is accessed.

The arguments of the method are handled as follows: parameter `allowance` is immediately captured because it is a primitive values and it cannot be observed by JDI whether it is accessed or not. During a *full capture*, the reference parameter `e` of type `Employee` would also be captured on entering the method (again by processing its fields and traversing all subsequent references). Contrary to that, for *filtered capture* again reference arguments are stored when exiting the method because then all the accessed elements have been recorded and are ready to be processed. In the example, the method `getSalary` is invoked on `e` which leads to its fields `salary`, `bonus` and the static field `globalBonus` of the class `Employee` being read. Furthermore, the argument `e` is used within the loop where it is passed as parameter to the function `getBrokerage` which is invoked on every project in `projects`. Those calls per project cause a calculation of the salary of the employee (by means of `getSalary`) which is without effect though, because those values have already been read.

Effectively, *filtered capture* stores a representation of this object instance of `Company` and e that only consists of the yellow-framed elements shown in the figure whereas *full capture* bascially stores "everything".

As a side-effect *filtered capture* implicitly tells you whether a method is *stateless* or *stateful*. If an empty pre-state is detected, it indicates that a method is likely to be stateless. However, this may still depend on the currently used input arguments. Other input values might lead to different branches being taken and the method could still be stateful in those cases.

Finally, when the method exits, the first thing to do is to capture the return value. In case of `calcSalary`, it is quite trivial because it is just a primitive numeric value (in this case `5231.897`). However, if it were an object value, there would be again the problem of traversing all possible references. The problem is that you cannot imply how the return value is used by the caller and therefore reduce or filter it. Thus, we do not perform a reduction of return values at the moment.

Additionally, the argument values are processed again in order to capture the post-state of reference arguments because they may have been modified by calling the method and this belongs to the behavior of the method has to be captured. *Full capture* simply stores the objects passed as arguments by traversing them again. *Filtered capturing* can make use of primitive fields that have actually been written for determining the effective changes.

4.1.2 Replay

Due to the design of the data model where the captured data is stored in, we are flexible how to replay it in order to perform regression testing. It is possible for us to replay a whole *application* (that means everything that has been recorded for it), certain *units under test*, single classes or methods or even just a single object instances of a class.

The prototype features two different replay modes, one being *method-based* and the other one being *instance-based*. There is no particular advantage of using one of these methods as they are just two different point of views and show that the data model allows it to use different ways to replay method calls. *Method-based replay* queries all captured calls of a certain method, restores the values of possible existing static primitive fields of the class the method belongs to and then creates an object instance using the captured pre-state of the call. After that, the call is replayed on the created instance by reconstructing the captured argument values and passing them to the corresponding method.

This is followed by comparing the results of the captured and the replayed call. First of all, there is a distinction whether the original call resulted in an exception or not. If it is the case, the resulting exception must be the same of course. If it is not the case and the method has a return value (not `void`), then the return values have to be compared. Additionally, it has to be checked if there was an unexpected exception of course. In case of a normal return of the captured call (no exception occurred while executing it), there are also two additional results to verify: If it is a non-static method, the post-state of the called instance is compared in order to ensure that the same modification of the state have been made. Secondly, the state of all passed reference arguments must be checked, in terms of whether they were changed in the same way.

The *instance-based replay* replays all calls which were made on a specific object instance (and you can query for example all instances of a certain class) in the same chronological order like they occurred during capture. The object instance itself is created by replaying the originally captured call to a constructor of the class. All in all, this also implicitly restores the same state of the object instance.

Example

Again looking at the code example in figure 1.5, replay can be explained as follows: Since the class `Company` is our unit under test we either look up all methods of this class (for *method-based* replay) or we load all captured instances of this class (for *instance-based* replay).

For the *method-based* replay all public methods which are not constructors are queried (i.e. `calcSalary`, `addEmployee`, `addProject` and so on). Replaying `calcSalary` means that the calls to it are queried (here it is just one call). Due to the fact that `Company` does not feature static

primitive fields which could be part of a necessary pre-state, there is nothing to restore here. However, the captured pre-state (it may be *full* or *filtered* i.e. only the yellow-framed elements) that was saved for the call is restored by reconstructing the instance of `Company`. After that, the arguments to the method call are fetched and in case of the reference argument `e` reconstructed (either of a `full` or `filtered` serialization). These arguments are passed again to `calcSalary` on the recreated instance of `Company`. Since `calcSalary` has a return type (non-`void`) the return value of the replayed invocation is compared to the expected result of this call (`5231.897`) and the result of that comparison is stored. No occurrence of an exception is expected for this call and there also will not be any on replay if the pre-state is restored correctly. This is stored as a result, too.

Furthermore, the post-state of the `Company` object and the reference argument `e` are compared with the corresponding values saved along with the method call (they also have to be reconstructed like the values described above). In the example, only on a *full capture* there will actually be values to compare because the *filtered* post-state(s) are empty as no values were written during the execution of the call.

On the other hand, *instance-based* replay fetches the unique identifiers of instances of `Company` (there is one such instance in the example). All method calls which were made on a specific instance are fetched ordered by their timestamp. The first call is assumed to be a constructor and is used to create the object instance. In the example, we have observed a constructor call of `Company` and therefore we can utilize it to instantiate our object. Under certain circumstances it might be that we did not observe a call to a constructor and in these cases there is a fallback strategy where again the instance is created by means of the stored pre-state of the first call. Anyway, after that all calls are executed like they occurred on capturing and in the example this leads to the two calls of `addEmployee`, also two calls of `addProject` and finally `calcSalary` being replayed. Replaying the calls themselves happens just like it is described above.

4.2 Implementation

4.2.1 Architecture

In this section, the most important APIs and third-party libraries, which are used as part of the implemented approach are introduced. It is described what they are used for, why they were chosen and which advantages or limitations they feature. Figure 4.1 gives an overall overview of the structure and dependencies within the approach.

Java Development Kit

The prototype is implemented on the basis of the Java Development Kit in version 7 (1.7.0). Some important APIs of it are described in the following sections (4.2.1 and 4.2.1). Although there were some *debug builds* of the JDK available in the past, we do not use them but rely on standard builds. Debug builds include more information in the class files (like parameter names) which is then read by the Java Debug Interface and you might end up with exceptions if it is missing. In general debug builds are hardly available and not released on a regular basis nowadays. Since we do not consider the standard class library to be code under test, which is observed, a standard build is sufficient.

Java Debug Interface

JDI is part of the Java Platform Debugger Architecture[1] and provides a high-level frontend interface to the debugging features of the Java Virtual Machine. Apart from the interface docu-

[1] http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/

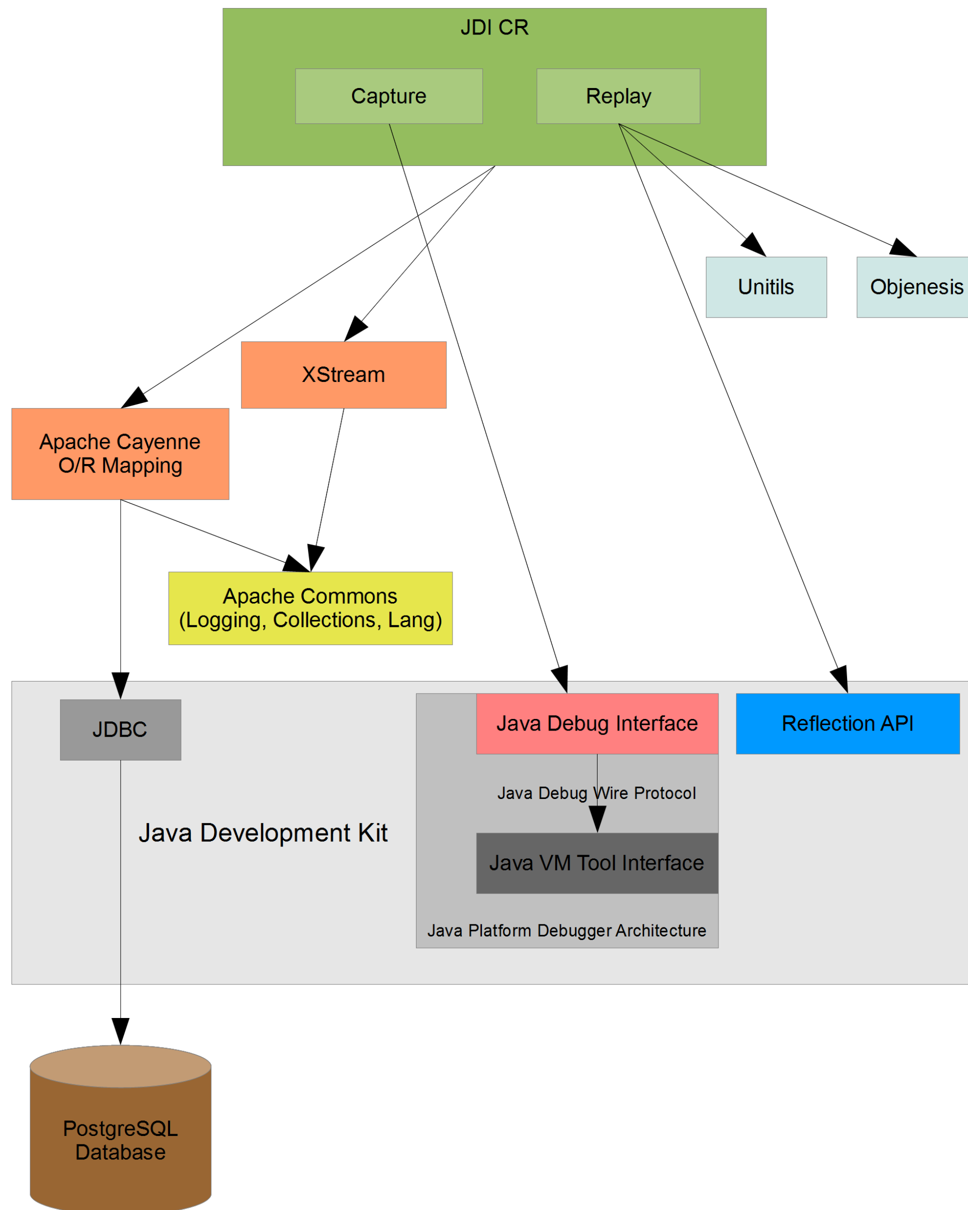

Figure 4.1: Overview of used libraries and SDKs

mentation[2] and some small example applications (contained in the JDK) there is not very much information about JDI available. But these two things are basically all you need to get started.

JDI is the primary technology to realize the capturing process in our approach. It is used to retrieve arguments and return values of method calls respectively the state of object instances (e.g. before and after the method call).

In figure 4.2 an overview of the most important interfaces of the API is shown. Please note that for the purpose of clarity some parts which were not relevant for our approach are omitted (e.g. the `DoubleType` and `DoubleValue` can be seen as representatives for all numeric types/values as there additionally are `float`, `long`, `int`, `short`, `byte`). As you can see there are two kind of similar hierarchies for types and values, each of them splitting up in primitives and references (latter with specializations for arrays and so on). In addition to that, all you need is a location to access this values (respectively their types) and for this purpose there are the interfaces `Method`, `Field`, `StackFrame` and `LocalVariable`.

[2]http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html

Figure 4.2: Diagram of the most important JDI interfaces

However, JDI works based on events, you cannot just query some information you need. The different kinds of events you can register for are shown in figure 4.3. Some interfaces are left out in this figure as well because they are not used in the implementation.

The basic procedure for using JDI is as follows:

1. Connect the *debugger* (our application) to a running VM process (the *debuggee*)
   In our case the debugger launches the debuggee and immediately connects to it.

2. Request to be notified for certain types of events (see figure 4.3)
   Along with the `EventRequest` it is possible to define filters so that you only get notified if events of specific classes, in specific threads or on specific object instances occur.

3. Handle events
   As long as the debuggee is alive, wait for new events and process them.

When it comes to capturing there are several limitations which are not necessarily shortcomings of JDI per se but that are caused by the underlying concepts and the debugger architecture.

First of all, with regard to method calls there is no mapping between a method entry event and the corresponding exit event. As a simple solution to overcome this problem we used a stack of calls per method to determine the currently active call on exit (because in case of recursion there can be multiple of course) and connect it with the data gathered at the corresponding method entry.

One rather annoying but on second thought plausible obstacle is that you can not directly access object references in the debuggee (since it might be in another process or even on another computer). As a consequence, you have to manually traverse the representation (called *mirror* in the terminology of JDI) of that object until you finally reach primitive values which can be retrieved out of the remote virtual machine by invoking corresponding methods on the object reference.

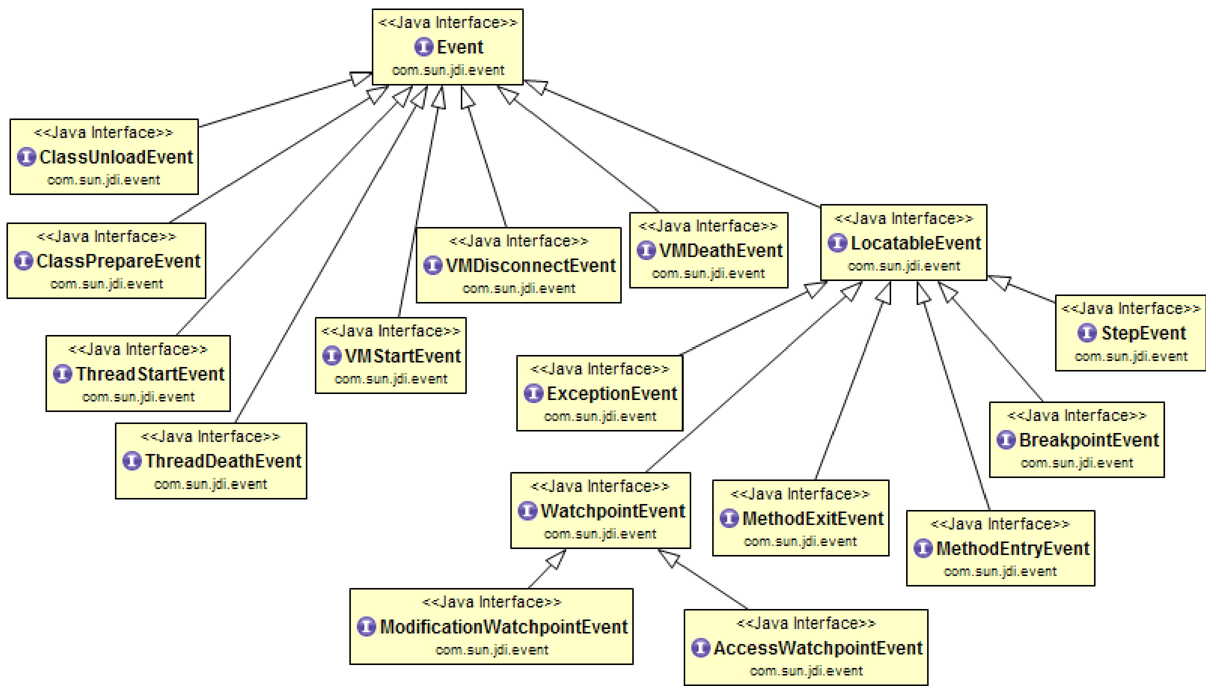

Figure 4.3: Diagram of most important JDI events

Reflection

Reflection in matters of programming languages is the ability of a program to deal with a representation of its state during its own execution. This ability is divided into two parts: *introspection* and *intercession*. Introspection means that the program is able to monitor and analyze its own state. Intercession means that the the program can modify its own state of execution and interpretation. [31]

In Java, the Reflection API in the package `java.lang.reflect` contains classes (like `Class`, `Method` and so on) that provide access to the elements of a class (i.e. querying the fields or methods of a class). It is also possible to dynamically create instances of a class, invoke methods on instances or set values of fields [32].

In our approach, we use Reflection for the whole replaying process. It is necessary to create appropriate objects and parameter values for replaying method calls (by just invoking the corresponding methods). Although JDI (4.2.1) has similar capabilities as well, we chose to use Reflection because it is the "more obvious" solution for such a problem in Java. Additionally, Reflection is part of the common Java class library whereas JDI is only included in the JDK.

XStream

XStream[3] is an open source library for serializing arbitrary Java objects to a XML string and deserializing them from XML. It does not require any special design or implementation of the Java classes to process them because it uses meta-programming techniques like Reflection to handle "unknown" classes. This means that you do not have to implement any kind of interface in a class or add a default constructor to a class in order to be able to (de)serialize objects of it. Additionally, it has built-in converter for many classes which are part of the Java class library (e.g. collections) in order to transform them into a compact and human-readable serialized format. It is also possible to tweak the output and hence avoid using actual names of classes or fields in order to produce a more concise output (this is called *aliases*).

[3] http://xstream.codehaus.org

In the approach, XStream is used to (de)serialize all the wrapper objects which contain the structure and finally the primitive field values of "complex" objects (contrary to primitive values). This may be a parameter to a method or the state of the object where a method is invoked on. Although XStream has quite a bunch of features, basic functionality was sufficient for our approach.

Apache Cayenne

Cayenne[4] is an open source persistence framework which is developed as a top-level project of the Apache Software Foundation. Like any other object-relational mapping tool, it reduces the implementation effort required to store/load (Java) objects into/from a database. The information about the Java classes, the database tables and the mapping between them is stored in XML configuration files. The project also offers a GUI modeling tool which allows to comfortably create and edit the configuration files including features such as database reengineering and Java source generation. The generated Java source is created in a way that allows you the easily extend the model classes.

Since there are no configuration or capturing log files used in our approach, basically all required data is written to the database during capture and retrieved from it on replay. Using a persistence layer naturally introduces an additional overhead which we were willing to accept. However, compared to the major performance related issues which slow down the application (see section 4.3), the overhead caused by Cayenne seemed insignificant to us (keeping in mind that it significantly simplified development).

Unitils

Unitils[5] is an open source project that provides a library that contains framework or utility classes which lower the effort for writing unit tests respectively enhancing the capabilities of unit tests in general.

In our approach we only use one feature called *Reflection assert* (part of the core module) for in-depth comparing during replay. This is necessary for non-primitive objects of classes which do not implement the `equals()` method.

Objenesis

Objenesis[6] is a small and lightweight open source library for instantiating objects. There are some cases where you cannot instantiate objects using the Reflection API (e.g. if the corresponding class does not have a default constructor and you do not know which values to use for a constructor with arguments or you want to avoid that constructor code is executed). Objenesis handles all these cases and hides the complexity of different Java VM vendors and versions.

During replay it is used to create "empty" objects because the field values are set after the creation anyway.

4.2.2 Data Model

Naturally, the class design of our tool (shown in figure 4.4) is influenced by how JDI is designed and how it works. Please note that due to the prototypical stage of the tool, this design is still quite simple and some classes could be enhanced with more fancy properties. A lot of classes have a prefix of "Captured" because otherwise their names would clash with those of the packages `com.sun.jdi` respectively `java.lang.reflect`. This prefix is not used for the database relations which can be seen in figure 4.5 (because it is not necessary there and would lead to needless

[4]https://cayenne.apache.org
[5]http://www.unitils.org
[6]https://code.google.com/p/objenesis

long names). Due to the fact that the classes are automatically generated by means of Apache Cayenne (4.2.1), a lot of dependencies or associations are not visible in the class diagram (created by ObjectAid UML Explorer for Eclipse) but only in the entity-relationship diagram.

Figure 4.4: Class diagram of the core capture and replay classes

The root object in our tool is an `Application` which has a unique name and the fully-qualified class name of the main class (i.e. a class with a method `public static void main(String[] args)`) to start. The classpath that is used for executing the application is configured via the `ClassPathEntry` class defining a path to a directory or JAR file. These entries are sorted by a defined order for obtaining uniqueness which class files to use. The class files are the only part of an application that must be available on the local computer (hence, this paths apply to the local workstation which is running the tool) and is not stored in the database. This is not really satisfying but for the time being we accept that difficulty. An application also consists of multiple `TestSet` objects and those testsets are constructed with objects of the class `TestSetClassFilter` (defining class or package names using wildcards).

The class information that is collected while capturing is stored in the following classes: `CapturedClass` represents a single java class by just storing the fully-qualified name and whether it is primitive or not. At the moment a `CapturedClass` belongs to single `Application` making it possible to have multiple versions of the same class in different applications. In addition to that, there is the class `CapturedType` that stores the required information for any type declaration in a Java program (like variables types, return types and so on). In essence, it means that you have

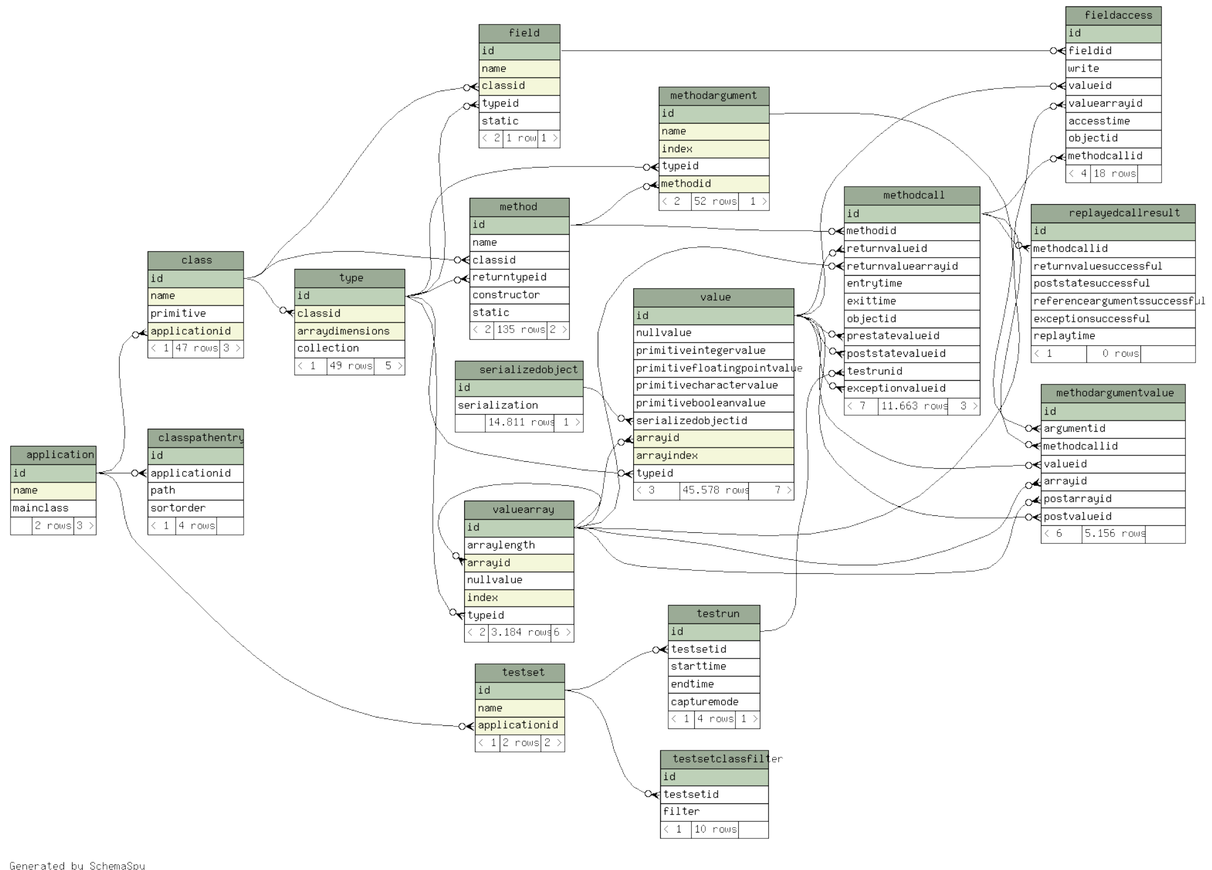

Figure 4.5: ERD of the physical data model on PostgreSQL

to store if it is an array (and how many dimensions it has) along with the corresponding class. Because of a special case for collections (they are treated like arrays) it is stored for the type if it is an collection. The contents of a Java class is held in the classes `CapturedField` (name, type and whether it is static) and `CapturedMethod` (name, return type, whether it is static or a constructor). The arguments of a method (with their names, types and order) are saved in `CapturedMethodArgument`.

Whenever a `TestSet` is observed during a run of the application, a `TestRun` bundles the captured values which are handled by the following classes: `CapturedValue` either directly contains a primitive value, defines it as a null value or links it to a serialization of an object as `SerializedObject`. `CapturedValueArray` allows packing `CapturedValue`s into arrays of arbitrary depth because a `CapturedValueArray` can also be part of a `CapturedValueArray` again.

As already mentioned, serialized objects are stored by the class `SerializedObject` which contains the serialization as a string value. However, before it is serialized, an object is represented by means of the class `CapturedObject`. Such a `CapturedObject` holds all the information for serializing a actual Java object. It uses the `Member` class respectively its subclasses `PrimitiveMember` for primitive field values, `ObjectMember` for field values that are again objects (which hence contains other objects of `CapturedObject`) and `ReferenceMember` for object field values that have already been processed (to avoid cycling references).

Arrays within objects values are kept as `CapturedArray`. Either objects of `CapturedValue` or `CapturedValueArray` are used by a `CapturedMethodCall` to define values for its arguments (linked via `CapturedMethodArgumentValue`), its return value, the pre-state and post-state of the called object instance and a possibly occurred exception. Additionally, it features the time of its entry and exit and the identifier number of the called instance (from JDI) which is unique within the `TestRun`. Values (or arrays) are also captured for a `CapturedFieldAccess` together

with the time of access and whether it was a read or write access.

The model for replaying is rather limited at the moment and only consists of a single class, namely `ReplayedCallResult`, that stores whether the return value, the post state of the called instance, the state of reference arguments or a possibly occurred exception could be verified as expected on replay.

The algorithms for capture and replay per se are implemented in the classes `DBCapturer` respectively `DBReplayer`. Entry point to observe a `TestSet` is the class `CapturingRunner` which starts the application itself as well as the `VMObserverThread` (it contains the JDI event loop and handling) and the `StreamRedirectThread` (it is necessary to read the output and error streams so that the application runs and exits normally).

4.2.3 GUI

The graphical user interface of the prototype was implemented using WindowBuilder[7] in Eclipse based on the Swing widget toolkit of Java. Figures 4.6 and 4.7 are screenshots that give a exemplary glance at the (rather simplistic) graphical user interface of the prototype.

An overview of the classes which are used to realize the GUI is provided in figure 4.8. It mostly consists of `Action` classes for handling the different executable commands, `TreeNode` classes for displaying an application's structure and `Panel` classes for editing that structure.

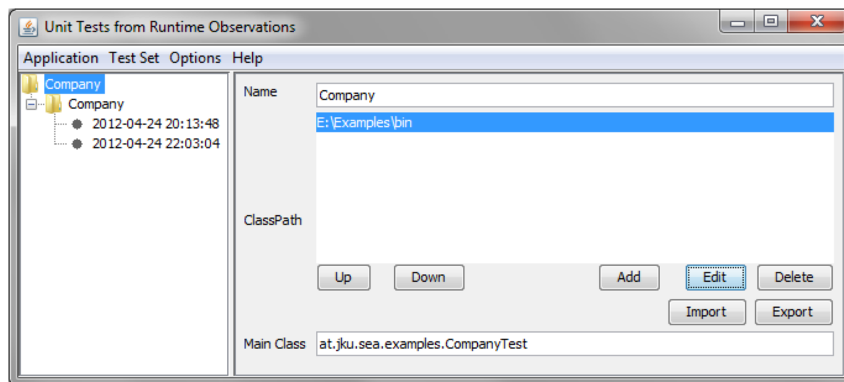

Figure 4.6: Screenshot of an example application within the prototype

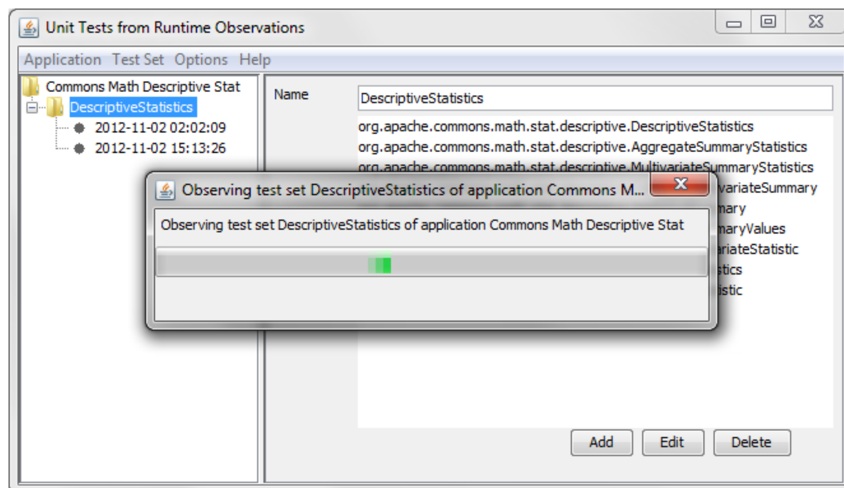

Figure 4.7: Screenshot of observing an application

[7] https://developers.google.com/java-dev-tools/wbpro

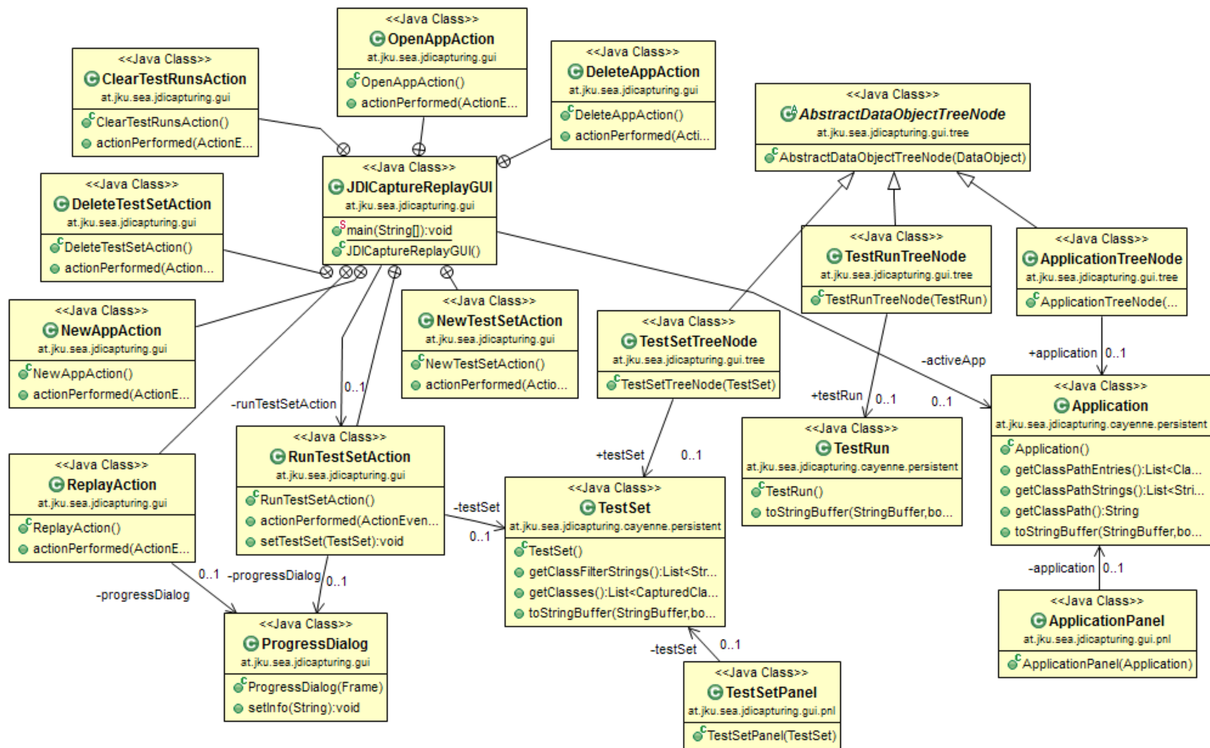

Figure 4.8: Class diagram of the GUI component

4.3 Evaluation

In this section, the different principles are analyzed regarding their efficiency by means of exemplary program executions.

First of all, the code example shown in figure 1.5 was used as a really tiny test execution (referred to as *CompanyTest*). The second scenario called *DescriptiveStatTests* consisted of several JUnit test cases of the open source Apache Commons Mathematics Library[8]. In table 4.1 some metrics of the scenarios are given for outlining their dimensions.

The workstation used as testing platform featured the following specifications in terms of hardware and software.

- AMD Phenom II X4 905e Processor (4 cores, 2.5 GHz)
- 3.75 GB RAM
- a standard HDD (no SSD)
- Windows 7 Professional Servicepack 1 (64 Bit)
- Java Development Kit 1.7.0 Update 7 (both 32 and 64 Bit versions)
- PostgreSQL 9.1 (database locally installed on the workstation)
- Cayenne 3.0.2

The system was not especially prepared for testing but was used for developing and other common purposes at the same time.

[8]http://commons.apache.org/math/

Scenario	Classes	Methods	Method Calls
CompanyTest	3	4	6
DescriptiveStatTests	44	131	1367

Table 4.1: Scenario Metrics (captured items)

Runtimes as shown in table 4.2 were measured for normal execution by means of simple manual time calculations based on timestamps before and after calling the program. On the other hand, for the observed executions the duration was calculated using the start and end time of the respective `TestRun`. This does not include the time needed to setup and launch the required separate virtual machine. It comprises the initial requesting of JDI events and the shutdown of the observed Java process though. All values are the mean of five runs.

The collected data unfortunately shows that the observation of a program execution by means of the Java Debug Interface produces a tremendous overhead and is therefore not really practical for capturing real world applications.

A more detailed performance analysis of program observations based on the Java Platform Debugger Architecture is provided by Mehner in [33]. That article shows that the features of JDI which we definitely need (suspending threads and filtering events), have an additional negative impact on the performance.

As another result, the data tells us that despite its advantages concerning the reduction of state size, *filtered capture* is about 5-8 times slower than *full capture*. The reason for that is that it requires the request and handling of a lot more events in JDI (those `AccessWatchpointEvents` for observing accesses on fields of object instances) which also leads to threads respectively the whole virtual machine being suspended a lot more often.

Scenario	Normal Execution	Full Capture	Filtered Capture
CompanyTest	2.6 ms	339 ms	1803 ms
DescriptiveStatTests	86 ms	41577 ms	328360 ms

Table 4.2: Runtime Comparison of Capture

The size of captured data (listed in table 4.3) was computed by summing up the sizes of the tables `Value`, `ValueArray` and (most significant) `SerializedObject` on database level (see figure 4.5 for the ERD). The test results clearly evidence that *filtered capture* is capable of reducing the amount of captured data in a distinguishable manner. In the chosen examples, the size of the state resulting of full capture was 2-5 times larger than the filtered state. Hence, it was at least successful in achieving its primary goal.

Scenario	Full Capture	Filtered Capture
CompanyTest	16 kB	7.2 kB
DescriptiveStatTests	1392 kB	272 kB

Table 4.3: Size of captured data

The overall conclusion based on the evaluation of the capturing processes is that sadly neither *full* nor *filtered* capture give the impression of being practical to applied on everyday software projects. *Full capture* is likely to be just wasteful in case of larger program states and already causes a remarkable overhead during observation. *Filtered capture* might be adequate in terms of amount of stored state but the execution overhead is just ridiculously high.

Regarding replay, tables 4.4 (based on a *full capture*) and 4.5 (using a *filtered capture*) oppose the normal execution time to the time necessary to replay the captured testruns. Looking at the runtimes in both tables you can see that *instanced-based replay* appears to be faster than *method-based replay*. This can be explained by the fact that the former method requires less reconstruction of state because it relies on constructor calls.

Furthermore, comparing the two tables it is obvious that the enormous effort for *filtered capture* pays off here. Reconstructing the filtered state is of course much faster due to its smaller size. In the exemplary cases, the differences are quite remarkable since replaying a fully captured state is from 7 up to 20 times slower than replaying a filtered state.

Scenario	Normal Execution	Method-based Replay	Instance-based Replay
CompanyTest	2.6 ms	475 ms	352 ms
DescriptiveStatTests	86 ms	84224 ms	45674 ms

Table 4.4: Runtime Comparison of Replay based on Full Capture

Scenario	Normal Execution	Method-based Replay	Instance-based Replay
CompanyTest	2.6 ms	60 ms	17 ms
DescriptiveStatTests	86 ms	5772 ms	3421 ms

Table 4.5: Runtime Comparison of Replay based on Filtered Capture

Chapter 5

Conclusion

It has been quite interesting to analyze the state-of-the-art in software testing and to research which approaches to increase automation and effectiveness exist in this area. In the particular topic of the thesis, excellent work has already been published, especially by Elbaum *et al.* [22].

The *Java Debug Interface* is a very elegant solution in terms of implementing a capturing process that determines only required state. A major advantage is that is does not require special preparation or pre-processing of the code respectively classes under test and can be used with a standard and unmodified Java Development Kit. However, it turned out to be extremely inefficient regarding execution time overhead. Especially those features, which are needed to reduce the amount of captured state to a minimum, cause a further slowdown. Hence, it does not seem to be practical for software of scale as it appears in common use. In its original application, namely debugging, the extraordinary low speed is not of big consequence because only small parts of code are observed by a human and human observer cannot process that information fast enough anyway.

It has also been very informative to implement this prototype because you have to keep lots of special cases in mind in order to support different features of programming languages (in this case Java) and it makes you think a lot about the principles of Java.

At the current state of work there are still parts of the language left to be supported and bugs to be fixed. Not all implementation details or encountered problems during the development of the prototype are mentioned in this thesis.

Future work could include stabilizing the prototype, covering more features of the Java language as well as extending replay mechanisms (which are rudimentary at the moment) and use the power that lies in the database-based storage of the captured to create sophisticated analysis and reports. Other possibilities would include to reason about the faultiness of units respectively them being suspicious to be faulty.

Bibliography

[1] P. Bourque and R. Dupuis, *Software Engineering Body of Knowledge (SWEBOK).* IEEE Computer Society, 2004.

[2] T. Xie, "Improving automation in developer testing: State of the practice," tech. rep., Technical report, North Carolina State University, 2009.

[3] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, "Carving differential unit test cases from system test cases," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, (New York, NY, USA), pp. 253–264, ACM, 2006.

[4] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, pp. 100–107, December 1998.

[5] D. Saff and M. D. Ernst, "An experimental evaluation of continuous testing during development," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '04, (New York, NY, USA), pp. 76–85, ACM, 2004.

[6] T. Y. Chen and F.-C. Kuo, "Is adaptive random testing really better than random testing," in *Proceedings of the 1st international workshop on Random testing*, RT '06, (New York, NY, USA), pp. 64–69, ACM, 2006.

[7] T. Chen, H. Leung, and I. Mak, "Adaptive random testing," in *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making* (M. Maher, ed.), vol. 3321 of *Lecture Notes in Computer Science*, pp. 3156–3157, Springer Berlin / Heidelberg, 2005.

[8] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, (New York, NY, USA), pp. 213–223, ACM, 2005.

[9] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, (Washington, DC, USA), pp. 75–84, IEEE Computer Society, 2007.

[10] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, July 1976.

[11] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, (New York, NY, USA), pp. 1066–1071, ACM, 2011.

[12] K. Sen, "Concolic testing," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, (New York, NY, USA), pp. 571–572, ACM, 2007.

[13] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "jRapture: A capture/replay tool for observation-based testing," in *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, (New York, NY, USA), pp. 158–167, ACM, 2000.

[14] A. Georges, M. Christiaens, M. Ronsse, and K. De Bosschere, "JaRec: a portable record/replay environment for multi-threaded Java applications," *Softw. Pract. Exper.*, vol. 34, pp. 523–547, May 2004.

[15] S. Artzi, S. Kim, and M. D. Ernst, "ReCrashJ: a tool for capturing and reproducing program crashes in deployed applications," in *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC/FSE '09, (New York, NY, USA), pp. 295–296, ACM, 2009.

[16] H. Prähofer, R. Schatz, C. Wirth, and H. Mössenbock, "A comprehensive solution for deterministic replay debugging of SoftPLC applications," *Industrial Informatics, IEEE Transactions on*, vol. 7, pp. 641 –651, nov. 2011.

[17] G. Xu, A. Rountev, Y. Tang, and F. Qin, "Efficient checkpointing of java software using context-sensitive capture and replay," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, (New York, NY, USA), pp. 85–94, ACM, 2007.

[18] A. Orso, "Monitoring, analysis, and testing of deployed software," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, (New York, NY, USA), pp. 263–268, ACM, 2010.

[19] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang, "OCAT: object capture-based automated testing," in *Proceedings of the 19th international symposium on Software testing and analysis*, ISSTA '10, (New York, NY, USA), pp. 159–170, ACM, 2010.

[20] K. H. Lee, Y. Zheng, N. Sumner, and X. Zhang, "Toward generating reducible replay logs," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, (New York, NY, USA), pp. 246–257, ACM, 2011.

[21] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ICST '10, (Washington, DC, USA), pp. 137–146, IEEE Computer Society, 2010.

[22] S. G. Elbaum, H. N. Chin, M. B. Dwyer, and M. Jorde, "Carving and replaying differential unit test cases from system test cases," *IEEE Trans. Software Eng.*, vol. 35, no. 1, pp. 29–45, 2009.

[23] S. Joshi and A. Orso, "SCARPE: A technique and tool for selective capture and replay of program executions," in *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on*, pp. 234 –243, oct. 2007.

[24] S. Joshi and A. Orso, "Capture and replay of user executions to improve software quality," tech. rep., Technical report, Georgia Institute of Technology, 2006.

[25] A. Orso and B. Kennedy, "Selective capture and replay of program executions," in *Proceedings of the third international workshop on Dynamic analysis*, WODA '05, (New York, NY, USA), pp. 1–7, ACM, 2005.

[26] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," in *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, (New York, NY, USA), pp. 128–137, ACM, 2003.

[27] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, "Automatic test factoring for java," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, (New York, NY, USA), pp. 114–123, ACM, 2005.

[28] D. Saff and M. D. Ernst, "Mock object creation for test factoring," in *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '04, (New York, NY, USA), pp. 49–51, ACM, 2004.

[29] T. Xie, K. Taneja, S. Kale, and D. Marinov, "Towards a framework for differential unit testing of object-oriented programs," in *Proceedings of the Second International Workshop on Automation of Software Test*, AST '07, (Washington, DC, USA), pp. 5–, IEEE Computer Society, 2007.

[30] S. Zhang, D. Saff, Y. Bu, and M. D. Ernst, "Combined static and dynamic automated test generation," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, (New York, NY, USA), pp. 353–363, ACM, 2011.

[31] D. G. Bobrow, R. P. Gabriel, and J. L. White, "CLOS in context: the shape of the design space," in *Object-oriented programming* (A. Paepcke, ed.), pp. 29–61, Cambridge, MA, USA: MIT Press, 1993.

[32] G. McCluskey, "Using Java Reflection." `http://java.sun.com/developer/technicalArticles/ALT/Reflection/`, 1998. Accessed on 2012-08-13.

[33] K. Mehner, "Zur Performanz der Überwachung von Methodenaufrufen mit der Java Platform Debugger Architecture (JPDA)," *Java Spektrum*, November 2003.

List of Figures

1.1 Unit Testing 2
1.2 Integration Testing 3
1.3 System Testing 3
1.4 Generating Unit Tests from Executing System Tests 6
1.5 Code example showing amount of required state 7

2.1 Classification of unit tests regarding their usage of state 9
2.2 Classification of unit tests regarding evolutionary changes 10

4.1 Overview of used libraries and SDKs 20
4.2 Diagram of the most important JDI interfaces 21
4.3 Diagram of most important JDI events 22
4.4 Class diagram of the core capture and replay classes 24
4.5 ERD of the physical data model on PostgreSQL 25
4.6 Screenshot of an example application within the prototype 26
4.7 Screenshot of observing an application 26
4.8 Class diagram of the GUI component 27

List of Tables

4.1 Scenario Metrics (captured items) 28
4.2 Runtime Comparison of Capture 28
4.3 Size of captured data 28
4.4 Runtime Comparison of Replay based on Full Capture 29
4.5 Runtime Comparison of Replay based on Filtered Capture 29