Using Arquillian to Improve Testing

Tobias Evert

January 18, 2015
Master’s Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Mikael Rännar
Examiner: Jerry Eriksson

UMEÅ UNIVERSITY
DEPARTMENT OF COMPUTING SCIENCE
SE-901 87 UMEÅ
SWEDEN
Abstract

Cábra is the main case handling system of the Swedish Prosecution Authority. It is being developed and maintained by Decerno, a consultant company specializing in dependable custom made business support systems. In an effort to increase cost efficiency and software quality Decerno wants to implement more automated testing. In order to make this easier this thesis project was started to evaluate available testing frameworks that can be used to test a Java Enterprise system like Cábra. The main goal of this thesis project is to survey the available tools and pick one (or a set of tools that can be used together) to use evaluate further by implementing a proof of concept testing framework for Cábra.

As part of this thesis project a literature study was conducted to find out what testing techniques could benefit a system like Cábra. Different testing techniques and testing evaluation metrics were examined during the study.

The result of this thesis project is a survey of Java Enterprise testing tools available, an implementation of a testing framework using Arquillian and an analysis of the potential benefits of using Arquillian for future testing work in the Cábra team. Arquillian was picked and analyzed based on the information provided by the literature study.
# Contents

1 Introduction .................................................. 1  
   1.1 Decerno .................................................. 1  
   1.2 Câbra .................................................. 1  
   1.3 Definitions ............................................. 2  
   1.4 Outline .................................................. 2  

2 Problem Description ........................................ 3  
   2.1 Problem Statement ....................................... 3  
   2.2 Goal .................................................... 3  
   2.3 Purpose .................................................. 3  

3 Software Testing ........................................... 5  
   3.1 Module Testing ........................................... 5  
      3.1.1 Mocking ............................................. 6  
   3.2 Integration Testing ...................................... 6  
   3.3 System Testing .......................................... 7  
   3.4 Black Box Testing ....................................... 8  
   3.5 White Box Testing ....................................... 8  
   3.6 Test Automation ........................................ 9  
      3.6.1 What to Automate? .................................. 9  
   3.7 Evaluating Testing Strategies .......................... 11  
      3.7.1 Fault Statistics Metrics .......................... 12  
      3.7.2 Mutation Testing .................................. 14  

4 Accomplishment ............................................. 17  
   4.1 Preliminaries ............................................ 17  
   4.2 How the Work Was Done .................................. 18  
      4.2.1 Literature Study .................................... 18  
      4.2.2 Survey .............................................. 18  
      4.2.3 Studying Câbra ..................................... 18  
      4.2.4 Implementation ..................................... 18
4.2.5 Analysis and Report ........................................ 18

5 Results ......................................................... 19
  5.1 Existing Framework ........................................ 19
  5.2 Available Java Testing Tools ............................... 20
      5.2.1 TestNG .............................................. 20
      5.2.2 Behavior Driven Design Tools ...................... 20
      5.2.3 Java Enterprise Testing Tools ...................... 20
      5.2.4 Mocking Frameworks ................................ 21
  5.3 Which Tools to Use in the Prototype? ..................... 22
  5.4 Implementation ........................................... 23
      5.4.1 Test Executor ..................................... 23
      5.4.2 Test Data and Expected Results .................... 23
      5.4.3 Data Handling ..................................... 23
      5.4.4 Deployment Package Creation ....................... 24
      5.4.5 EJB Mapping ...................................... 25
      5.4.6 Arquillian compared to the Current Testing Framework .... 25

6 Conclusions .................................................. 29
  6.1 Test Automation .......................................... 29
  6.2 Test metrics ............................................. 30
  6.3 Evaluation of Arquillian .................................. 30
  6.4 Limitations .............................................. 31
  6.5 Future work ............................................. 31

7 Acknowledgments .................................................. 33

References .......................................................... 35
Chapter 1

Introduction

Once a software product grows big or complex enough it will contain bugs. If a bug is not found until it affects the system in a harmful way it can cause great costs, both by the harmful behavior itself or by the cost of fixing it. Generally speaking it is cheaper and easier to fix a bug the earlier in the development process that it is found. In order to find the bugs before the code goes into production, software is tested, since there is never any guarantees that all bugs are found deciding how much is enough testing is a reoccurring question in software development.

Since a lot of testing tasks are very repetitive and time consuming in nature, automation of software testing is something that could potentially save a lot of effort. There exist a huge number of different testing tools that are meant to make software testing easier, less time consuming and more manageable. For something as old and well known as Java Enterprise Edition there are a huge number of tools and toolkits for different kinds of testing and test automation.

In order to objectively evaluate the quality of the testing effort in a product some kind of measurements or metrics has to be collected to form a basis of such an evaluation. There are a several metrics that can be collected or calculated that can be used to measure the quality of the testing effort connected to a software product, but none of them are feasible in every situation, selecting which metrics to use will influence your testing effort, care should be taken when making this choice.

This thesis focuses on automated software testing, testing tools and testing evaluation criteria. As part of this thesis I will evaluate which current tools could be used to improve automated testing in a Java Enterprise system.

1.1 Decerno

Decerno is a Stockholm based software consultant company founded in 1984 that today employs around 35 people. The company specializes in delivering reliable customized systems to support the customers’ business workflows.

1.2 Câbra

Câbra is a case management system developed and maintained for The Swedish Prosecution Authority by Decerno. The system is used by almost all Swedish prosecutors in their
daily work and also interacts directly with computer systems operated by other members of Swedish justice system, such as the Police or the Swedish Tax Authority. Availability and reliability is of course very important in a system like Cäbra, which in turn makes testing and software quality important.

Cäbra is an Enterprise Java system running on Oracle Weblogic 10 with an Oracle database back-end. Users connect to the system using a client based on Java Swing. Part of the client is the delegate layer that exposes the server API to the client, its main function is to forward calls to the server. On the server there is a facade layer that receives the calls from the clients, handles them, and if necessary forwards them to the stateless session layer that can access the database and storage. The two server layers are specified by Java interfaces and implemented as Enterprise Java Beans.

1.3 Definitions

These definitions are used throughout the thesis:

**Enterprise Java Beans (EJBs)** is a part of the Java EE standard and is an architecture for making interchangeable encapsulated modules for handling different kinds of business logic on the server side of an enterprise application. They can be specified by a Java interface (with one or more implementations) and can be called both locally (in the same Java Virtual Machine) or remotely.

**Bug** is a software defect that causes the system to behave in an unwanted way, either by not conforming to the systems specification or be acting in a way that is in contradiction with the purpose of the system. Bugs can be present already in the specification, but in this project the focus is mainly on defects introduced in the implementation of the system.

**Regression tests** are tests run before a new release of a program that are meant to ensure that old features in the program still works as intended and have not been unintentionally altered when new features were added.

1.4 Outline

This is an overview of the following chapters of this thesis report:

**Chapter 2** contains the problem description, the goals and the purposes of the thesis project

**Chapter 3** contains an in depth study of software testing and testing metrics that was part of the thesis project

**Chapter 4** contains a description of the work process of the project and the project’s accomplishments

**Chapter 5** contains the result of the thesis project

**Chapter 6** contains the conclusions made after the thesis project.
Chapter 2

Problem Description

2.1 Problem Statement

The Cåbra system is developed in iterations. When a new iteration is complete it is deployed to a test environment where it is tested both by professional testers and intended end users of the system. If the testing goes well enough, the iteration is later deployed into the production environment. If any stopping bugs are found, a new iteration is developed and deployed to test.

At present almost all the testing done on the Cåbra system is done manually. First the developers test that any new functionality or bug fixes they have created actually works, then the testers manually run through a large set of test cases in the testing environment. Since very little testing is done before deploying a new iteration to test, it happens that an iteration deployed to test has stopping bugs that prevents any meaningful testing from taking place. There are hopes that this situation can be avoided in the future if the code is more properly regression tested before it is deployed to the testing environment. Attempts have been made in the past to introduce more automatic testing in the development environment, but they have not been very successful.

2.2 Goal

The goal of this thesis project is to, based on current testing literature, evaluate existing testing tools and implement a new testing framework for more efficient automatic testing of the Cåbra system. The framework will be implemented using the most promising testing tools; the benefits of the new framework will then be evaluated.

2.3 Purpose

The purpose of this thesis project is to study the current testing methods, techniques and evaluation criteria available today and propose a new automated testing framework for the Cåbra system. This new framework will be compared to the current existing framework.
Chapter 3

Software Testing

Software is tested to find bugs in it before those bugs are found by users after the software is released. Why bother with software testing at all? It is time consuming and developers hate it. Any errors in the code that have an effect on the average user will be discovered quickly anyway and others may never actually be triggered, why go looking for trouble?

According to [13] the cost of fixing a bug increases logarithmically for each step in the software design process. Patton claim that it is 10 times more expensive to fix a bug in the coding phase than in the design phase another 10 times more expensive in the test phase and yet another 10 times more expensive after the software is released. Similar claims are made in [16], but the only reference given is unspecified "Studies at IBM". As [16] also mentions the cost of fixing a bug increases if the developer has to spend time re-familiarizing herself with old code before fixing the bug. The factor with which the cost of fixing a bug increases over time in a project probably varies between projects and bugs, but if this logarithmic cost increase is even remotely applicable in the general case, software testing makes economic sense even before you consider any loss of customer goodwill if customers are badly affected by a bug in your software.

According to Chen et al software testing can be said to have three goals [3]:

1. Increased quality
2. Decreased time to market
3. Decreased cost to market

For the Càbra team these are all relevant, but the main goal is to increase the quality with the same amount of time and resources spent on testing in the long run. The decreased cost to marked achieved by finding bugs earlier in the development cycle and quicker after they are introduced would motivate the introduction of tests that run as soon as possible after code is committed to the source code versioning system, preferably after each build, but at the very least the tests should run once a day.

3.1 Module Testing

If the only tests done during development of a new system are done on the whole system at once it can be hard to figure out what is causing any errors found. There is also a risk of faults hiding other faults, so that when the cause of one error is found and fixed, the
next test run reveals a new fault previously hidden by the first one. To avoid the problems with testing a complete system all at once, there is Module Testing (or Unit Testing) where you divide the system into modules (or units) and then verify these modules independently to verify that they behave as expected in various situations[13]. To do this you isolate the component under test from the rest of the program and then use a Test Driver to execute the code in the component with different inputs. Any parts of the system that the component interacts with are replaced by stubs mimicking different behaviors that the system can normally expose the component under test to[1].

The main downside of isolating modules is that it usually requires a lot of extra test code, usually referred to as a "test harness" needed to execute the code in the tested units without the rest of the code in the system. This enables the test case to execute exactly the parts of the module under test that the test is supposed to cover[13]. This is of course extra work, but as Ellims, Bridges and Ince report unit testing can actually be more cost efficient than functional testing. They also state that unit testing can achieve coverage of code that is hard to reach by other means, but also that it is hard to maintain the tests and coverage does not imply lack of errors[6].

In Java systems like Cäbra the obvious approach to divide the system into modules is to isolate a class from the rest of the system and test the behavior of that class. When testing server side code in a system like Cäbra that uses Session Enterprise Java Beans for most of its server side business logic focusing on testing the classes implementing these beans would seem like a good focus for module tests. When isolating a class in Java any other classes can be replaced by Mocks (see section 3.1.1). This requires some planning when writing the original code, or might require refactoring of existing code.

3.1.1 Mocking

A common way to achieve isolation of a module in an object oriented programming language like Java is to use mock objects to substitute any external dependencies. A mocked object (or just "a mock") replaces a real object and provides all the necessary interfaces that the code under test uses, but lets the test itself control how those interfaces behaves. They are similar to stubs, but stubs tend to have more logic implemented and are not controlled by the test code[17]. In Java both concrete classes and interfaces can be mocked, but having to mock concrete classes can be seen as a sign of bad object oriented design. Likewise running into objects that cannot be replaced by a mock is a sign of bad design since classes are obviously tightly coupled. Classes representing simple values do not need to be mocked, and should not be mocked so creating an interface for the value class is unnecessary overhead[8].

When working with legacy code, mocking can be hindered by previous design choices that makes it hard to inject a mock object into the object under test or that forces you to mock a concrete class instead of an interface. If possible refactoring the code to avoid such tight coupling and to enable more mocking should be considered[8]. Mocking can be a very useful tool when you are trying to test out pieces of legacy code in an isolated environment[7].

3.2 Integration Testing

Once all units/modules have been tested to a satisfactory extent they are tested together; this is referred to as integration testing[13]. It should be pointed out that integration testing does not only concern the systems internal interfaces, but also any external in-
interfaces exposed to other systems, and that testing needs to keep in mind both the explicit (documented) interfaces and any implicit interfaces between modules or systems[4].

When scaling up from modules to the whole system it is beneficiary to scale up gradually, since this makes it easier to locate any errors found in one incremental integration step. The module sets tested together form new modules that are in turn tested in isolation[13]. When integrating internal modules there are four ways to do it[4]

**Top-down** integration where you start from the top (Module 1 in figure 3.1) and work your way downwards in the order: 1-2, 1-3, 1-2-4, 1-3-5, 1-3-5-6 and finally 1-2-3-4-5-6, which includes the whole system. This is recommended when the design and requirements are clear and unchanging during the project.

**Bottom-up** integration where you start from the bottom (Module 4 in figure 3.1) and work your way up in the order: 4-2, 5-3-6-3, 6-3-5, 4-2-1, 6-5-4-3-2-1 which includes the whole system. This is recommended when the requirements and design are dynamically changing during the project.

**Bi-directional** integration where you combine more modules at each step, the order in figure 3.1 would be 2-4, 3-5-6, and finally 1-2-3-4-5-6 which includes the whole system. This is mainly useful when you are changing the architecture of the system without altering the specification.

**Big bang** integration where you just test everything at once could be motivated by small changes to the architecture with limited impact.

### 3.3 System Testing

Some consider System testing to be what happens when integration testing has gone so far that it includes all or most of the system’s components[13]. Others see system testing as a
different kind of testing; like in the first case it involves the whole system. It includes both functional and non-functional testing[4].

The functional system testing includes actual users testing the completeness of the system’s functionality, testing general purpose systems in different use case domains, and testing deployment of the system in an environment as similar to the expected target environment as possible. Functional system testing is concerned with actual use cases, preferably by actual customers. Any certification processes conducted by a third party are also considered system testing. Non-functional system testing tests things like scalability, reliability and performance testing[4].

System testing is as stated above the testing of the whole system. It relies on providing the system with inputs using the available input interfaces and checking output on the available output interfaces, this makes it a form of Black Box Testing[1].

3.4 Black Box Testing

Black Box Testing is a term used to describe a testing practice where the tester treats the system under test as a black box (hence the name), that is a box with unknown content. The only information available to the tester is the system specification and the only interfaces that can be used to verify that the specification is correct are the system’s external interfaces. The tester can input data in the same way as data would be input into the deployed system, and read any output that the system would normally have. The tester then verifies that the system behaves according to the specification by entering test data and verifying that the output complies with the specification[13].

A special case of this kind of testing is exploratory testing, which is an alternative if the software’s specification is not good enough. In this case the tester learns the software by exploring it while testing it. Building a specification from the software, describing all existing features and how they work. This is not the best situation for testing, but once a specification document is built from methodically exploring the software it is at least clear how the software works and any obvious errors in the workflows will have been documented[13].

Black box testing is good for verifying adherence to the specification and has the advantage of not requiring any understanding of, or insight into, the inner workings of the program. This means that it can be used to verify software from a third party and that the verification can be done by someone not familiar with the program. The drawbacks of black box testing is that the lack of the extra information available during white box testing makes it a lot harder to use it to find bugs and almost impossible to know how well the software has been tested.

3.5 White Box Testing

In white box testing the tester uses knowledge about the program’s inner workings when testing it. This enables the tester to use the program’s source code as a basis when designing tests, analyzing it for potential sources of errors and trying to create a test case that exposes any such errors. While this can enable the tester to prove the existence of hard to catch bugs, it can lead to a bias where the tester’s understanding of the code influences how the tests are designed[13].

White box testing also enables the practice of using different testing metrics as guides when designing the test cases. For example by adding more test cases for the parts of the program where statement coverage is really bad or, if using mutation testing where you seed
the program with generated errors called "mutants" (for more info see section 3.7.2), in areas where very few generated mutants were successfully eliminated by the test suites[11]. For more information about these metrics see section 3.7.

White box testing is good for finding errors and reaching testing metrics goals, but when used for verification of the software specification the added insight into the program can lead to dangerous assumptions and loss of objectivity. To be able to use metrics to guide your testing efforts, these metrics have to be collected on a regular basis, which can require some effort.

## 3.6 Test Automation

Some repetitive testing tasks in software testing can be set up to run automatically or with minimal user intervention, this is referred to as test automation or automated testing. This process will introduce extra costs initially but if done correctly it can remove work load from manual testers.

### 3.6.1 What to Automate?

The Cábra team and stakeholders agree that testing should focus on the most business critical parts of the system. This would be the main workflow for handling incoming standard criminal cases from the Swedish Police from the point when they are sent to a prosecutor to the point when they are either closed by the prosecutor assigned to the case or they are sent to court for trial. This would be the test cases with the highest priority when performing regression testing before a new release.

This approach would fall into the category that Kasurinen, Taipale and Smolander call risk based testing, as opposed to design based testing. Where risk based testing tries to minimize the impact of any errors and still keep the deadlines in the project, the design based testing focuses on thoroughly checking that the program conforms to the specification. This is more resource intensive, which would explain why risk based testing is economically sound in organizations (like the Swedish Prosecution Authority) where there are not enough resources available to test everything before every release [9].

Automated testing is considered to make economic sense if the cost of implementing and maintaining the automated test cases is less than the cost of running the manual tests. Whether this is true depends on several factors: the cost of test implementation (including maintenance), the cost of test execution and the expected number of times a set of test cases will be executed over the life cycle of the software under test[10]. This can be formalized to the following formulas describing the cost of the testing[5]:

\[
A_M = V_M + n \ast D_M
\]

for a manual test, case where:

- \( V_M \) is the cost for specifying manual test case
- \( D_M \) is the cost of executing the manual test case and
- \( n \) is the number of times the test case is run

\[
A_A = V_A + n \ast D_A
\]

for the automated test cases, where:

- \( V_A \) is the cost for specifying and implementing the automated test case,
- \( D_A \) is the cost of executing the automated test case
- \( n \) is the number of times the test case is run
Then the break-even point for automating a test case can be calculated as:

\[ E(n) = \frac{A_A}{A_M} = \frac{V_A + n * D_A}{V_M + n * D_M} \]

and in their case study is to be a number somewhere between 1.4 and 6.2\[5\]. Others claim that \( n \) varies between 2 and 20 in testing literature \[14\].

Ramler and Wolfmaier claim that the previously mentioned model for evaluating the economics of automated testing is too simplistic. Their criticisms include the following points:

1. It considers manual and automated testing equal in everything but cost, but they claim it could be argued that they are not and that they provide different benefits.
2. Additional costs, mainly for test case maintenance, are not considered.
3. All test cases are considered to be of equal importance and their costs are not tied to any real world project context\[14\].

They propose a more realistic model to do cost/benefit analysis of test automation. In this model the budget for testing a system is fixed (which they claim is common in real life projects) and the goal is to find the optimal allocation of this budget between manual and automated tests. Their model looks like this\[14\]:

\[ n_A * V_A + n_M * D_M \leq B \]

Where:

- \( n_A \) = number of automated test cases
- \( n_M \) = number of manual test executions
- \( V_A \) = expenditure for test automation
- \( D_M \) = expenditure for a manual test execution
- \( B \) = fixed budget

In this model they make the assumptions that the cost of defining a test case is not part of the testing budget and that the cost of running an already implemented automated test case can be considered to be zero. With this model, the cost of manual testing is dependent mainly on the number of manual test runs and the cost of automated testing is dependent mainly on the number of automated test cases\[14\].

The authors propose that you use the whole testing budget so that:

\[ n_A * V_A + n_M * D_M = B \]

Both suggested economical models for evaluating if tests should be automated or not rely heavily on numbers that are estimated, and that can be quite hard to estimate correctly. I would argue that these models both describe the economy of test automation, but since they rely heavily on estimated numbers it is likely that having a good idea about implementation costs for the test cases and the expected number of times the test cases will be run is more important than the difference between the models. The important point that both models make still stands however, if you think you will run the tests enough times to make up for implementing and maintaining them as automated test cases you should do it, otherwise you should not. One important side effect of this point is that in most development projects there will be tests that should be automated, and tests that should not be automated.
3.7 Evaluating Testing Strategies

The aim of testing is to find bugs, and by doing so decrease the amount of bugs that remains in the system when it goes live. It is unlikely that all bugs are found, and even if they were there is no way to know this for sure. The best thing one can hope for is to test enough to find all serious bugs that are likely to be executed. Finding this enough-level is of course not a trivial problem, answering the question "What is adequate testing?" is not easy.

There are several ways to determine test adequacy. Usually you pick a measurement and you run enough tests to get this measurement to a high enough value. Common test adequacy metrics are[18]:

**Statement coverage** - If a high enough percentage of the statements in the program under test are executed by the selected test cases, then testing is considered to be adequate[18].

**Branch coverage** - Similar to statement coverage, but focuses on which of the different decision branches of the program under test that are executed by the selected test cases[18].

**Path coverage** - Measures the percentage of the paths through the program flow graph that are used during the execution of the selected test cases[18].

**Mutation adequacy** - Measures the amount of mutants generated by applying mutant operators to the original program that are detected, or “killed” by the selected test cases[18]. For more information on mutation testing see section 3.7.2.

Testing adequacy criteria can be based on either the specification of the program or the program itself. That is, one can either measure the amount of the specification of the program which has been tested or one can use knowledge about the program’s internal structure to determine which tests should be run. The former belongs to a method of testing that is usually referred to as black box testing, since it treats the program under test as a black box with unknown content. The latter belongs to what is referred to as white box testing, where the program itself can be studied when designing the tests. It is generally considered wise to use a combination of both when testing software[18].

Another way of categorizing a testing adequacy criteria is to look at the testing approach on which it is built[18]:

**Structural testing** focuses on coverage of code structure elements (statements, branches, etc)[18].

**Fault based testing** focuses on detecting faults in the software itself (even if the output is still correct). To measure adequacy we must somehow measure the fault detecting ability of the selected test cases, mutation testing would be categorized under this category[18].

**Error based testing** focuses on detecting deviations from the program’s specification. Here one would test the parts of the specification where the program is most likely to diverge[18].

The Caba team today uses Error based testing, where they test that the system behaves as specified. They test as much as there is time for before a new release and focuses on the things that are most likely affected by code changes in the new version. A new automated testing framework should have the ability to at least measure some structural testing metrics.
like statement or branch coverage so that it can be clearly documented what the automated tests actually test.

3.7.1 Fault Statistics Metrics

In order to evaluate and compare testing strategies and evaluate test work one needs an objective measurable way to decide the performance of a given test strategy. Chen, Probert and Robeson provide a summary of commonly used metrics to evaluate the test effort in a project. They then go on to propose some new ones. They categorize the different metrics under three goals for software testing[3]:

– Increased quality
– Decreased time-to-market
– Decreased cost-to-market

They then conclude that no single metric is enough to sufficiently measure the ability of a testing strategy to reach all of these goals. They propose to use a combination of several metrics. The metrics they suggest that you choose from are [3]:

**Quality Metrics**

**Quality of Code (QC)** measures the frequency of errors present in the code before it is sent to test[3]. If you weight errors found in test as less severe than errors found in a live system this will of course increase with more thorough testing. Otherwise this will not necessarily increase with more testing. If faults are found in testing that would have been undetected in production, then QC will actually go down.

\[
QC = (W_T + W_F)/KCSI
\]

Where:

– \( W_T \) = The number of defects (weighted by severity) found during test.
– \( W_F \) = The number of defects (weighted by severity) found after the system has gone live.
– \( KCSI \) = The number of thousands of changed lines of code.

**Quality of Product (QP)** measures the frequency of errors discovered in a live system[3],

\[
QP = W_F/KCSI
\]

Where:

– \( W_F \) = The number of defects (weighted by severity) found after the system has gone live. \( KCSI \) = The number of thousands of changed lines of code.

**Test Improvement (TI)** measures with which frequency faults were found by the test team, the higher TI value the greater is the test teams contribution to the quality of the product[3].

\[
TI = W_T/KCSI
\]

Where:

– \( W_T \) = The number of defects (weighted by severity) found during test.

**Test Effectiveness (TE)** measures the percentage of all found defects that were found before the system went live[3].

\[
TE = W_T/((W)_T + W_F)) \times 100\%
\]

Where:

– \( W_T \) = The number of defects (weighted by severity) found during test.
– \( W_F \) = The number of defects (weighted by severity) found after the system has gone live.
3.7. Evaluating Testing Strategies

Time to Market Metrics

**Test Time (TT)** measures the test time required in the project per thousand lines of code\[^3\].

\[
TT = \frac{T_T}{KCSI}
\]

Where: \(T_T\) = The number of business days spent on testing. \(KCSI\) = The number of thousands of changed lines of code.

**Test time over development time (TD)** measures the percentage of the total development time that was spent on testing\[^3\].

\[
TD = \frac{T_T}{T_D} \times 100\%
\]

Where: \(T_T\) = Time spent on testing. \(T_D\) = Total time spent on project.

Cost to Market Metrics

**Test cost normalized for product size (TCS)** measures the test cost per line of changed code.

\[
TCS = \frac{C_T}{KCSI}
\]

Where: \(C_T\) = total test cost. \(KCSI\) = The number of thousands of changed lines of code.

**Test cost as ratio of development cost (TCD)** measures the percentage of the development cost that was spent on testing.

\[
TCD = \frac{C_T}{C_D} \times 100\%
\]

Where: \(C_T\) = Total test cost. \(C_D\) = Total development cost

**Cost per weighted defect unit (CWD)** measures the ratio between testing cost and number of found defects

\[
CWD = \frac{C_T}{W_T}
\]

Where: \(C_T\) = Total test cost. \(W_T\) = Sum of all defects weighted for severity.

The problem with all these metrics is that they rely on knowing the number of faults found in the system in production. This is not always feasible, especially not for comparing different strategies which each other for the same code base. You cannot implement two different strategies and run them in parallel, with individual production systems and so on for the time needed to compare the results. You can however use these metrics to measure your organization’s test efficiency as it changes over time. Currently there are no metrics that are being used to evaluate the testing effectiveness in the Câbra team.

To be able to track the effects of changes to the team or the workflow it might be a good idea to start collecting a few of these metrics, since the idea is to use automated testing to find bugs before a release candidate goes from the development team to the testing team it could be interesting to measure the Test Improvement or Test Effectiveness of the automated test suites to see if it is giving good return of investment.
3.7.2 Mutation Testing

Ma, Harold and Kwon describes mutation testing as a way to test a test suite. Mutation testing means that errors are introduced into programs creating faulty versions of the program called mutants. These mutants are created by applying so called mutation operators to the original program. Ma et al divides mutation operators into two categories: class mutation operators (specific to object oriented programming languages) and traditional mutation operators (that are more generally applicable) [14]. This has the potential to make mutation testing more relevant to object oriented languages since errors specific for object oriented programs are not generated otherwise.

When experimentally evaluating testing methods and strategies it can be problematic to find programs of appropriate size with enough errors to provide a relevant testing sample. One way to overcome this is to seed an existing program of appropriate size with errors, either by hand or by having another program generate erroneous versions of the original program. The latter can be accomplished by applying mutation operators to the original program thus generating a set of mutants [2].

The main advantage of mutant generation over hand seeded errors is that the former provides a well-defined, reproducible way to introduce a large amount of errors into a program. This is important for others to be able to verify the results from past experiments, which is necessary for a scientific approach to test evaluation. Andrews et al admits that it could be argued that manually introduced errors would be more realistic, but also points out that any argument about how realistic an error is would be highly subjective in nature [2]. I would say that this might be a bit of a simplification; surely you could collect information about what kind of errors programmers actually make? Then there might be non-subjective arguments about the realism of an error. Realism aside though, being able to introduce large amounts of errors into a program in a reproducible way can still be very useful when evaluating testing in a scientific manner.

In order for mutation testing to be a meaningful way to evaluate test suites and testing strategies there must be a correlation between a set of tests detecting a mutation (“killing a mutant”) and the same set of test being able to detect an actual error in the program. Andrew et al tries to show that such a correlation exists by running test suites constructed by picking test cases randomly from a huge pool of test cases. Then running these test suites on versions of the tested program that are known to contain real defects, on versions of the same program with all known errors corrected but new errors introduced manually, and on versions of the same program with all known errors corrected but new errors introduced by applying a set of mutant operators [2].

Their set of mutation operators is a small one:

1. Replace an integer constant $C$ by 0, 1, -1, $C+1$ or $C-1$

2. Replace an arithmetic, relational, logical, bitwise logical, increment/decrease or arithmetic-assignment operator with another operator from the same class

3. Negate a condition in an if or while statement

4. Delete a statement

The conclusion reached is that if one is applying carefully selected mutation operators and eliminating mutants equivalent to the original program, one can use the result of the mutation testing as a reliable indicator of the fault detection ability of the test suite. They further conclude that faults introduced manually by humans into the program are harder to detect than both faults introduced through mutant generation and also harder to detect than
real errors. Results depending on a human subjectively choosing which faults to introduce into a program also impairs the reproducibility of the test evaluation[2].

Namin & Kakarla continue the work of Andrews et al. While they agree that mutation testing is considered a good way to assess the cost-effectiveness of test techniques, and that it has an important role to play in test evaluation more studies are needed to determine how dependent the quality of the mutation testing is on external parameters such as programming language and number of test cases[12].

Equivalent Mutants

Schuler, Dallmeier and Zeller points to cost as the biggest problem with mutation testing. The main focus when discussing the cost of mutation is the computational cost of generating all the mutants, compiling them (if they are not generated from already compiled code) and running the test suites on all of them. Another big cost in mutation testing is detecting mutants that are equivalent to the original program. This is an issue since some of the mutated programs will be functionally equivalent to the original program, and there is no possible test suite that could distinguish such a mutant from the original program. This mutant will never be killed and using it as part of you test suite evaluation will not be meaningful. Determining if a mutant is equivalent to takes on average 30 minutes of manual labor, considering the usually huge amount of generated mutants it is generally not possible to check all mutants manually. Heuristics and guiding mutation generation with genetic algorithms have been suggested to solve this problem[15].

Schuler, Dallmeier and Zeller suggest an algorithmic way to determine which of the generated mutants are more likely to be nonequivalent to the original program. This allows manual verification of nonequivalence to focus on the mutants that are most likely to provide a good test for the test suite. Their algorithm is based on first detecting the invariants of the program under test. The hypothesis is that mutants that violate at least one of these invariants are less likely to be equivalent to the original program[15].

The experimental tests conducted as part of Schuler, Dallmeier and Zellers article indicate that their hypothesis hold, in most of their tests they can even establish a direct relationship between the number of violated invariants in a mutant and the probability of that mutant being nonequivalent to the original program. Assuming that the results can be generalized to most Java programs, applying this method of selective mutation testing would introduce some bias to what kind of mutants would be used to test the efficiency of the test suites. This would in turn introduce a bias in which test cases one would implement in order to kill those mutants. The authors claim that this bias is actually something positive, that it will encourage tests of the parts of the code where an error would have the biggest impact[15].

In their experiment mentioned above Andrews et al have worked around the equivalence issue by making the assumption that the already existing test suite can find any errors in the program under test. Any mutant not killed by this test suite is then considered to be semantically equivalent to the original program[2]. This approach is very convenient for testing sets of tests that differ from this (assumed) complete and perfect set of tests, but the requirement of a large and known to be as good as perfect set of tests is a huge obstacle in a lot of cases.

There is no feasible way for this project to eliminate equivalent mutants. There is no library of old test cases to use as an equivalence check, there is no time to manually analyze all mutants and there is no good way to automatically determine mutant equivalence. Even without the obstacle of mutant equivalence, using mutation testing as a way to evaluate
testing is far too time consuming for this project.
Chapter 4

Accomplishment

4.1 Preliminaries

The original plan for the thesis project was to pick a few different testing tools and then quantitatively compare them to each other. This was based on the assumption that I was going to be able to acquire the help of some of the Cäbra developers in implementing the actual test cases; otherwise there would not be time enough to implement the amount of test cases necessary to compare different tools and strategies. When this assistance turned out to be unavailable while I was still working on the in-depth study and had to rethink the approach to the practical part of the thesis project.

Instead of doing a quantitative analysis of different tools, I decided on doing a survey of the different testing tools available that were applicable for a system like Cäbra. Then based on that in-depth study and the requirements of the Cäbra system pick one of them (or a set of them that could be combined) that I would implement a proof of concept testing framework and some test cases chosen to provide a basis for doing a more thorough evaluation of whether the chosen tool-chain would offer enough benefits compared to the current test framework to warrant rebuilding the existing tests with the new framework and including the new tools and libraries required in Cäbra’s external dependencies.

The plan was to spend 40 weeks working half time on the thesis project and half time as a consultant for Decerno. I planned to divide these weeks as follows:

An in depth study of testing literature where I read up on current testing research and methods. I have planned 10 weeks for this, the first two weeks I will be focusing on only this, but then I will also start the survey of available testing tools and the study of the Cäbra system. I plan to use both published books and articles from the ACM library as my sources.

A study of the Cäbra system architectural design and source code. Includes documenting the existing testing framework. This is part of the implementation work rather than the theoretical study. I estimate that this will take around five weeks. I plan to start this as soon as I get access to the testing environment and the source code.

A survey of available testing tools relevant to the project. From this I select one/a few that can be used together for an in depth evaluation. Finding candidate tools for this survey will be done using web search engines and checking current topics on test focused Java conferences. Estimated time for this is four weeks.
An implementation of a new testing framework using the tool(s) selected above, estimated to take 12 weeks.

An analysis of whether the new testing framework offers enough benefits to warrant a switch from the old one. I estimate four weeks for this.

This report which will take another five weeks of work spread out over the whole project.

This sums up to 40 weeks of half time work.

4.2 How the Work Was Done

There were some delays to the thesis project due to other project of higher priority to Decerno demanding attention.

4.2.1 Literature Study

After initial discussions with Decerno about my thesis project specification I started the literature study as planned. In the beginning the exact focus of the thesis project was a little unclear, so some time was wasted on reading up on unrelated topics. Other than that and having to add some details towards the end of the project the theoretical study went as planned.

4.2.2 Survey

It was very hard to draw the line on when to stop looking for new tools, so I spent more time than planned on this activity. I had a tendency to go back and look for more potential tools whenever I got stuck in another part of the project. In retrospect I should have set a very firm end criterion before I started looking and then not allowed myself to go back to looking again.

4.2.3 Studying Câbra

It took longer than I expected to understand the system, I grasped the architecture, database model and the source code structure fairly quickly, but there is a lot that you need to know about the workflow of a Swedish prosecutor if you want to write good automated test cases that mimics actual use of the system.

4.2.4 Implementation

The implementation was done in parallel with the later part of understanding the Câbra system. At some points it was hard to differentiate what is implementation and what is study of Câbra since implementing test cases required me to know how the system was supposed to be used and it was supposed to behave.

4.2.5 Analysis and Report

Writing the report took longer than expected, in general I think I did not take report writing time into account properly when estimating the time required for the other parts of the project, since documenting for the report has to be done in all parts of the project.
Chapter 5

Results

The results of this thesis project is a small survey of Cäbra’s current automated testing work, a survey of available testing tools that could be used to improve Cäbra’s test framework, followed by an in depth study with a proof of concept implementation using a selection of the tools that are deemed most likely to be able to offer benefits to Cäbra’s test framework.

5.1 Existing Framework

The existing tests are a bit of a mixed bag. They are all implemented using some version of the JUnit library. Some of them are pure unit-tests while some are integration tests. The unit tests use a recent version of JUnit4 while the integration tests are still running on JUnit3. The JUnit4-tests are run every build and run very quickly. The JUnit3 tests have been implemented ad-hoc when it has been necessary in the past to make sure a new piece of functionality works as intended, or to ensure that a particular bug does not re-surface again. Most of the tests have not been maintained for a while and it is not known whether they will run or not at present.

All the JUnit3 test cases share some common characteristics:

They use delegates to interact with the server. This limits the amount of white-box testing that can be used slightly since the tests only have access to what the normal clients can access. White box meta data such as statement coverage etc. can still be used, but if the tests are going to access any server code to trigger specific events or do specific checks support for this must be added all the way from the delegate layer to the session layer. The benefit of this approach is that it makes the test code behave more like the production code client, which reduces the amount of untested code.

They rely on code that is used for non-testing purposes. In order to educate prosecutors in the use of the Cäbra system there is a generator that sets up fake prosecutor’s offices with fake case data. This code is also used by the test cases. Any commonly used data generator code should be moved to a specific library.

They do few asserts and mainly test that no exceptions are thrown during execution, although there are a few exceptions to this rule.

They all use some fixture class to connect to the server and login. This fixture class usually also contains some test suite specific utility methods and most of the test suites have their own fixture-class, this causes a lot of code duplication.
Two test suites are still being maintained properly and run on a regular basis, they both consist of a set of tests for receiving data from external systems. These two suites are running the same set of tests but with two different formats for the input data. They run every night and the results are actively monitored by one of the Cäbra team members. These are the only JUnit3 test cases that actually do any checks on the object states and data to ensure correct behavior of the program, but even here there is room for improvement. They are dependent on data files that are time consuming to update if the data format should change, but so far this has not happened too much.

The module-tests are not really in need of updating, more module tests would be useful but the ones that exists use up to date versions of libraries and are in good shape. The integration tests should be updated to JUnit4 and any code duplication that can be avoided should be avoided, so even without a new test framework there is work to do on the current tests, not to mention that there is definitely room to implement more tests.

5.2 Available Java Testing Tools

5.2.1 TestNG

TestNG is an alternative test runner that can be used instead of JUnit. It was created to overcome some problems in JUnit3. Most of these problems were addressed in JUnit4, but TestNG still offers some functionality not available in JUnit. There is a more powerful and flexible system for grouping test cases together, support for specifying dependencies between test cases and better support for parameterization of test cases. Cäbra’s test suites are not expected to grow beyond what JUnit is capable of grouping and organizing and there is currently no use case for parameterization, so the extra functionality does not offer any benefits at this point. Besides that there is the fact that TestNG uses XML files to handle some things that in JUnit4 is done in Java code, where the team explicitly prefers the latter option. There is there is not enough motivation to replace a tool the team already knows and that is well integrated in the development environment.

5.2.2 Behavior Driven Design Tools

Behavior driven or test driven design are two very similar and interesting approaches to development where you always implement the tests before the actual functionality. There are a number of tools available for working like this in Java. This includes JBehave, JDace and Cucumber-JVM. While it is possible that these tools are very good at providing support for behavioral testing, behavioral testing does not offer enough benefits to the Cäbra system at this point since they would require the team to change its way of working a lot to be useful.

5.2.3 Java Enterprise Testing Tools

Testing Java Enterprise systems comes with a set of problems that you do not encounter when testing classic applications. In order to do integration testing of the system, you need to deploy it in a Java Enterprise Container; usually this also needs to have a database back end. Apart from these dependencies, Cäbra is also dependent on certain basic data domains being available in a database back end to function properly, this makes unit testing difficult and time consuming in the current system. There are a few testing tool for testing Java Enterprise applications that tries to remedy these problems, they all to some extent let
you run your tests inside an actual Java Enterprise Container without having to handle packaging and deployment using external tools.

These tools can interface with containers in a few different ways. They can run an embedded container, that is a container that is started in the same Java Virtual Machine as the test run itself, they can deploy the test package in a local container that runs on the same machine as the test code or they can deploy the test package in a remote container that they access using TCP/IP.

**Pax Exam**

Pax Exam is a tool to test Java applications that depend on either Java Enterprise Edition or OSGi for dependency injection and configuration, where OSGi is the main focus. Since Câbra is not built as an OSGi application it is the Java EE parts that are interesting. Pax Exam features automatic construction of deployment packages at the start of a test run when testing Java EE applications. It does this by including all classes that are in the classpath when the test is started. Pax Exam supports several embedded and local Java EE containers. The framework is being developed as an open source project that is free to use and distribute.

**Cactus**

Cactus is an older framework for running tests on Enterprise Java Beans in an actual EJB container. Apart from this it also has some support for calling servlets and Java Server Pages from the test cases. The ability to deploy code to an EJB container and run tests inside that container could be useful, but they are also provided by Arquillian and Pax Exam. The web related features are less useful for testing Câbra since there are no web based components in the system at this point. Like many other less popular open source projects Cactus has seen very little development in the last few years and are falling behind the rest of the Java world technology wise. Chances that Cactus in its current state would attract enough contributors to catch up with the other frameworks are slim.

**Arquillian**

Arquillian leaves it to the developer to construct the deployment package at the start of the test case, using a library called ShrinkWrap. If the system under test is well modularized this gives the tester the opportunity to deploy only the subset of the system required for the test suite before running the test, cutting down on test turnaround time. The generated package can then be deployed in an embedded, local or remote container. Arquillian supports a wide range of different containers and has a plug-in system to make it easier to add support for more containers if necessary. Arquillian supports running tests in the server container, tests running outside the server (like a client would) or interfacing with the server through a web browser, the first two could be useful for testing Câbra, but at this point there are no web components to test in the system.

### 5.2.4 Mocking Frameworks

Mockito is a mocking framework that is already in use in the Câbra team. In order to for a change of mocking framework to make sense, the new one has to offer something really useful that Mockito does not. From the other available object mocking frameworks, JMockit seem to offer the most interesting new features, mainly the ability to mock static methods,
but this is also available in the PowerMock extension to the Mockito API. Since the people involved are already familiar with Mockito and there is little need for new features there is not enough motivation to introduce another mocking framework at this point.

5.3 Which Tools to Use in the Prototype?

The tools that are mainly focused on Java EE testing seem to offer the biggest potential benefit. They could in theory replace a lot of the current test fixture classes and eliminate the need for any delegates used only for testing purposes. Another very interesting prospect is that they could, even though they are technically integration testing frameworks, offer a way to implement more module testing in Cábra. A lot of the business logic in Cábra is controlled by the data in the database. There is a lot of behavior that classic unit tests, which traditionally should not rely on any external components such as databases, cannot be used to verify. Isolating modules in the session layer of Cábra would require implementing special testing facades and delegates since the tests are run as client code. These tools offer a way to run the tests on the server, in the EJB container, with full access to all the Session EJBs which would eliminate the need for extra test specific client code. The tools that offer dynamic creation of deployment packages for a test suite would make it possible to, in an easy way, replace some of the enterprise Java beans with stubs, which would make it easier to isolate the modules under test and achieve a more controlled environment.

Apart from the module testing opportunities offered by Arquillian, the potential for shorter test turnaround times could mean that it would be feasible for developers to run some in-container testing before checking in modifications. Since Arquillian builds a development package as part of the test suite it could be possible to test the current working copy of the code in the developer’s development environment without having to re-package and re-deploy using build scripts. If deployment packages with only a subset of the Cábra system could be used to test parts of the Cábra system then test turnaround time would most definitely be quicker with Arquillian, but if the whole system needs to be packaged and deployed anyway, it is just a matter of the developer having to push one button less. If tests are run on a nightly basis, easy deployment without build scripts and quicker run times are of course less important.

Of the three tools listed above Cactus is easily the least interesting, it is not well maintained and it lacks support for modern Java EE code. Arquillian and Pax Exam are harder to choose between. Pax Exam’s main benefit over Arquillian is that it supports automatic generation of deployment packages from the test’s classpath. Arquillian’s main benefit over Pax Exam is support for Oracle Weblogic, which is what Cábra is currently running on. In the end it seems easier to implement a deployment package creator than to add support for Weblogic to Pax Exam. Running the Cábra tests in another container could be an option if the goal was to only create module tests, but this is not the case, integration testing is a requirement and then we want the test environment to be as much like the production environment as is reasonably possible.

The test database handling toolkit DBUnit is included in the Arquillian framework. It offers support for storing test data sets in files. These data sets can be imported into the database for test setup or used to verify that the state of the database is as expected after a test has been run. This could provide a good way to handle test data.
5.4 Implementation

In order to be able to do a more in depth compassion of Arquillian and the current testing framework and to be able to analyze what benefits Arquillian could potentially bring I had to actually use it to write automated test cases for Cábra. In order to be able to do this I had to make a proof-of-concept implementation of a new test framework for Cábra using Arquillian, DBUnit and VirtualBox. A lot of time was spent tweaking configurations to get Arquillian, VirtualBox, Eclipse and WebLogic to work well together. To be able to compare the new test framework to the old one in terms of ease of use and test execution time I started by implementing some of the existing test cases in the new framework. These test cases were followed by some illustrative test cases to show what was possible to do when testing Cábra with Arquillian.

5.4.1 Test Executor

Arquillian supports both TestNG and JUnit as the test executor for the test cases. The main benefit of TestNG is that it supports more advanced dependency relations between different test cases and has better inheritance support for test case types extending other kinds of test cases. These benefits were not considered to be important enough to introduce a new test executor to a team that is already familiar with JUnit.

5.4.2 Test Data and Expected Results

Creating good test data is not a trivial task. The test data from the previous test cases will be re-used. Since it is all stored in code, it might have to be exported to another format for DBUnit. Any new test data created manually will be based on anonymized production data to be realistic, or if necessary custom tailored to trigger specific behavior in Cábra.

Generating the expected result of tests can be very time consuming if done manually, the problem with generating this automatically is that it assumes that the program making the generation is correct. Since the system is quite mature and the project is focusing on regression testing, I will use the system under test to generate the expected output from different tests since you want to test that future changes in the program does not unexpectedly change any old behaviors.

5.4.3 Data Handling

To get the correct data into the database before a test and to check that the database is in the expected state after the tests have been run is a time consuming process. In the old Cábra tests this is done by using the Cábra API, both for checking the state of the data container objects and for creating new data. Some of this data is kept in the source files and some in external files that are sent to Cábra as messages from external systems. Storing data in source code can be convenient since both the test case and its data can be in the same file, but it also makes it harder to re-use data between different test files. Using the Cábra API to get the data into the database makes it somewhat easier to keep the test data up to date with the database model, any changes hidden by the API will not affect the test data and any changes resulting in compilation errors will be easy to spot. The downside of using the API is that it is a lot slower than just reading data straight into the database.

Using DBUnit data stored in files with different file formats can be read into the database, deleted from the database or used to assert that the database contains certain rows. Since there is a lot of relations and constraints in the Cábra database, I had to implement an
extension of the DBUnit XmlDataSet class that gave the user more control over in which
data was inserted and removed from the database. Even with this customizable
insert and delete order there were deletes that you could not execute without first removing
some references to other tables, this is a known problem and there is a stored procedure
implemented in the database for deleting a complete case with all data that it references, as
long as no other case reference the same data. Performing a complete wipe of the database
works but that has some unfortunate complications:

– All developers use the same database as a back-end for their locally running version
  of the Câbra server, removing all the data will cause a lot of problems for others

– A lot of the business logic of the system is stored in the database and this data is
  required for the system to work properly

One way to overcome the first problem would be to have developers run their own
database servers, but keeping everyone up to date with the latest database schema and
business logic control data would be an increased work load for the team, and since most
of the developers need some existing data for manual testing of new features cleaning out
all the data is still not optimal. If Câbra had used Java Persistence Architecture (JPA) then
DBUnit could supposedly have generated a test database from the JPA specification in some
kind of light weight in memory database server, but this is not the case. In the end DBUnit
is best used for putting data into the database or checking the contents of the database, the
deleting can be done by calling the stored procedure already in place for this.

Using DBUnit to create complex data with a lot of relations are time consuming since
you have to set up all the relations manually in the data, and reusing part of the data in
another context usually does not work. If the data set is too large the xml file gets very
hard to understand and manage. During this project there was never any need to update
any of these data sets, but this would be an issue even for data sets of limited size. Setting
up data by calling the Câbra API directly gets the relations correct (or as correct as they
would be in reality anyway), let you use loops and parameterization to a greater extent and
makes keeping the data up to date easier since most database changes are hidden by the
API. For this reason I complemented the existing helper methods for data creation with
some new helper classes designed using the builder pattern that is used in for example the
Java StringBuilder class that helps the developer of a test case create instances of the most
common database entities such as cases and actors.

An alternative to manipulating the database when setting up a desired state before
running a test suite is to run the database in a virtual machine that permits loading a
known state from disk before the tests start. The result of the test runs are then stored in
da delta file which can quickly be thrown away when it is no longer needed, making resetting
to the known starting state easier and faster. The main drawback to this approach is that
the team would need to distribute large files containing the disk image of the latest version
of the system after each upgrade, or that every team member would have to keep their own
virtual test server up to date with all new developments. Since this and the licensing of
virtualization software was deemed not cost efficient enough the idea was scrapped after
some initial tests.

5.4.4 Deployment Package Creation

One of the most prominent features of Arquillian is that it generates a deployment package
at the start of a test suite, deploys that package, runs the test suite and then undeploys the
package. The package has to be created programmatically in the setup phase of the test
suite. In the simple examples used by the team behind Arquillian to illustrate this feature this looks very straightforward and easy, but with a bigger system it is not as simple. The sprawling dependency graph of Cábrea presents a problem here, I considered trying mapping the dependencies programmatically, but I decided it made the code too complex and too time consuming to run for a limited potential benefit from decreased deployment time. Instead I used the existing Java package structure to add all classes in the specified packages to the deployment package. It is still left to the developer of a test suite to specify which packages are necessary to run the test suite.

Unless the test suite is really long, packaging and deployment of Cábrea consumes a large part of the test run time. If all tests are run in a single suite this is of little concern, since any change in the server code requires a re-build and re-deploy of the system anyway, but the more test suites you want to run after a server code change, the more time is going to be spent packaging and deploying compared to if you use the current test framework. When implementing test suites in the Arquillian based framework the test cases should be put in as few test suites as possible if test run times are a concern. With the current testing approach tests are only run during the night, so run time is not a big concern.

It is possible that package size can be reduced by re-working some dependencies and trying to make different parts of the system more independent from each other. The most affordable and probable way to reduce package size however would be to replace some modules in the system with mocks or stubs. Since most central functions in the system are accessed by loading a class that implements a certain Enterprise Java Bean interface, test suites that focuses on some specific functionality could replace the other parts with mocks. This would provide better opportunities for module testing the system in a controllable way and decrease the size of the deployment packages.

### 5.4.5 EJB Mapping

When you run test cases through the facade layer like a normal client would; any session Enterprise Java Beans that you used are mapped in Cábrea using Java Annotations. This does not work if you try to lookup any session beans in another context when executing an Arquillian test case. An alternative way to map these session beans from their specifying interfaces to the implementing classes is to use an xml-file in the enterprise archive. By using the Java Reflection API and the existing annotations in the Cábrea code I generate this xml-file when I generate the deployable test archive.

Any of these mappings can be substituted by a mock implementation of one of the bean interfaces, this way the real implementation of mocked beans can still be present in the package, but since the specifying interface is mapped to the mock implementation the mock will be used instead. This allows the developer of a test case to use mock implementation of EJBs without having to remove any specific classes from the deployment package, just changing the mapping will be enough. Since module testing can be very useful and cost efficient and the most natural way to divide the server part of Cábrea into modules is to divide it into the separate beans this should be enough to support cost effective module testing in Cábrea.

### 5.4.6 Arquillian compared to the Current Testing Framework

When comparing Arquillian to the existing framework, the first observation one should make is that it is entirely possible to run the old test cases in Arquillian. This is made possible by Arquillian’s support for running test cases that behave like a client, even though the main
focus is running test cases deployed in the server environment. Running the old test cases in Arquillian would add the extra overhead of package creation on deployment, but as long as test run time is no issue Arquillian offers all the functionality of the old testing framework, and then add other features on top of that. If the question was only which framework offers the most functionality then the answer is obvious, the question should rather be if implementing the use of Arquillian in testing in the Câbra team is worth the effort and a new external dependency.

Arquillian seem very promising at first when you read the official guides, a simple way to run tests on individual enterprise Java beans in an actual EJB container connected to an actual data source. The promise of "No more mocks" is partly true, you no longer need to create mock implementations of data sources or other services unavailable in your testing environment, but if you want to test on EJB separately you still need to create mocks of any beans it interacts with.

The ability to easily replace parts of the system with mocks is the main advantage over the old system. Module testing is considered to be a cost effective method for achieving good testing coverage and with the current framework there is a lot of work involved if you want to run your test with parts of the server side code replaced by mocks or stubs. Since EJBs are meant to be modular and possible to substitute with another implementation of the same bean interface it is quite easy to replace a bean with a mocked implementation with a fixed behavior. With the EJB mappings in an xml-file generated in the test code, a mock implementation can easily be substituted for the original implementation when the deployment package is created in the beginning of the Arquillian test suite.

The second main advantage when using Arquillian is that it eliminates a lot of extra test code, the test fixtures and extra delegates are no longer necessary if you run your tests inside the container. To fully take advantage of this though, you have to stop running tests that interact with the server via delegates like the old test framework does and that would mean that you no longer run any delegate-code in your tests. This might be a problem, but in general the delegate code is pretty simple, it prepares data that is then sent to the server. This could probably be more efficiently tested using classic unit tests that replace the server with mocks that responds in different pre-programmed ways dependent on the test case. Even with the added extra code for preparing deployment packages you should be able to reduce the amount of source code that you need to maintain in your test cases by using Arquillian.

In the old testing framework data is entered into the system in two ways. By simulating received messages from external systems and by creating the data by calls to the Câbra API in the test code. In the new test framework there is a third option, using DBUnit to write data directly into the database. This has the advantage of being faster at run time, but it requires a lot more work when setting up the data and has fewer opportunities to use parameterization and loops to reuse parts of the data. As stated earlier there are problems with using DBUnit to delete test data after the test run. All in all DBUnit does not add a lot of value to the testing in Câbra. If you want to import a lot of simple rows into the database there might be a point in using DBUnit, otherwise the Arquillian based tests will generally use the same methods as the old framework to set up test data. The builder classes created for the new testing framework does not rely on Arquillian so they could also be used if new test cases are implemented in the old framework.

The main drawbacks of using Arquillian is the extra work involved initially, the old framework is already in place and the few people in the team that work with it know it well. Moving the tests to Arquillian means adding new dependencies, updating build files and re-implementing some parts of the test cases. When the project started the current level of
ambition with automated testing in Cäbra was quite low and no mayor additions to the two currently running test suites were planned. There was however hope that this project could spark some initiative and enthusiasm for more automated test cases. Whether migrating the test cases to Arquillian or not is worth the effort depends on how much modularization of the test cases that the team intends to do and how much new test cases will be added. Arquillian on its own will not create better tests, but it might make it easier and less time consuming to add more test cases in the future.
Chapter 6

Conclusions

Testing is an important part of software development, all serious development organizations should spend some effort at testing the system they are developing before releasing it. The cost of testing can be motivated by the lower cost of fixing a bug early in the development cycle compared to later; this is even before you count the costs a fault in a production system can cause, either by loss of income or user goodwill. Testing is an expensive, time consuming and sometimes tedious and repetitive task. Like many other tedious repetitive tasks a lot of the work involved in testing can be handled well by computers, automating tests can be a way to achieve more testing without increasing the costs.

The outcome of this thesis project was an evaluation of the possibility of using the testing framework Arquillian for testing the Cäbra system. Arquillian was chosen for an in depth evaluation after studying the current test frameworks of the system and doing a survey of currently available free testing software for Enterprise Java systems. The project went well, but the time plan was not perfect, a slow start and initially aiming for a more ambitious project that turned out to be impossible with the time and resources available combined with having to put the project on hold to focus on other assignments added a lot of lead time and some extra work to the project.

6.1 Test Automation

Not all tests are suitable for automation; in some cases the initial cost associated with the implementation effort involved in automation is too high. In order for automation to make economic sense the expected long term savings in reduced costs for running the test cases most outweigh this implementation cost. This depends on the cost of automating the tests, the costs of running the tests (manually and automatically), the cost of maintaining the test code for automated tests and the number of times the tests in question are expected to be executed. Apart from economic reasons there are other reasons why a test should not be automated, tests of explorative nature or user acceptance testing are examples of tests that rely on a human tester.

There are different ways to prioritize testing efforts, you can prioritize the most critical code where a fault would be most costly, the most central code that is used the most by the program or prioritize to test as much of the code as possible measured in lines of code. When doing white box testing different metrics can be used to guide testing implementation to cover areas not tested properly yet. At this point the tests in the Cäbra system cover very little code and very little functionality. Since the automated tests will be run to verify
that the system is ready for manual testing the most central functions of the system will be initially prioritized in test automation. After the most important functions have automated testing some coverage data should be used to guide further testing to avoid unnecessary duplication of tests.

Testing can be done at different detail levels, from testing individual method calls in unit tests to the whole system and all its external interfaces in systems testing. Only testing on one level is not optimal. The smaller the component under test the easier to create test cases that isolate extreme use cases to make sure they work correctly and the easier it is to find what is causing any tests to fail. The bigger the component the more certainty that the components of the system behave well together and don’t cause each other to fail. A combination of several different levels should be used preferably.

6.2 Test metrics

To have any idea on how your testing efforts are going you have to somehow measure their success. One should take an interest in the long term quality trends. There are several different metrics that are easy to track in an organization where the source code version control system can report the number of changed lines between two releases and where bugs are reported in a bug tracking software. In an organization like the Câbra team where the number of available testers and developers do not vary so much over time it might be more interesting to measure the quality or the time to market rather than cost to market.

Using metrics such as code coverage or mutation testing to guide your testing efforts can be a good way to avoid testing the same thing more than once and to find out which parts of the code need more tests. This is more easily done with any automated tests since it is easy to run them all in the same test environment and then collect the coverage data from there. Once the automated tests of the Câbra system starts to grow beyond the most basic tests, efforts should be put into collecting some kind of coverage data to be able to track what the automated tests cover. Mutation testing is an interesting way to measure the quality of a test suite, but it is still too costly to implement and run.

Today all the tests in Câbra are black box tests. They interact with the system through the normal external interfaces and they verify the test results the same way. If you would begin collecting metrics from the test cases you could use this data to guide your testing efforts and by doing so combine the black box approach in place today with some white box testing. White box testing is not without faults, but it is very worthwhile to use both approaches.

6.3 Evaluation of Arquillian

Arquillian may promise more than it can deliver at the moment, but that does not mean that it is not a useful tool for software testing. It offers a way to run tests from JUnit (or TestNG) in a remote EJB container. The tests run inside the container and can access data sources and Enterprise Java Beans just like the production code, so no extra test fixture is needed to expose anything to the tests that are not normally exposed in a production environment. This has the potential of reducing both implementation and maintenance costs for automated testing, making more test cases economically feasible.

The packaging of deployment packages at the start of the test adds a bit of implementation work and increases test run time. Since this only changes when the set of classes needed for a test changes it should at least require less maintenance work than implementing your
6.4 Limitations

own client. An advantage that this procedure brings with it is that it makes it very easy to replace any of the EJBs with another implementation of the same interface, for example with a testing stub providing a certain set behavior to an interface.

Easily substituting the normal implementations of EJB interfaces would enable better module testing, which is today hard to achieve in the Cabra system. Currently the Cabra automated testing effort focuses on running tests on the complete system, this should be complimented with module tests. While you should not rely entirely on module testing, it has proved to be cost effective, especially for increasing testing coverage and in executing hard to reach sections of the code, it is also easier to debug a failing module test than a failing system test.

Arquillian also enables easier testing of the code in the session layer since it can call the methods on the session beans directly. This would make it easier to achieve test coverage of rarely occurring cases in the session layer of the architecture.

6.4 Limitations

There is no way to automatically package all dependencies for a certain set of Java classes in the packaging step of the test suite, so either the person implementing a test suite has to package the complete Cabra system (maybe with some parts replaced by stubs) or manually construct a smaller independent subset of the system that can be deployed without the rest, if such a subset exists. Developing some kind of tool for this would make it easier to test smaller parts of the system, without having to create stubs.

6.5 Future work

The current implementation is set up to run the tests on Oracle WebLogic server, either local or remote. Since this is the environment used both for test and production there is no pressing need to try other configurations, but it could be interesting to investigate running tests in a simplified embedded EJB container that is more light weight and does not run a full database server. This could maybe be used for some module testing that does not rely so heavily on the database, if a relevant test case can be specified that would benefit from this setup.
Chapter 7

Acknowledgments

I would like to thank Decerno for giving me the opportunity to work on this project at all especially Mats Dahl and Fredrik Lann who have been my supervisors at Decerno. I would also like to thank Axel Olsson who has supplied knowledge about Câbra as well as some much needed enthusiasm for this project.
References


