Unit Testing of Java EE Web Applications

CHRISTIAN CASTILLO
and
MUSTAFA HAMRA

KTH Information and
Communication Technology

Bachelor of Science Thesis
Stockholm, Sweden 2014

TRITA-ICT-EX-2014:55

Unit Testing of Java EE Web Applications

Christian Castillo
Mustafa Hamra

Bachelor of Science Thesis ICT 2013:3 TIDAB 009
KTH Information and Communication Technology
Computer Engineering
SE-164 40 KISTA

KTH Industriell teknik och management

Examensarbete ICT 2013:3 TIDAB 009

Analys av testramverk för Java EE Applikationer

Christian Castillo
Mustafa Hamra

Godkänt	Examinator	Handledare
2014-maj-09	Leif Lindbäck	Leif Lindbäck
	Uppdragsgivare	Kontaktperson
	KTH/ICT/SCS	Leif Lindbäck

Sammanfattning

Målet med denna rapport att är utvärdera testramverken Mockito och Selenium för att se om de är väl anpassade för nybörjare som ska enhetstesta och integritetstesta existerande Java EE Webbapplikationer. Rapporten ska också hjälpa till med inlärningsprocessen genom att förse studenterna, i kursen IV1201 – Arkitektur och design av globala applikationer, med användarvänliga guider.

KTH Industrial Engineering and Management

Bachelor thesis ICT 2014:6 TIDAB 009

Unit Testing of Java EE web applications

Christian Castillo
Mustafa Hamra

Approved	Examiner	Supervisor
2014-maj-09	Leif Lindbäck	Leif Lindbäck
	Commissioner	Contact person
	KTH/ICT/SCS	Leif Lindbäck

Abstract

This report determines if the Mockito and Selenium testing frameworks are well suited for novice users when unit- and integration testing existing Java EE Web applications in the course IV1201 – Design of Global Applications. The report also provides user-friendly tutorials to help with the learning process.

PREFACE

The report is a Bachelor Thesis that has been written in collaboration with the Department of Software and Computer Systems (SCS), School of Information and Communication Technology (ICT), Royal Institute of Technology (KTH). The purpose of this thesis is to analyze which unit testing frameworks and integration testing frameworks are well suited for Java EE applications for the course *Design of Global Applications*, IV1201. Being an academic report meant a close cooperation with our supervisor/examiner. Specifically, this study meant acquiring a strong grasp on the different frameworks such as Mockito framework extension over JUnit or JSFUnit, before implementing these on our previous Java EE code projects from when we attended the course.

With this in mind, we like to thank our examiner and supervisor Leif Lindbäck at the Royal Institute of Technology (KTH) for his immense support and time dedicated into helping us throughout the project.

Christian Castillo and Mustafa Hamra

Stockholm, June 2014

Abbreviations

CDI	Context Dependency Injection
GUI/UI	Graphical User Interface/User Interface
HCI	Human-Computer Interaction
ICT	Information and Communications Technology
IDE	Integrated Development Environment
IMRaD	Introduction, Method, Results and Discussion
KTH	Royal Institute of Technology
OS	Operating System
OSGi	Open Services Gateway Initiative
PC	Personal Computer
SCS	Software and Computer Systems
SUT	System Under Test
TDD	Test-Driven Development
URL	Uniform Resource Locator
XP	Extreme Programming

TABLE OF CONTENTS

PREFACE ..... **IV**

NOMENCLATURE ..... **V**

1 INTRODUCTION **1**

1.1 BACKGROUND 1
1.2 PURPOSE 1
1.3 DELIMITATIONS 2
1.4 METHOD 3
1.5 DISPOSITION 4

2 FRAME OF REFERENCE **5**

3 THEORY **6**

3.1 UNIT TESTING 6
3.2 INTEGRATION TESTING 7
3.2.1 Big Bang Approach *7*
3.2.2 Top-down Approach *9*
3.2.3 Bottom-up Approach *11*
3.3 MOCKITO 12
3.4 SELENIUM 14

4 THE PROCESS **17**

4.1 TEST CASES 17
4.1.1 Test the logger *17*
4.1.2 Test of login method *18*
4.1.3 Test of getters and setters *20*
4.1.4 Test of login interaction *21*
4.1.5 Test the login interaction & update status *22*
4.1.6 Test of creating an application *22*
4.2 TUTORIALS 23

5 RESULTS **25**

5.1 MOCKITO TEST RESULTS 25
5.1.1 Results for test case: Test the logger *25*
5.1.2 Results for test case: Test of login method *28*
5.1.3 Results for test case: Test of getters and setters *30*
5.2 SELENIUM TEST RESULTS 33
5.2.1 Results for test case: Test of login interaction *33*
5.2.2 Results for test case: Test of login interaction & update status *34*
5.2.3 Results for test case: Test of creating an application *35*
5.3 EVALUATION OF TUTORIALS 37
5.3.1 Resulting structure of tutorials *37*

6 DISCUSSION AND CONCLUSIONS **39**

6.1 DISCUSSION OF TEST CASE RESULTS 39
6.1.1 Discussing results for test case: Test the logger *39*
6.1.2 Discussing results for test case: Test of login method *40*
6.1.3 Discussing results for test case: Test of getters and setters *42*

6.1.4 Discussing results for test case: Test of login interaction *43*
6.1.5 Discussing results for test case: Test of login interaction & update status *43*
6.1.6 Discussing results for test case: Test of creating an application *43*
6.2 DISCUSSION OF TUTORIALS 44
6.2.1 Tutorial for Mockito *44*
6.2.2 Tutorial for Selenium *45*
6.3 CONCLUSION 47
6.3.1 Frameworks *47*
6.3.2 Tutorials *47*

7 RECOMMENDATIONS AND FUTURE WORK 48

7.1 RECOMMENDATIONS FOR A SUSTAINABLE FUTURE 48
7.2 FUTURE WORK 49

8 REFERENCES 50

APPENDIX A: MOCKITO UNIT TESTING TUTORIAL 52

A.1 DOWNLOADING THE NECESSARY FILES 52
A.2 IMPLEMENTING MOCKITO TO YOUR JAVA EE WEB PROJECT 52
A.3 SETTING UP TEST ENVIRONMENT FOR MOCKITO 53
A.4 WRITING A SIMPLE TEST WITH MOCKITO 55
A.5 EXECUTING A TEST 59
A.6 THE MOCKITO API 60

APPENDIX B: SELENIUM FRAMEWORK TUTORIAL 66

B.1 DOWNLOADING THE NECESSARY FILES 66
B.2 IMPLEMENTING SELENIUM TO FIREFOX AND NETBEANS 68
B.2.1 Creating test through Netbeans IDE *70*
B.2.2 Exporting recording to Netbeans IDE *73*
B.3 RECORDING WITH SELENIUM IDE PLUG-IN 78
B.4 IMPLEMENTING A TEST THROUGH NETBEANS IDE 81
B.4.1 Guidelines for a manually coded test *81*
B.4.2 Implementing an exported recording *86*
B.5 EXECUTING A TEST 88
B.5.1 Executing a manually coded test *88*
B.5.2 Executing recording in Selenium IDE *89*
B.5.3 Executing exported recording *90*

1 INTRODUCTION

This chapter covers the background, product and why our thesis project is needed. Also, our tasks are explained in detail.

1.1 Background

Unit testing is an optional part of a major Java EE project. This project covers most of the time for the course *Design of Global applications, IV1201*. It is a web-based recruiting system where applicants can apply for a job by filling out a form. The project also lets a recruiter log in to the system and read the applications in order to decide which applicant or applicants to hire for a certain job.

The recruitment system is fictive and the jobs are not real. It only exists for academic purposes and the point is to teach students on how to code a Java EE Web project from the ground up. It is also the code base used for testing during this thesis project.

During the course project, a set of goals are given to the students. The goals are in the form of different functionality that, if implemented, yields a certain letter grade. One optional goal is to implement testing and it is meant to teach the importance of it and why code should be tested. Another goal is to teach the students how to test, in general but also Java EE Web projects specifically.

This goal in particular, is rarely implemented in the project by the attending students. The reason for this might be that the amount of time and effort required to achieve this goal is too much for the students to implement testing into their project. This is something that the course responsible would like to change.

A way of changing this is to facilitate the use of different testing frameworks by explaining when and how to use them for testing a code base. Another way is to provide the students with easy to follow tutorials for a given framework with simple testing examples. This is what our thesis project is for. To alleviate the entry barrier for learning about testing code so that more students choose to implement it on their project.

1.2 Purpose

To achieve these goals, a study is needed where a specified number of unit-testing frameworks, specifically for Java, are analyzed and compared against each other. Advantages and drawbacks in different areas of the project are considered in order to arrive at a conclusion that helps the course responsible decide which unit testing framework, or frameworks, is/are best suited for the course project.

Furthermore, tutorials need to be created for each framework that is analyzed in order to ease the use of that framework. The purpose of the tutorials is to shorten the time it takes to learn how to install and implement the frameworks so that more time is spent actually testing.

The course responsible is also our mentor and examiner, Leif Lindbäck and the thesis is conducted at KTH.

1.3 Delimitations

The frameworks this thesis project focuses on are Mockito and Selenium. The reason for why these are chosen over other frameworks is explained next.

Early in the thesis project when deciding upon which frameworks to focus on, our mentor provided a list over different testing frameworks. This list is roughly sorted after importance and relevance to the course project. Because of this sorting, the initial frameworks to focus this report on are the first three frameworks from this list namely, Pax Exam, Mockito and JSFUnit. Each framework covers a different aspect of testing and the idea is to cover all parts of the course project.

The thesis begins with the analysis of Pax Exam and upon further investigation we come to the conclusion that this framework is too complex for the scope of this thesis. Pax exam covers certain aspects that are relevant in of themselves to the course project but not the testing of them.

Also, implementing Pax Exam into the course project proves to be too difficult considering the students, for whom the result material is partially for are in general at beginner level. They have usually very little experience with code testing prior to attending the course. This is the reasoning for why it is decided not to include Pax Exam in this thesis project. Instead, Pax Exam is mentioned as a potential future study.

The next framework to look into is Mockito. Mockito is easy enough to implement into the course project and learning to use it is as difficult as Pax Exam. This means can be used Mockito is a fitting starting point when implementing testing for the first time and it is decided that it will be the first framework to be analyzed.

Mockito covers only a certain part of testing. An area called unit testing that is explained in detail in section 3.1. This aspect is an important one but it does not cover all aspects of the course project. The remaining two frameworks at this point have to cover the rest of the course project.

One area that is not covered by Mockito is some form of testing for the top layer of the course project, the layer where the client side resides. This layer contains the part of the project that a user interacts when using the Recruitment system, which is what the course project code base is once it runs. It is important that this aspect of the project is tested.

Following the list, the next framework to be studied was JSFUnit. JSFUnit specializes in testing JSF, framework used at the top layer of the course project. Without going into too much detail, it covers testing where Mockito does not. Specifically, the communication between the java code base and the JSF web interface of the course project.

At first it may seem natural to include JSFUnit in this thesis project but the more it is known about it, the more it is apparent how hard it is to implement JSFUnit into the course project. Most of the literature around JSFUnit deals with web project built with JSP web pages and not JSF pages, as it is in this case. To incorporate JSFUnit with JSF page proves to be too much of a hassle. Instead the focus turns on another testing framework called Selenium.

It is crucial to remember that the students that will use this material usually do not have much experience with testing and because of this, JSFUnit is deemed too complex for beginner level testing. The reasoning, in this sense, is similar to why it is decided to remove Pax Exam.

Selenium on the other hand is much easier to implement and, as a consequence, it can test the top layers of the system with ease. Selenium focuses on something called Black-box testing of the web interface of the course project. It does not cover testing of the JSF framework present in the project but it is regarded to be enough for the scope of thesis. Selenium is easy to implement and easy to use. This is a big advantage and because of this, it is decided to add Selenium to the thesis project instead of JSFUnit.

1.4 Method

Firstly, a decision has to be made on which frameworks to focus on. A list of existing frameworks is provided by our examiner and, from this list a number of frameworks that fulfills a certain set of requirements are chosen. These requirements are set up after deciding the demographic that will actually use the appendices provided in this report. That is, the students that attend the course. The assumption is made that these students are new to software testing. It is also assumed that their previous knowledge of testing frameworks is basically zero or close to zero. With this in mind, the goals with this thesis project are the following:

A framework has to be user-friendly. It has to be simple enough so that it can be learned by the students within the time frame of the course. Also, the relevance of testing the course project has to be taken into account. Testing is only one of several optional goals to achieve a certain overall letter degree on the project. If the framework is to complex, it will be discouraging for the students and they might choose not to implement testing at all.

The second goal with this thesis project is to help the students to implement a framework and create tests using it. To this end, a tutorial for each testing framework is created for the students to follow. The purpose of the tutorials is to speed up the learning process as much as possible so that the students can spend their time actually testing their code. For this to happen, **the tutorials themselves have to be user-friendly**. They have to be easy to follow.

1.5 Disposition

Overall, this thesis report follows the IMRaD disposition. The contents of each chapter are explained in a short manner.

In chapter 2, FRAME OF REFERENCE, other work on the subject, if any, is brought up. It is also explained how this thesis project differs from other eventual projects that discuss the same topic.

In chapter 3, THEORY, all the different technologies that are used throughout the course of this project are explained. The solutions that are used and any new knowledge acquired during the project is elaborated.

In chapter 4, THE PROCESS, the work process is described in detail. Any software development process is followed is also explained.

In chapter 5, RESULTS, the results from the analyses of the different frameworks are presented.

In chapter 6, DISCUSSION AND CONCLUSIONS, the results from chapter 4 are discussed. Conclusions that have been brought up during the thesis are presented here. These conclusions are based from the analysis with the intention to answer the questions formulated in Chapter 1.

In chapter 7, RECOMMENDATIONS AND FUTURE WORK, the ethics of the work is taken into account and future work in this field is presented.

In chapter 8, REFERENCES, all references to literature used in this thesis are listed in this chapter.

In APPENDIX A: MOCKITO UNIT TESTING TUTORIAL, a tutorial for how to implement Mockito Unit Testing/mocking framework is described with the help of examples.

In APPENDIX B: SELENIUM FRAMEWORK TUTORIAL, a tutorial for how to implement Selenium IDE for black-box testing of the user interface of the course project is described.

2 FRAME OF REFERENCE

The reference frame is a summary of the existing knowledge and former performed research on the subject. Earlier thesis projects, if any, are presented and it is explained how this material differ from this thesis report.

Since this project focuses on a specific Java EE Web project from a course, it is hard to find other work that has similar goals. Nonetheless, the topics covered in this paper, such as unit testing or integration testing, are widely discussed and written about.

In *Integration Testing of Object-Oriented Software* by Alessandro Orso **[1]**, integration testing of Object-oriented software is analyzed. The point is made that the complexity moves from individual modules to the interfaces between them in Object-oriented software. As a result, testing module interactions becomes the more difficult part as opposed to testing code within modules, such as when unit testing. New problems arise in this environment and these are examined in the thesis in order to hopefully define solutions for them. These solutions are new strategies for integration testing accompanied with new techniques for testing the interactions between modules.

Because this report uses a Java EE Web project that is Object-Oriented, the report by Orso is highly relevant when defining the integration tests for the Web project. However, the reports differ when it comes to scope and focus. While Orso's report focuses in the methodology and strategies for an integration test, this thesis project focuses on the actual frameworks that enable integration testing. Another difference is that the report by Orso does not examine frameworks for unit testing. It is only mentioned in his report.

In *The Development and Evaluation of a Unit Testing Methodology* **[20]**, a master thesis by Stefan Lindberg and Fredrik Strandberg, unit testing as methodology is discussed in detail. The thesis aims to develop and document a new unit testing methodology for testing the processes done by a certain company's software developing department. To this end, an evaluation of existing best practices for doing a successful unit test is done and from the data collected, a new methodology is derived tailored to the company's software.

The master thesis mentioned above focuses on unit testing and, even though no current best practices are applied directly into the process of this thesis, it still proves useful when documenting the theory behind unit testing in chapter 3.1 Unit testing. The focus of the master thesis is not any particular framework. Instead the master thesis establishes the methodology behind a unit test in order to develop a new methodology. In conclusion, the thesis by Lindberg and Strandberg gives good insight and another perspective on the theory of unit testing.

3 THEORY

In this chapter, all the different technologies that are used throughout the course of this project are explained. Any solutions that are used and any new knowledge acquired during the project is elaborated.

3.1 Unit testing

When talking about testing software, the terms unit testing and integration testing are often used. In these cases, the developer is not only interested in verifying the behavior and logic of the code but also how well all the parts in the code project interact with each other. It is important that these two methods of testing are well understood before analyzing existing testing frameworks.

The reason for this is that, in some cases, testing frameworks are capable of doing both a unit test and an integration test. Furthermore, some integration tests can also be a type of unit test. This is important to consider in order to yield a fair and giving analysis.

The word unit in unit testing has different meanings depending on the environment from where the test is conducted **[17]**. For the purpose of this report, the environment is determined by the course project for which the tests are created.

The course project is written in Java. In this case, a unit refers to either a single method in a class or an entire java class. Consequently, a unit test means testing a method in a class or the logic of a class. In other words, a unit test ensures that a specific piece code from the course project behaves as intended.

There are different arguments on how much of a particular piece of code should be covered and often, a percentage is set on how much of the code is tested **[2], [6], [19]**. In this thesis however, this subject will not be discussed nor will a stance be taken on the optimal percentage of code that should be tested for the best results. Instead, this report will focus on explaining the theory behind the methodology and to evaluate testing frameworks used for automated unit testing and integration testing.

In general, the more test coverage the more are the benefits of unit testing become noticeable. However, 100% test coverage is not always possible in real life scenarios due to other outside-factors like scarce resources or time constraints. One benefit of unit testing is a reduced number bugs in the source code. Bugs are usually not spotted until run-time, once the source code has been compiled and run. A typical bug is a behavioral error or ill-implemented logic in a particular method.

This type of problem is what a unit test aims to find and to make sure that the unit acts as intended. The developer is forced to confront the problem in a very concrete manner when testing that method. At that point, the code structure is scrutinized and its behavior analyzed which is necessary in order to create a test for it.

When a unit test passes, given that the test is well defined and valid, the developer can be sure that the piece of code works. This yields confidence to the developer, leading to another benefit

of unit testing, namely robustness of the source code. Also a developer confident in the code is not afraid to change it in order to improve it.

A well tested source code is easier to maintain and to further develop without the fear of breaking the code while changing it. If a method is changed or extended, it is as simple as running the unit tests to make sure that the logic is not broken or that any new bugs have appeared because of the newly changed method. As a result, a lot of potentially time consuming debugging is saved. This is another benefit to unit testing.

3.2 Integration Testing

Seen as a natural continuation or extension of unit testing **[9]**, integration testing involves grouping a number of units into one or more components or modules, finally testing the interfaces between the modules. This assumes that the individual units have already been successfully unit tested.

A project can have several components/modules of varying size and complexity and a module represents a specific business function in the project. The purpose of integration testing is to test the interaction between modules in the project as a whole.

At this point, any problems that may happen when testing the integration modules are most likely caused by the interfaces used for the integration test and not by a unit itself. This effectively reduces the complexity of the system, making it easier to find the root cause of a problem.

Integration testing looks at some issues that are not addressed during unit testing namely the interfaces needed for the modules to interact with each other and the different outcomes when several modules start to pass information between one another.

An interface is what helps the modules interact with other modules in the system. They are created so that data can be transferred between the modules. To keep this data from being unwillingly changed or corrupted, the interfaces must be tested to make sure that they are working as they should. This is called interface integrity.

Another way of seeing it is that by testing the interfaces, the data is tested when passed between the modules or components, as they are also called, during an integration test.

This type of data corruption becomes more relevant when more than two modules interact with each other. Any global variables may be changed involuntarily and different module unions may yield unforeseen data output. Data may also be lost during this interaction.

There are several ways to do an integration test. This report will focus on three common strategies called the Big Bang approach, Top-down testing and Bottom-up approach.

3.2.1 Big Bang Approach

The idea behind the Big Bang approach is to test the whole system at once. This means that all units are first integrated into one or more components/modules, depending on the business logic and then integration tested, all at once. The arrows in Figure 3.1 show the direction of the method calls done by the modules. It shows a simple interaction between modules.

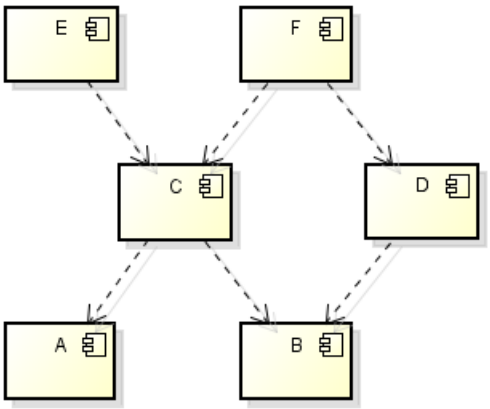

Figure 3.1 - Modules A to F with arrows showing their respective method calls.

The integration testing is done all at once, for all modules. See Figure 3.2.

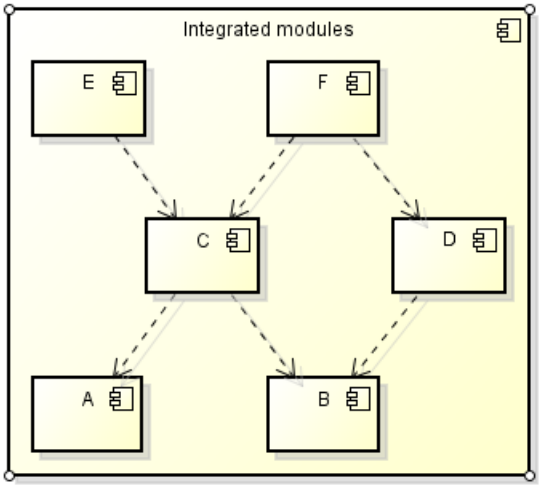

Figure 3.2 – Modules A to F all integrated and their interaction tested at the same time.

Other approaches involve division of the code project into modules, from higher level logic to sub system interactions between different frameworks like JPA and JSF, for instance.

The tests are then done by parts, testing the first modules in isolation and then either using drivers **[21]** to simulate calling modules or stubs **[21]** to simulate called modules. Some of these approaches are covered later.

Continuing with The Big Bang approach, it does not involve any division of the project. Instead all components, or modules, along with the interfaces are tested simultaneously.

This approach is best suited for smaller sequential applications where the unit tests are thorough with properly defined interfaces between the modules.

Problems to this approach arise when the testing fails or there are defects with either the modules themselves or the interfaces between them that help the modules interact with one. Since all modules are integration tested at the same time, it can be hard for the developer to know where the problem is coming from.

The number of possible bug sources in the code varies depending on the scale and complexity of the system. Also, any defects to the modules with corresponding interfaces are detected later in the testing process and the project can therefore be harder to debug. This is why this approach yields the best results when it is used on smaller projects.

There are more disadvantages to this approach which keep developers from using it to any significant extent during software testing. Since all modules are tested at the same time, there is no difference made between the modules. Some modules that handle a particular part of the business functions may be considered more crucial to the project than other modules.

This information becomes relevant for instance when there are time constraints to consider and all modules cannot be tested as thorough. In these cases, it is better to focus on the more critical modules for testing instead.

Another negative aspect to the Big Bang approach is that, in order to use this test all modules must be completed first. This means that, unlike the other two approaches, the integration test cannot be done until very late in a development cycle.

There are not many advantages to this approach unfortunately. In comparison to the Top-down or Bottom-up approach, the big bang approach has the potential to saves some time if the project is small in size. In this case, it can be easy to set up an integration test with this approach.

Yet, even this advantage is not compelling enough to recommend this approach. If the developer is very comfortable with, for instance the Top-down approach, such an approach can be used to set up an integration-test just as fast.

Another advantage is that all parts that go into the integration testing must be finished before the test itself. A lot of preparation must be done naturally but once the system is ready to be tested, all the different modules and interfaces are ready.

3.2.2 Top-down Approach

This approach is based off of an incremental testing mentality where each module, like the Bottom-up approach, is tested one by one until the whole system has been integrated and all modules are communicating as designed **[9]**.

The idea is to begin testing the module that exclusively makes calls to other modules and is never called by other modules. In order to do this a hierarchy among modules is needed to see which modules are called, which make the calls and which do both.

This differs from the big bang approach where all module interactions are tested at once. The drawback being that all modules must be coded and be ready before the integration test can begin when using the Big Bang approach.

When using the top-down, stubs **[21]** are created to simulate unfinished modules that are called by the module under test. This is similar to the mocking concept when unit testing with the Mockito framework.

The gain in using this approach is that the system is more easily debugged since it is divided into testing compartments that are individually tested. Also, it saves time if the project is still under development because it allows integration testing of modules incrementally as they finish development. No need for idle testers waiting for other modules to finish development.

Looking at the module composition in Figure 3.3, the testing begins by testing module *E* and *F* in isolation because these two are at the highest code level and only make calls to other modules. Modules *E* and *F* are never called by other modules in this system.

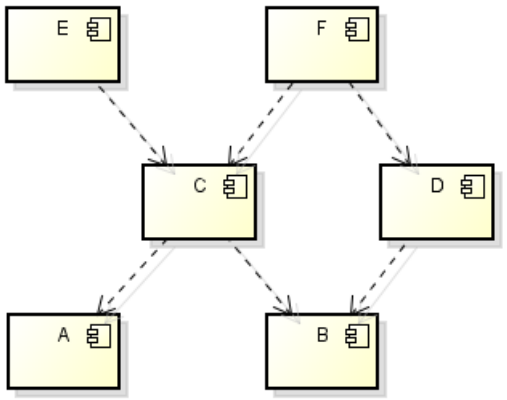

Figure 3.3 – Same module interaction structure as in Fel! Hittar inte referenskälla..

Next step is to test the call made by module *E* to module *C*. If an error occurs, it is coming from either module *C* or the interface between *E* and *C*. This is why this approach is better at finding problems then the Big Bang approach.

The steps done for module *E* are repeated for module *F* as it is at the same code level as *E* and test its interactions to both *C* and *D*. Modules *E*, *F* and *C* are then merged into a single Module. Its interactions with module *A* are subsequently tested. If module *A* passes the test, it is absorbed into the larger module containing *E, F* and *C* seen in Figure 3.4. This is done incrementally until the whole system has been integrated.

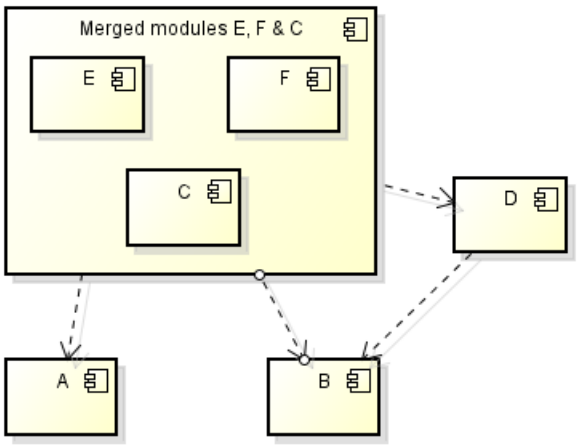

Figure 3.4 – Start from the top (calling modules) and merge after each tested interaction.

Having in mind that this report revolves around a web project that has already been finished, this time saving advantage of being able to do the integration tests even when the modules are not finished may not be as relevant. With that said, this approach can still be applied to the project and its other advantages over the big bang approach are still relevant.

The benefit of applying this approach to a system has already finished development is that there is no need to code stubs to simulate called modules as those are already finished. This in itself saves time.

3.2.3 Bottom-up Approach

With this approach, the module that is first tested is the one that has no calls to other modules but is only called by other modules. This module is tested in isolation and modules are incrementally added, opposite to the Top-down approach. Since Top-down and Bottom-up each other's opposites, the same illustration can be used in Figure 3.5. The approach starts with module *A* by having a driver **[9], [21]** to simulate the call done to it by module *C* if *C* is not yet finished. If *C* is finished, its real methods are used instead, of course. If module *A* acts as expected, it passes the test.

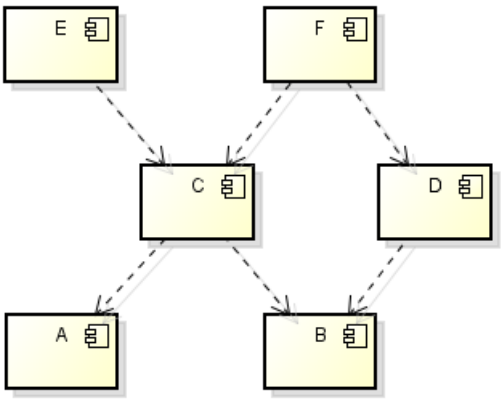

Figure 3.5 – Same module interaction structure as in Figure 1 and 3.

Next step is to do the same with module B and test its interactions between it and modules *C* and *D*. Once *B* has passed, the integration testing continues by merging modules *A, B* and *C* into a single module and the calls done to it by other modules, in this case module *D, E* and *F*, are tested.

The process continues by merging more and more modules until the whole system has been integrated and all the different module interactions are tested. Instead of stubs for simulating called modules, drivers are used with this approach to simulate calling modules that have yet to finish development in order to make the test, as shown in Figure 3.6. Because of the nature of starting with the module that never calls other modules, a suite of advantages and drawbacks arise.

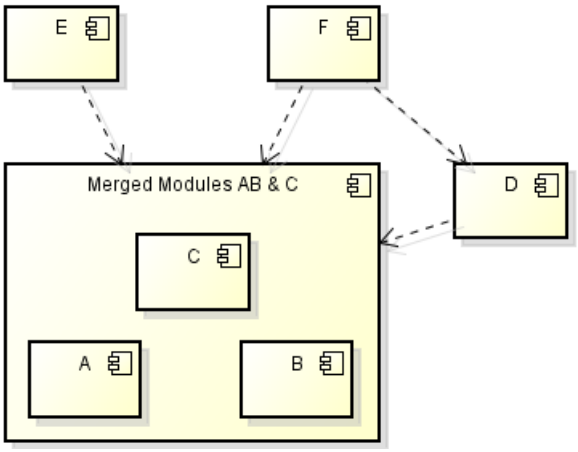

Figure 3.6 – Start from the bottom (called modules) and merge after each tested interaction.

This approach is generally easier to implement than the top down and Big bang approach but if the system is still under development, it will take longer until a working build can be presented to the end user since the highest code level modules are handled last. It also is easier to plan ahead and adjust the higher level modules to work better with the more utilitarian modules at the lower level part of the system.

Continuing with a drawback, drivers are often harder to create than stubs due to the nature of having to predict how the calling module will behave once it is finished. Again, this fact only comes into consideration if the system is still under development. In this case, the project is already finished and the design of the higher level module is already known.
It might still be necessary to create a driver to simulate the call but it is easier to do if the design of the calling module is known beforehand.

Other methods of testing are the Umbrella approach and sandwich testing. These two combine or expand upon one or more of the earlier three approaches and will not be covered in this report.

3.3 Mockito

Mockito is an open source unit testing framework developed for use with Java. It is used as an extension to the JUnit testing framework. This means that all the methods in the JUnit library can be used with Mockito as well. A downside to this is that Mockito cannot be used as a stand-alone framework and requires that JUnit is installed and implemented in the code project beforehand.

The goal of Mockito as a testing framework is to simplify the use of mock objects. A mock object is simply a fictive object that simulates external dependencies of a real object in order to test an object or class **[10]**. The object under test is often called system under test, shortened SUT.

The meaning of SUT differs depending on the topic of discussion. In integration testing, an SUT can be a group of objects. For instance, when making an integration test, the SUT is often a module which, in turn, is usually a composition of units that cover a specific part or role in the code-project. In Unit testing however, an SUT is referring to a unit which is a single class or object of this class.

Mockito differs from other testing frameworks, by giving the developer the ability to test without using the expect-run-verify pattern **[11]**.

Mockito accomplishes this by removing the expectation part when setting up a test in order to check the behavior of the SUT. An SUT, or System under test, is the system that is being tested.

What this means in practice is that the developer does not need to set up expectations when verifying behavior of a method. Instead, the verification is done after the fact.

For example, instead of expecting that a method is going to be called the developer can verify if the method was invoked after the call is made. Furthermore, Mockito lets the developer be as specific as needed for the test. For instance, the developer can verify if the method was invoked exactly 3 times or that it was invoked with the right parameters and so on.

In Figure 3.7, the first thing that happens in the test is that method A is called with a specific parameter "*anyString"* with no expectations set up before that point. Method *A* in turn calls another method B which makes *A* dependent of *B*. This dependency is mocked out before the test in it is not shown in Figure 3.7.

```
/**
 * Test run of methodA, of ClassA. Test verifies different
 * things about methodB in classB. This is done by mocking dependencies
 * to ClassB.
 */
@Test
public void testMethodA1() throws Exception
{
    // Call methodA in ClassA.
    a.methodA("anyString");

    // Verify that methodB was actually invoked with the correct
    // parameter when methodA was called.
    verify(b).methodB("anyString");
}
```

Figure 3.7 – Pseudo code over a test with Mockito. Notice the run-than-verify structure of the test.

Instead of expecting a behavior before calling method A, the test checks if method *B* was actually invoked by *A* with the correct parameter using *verify()* after method *A* is called.

A drawback that is often mentioned about Mockito is that the framework does not allow mocking of static methods. This is a problem that requires tempering of the SUT. This issue becomes relevant when trying to solve test case *3.1.1 Test the Logger* with Mockito.

In this case, a test is needed to check if a specific number of exceptions are logged when thrown during execution of the test. Unfortunately, this method is static which means that it would require one or more changes to the code of the SUT mainly, *Logger.java*.

This goes against the purpose of testing since the SUT is changed just for the sake of running the test. The purpose of a unit test is to test existing code to see if it still performs to specifications, even after further development.

The answer to this critic is that a static method is usually a sign of bad design of the SUT itself but in test case *3.1.1* in particular, the method is static because it tries to write to external text files. In such a situation the method should be static according to conventions in the Java programming language.

The reason for why the method in the class *Logger* is static is because the method accesses external files. In order to access these, the method that is named *log* invokes a specific method from the servlet context class called *getRealPath* to get the real paths to the external files.

Unfortunately, when mocking the servlet context, it does not have a real path to any file since it is just a mock and the method and an exception will be thrown. It does not actually set up a new servlet context.

3.4 Selenium

The main purpose of this framework is to automate the browser. This allows for black-box testing of the user interface by setting up automated tests without the need to know any scripting language. Selenium is open source and distributed under the Apache License 2.0.

Selenium is comprised of a number of components that give the user different ways to test. Probably the most common of these components is **Selenium IDE** which is implemented as an Add-on for the Mozilla Firefox web browser. With the IDE, recording and editing tests is facilitated through the IDE interface. Once the recording has started, every command done by the user on the project website (which is the user interface of the system) is recorded (Figure 3.8).

For instance, every click done on an HTML element or any text box filed is recorded with their respective values. This information can be used to track where the commands go and, by knowing this, determine if the web page is acting as it should. The recording can be played back which simulates every step taken on the web page.

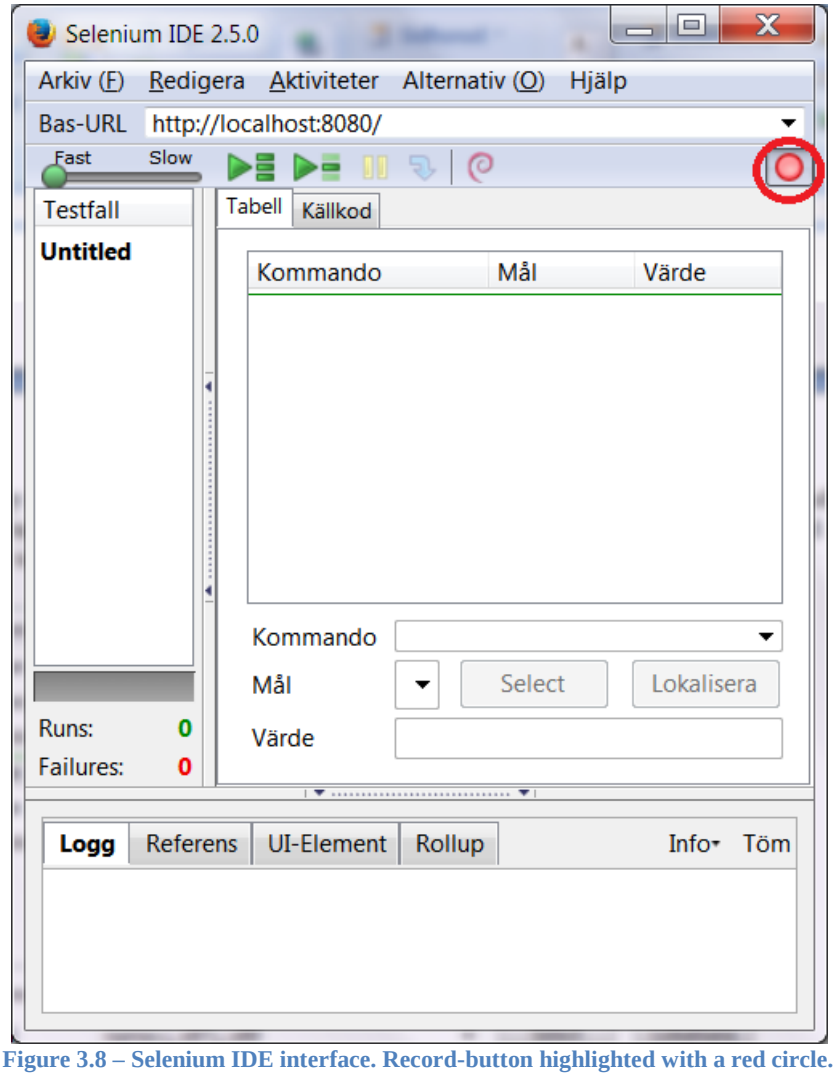

Figure 3.8 – Selenium IDE interface. Record-button highlighted with a red circle.

All recordings are constructed in the scripting language Selenese. Selenese commands represent every action done on the web page and is displayed in a log window at the middle of the IDE interface as shown in Figure 3.9.

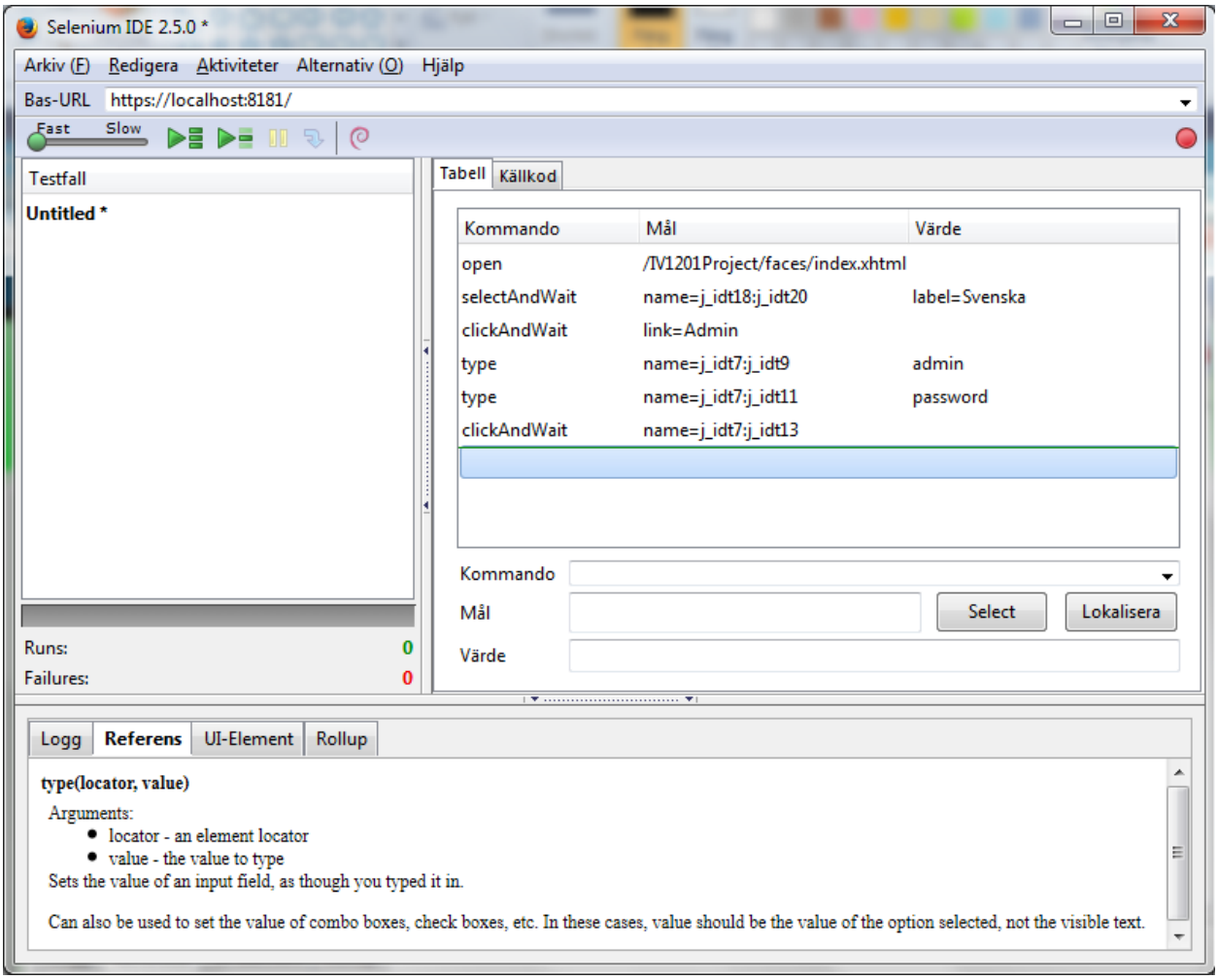

Figure 3.9 – Selenese script language example from a recording.

The user-friendly interface of this component works as a good entry point into Selenium and software testing as a whole, which makes this component the most used of the rest. The next component is **Selenium Client API**. It allows the user/developer to write languages other than Selenese, like Java. The goal with this component is to provide more ways to write tests. Without this component all test would have to be written in Selenese and they would only able to run through the Selenium IDE.

One advantage of having the test written in Java, for instance, is that the test can be executed in an IDE other than Selenium IDE like NetBeans, with the help of third component called Selenium WebDriver but more on that later. Furthermore, the project does not have to be deployed for the test to execute. For obvious reasons, this is not the case if the test is recorded using the Selenium IDE. If the project website is not up and running, there is no way of recording a test on it.

Thirdly, there is **Selenium WebDriver**. This component works as a handler of the commands sent by Selenium Client API to the browser instance and retrieves the results. This component is packaged together with the client API and implemented with a driver that is browser-specific. As mentioned earlier, the project website is not needed. Instead, when a test is executed in NetBeans for example, the WebDriver initiates a new browser instance. The driver takes control of it and runs the test. It is the browser driver that dictates which browser to use. As an alternative, a special browser driver called HtmlUnit Driver can be used to simulate a browser instance.

As of February 2014 only Firefox is directly supported by the creators of Selenium (a.k.a. seleniumhq) but there are third-party browser drivers for other browser applications such as Chrome and Internet Explorer. The browser drivers are available for download from the official Selenium download page.

As a final component to Selenium, there is the **Selenium Grid**. Grid is a server that lets the developer run tests on a browser instance located on a remote machine. In this structure, there is a central server that handles the different browser instances and each test asks the server, or hub, for permission to access a certain browser instance. The main point with Grid is to allow for parallelism among the tests. In other words, the tests can run in parallel on different remote machines. The thesis will not touch on this subject and will not use Grid in any extent because it falls outside of the main goals of the thesis.

4 THE PROCESS

Here, the work process is described in detail. Any software development process that that is followed is explained.

4.1 Test cases

The way this project analyzes the frameworks is by creating a set of test cases for each framework. Also, an evaluation is done on how easy the frameworks are to use. The test cases are designed to evaluate the capabilities of each framework in terms of concrete testing cases. The frameworks focus on different aspects of testing. It is therefore not possible to compare them to each other. Since no comparison is possible, the test cases instead show if a particular framework is capable to test a certain aspect of the course project. If so, the test case yields valuable insight on how to test with that framework. In that case, the results of a test are used as concrete examples for how to go about with similar problems using a particular framework.

When a framework fails a test case, the validity of the test is questioned to determine if the test case is properly defined for that testing framework. For example, a test case may be about Integration Testing and Unit Testing. The two often work together and some frameworks are not designed for this type of testing, which makes the test unsuitable for that particular framework.

For each of the three unit testing frameworks, test cases are developed for evaluating the effectiveness of the frameworks. Some of the cases are implemented in more than one framework. In this way, the frameworks can be compared in order to determine which one is better to use in a specific case. All test cases created for this project are explained in detail here. In essence, if a testing framework passes a test case, it is a well suited framework for the Java EE web project.

4.1.1 Test the logger

Make a test for the logger method to see if the method actually logs in the files *database_log.txt, login_log.txt* or *exception_log.txt*. In this test case, the SUT is the Java class *Logger.java*. Figure 4.1 shows the method that is tested in this test case. The illustration only shows pseudo code and the whole method can be found in the Java class *Logger* under the project source package *model.log*.

```
/**
 * Takes a message and appends it to one of the available log text files.
 *
 * @param msg the message to be logged
 * @param pointer points to a specific log
 */
public static void log(String msg, int pointer)
{
    try
    {
        String fileName = log_map.get(pointer);

        ServletContext ctx = (ServletContext) FacesContext.getCurrentInsta
        String path = ctx.getRealPath(fileName);

        PrintWriter out = new PrintWriter(new BufferedWriter(new FileWrite
        out.println(msg);
        out.flush();
        out.close();
    }
    catch (Exception e)
    {
        System.err.println("Logger.java failed. Exception: " + e);
    }
}
```

Figure 4.1 – Method under test, i.e. the SUT in test case 4.2.1.

The method that is tested receives two parameters from the calling class. The first is the message that needs to be written to a text file and second is a pointer that tells which file to write the message to. The external text files that are written to by this method are referenced to by the real path found through the servlet context. This path is then saved as a string called *path*. A buffer is later opened to the file and the message is sent. Access to the log-files is of the nature write-only due to security reasons **[13]**.

Logs are an important part of debugging and troubleshooting **[18]** because they allow a retrace of all actions taken that caused an error **[4]**. It is therefore easier to find what caused the error and why, so that the fault can be corrected faster **[18]**. This is why this test case exists. To make sure that the logging procedure is working as it should and that the errors are cataloged.

4.1.2 Test of login method

Make a test of the login method in *AuthenticationBean.java* that verifies a specific method call. In this test case, the SUT is *AuthenticationBean* and the method that tested is called *login()*. The method passes the input from the user to the Controller, *DAOFacade.java*. If the controller returns 0, it means that the username and password provided is correct. At this point, a string called *AUTH_KEY* is set to the session so that the user can access restricted pages. This string acts as an authentication key and it is removed once the session ends. If the input is incorrect, a non-zero value is returned from the controller and the user is not able to log in to the system.

Figure 4.2 shows pseudo code of the login method that is tested in this test case. The complete method is found in the Java class *AuthenticationBean* in the project source package *view*. The SUT decides which page to send the user to depending on the outcome of the method. Upon providing the correct login information, the user will be sent to the admin page. If not, an error page is shown instead.

```java
/**
 * Passes method to DAOFacade to check with database if the user exists.
 * Sets the AUTH_KEY session
 * so that the user can enter restricted pages.
 *
 * @return the admin page if success, otherwise the login_error page
 */
public String login()
{
    if (daof.login(username, password) == 0)
    {
        FacesContext.getCurrentInstance().getExternalContext().
                getSessionMap().put(AUTH_KEY, "the admin");
        return "admin";
    }
    else
    {
        return "login_error";
    }
}
```

Figure 4.2 - Method under test in the SUT for test case 4.2.2.

4.1.3 Test of getters and setters

Make a test for the get/set methods in *AdminBean.java*. These methods are required so that a recruiter can access necessary information about applicants. *AdminBean* is the SUT here. Test at least 75% of the methods to pass the test case. Figure 4.3 shows the code for some of the get/set methods found in the Java class *AdminBean*. The java class is located in the project source package *view*.

```
// GETTERS & SETTERS
public String getFnamn()
{
    return fnamn;
}

public void setFnamn(String fnamn)
{
    this.fnamn = fnamn;
}

public String getEnamn()
{
    return enamn;
}

public void setEnamn(String enamn)
{
    this.enamn = enamn;
}

public String getEmail()
{
    return email;
}

public void setEmail(String email)
{
    this.email = email;
}

public Date getDate of registration()
```

Figure 4.3 – Some of the methods tested in the class under test, i.e. the SUT in test case 4.2.3.

The information about the applicant can be expanded upon, which means manipulating *AdminBean* by changing, adding or removing code. It is important that current requirements are not involuntarily changed. For this reason, an automated test is needed in order to check that current requirements are not altered unwillingly. If they are, a test will fail. This test can also be used for future reference on how to test get/set methods.

4.1.4 Test of login interaction

Make a black-box automated test of the login process and the internationalization support of the project website. The goal with this test case is to verify that the relevant pages access the right elements, call the correct methods in the java bean at the layer below, and changes to the right language. For this purpose the error handling of the site is checked by deliberately entering the wrong login information first and then entering the right information. The SUT in this case is a collection of JSF pages involved in the login process. These pages are *index.xhtml, login.xthml, login_error.xhtml* and *admin.xhtml*. The login interaction consists of the following 4 steps:

1. Click the drop-down list in *index.xhtml* and change the language to Swedish.
2. Click the link called *Admin*.
3. Enter wrong login information and click on the button *Login*.
4. Enter correct login information and click on the button *Login*.

The SUT is located in the Java EE Web project folder called *Web pages*. This folder contains all JSF pages that make up the project web interface from which the user interacts with. The Login process accesses the project database to verify that the user input is correct. This means that the process passes through all the layers of the project. Some steps are transaction based. A transaction means that, if a failure occurs during such a step, all actions taken during the step are rolled back to a point right before the start of the transaction. This step is taken when accessing the database at the lowest layer of the project.

When the user provides the login information, the JSF page *login* calls the method *login()* in the Java bean *AuthenticationBean.java.* This bean passes the information down to the other layers of the project. Depending on the result returned from the lower layers, the bean redirects the user to either the JSF page *login_error* or *admin*. The result is binary, either the login fails or it passes. *AuthenticationBean* is found in the project source package *view*.

Going back to the test case itself, it is important that the login interaction works as it should for the user and not just in the logic behind the login process. By recording such an interaction, it is checked that this crucial part of the web interface has no bugs that might appear only at run time. This test case is looking for interaction problems not visible from a source code point of view and that only appear when interacting with the system.

Since the interface can change so rapidly and in major ways, bugs might appear that are not present before the changes to the interface are made. By making an automated test of an important interaction, it can quickly be checked that no new bugs to this particular part of the interface breaks due to a change in some other parts. This is why an automated test gives good supports for agile and extreme development methodologies **[24]**.

On the other hand, sometimes it may not wise to apply automated tests on, for example, the login process in the UI. Every time an aspect is changed, there is always a risk that its corresponding automated test becomes invalid. Therefore, if a specific part of a project such as the login process is expected to change a lot within the near future, it is better to wait for the code to become more stable before creating automated tests for it. In such occasions, it may be better to write manual tests for it **[24]**.

An automated test basically checks, among other things, if a change somewhere else has broken the code under test, provided that the code itself has not been changed after the creation of its unit test.

4.1.5 Test the login interaction & update status

Make an automated test of the login interaction and the internationalization support of the site, just as in test case 4.1.4. Also, extend the test by testing the ability to select an application and accept or reject it for a certain job opening, as an admin. The goal with this test case is to test another important part of the user experience as an admin, namely reviewing applications. By testing the login method again, it is checked that is still works when following a different possible interaction path that the user might take.

The SUT in this test case is a number of JSF pages. These are *index.xhtml, login.xthml, admin.xhtml* and *application_profile.xhtml*. The process follows the steps described below.

1. Click the drop-down list in *index.xhtml* and change the language to Swedish.
2. Click the link called *Admin*.
3. Enter correct login information and click the button *Login*.
4. Click the button called *Show* for the first application in the list of all available applications.
5. Click in the check box to change the status of the application from either "*ANTAGEN"* to "*NEKAD"* or "*NEKAD"* to "*ANTAGEN"*.
6. Click the button called *Uppdatera*.

The SUT if found under the project folder called *Web pages*. The test case finds potential bugs in the system that only appear during run-time. Due to the nature of the web-development, major parts of the site can change quickly, which can lead to new bugs in the system interface. By having an automated test of a crucial part of the system, a quick check can be done to make sure that it still works as it should after a change of something else has been committed **[25]**. Such a change if often a visual one that has to do with improving the user friendliness of the web site.

4.1.6 Test of creating an application

Make an automated test for the process of applying to a job and test the internationalization support of the system by changing the language from English to Swedish. The test is needed to check that part of the project web site works properly. It is not a test of usability of the system but more of a bug test of this particular part. The SUT in this case consists of a number JSF-pages involved in this process. These are *index.xhtml, apply_step1.xhtml, apply_step2.xhtml, apply_step3.xhtml* and *apply_success.xhtml*. The process follows these steps:

1. Click the drop-down list in *index.xhtml* and change the language to Swedish.
2. Click the link *apply* at the left side of the page.
3. Enter incorrect first name, last and e-mail address.
4. Enter correct first name, last name and e-mail address.
5. Click *Nästa*.
6. Click the drop-down list for all competences and choose the competence called "*Kock"*.
7. Fill in *"x"* as years of experience for the competence *"Kock"* in the right text field.
8. Fill in seven years of experience for the competence "*Kock"* in the right text field.
9. Click *Lägg till* and then click *Nästa*.
10. Fill in an incorrect availability period of when it is possible to work.
11. Fill in the availability period of when it is possible to work. Choose the period 2014-01-01 to 2015-01-01.
12. Click *Lägg till* and then click *Klar*.

The SUT is located in the project folder called *Web pages*. By testing such an interaction, it is verified that a crucial part of the web interface has no bugs that might appear only at run time. This test case is looking for interaction problems not visible from a source code point of view and that only appear when interacting with the system.

Since the interface can change so rapidly and in major ways, bugs might appear that are not present before the changes to the interface are made. By making an automated test of an important interaction, it can quickly be checked that no new bugs to this particular part of the interface breaks due to a change in some other parts. This is why an automated test gives good supports for agile and extreme development methodologies **[24]**.

4.2 Tutorials

In addition to the test cases for each framework, tutorials are created in order to demonstrate how to install, implement and use the different frameworks. To evaluate their effectiveness, a number of students are chosen to simply use the tutorials and give direct feedback on what they think about them. If they cannot follow a tutorial for any reason or find it hard to do so, it means the tutorial is not user-friendly enough and it has to be revised.

Their design is the result of a process of iterative testing where the students are asked to follow each of the tutorials. The feedback is then used to revise the tutorials. The process is repeated and the students are once again asked to follow the tutorials until there is no confusion and they feel like they can follow the tutorials with as little effort as possible.

Having basic knowledge about HCI is proven to be very useful in several areas. One of these areas is when setting up an environment for the students where they follow the tutorials under a set of given conditions. A certain scenario needs to be set up where it is decided how much prior knowledge the candidate should have and what they are supposed to do. In this scenario, the students are asked to pretend that they are attending the course and that they want to implement the testing framework into their course project. To achieve this, they have been given a tutorial whose goal is to teach them how to do this. Furthermore, they are not allowed to interact with the tutorial designers. They are only there to observe.

The students are given a pen and paper to write down any thoughts that may come up when following a tutorial. As mentioned before, once a student has started following a tutorial, the designers are not allowed to intervene in any way with the student. If a student gets stuck, for any reason, they are not allowed to ask the designers for help. Instead they may write down any problem they come across and a discussion is had once the scenario is over.

The main reason for this type of set up is that some problems with usability or effectiveness of the tutorial may be lost if the designers intervene. As an example of why the designer should not intervene, imagine that some crucial information that is supposed to help the student get through the tutorial may not be conveyed in an effective enough manner. This can lead the student to get stuck in a real world scenario and instead of rooting out the problem, the designer tells the student where the problem is and the student solves the issue that way, even though there is a fault in the design of the tutorial. In other words, design issues with the tutorial may be lost if the students are allowed to interact with a designer that is observing the process. The goal with setting up an environment is to simulate a real world scenario as accurate as possible.

Another area where HCI knowledge comes in great effect is when choosing the right candidates to follow the tutorials. In order to decide what type of person to ask about following a tutorial,

factors like age, prior relevant knowledge about the theme or general experience with computers must be taken into account. The optimal candidate is the person who will use the tutorial in a real life scenario, in other words, the end user of the appendices. In this case that person would be a student form the course IV1201. The age and gender of the person is not relevant in this case.

Once the optimal candidate is defined, it is not a guarantee that such a person is found or if the person is willing to participate. For this reason, the search for candidates is widened to include any student from KTH with basic knowledge about Java EE design with a web-based GUI. These criteria infer that the candidates know about the MVC-model but, as a precaution, it is explicitly asked if they do. To find the candidates, students are approached at random within the university, a short introduction is given and then they are asked if they would like to participate. Friends and relatives that fulfill the knowledge requirements are also contacted.

The criteria must be defined well enough so that a small number of about 3-5 students are enough to achieve a satisfying design of the documents. It is important that the number of candidates is low so that the evaluation does not take too much time. Basically, the better the criteria, the better the evaluation from each student is and the pool of candidates can be smaller but still achieve acceptable results.

Each student goes through each tutorial once and not more. This is because, once a student has followed a tutorial to the end, the student learns its structure. At this point, it is difficult for the student to be put in the same scenario where the student is not supposed to have read the tutorial. Once this is known, some crucial information may be lost. The rationale is similar to the reason for why a scenario needs to be set up in the first place. The results of the evaluations are presented in section 5.3 Evaluation of Tutorials.

Here, the results from the analyses of the different frameworks are presented.

5.1 Mockito Test Results

Analysis of this framework is based off of three test cases. With Mockito, all three test cases passes, with the first partially modified. The results of the test cases are presented with pseudo code of the test classes.

5.1.1 Results for test case: Test the logger

The test is completed using JUnit with some modifications to the SUT. To be able to execute this test, a dummy class has to be created. This class contains three modified methods of the original log-method used in the real SUT, *Logger.java*. All three methods contain hardcoded paths to their respective log files instead of having the paths extracted from the servlet context, as it is done in the original method.

Figure 5.1 shows pseudo code for one of these three new methods found in the dummy class *DummyLogger.java*. The source code is located in the test package *model.logs* along with the original logger class *Logger*. *DummyLogger* is now the SUT with three methods, *logLogin*, *logDatabase* and *logException*. These contain the paths for each the text file. The login text file in particular contains logs of all error messages concerning the login procedure of the system that are generated by a particular set of exception handlers.

```
public static void logLogin(String msg, int pointer)
{
    try
    {
        /*
         * The real method to get the pathname for the log-file.
        String fileName = log_map.get(pointer);

        ServletContext ctx = (ServletContext) FacesContext.getCurrentInstanc
        String path = ctx.getRealPath(fileName);

         *
         */

        /**
         * Hard-coded the path to be able to write to the log.
         * Not able to find the path to the fileName when using getRealPath(
         * because its a mocked ServletContext.
         */
        String path = "C:\\Users\\Mustafa\\Desktop\\Examensarbetet\\build\\w

        PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter(
        out.println(msg);
        out.flush();
        out.close();
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }
}
```

Figure 5.1 – One of three modified log methods. This contains a hard-coded path to log file for login errors.

The exception handlers that calls the method in the original *Logger* are found in the login method called *login* of the calling Java class *Logic.java*. *Logic* is part of the project source package *model.dao*. In Figure 5.2, some of the exception handlers are shown. The pseudo code is from the Java class *Logic*, the calling class.

```
catch (Exception e)
{
    String msg = new Date(System.currentTimeMillis())
                            + ": Runtime exception: " + e.getMessage();
    model.logs.Logger.log(msg, model.logs.Logger.EXCEPTION);
}

if (r == null)
{
    String msg = new Date(System.currentTimeMillis())
                        + ": Failed login attempt: user: " + username;
    model.logs.Logger.log(msg, model.logs.Logger.LOGIN);
    return 1;
}
```

Figure 5.2 – Exception handlers found in *Logic.java*, the calling class. Exceptions caught here are logged in respective text files.

Figure 5.3 shows the test class for the dummy class called *DummyLogger*. The test class is found under the project test package *model.logs* and it is called *LoggerTest.java*. Due to the modification required to make this test, Mockito is not needed since there is nothing to mock. Another consequence is that the SUT is not tested directly but indirectly by testing the dummy class. In other words, the actual SUT is *DummyLogger* and not *Logger*, as it was planned when defining this test case. The only difference is in how the log methods get the real paths to the text files. In the Original SUT, they are extracted from the servlet context and in the dummy class, the paths are hardcoded.

```
public class LoggerTest
{
    @Test
    public void TestLoggerForLogin() throws Exception
    {
        String msg = new Date(System.currentTimeMillis()) + ": This is a test "
                                + "to see if the method logs to login_log.txt";
        DummyLogger.logLogin(msg, model.logs.Logger.LOGIN);
        System.out.println("Login_log test finished!");
    }

    @Test
    public void TestLoggerForDatabase() throws Exception
    {
        String msg = new Date(System.currentTimeMillis()) + ": This is a test "
                                + "to see if the method logs to database_log";
        DummyLogger.logDatabase(msg, model.logs.Logger.DATABASE);
        System.out.println("Database_log test finished!");
    }

    @Test
    public void TestLoggerForException() throws Exception
    {
        String msg = new Date(System.currentTimeMillis()) + ": This is a test "
                                + "to see if the method logs to exception_log.txt";
        DummyLogger.logException(msg, model.logs.Logger.EXCEPTION);
        System.out.println("Exception_log test finished!");
    }
}
```

Figure 5.3 – Test class that tests the modified SUT *DummyLogger.java*.

The reasons for why this modification is needed are discussed in chapter 6. Among other things, the validity of the test is questioned and any conclusions that are arrived to are also mentioned in that chapter.

5.1.2 Results for test case: Test of login method

Test case passed using Mockito. A new Java class called *ContextMocker.java* **[7]** is needed in order to mock *FacesContext* **[15]** and yield an authentication key for the session. The key is set through *ExternalContext.java* **[14]** which handles the behavior of the Servlet implementation. This is needed to simulate a session with the right permissions. *ContextMocker* is not proprietary. The code is taken from the blog Illegal Argument Exception – Miscellaneous Computer Code **[7]**.

Figure 5.4 illustrates the class *ContextMocker*. The pseudo code shows the implementation for releasing the mocked *FacesContext* for garbage collecting once the test is over. The method named *mockFacesContext()* creates a mock of *FacesContext* and sets it to be the new instance for the test container. Once the test is over, the mocked *FacesContext* calls the method *release()*. This is done during the tear-down phase of the test class.

The real implementation of this method, described in the Java API **[16]**, releases all resources associated with *FacesContext*. For this test however, this call is overridden by a new implementation called *answer()*. This method is defined in the inner class *Release*, found in *ContextMocker*. The override is done using the Mockito method *doAnswer()* which specifies what to return when a specific method is called, in this case *release()*.The method *answer()* removes the mocked instance by setting the current instance of the test container to null. At this point, the mocked *FacesContext* can be garbage collected.

```
// Define inner class Release
private static class Release implements Answer<Void>
{

    /*
     * Take down current context for garbage collection. Overrides method
     * release() in FacesContext when called during tear-down phase of
     * the test in test class AuthenticationBeanTest.
     */
    @Override
    public Void answer(InvocationOnMock invocation) throws Throwable
    {
        setCurrentInstance(null);
        return null;
    }
}

/*
 * Mock FacesContext using Mockito. Set the instance to be
 * the mocked FacesContext while running the test. Release the mock
 * to be collected by the garbage collector once test finishes.
 */
public static FacesContext mockFacesContext()
{
    FacesContext context = Mockito.mock(FacesContext.class);
    setCurrentInstance(context);
    // When context is released, run method answer()
    Mockito.doAnswer(RELEASE).when(context).release();
    return context;
}
```

Figure 5.4 – Release implementation and method mocking *FacesContext.java*. Used by test class.

The test then verifies that the right method in the controller, *DAOFacade.java*, was called exactly one time with the right parameters. The SUT needs access to *DAOFacade* in order to be able to pass a mocked object of type *DAOFacade* to the SUT during the test. This is possible using the injection point for *DAOFacade* found in the SUT. In Figure 5.5, code to the SUT is shown together with test code from the test class. Why this is needed and other reflections over this particular solution are brought up in chapter 6.

```
/*
 * Injection point for DAOFacade. Instead of accessing it as an EJB.
 */
@Inject
void setDAOFacade(final DAOFacade daof)
{
    this.daof = daof;
}

 ----------------------------------------------------------------------

/*
 * Tests the method used when admins try to login.
 */
@Test
public void TestAdminLogin()
{
    authBean.login();
    verify(mockedDAOFacade, times(1)).login("Terminator",
            "c00lk1ll3rb0y#96");
}
```

Figure 5.5 – Upper part shows Injection for *DAOFacade.java* in the SUT. Lower part shows the test.

The complete code for SUT called *AuthenticationBean* is found under the project source package *view*. The test class is called *AuthenticationBeanTest.java* and is located in the project test package *view*. The test itself is simple in nature. The complexity is in the set up of the test. This, among other reasons, is because the SUT uses contexts that need to be mocked but more on this in chapter 6.

5.1.3 Results for test case: Test of getters and setters

Test case passed partially using Mockito. The test case is a unit test and most methods are tested in isolation. Five line of code was added to SUT in order be able to pass a mocked object of type *DAOFacade.java* to the SUT during the tests. The code is that is added to the SUT is identical to the one added in the SUT for test case 5.1.2 and it is illustrated in the upper part of Figure 17.

Some of the methods require more advance parameters than others which means more set-up code in order to test. The advanced methods are *getApplicants()*, *getCompetenceList()*, *getCompetence_list()*, and *getAvalability_list()*. Most of the simpler methods do not require mocking. For these, the set-method is called with a specific parameter, during the test, followed by an assertion on the get-method to check if it returns the same parameter.

Two test classes are created, one that test the simple methods and one that test the methods with more complex parameters. These are named *AdminBeanTest.java* and *AdminBeanComplicatedTest.java* respectively. Both test classes are found in the project test package *view*. Figure 5.6 shows a code snippet from *AdminBeanTest*. The figure shows how the simple get/set methods are tested. The test makes sure that the methods return the right values, even if the logic is changed.

```
/*
 * Test the set- & get-methods in the class AdminBean.
 */
@Test
public void TestGetsAndSets()
{
    // Set name using the method setFnamn in AdminBean
    // Call getFnamn and check if parameter is the same.
    adminBean.setFnamn("Afatsumo -o");
    assertEquals(adminBean.getFnamn(), "Afatsumo -o");
```

Figure 5.6 –Testing a simple get/set method. From test class *AdminBeanTest.java*.

One of the methods in the SUT sets a certain boolean value to be either true or false depending on the parameter, when called. This value is then used in another method that passes it to the controller of the system. The controller is called *DAOFacade* and the boolean value determines if an applicant is accepted or rejected for a certain job. The dependencies between the SUT and *DAOFacade* are mocked using Mockito. The method that passes the value to *DAOFacade* is called *setAccepted()*. This method is tested differently than other set-methods. It is checked that the right method in *DAOFacade* was called with a specific set of parameters from the SUT. This is done using the Mockito method *verify()* (Figure 5.7). The test can be found in *AdminBeanTest*.

```
@Test
public void TestSetAcceptedToDAOFacade()
{
    // Update status to true.
    // Update the status for a specific applicant.
    // Verify that the correct method was called.
    adminBean.setAccepted(true);
    adminBean.setAccepted();
    verify(mockedDAOFacade).setAccepted(0, true);
}
```

Figure 5.7 – Verify that right method in *DAOFacade.java* is called from SUT.

As mentioned before, the get/set methods with more advanced parameters require a more elaborate set-up part in the test class. During the set-up, a fictive application is created in order to compare certain values from it with parameters that are returned from some of the get-methods. Figure 5.8 shows pseudo-code on how this is done. The rest of the code from *AdminBeanComplicatedTest* is located in the project test package *view*.

```
/*
 * Create an applicant with the regular parameters.
 * The parameters for the applicant is:
 * Id, name, surname, email, status of application (accepted/declined),
 * and the registration date.
 */
applichant = new Applicant(1337, "Darth", "Vader",
        "WhosYourDaddy-o@LukeSkywalker.x",
        false, new Date(System.currentTimeMillis()));

// Create a new availability.
List<Availability> availability_list = new LinkedList<Availability>();

-------------------------------------------------------------------------

@Test
public void TestGetApplicants()
{
    /*
     * When method getApplicants() in the mocked class, DAOFacade, is
     * called, return list of all applicants, listAppDTO.
     */
    when(mockedDAOFacade.getApplicants()).thenReturn(listAppDTO);

    // Call getApplicants().
    adminBean.getApplicants();

    // Verify that method getApplicants() in DAOFacade was called
    // exactly once from AdminBean.
    verify(mockedDAOFacade, times(1)).getApplicants();

    // Assert that getApplicants() returns the correct parameter,
    // listAppDTO
    assertEquals(adminBean.getApplicants(), listAppDTO);
}
```

Figure 5.8 – Part of the set-up code and one of tests for advanced methods.

All of the methods in test class *AdminBeanComplicatedTest* pass values to the mocked *DAOFacade*. Verification on all these methods is done to make sure that the right method was called in the mocked controller, just as the special test case of the simple set-method *setAccepted()*.

5.2 Selenium Test Results

Analysis and evaluation of this framework is based off of three test cases, just as when evaluating the Mockito Testing framework. Selenium passed all three test cases without any modifications needed. All three test cases are implemented using three different methods.

For two of those methods, the test is executed as Java code in NetBeans IDE. For this purpose, a new Java project, separate from the main Java EE Web project, is created to run all three test cases when using method 2 and 3. The new Java project is named *SeleniumTestcases* and it contains as two packages per test case, one test package for the second method and one source package for the third method of implementation. In Figure 5.9 the project tree of this new Java project is shown.

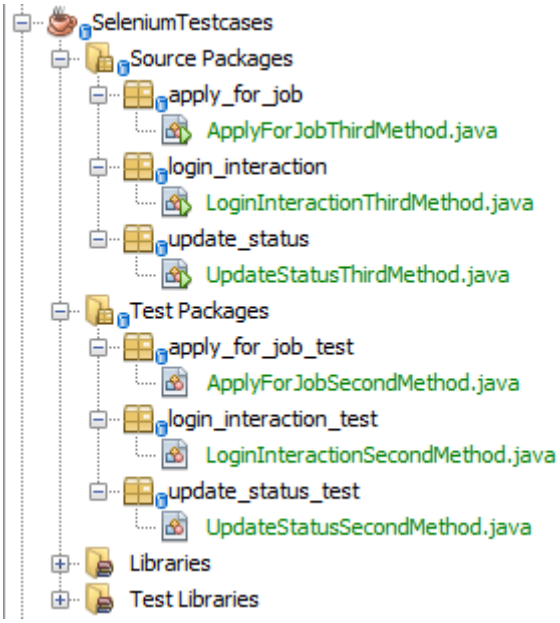

Figure 5.9 – Project tree for new Java Project.

The test results and the three different methods of implementation for each test case are described below.

5.2.1 Results for test case: Test of login interaction

Test case passed using Selenium. Test is implemented in three different ways since Selenium offers several ways of testing. All three methods follow the same interaction defined in section 4.1.4 but implemented differently and they are described here.

The first method is making the test using the recording tool of the Selenium IDE plug-in for Firefox. The recording is played back using the Selenium IDE plug-in and the flow of the recording is illustrated in Figure 5.10 where each box is one step of the login interaction. They are numbered in the order that they are done.

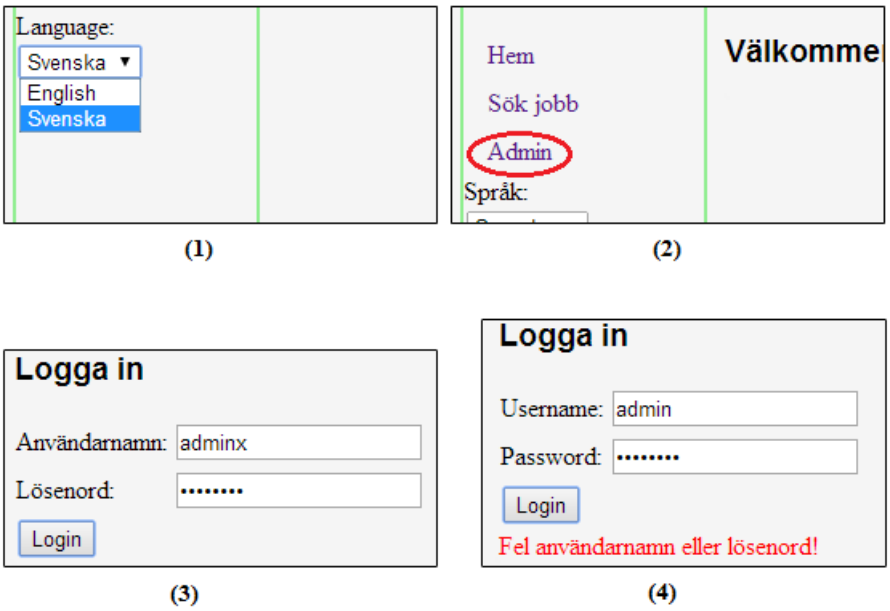

Figure 5.10 – These are the four steps described in the test case definition found in section 4.1.4.

The second method of implementing the test case is by exporting the recording to Java code using the Selenium IDE plug-in. The test is run in NetBeans IDE. The code is executed as a JUnit test and is run on the same server as the Java EE Web project, namely Glassfish, but in a new container. When the test is run, the Firefox WebDriver starts a new Firefox instance where the interactions are done. The name of the exported Java class is *LoginInteractionSecondMethod.java* and it is found under a test package called *login_interaction_test* for a new Java project named *SeleniumTestcases*. The new Java project is created for the purpose of running the Selenium test cases that require an IDE.

A third method is by manually writing a test in Java. Selenium supports other programming languages like C# and Ruby but this test case focuses on Java because the web project is written in Java. In order to write a Selenium black-box test in Java, the IDE in use must have access to the Selenium framework. To this test case NetBeans is used. This is accomplished by importing a certain number of libraries into NetBeans. More information on how to implement Selenium is found in APPENDIX B: SELENIUM FRAMEWORK TUTORIAL.

The written code simulates the different steps of the login interaction instead of showing them in a Firefox window. The code verifies that the test is passed by checking that a certain element is set to be visible since this element is only visible for the user once the login is successful. The Java class is called *LoginInteractionThirdMethod.java* and it is located in a Java package called *login_interaction*. This package is located under the source folder of the Java project *SeleniumTestcases*.

5.2.2 Results for test case: Test of login interaction & update status

Test case passed using Selenium. The test is repeated using three different methods of testing. In all three implementations, the same steps are taken as defined in section 4.1.5. The first method involves recording the interaction using the Selenium IDE plug-in for Firefox and playing it back directly through the plug-in. The second method of carrying out the test is by exporting the recording as a Java class and running the test in NetBeans IDE. The name of the exported Java class is *UpdateStatusSecondMethod.java*. The class can be found under the source package *update_status_test* for a separate the Java project *SeleniumTestcases*.

The third and final method concerns writing the test manually in Java and executing it in NetBeans IDE. This method completely avoids the use of the plug-in for Firefox.

The written code simulates the different steps for creating an application and does show them in a Firefox window. In order to verify that the status is changed, the value of the check box before the test is run is compared to the value of it after the test is executed. Figure 22 shows the code for the verification. The Java class is called *UpdateStatusThirdMethod.java* and it is located in a test package called *update_status*. The package is found under the source folder of the Java project *SeleniumTestcases*.

```
// Check the status of application and print out its value before
// changing it
try
{
    assertTrue(driver.findElement(By.cssSelector("BODY")).getText()
            .matches("^[\\s\\S]*REJECTED[\\s\\S]*$"));
    System.out.println("Current status is 'REJECTED'");
}
catch (Error e)
{
    System.out.println("Current status is 'ACCEPTED'");
}

// Click the "Update" button.
WebElement updateStatus=driver.findElement(By.name("j_idt26:j_idt30"));
updateStatus.click();

/*
 * Check the status of application after updating the application.
 * If the same value is printed twice, no change has been done and the
 * test is concidered a failure.
 */
try
{
    assertTrue(driver.findElement(By.cssSelector("BODY")).getText()
            .matches("^[\\s\\S]*REJECTED[\\s\\S]*$"));
    System.out.println("Current status is 'REJECTED'");
}
catch (Error ex)
{
    System.out.println("Current status is 'ACCEPTED'");
}
```

Figure 5.11 – Test case verification code. Check to see if application status was changed.

5.2.3 Results for test case: Test of creating an application

Test case passed using Selenium. The test is implemented three times, using a different method each time. The methods are the same as those described in sections 5.2.1 and 5.2.2. The first method involves the Selenium IDE plug-in for Firefox. The interaction described in section 4.1.6 is recorded and played back using only the plug-in. Here, the recording constitutes the test. If any change is done to the site, the recording can be played back to see if any crucial part has been altered unintentionally.

The second method of implementing the test involves another way of executing it and it requires an IDE. Instead of playing back the recording in the plug-in, it is instead exported as a Java class and the test is run in NetBeans IDE. When executing the test, a Firefox window opens where the interaction is simulated. This allows the developer to follow the interaction easier, instead of having to look at the source code. The exported Java class is named

ApplyForJobSecondMethod.java and is located in the test package *apply_for_job_test* in the new Java project *SeleniumTestcases*.

The third method does not use the Firefox plug-in at all. Instead, the test is written manually in Java and executed in NetBeans IDE. There is no visual feedback using this method, as it is in the first two methods. To compensate for this, a check is added to the code, similar to the one illustrated in Figure 5.11. In this test case, if the creation of a new application is successful, the user is sent to a new web page that has the message *"Success!"*. The verification checks that the message simply exists at the current page after the user finishes creating the application.

If the message is not found on the current web page, it is because the process of creating a new application failed. In this case, user is sent to an error page that shows the message *"Oops, something went wrong! Did you try something weird? :("* instead. In Figure 5.12, the code for the verification is illustrated. The Java class for this test is called *ApplyForJobThirdMethod.java* and is found under the Java source package called *selenium_testcases*. The packages in located under the source folder of the new Java project *SeleniumTestcases*.

```
/*
 * Verify that test passed by checking that the message "Success!"
 * is visible on the current web page. If not, the creation did not
 * succeed and the user is sent to an error page instead.
 */
try
{
    assertTrue(driver.findElement(By.cssSelector("BODY")).getText()
                                .matches("^[\\s\\S]*Success![\\s\\S]*$"));
    System.out.println("Application created.");
}
catch (Error e)
{
    System.out.println("Something went wrong.");
}
```

Figure 5.12 – Verify that test passed by looking for the message *"Success!"*

5.3 Evaluation of Tutorials

In this section, the results from the evaluations of the tutorials are presented. The resulting structure of them is explained in section 5.3.1.

Each tutorial has a different design, based off of the evaluations given by students that volunteered. Each document is evaluated by four different students, all of which has attended the course *IV1201 – Design of Global Applications* and has appropriate programming skills in Java to pass the course.

From the evaluations, the biggest gripe with the tutorials is the order in which the information is presented. Both tutorials have been restructured several times due to the feedback from the evaluating students. However, the actual information is not changed as often. A good example of the faulty order and the solution to the problem can be found in the Selenium tutorial. Because the tutorial explains how to use Selenium in more than one way, some information becomes irrelevant for a different course of action and may lead to confusion for the reader. In other cases, some methods overlap. This means that some sections are part of more than one method.

To solve this issue it is decided to introduce paths into the tutorial. The reader is recommended to choose only one path to follow and only read those sections that concern that specific course of action. If the student is required to skip a section, he is explicitly told where to continue on reading in order to stay on the chosen path.

One unforeseen side effect of this is that the paths lead to even more confusion. The evaluators claimed that it was not clear enough where to make the jumps to skip unnecessary sections and they found it hard to follow the path of choice.

To solve this new problem, a flowchart is created and put early on in the tutorial, in an effort to make the available paths even clearer. Also, each path is explained in a bullet list right before the flowchart in order to separate them further. This cleared any confusion about how to read the tutorial and the evaluators complained no more.

The evaluators where also an instrumental part when it comes to refining the language that is used in the tutorials. In general, the students are all good at pointing out information that feels redundant or confusing which helps in keeping the information short and to the point.

5.3.1 Resulting structure of tutorials

In this section, the resulting structure from the evaluations is described. The tutorial for Mockito begins with a short summary of what it is used for and gives a brief description of the target Framework. It then mentions the prerequisites such as system requirements and the applications needed to be able to follow the tutorial.

From this point forward, the tutorial shows in a step-by-step manner how and where to download the Mockito framework, how to implement it and finally how to create and run tests using Mockito. Each section contains appropriate illustrations to further clarify certain steps. The document ends with a section about the framework API to give a more in-depth insight about the framework using examples and illustrations of pseudo code. The tutorial is added to this report as APPENDIX A: MOCKITO UNIT TESTING TUTORIAL.

When it comes to Selenium, its tutorial follows a similar overall structure but differs significantly when explaining how to implement the framework and testing with it. This tutorial also has a short summary of the document itself and about the framework, at the beginning. This

is followed up by the prerequisites and where to download the necessary files, much like the Mockito tutorial.

After this, it explains how to implement the Selenium framework but here is where it starts to differ from the Mockito tutorial. The document is divided into three possible ways of following it. All three paths are presented and explained with further clarification using a flowchart and the reader should choose to follow only one path to avoid confusion. Each path has its own way of implementing the framework and testing with it. The document has no dedicated section about the framework API. Instead, it is explained along the way since Selenium is easy enough to learn and use this way. The tutorial is available as APPENDIX B: SELENIUM FRAMEWORK TUTORIAL.

6 DISCUSSION AND CONCLUSIONS

A discussion of the results and the conclusions that have been drawn during the Science thesis are presented in this chapter. The conclusions are based from the analysis with the intention to answer the formulation of questions that is presented in Chapter 1.

6.1 Discussion of test case results

In this chapter, the results of each test case are discussed and reflected upon in a separate section. Each section ends with some sort of conclusion.

6.1.1 Discussing results for test case: Test the logger

In section 5.1.1, no explanation is given for why the modification to the Original SUT is needed. One reason is in how the log method in the original SUT opens a write stream to a text file in order to log an error message. In the unmodified method, the real path to each text file is extracted from the servlet context using a library method named *getRealPath().* This method gets the real path that corresponds to the given virtual path, in this case to the text file.

With Mockito, the servlet context can be mocked. The problem arises when trying to get the real path to the text files. Since the context is mocked, it is communicating to a different servlet container. In this container, the context is simulated and all methods on it will return null, such as the library method *getContext().* For this reason, trying to extract a real path from a non-existing context will result in a null-pointer exception being thrown and the test will fail.

Another problem that has more to do with the limitations of Mockito is that the method in *Logger* is static. Mockito cannot mock static methods by design. The reason for this decision is that some actions, which will not be covered here, taken in order to mock an object, happen dynamically at run-time. An approach that is not possible with static objects due to the fact that static objects cannot be overridden. Arguments against static members in java say that it is often a sign of bad design and it should be avoided. However, the method in *Logger* is static because the method is not supposed to change or overridden. For instance, if it is allowed to override the method, the real paths to the text files can change involuntarily. By having a static method, it becomes hard to test anything that involves this method using Mockito.

The solution involving a modified SUT and not the original for the test renders Mockito useless. This means that there is no need to mock any of those classes, effectively avoiding the use of the unique properties of Mockito which is, simpler mocking. Instead, the test is done purely with the JUnit framework, of which Mockito is an extension of. Furthermore, as mentioned in chapter 2.1, it is conventionally wrong to create a completely new dummy class or somehow alter the SUT just to be able to test it. The test simply loses it validity and becomes unreliable when the SUT is changed just for the sake of running the test successfully.

Unless the behavior and logic of the SUT remains the same during testing as when during run time in a real-world scenario, there is no guarantee that the test itself is valid. Being sure that the SUT is not altered “too much” is very hard to determine and, for this reason, it should not be changed in order for a test to run properly. Nonetheless, the test case is to see if the log method actually writes to the various log-files, and it does. The difference being that the paths to each text file are hardcoded and not extracted from the servlet context.

This test case turns out to be difficult to implement with Mockito, in fact the solution effectively avoids the use of the framework entirely. It is decided to keep the test case anyway as it serves as a good example on where and why Mockito might be a bad choice as a testing framework. The test case also proves to be a good learning experience and it shows how the limitations of the framework can affect the testing.

As mentioned before, the test case was passed using JUnit but JUnit does not have similar capabilities compared to Mockito. In conclusion, this test case is not well defined for Mockito and, in the end, not suitable for Mockito.

6.1.2 Discussing results for test case: Test of login method

In order to pass the test case, some steps are taken before the test can be run. The method that is tested uses the class *FacesContext.java*. This class contains the state information of a JavaServer Face request. In this case, a method called *getCurrentInstance()* from *FacesContext* is used to get the instance that is being processed by the running thread. With the current instance, the external context is accessed using the method *getExternalInstance()*. This class specifies the behavior of the underlying servlet implementation such as HTTP servlet requests or HTTP sessions **[15]**. With this, an authentication key can be put into the session, giving the user access to restricted pages upon a successful log in.

All these steps need to be simulated during the test. To accomplish this, *ContextMocker.java* is created. As mentioned in the results, it mocks the context so that a key can be associated with it once the tested method is called from a test environment. If this was not done, there would be no context and the test would fail. The test class essentially creates a fake user session and simulates a successful log in to the system. This is done in order to check if *login()* passes to the right method in the controller of the Java EE Web project, namely *DAOFacade.java*. The steps taken before the test can be run are defined in the set-up stage of the test class. This is illustrated in Figure 6.1.

From Figure 6.1, it is rather apparent that the complexity of the test case is higher relative to previous test cases. However, what actually is being tested, or verified in this particular case, is not very advanced. In the test, a simple check is done to verify that the right method in the right class is called when the method in the SUT is called. In order words, a small-scale integration test is conducted using the top-down approach.

It turns out that Mockito is a good testing framework for integration testing using the top-down approach since verifications of this nature are easy to do. The mocks that are easily created using Mockito can be used to simulate called classes that are called from the SUT. In terms of integration testing, the mocks represent the stubs and the SUT the module subjected to the test. This is scalable to cover a higher percentage of the number of classes called by the module.

```
@Before
public void setUp()
{
    // Mock the class DAOFacade. The controller.
    mockedDAOFacade = mock(DAOFacade.class);

    // Pass the mocked object to the SUT.
    authBean.setDAOFacade(mockedDAOFacade);

    // Mock FacesContext using Class ContextMocker.
    FacesContext context = ContextMocker.mockFacesContext();
    // Create hash map of object-string pairs. This defines
    // user session permitions.
    Map<String, Object> session = new HashMap<String, Object>();
    // Put authenticaion key into hash map called session.
    session.put(AUTH_KEY, "..was here");
    // Mock the external context.
    ExternalContext ext = mock(ExternalContext.class);

    // When SUT tries to put authentication key into user session,
    // return session created above.
    when(ext.getSessionMap()).thenReturn(session);
    // When SUT wants the external context, return mocked context
    when(context.getExternalContext()).thenReturn(ext);

    // Set username & password to be able to verify that
    // login in DAOFacade was called with same parameters from SUT
    authBean.setUsername("Terminator");
    authBean.setPassword("c00lk1ll3rb0y#96");
}
```

Figure 6.1 – Set-up stage of the test. Run before the test code is run using JUnit @before annotation.

As mentioned in the test case results, the injection point for the controller in the SUT is used by the test class to set a mocked version of the controller when testing. The injection point is used by the SUT to access the controller during normal run time. Without this code, the SUT cannot access the controller, *DAOFacade*. Another common way of giving the SUT, *AuthenticationBean* access to *DAOFacade* without breaking the MVC-modeling rules is by using the EJB annotation. Defining the controller as an EJB in the SUT yields the same results as an injection point.

However, accessing the controller as an EJB becomes problematic when testing with Mockito. Since there is no explicit way of setting an object to be the mocked *DAOFacade* during a test, it is not possible to mock out the dependencies between the SUT and the controller, unless the SUT is changed and an injection point is added solely for the purpose of testing. Doing this change will not alter the SUT in any significant way. It only gives the SUT two ways of accessing the

controller. The problem lies in that the SUT is changed just for the purpose of testing it and this, as mentioned in section 3.3, should be avoided when designing unit tests.

If EJB's are used, after all, a developer must remember to add another entry point when designing the SUT. This design decision does not contradict any rules set by the MVC-model. In other words, it is tolerable to have access to the controller, both as an EJB and through an injection point.

In a simpler scenario, as the example used in section 3.4, the SUT contains a constructor that takes an object of the class upon which it depends, *ClassB*. *ClassB* is usually the class that is mocked. When running the SUT as usual, outside of a test environment, the method call from the SUT is passed to *ClassB* using an object of *ClassB*. The SUT has access to *ClassB* and can therefore generate the object with the help a constructor. During a test however, the object for *ClassB* is replaced by a mocked one.

In this case, the SUT does not have a constructor for *DAOFacade* because of rules dictated by the MVC-model. Instead, *DAOFacade* is accessed through an injection point. This method is called by the test class to set *DAOFacade* to be the mocked object. The injection point is illustrated in Figure 5.5.

Reflecting upon the test case itself, it turned out to be well suited for Mockito. Even though the solution required a more complex set-up relative to test case 5.1.1, it showed in a good way how to tackle a more complex scenario that is also common in Java EE Web projects. The scenario is the mocking and use of *FacesContext* and *ExternalContext* and how to work with EJB's and CDI using Mockito.

6.1.3 Discussing results for test case: Test of getters and setters

The reason for why the test case has two test classes is because it makes the test code easier to follow. It is possible to only have one test class but this solution would make the test class bigger and harder to grasp the full scope of the solution. Also, most of the get/set methods do not need any advanced parameters to be set up beforehand. To have the set-up code in the same file could become confusing since it is only used by 5 out of the 16 tested methods.

When it comes to the tests themselves, one of the simple set-methods is tested differently compared to the other set-methods, as mentioned in the results. This is because the method, called *setAccepted()*, in the SUT passes a value to the controller called *DAOFacade.java*. To begin with, this requires mocking out the dependencies to the controller using Mockito. Furthermore, the test verifies that the right method in *DAOFacade* is called from the SUT. This type of verification is actually an integration test with Mockito, using the Top-down approach. The test is not only a good example on how to implement integration testing with Mockito but it also shows how closely related unit testing and integration testing are.

As mentioned in the test case results, the test class uses the injection point in the SUT to pass a mocked object of *DAOFacade* during the test. The reasoning behind this is the same as for the test case discussed in section 6.1.2, fifth and sixth paragraphs. The injection gives the SUT the ability to handle mocked objects. Without this code, say using an EJB instead, the SUT does not use the mocked *DAOFacade*. Instead it calls the method in the real *DAOFacade* when running the test but, since *DAOFacade* is not defined in the test environment, the test fails.

The tests defined in *AdminBeanComplicatedTest.java* prove to be a good learning experience on how to deal with more complex methods using Mockito. Also, the tests give an insight on how hard it can be to test when mocking is not a viable solution. For example, a complete application

for a fictive applicant has to be created just for the sake of testing some of the get-methods. These methods ordinarily return the needed information from the system database, which makes them dependent on the database. Mocking dependencies is still needed but it is sometimes not enough, as this test case shows.

In retrospect, this test case is well suited for Mockito since it serves as a good example on how and when Mockito can facilitate testing of a more advanced code base.

6.1.4 Discussing results for test case: Test of login interaction

The reason for why the test case is implemented in three different ways using Selenium is because, if it did not, the test would feel too simplistic in nature. A good way of making the test case more substantial is by basically repeating the test case using another method and, by doing so, learning more about the different aspects of Selenium. This, in turn, yields better insight of the framework which helps when evaluating Selenium as testing framework.

When choosing the methods of implementation, the most prominent factor is the tutorial written for Selenium. In this tutorial, the same three methods are used when creating the example tests for it. These three methods are the first to be learned when evaluating Selenium during the tutorial. It is therefore natural to choose the same methods for the test case.

The fact that the test feels too basic when only choosing one method of implementation is, in itself, a testament to the ease of use of Selenium. Ultimately, the test serves as a good example on the different ways to use Selenium.

6.1.5 Discussing results for test case: Test of login interaction & update status

This test is more straight-forward since it is similar to test case 4.1.4 in its design and scope. It is also implemented in three ways for the same reasons as in the previous test case done with Selenium. The main goal with this test is to gain better test coverage of the code. There are arguments for how much of the code should be tested **[2], [6] & [19]** and a clear percentage seldom given.

For example, 100% test coverage of all functions in a SUT can still have bugs that only appear in a specific logic path that the user might take. A path that is unforeseen by the developers and, therefore, not handled properly. Instead, the focus should lie in creating good test cases that cover crucial parts of the system. In this case this means testing logic paths that are predicted to be used the most by the users, such as the login process or applying for a job. In this case, a path that partially covers the login process once more but it continues to change the status of an application.

Ultimately, achieving 100% test coverage of all possible logic paths is an unreasonable goal and that is not the goal with this test case. As mentioned before, this test case yield better test coverage but focuses on the more crucial parts of the web interface.

6.1.6 Discussing results for test case: Test of creating an application

The main difference between this test case and the other two done with Selenium is that this interaction creates a new application which means that a substantial amount of new information is added to the system database. In previous test cases, the database was used only to read from it or to change a single variable that indicates a status update. In this case however, several different variables are persisted and all must be written correctly to the database. As a consequence, the user must provide the right information which can lead an error prone process.

To make sure that all values given by the user are correct, several verification handlers verify that the given values are of correct format. As an example, one handle makes sure that the first name of the applicant is longer than two characters. If no, an appropriate error message is displayed and the user cannot continue until it is corrected.

Apart from verifying yet another interaction path, this test case also checks that the different verification handlers set in place are working as they should. This is the main purpose of the test case.

6.2 Discussion of tutorials

In this section, the tutorials are reflected upon. Each tutorial has its own section that focuses on the impact of the evaluations from chosen candidates that volunteered. The candidates are all students that fulfill a certain set of criteria defined in section 4.2 Tutorials.

6.2.1 Tutorial for Mockito

The goal with the tutorial is to create the means for the students to learn about unit testing with Mockito in a way that shortens the learning period as much as possible. If the learning process can be shortened, more time can be spent actually testing. This does not mean that it is the only way of achieving this however.

A different way would be to make a voiced video guide where each step is done by someone else and recorded. The user only needs to follow the steps. The benefits to this method might be that, by seeing the steps done, it might help against any confusion as opposed to letting the user interpret the instructions and doing the steps instead. This means that the instructions need to be clear and easy to understand to avoid confusion or misinterpretation, something that can be more relaxed when it is presented in video form. In such a case, the steps are shown and no interpretation is really required.

This is one of the main reasons for why the tutorials are evaluated by students. This way, any misinterpretation is corrected and different points of view help find different errors or problems. One ill-formulated instruction might be interpreted correctly by one student and miss the error but another student might find it confusing and point it out.

Also, making a video guide of good quality still needs to be evaluated and editing it is more time consuming than revising a tutorial in text form, one reason for why it was decided to do a texted tutorial instead.

When it comes to the structure of tutorial itself, it has section about the framework API not present in the Selenium tutorial. In this section, the framework is explained more in-depth with the intent of showing the many capabilities of the framework. This section is placed last with the reasoning that, at that point, the user should have basic grasp of the framework and knows how to test with it. The user is then ready to learn more advanced aspects in order to create better tests.

The point of including this section is to emphasize the complexity of the framework and to show all the different capabilities the framework offers, without making the reader feel like it is too much information.

The evaluations given by the students conclude, among other things, that the students prefer a more direct language where the information is presented as commands on what to do instead of writing the document as a user manual. If the text is written as a user manual, the information is instead presented as options on what a user could do or how a certain aspect works. The evaluations also tells that the instructions should be short and to the point as much as possible As an example, one instruction that tells the reader how to import the framework files to NetBeans IDE is shown below:

"Left-click Add Jar/Folder and then navigate on your PC to the folder which contains the Mockito library that you downloaded earlier from the Mockito website. Choose the Mockito library jar file called mockito-all-x.x.x.jar."

The same information can instead be presented as:

"To import Mockito into NetBeans, the IDE offers the option of adding the jar files for the Mockito framework into a test library that is separate from the Java EE project environment and can be run in a separate container."

The example is exaggerated in order to further emphasize the difference in presenting the information. Even though the second paragraph in italic may present more information about how the process works, the students are more interested on how to do it and they might get a sense of information overload. This is where the idea of adding a section solely about the Mockito API comes from. A place where, once the students are finished the different steps, they can read more about the intricacies of the framework, an idea that is well received by the students.

The structure of tutorial is also changed due to the feedback from the students. One such change, already mentioned, is the API section but also in the order that the information is presented. The first complete draft of the document explained the API of the framework in the same section where it is explained how to create tests with it. Furthermore, this section was placed before the section about how to execute a test. This design turned out to be confusing and the students felt like it broke the flow of the tutorial. The information about the API was therefore taken out and placed as an individual section, at the end of the tutorial.

To summarize, the evaluations are instrumental in the design, both overall and in detail, of the tutorial and even thought only four students evaluated the document, the resulting product is of satisfying quality. This might be because the criteria for an evaluator are well-defined. It would have been good to have more evaluators but, in the end, a small number of evaluations are better than no evaluations.

6.2.2 Tutorial for Selenium

The goal with this tutorial is the same as the tutorial for Mockito. Structure differs significantly from the Mockito tutorial in the way the information is presented. To begin with, the tutorial gives exactly three different ways of implementing the framework and testing with it. This forces the reader to make jumps to specific sections in the document. This can be a source of great confusion for the reader if it is not clear enough. The feedback from the students proves very useful in this matter.

By letting other students evaluate it, they can tell if there is any step that is convoluted. If so, the tutorial is changed accordingly until there is no confusion. One such change is the addition of a flowchart that illustrates what path the reader can take. The flowchart itself is color-coded with each path in a unique color, a decision also based off of the evaluations. The purpose of the different paths is to show the reader how flexible the framework can be. This is one of the strong suits of Selenium and it is important that this fact is portrayed in the tutorial. Selenium allows for even more ways of testing but the difference is minute between them. By having only three paths, the differences become more apparent.

This tutorial is written after the creation of the Mockito tutorial and a lot is learned about language and structure when creating the Mockito tutorial. This is knowledge is transferred to this tutorial as well and it shows for example, in the language. It is similar to that of the Mockito tutorial.

Another example is in its overall structure which is explained in detail in section 5.3 Evaluation of Tutorials. From that, the similarities between the two are described and, even though they are different in the later sections, they both follow a similar structure. In short words they both have a summary, a prerequisites-section, setting up testing environments and finally execution of tests, in that order. This structure is again a result of the feedback given by the students.

6.3 Conclusion

This chapter brings up the conclusion of the thesis project as a whole. The results are concluded to see if the original goals set in section 1.4 are met.

6.3.1 Frameworks

First of all, the frameworks need to be intuitive enough to use effectively and not too complex to learn so that they can be used by the students attending the course *IV1201 – Design of Global Applications*. It is important to acknowledge that every aspect of a Java EE web project is not covered by these two testing frameworks alone. However, this is not a criterion when choosing what frameworks to evaluate.

Instead, the frameworks need to be able make automated Unit tests and some simple Integration testing of the source code. This is solely accomplished by Mockito but having just one framework is too small of a scope for this thesis project. Because of this, it is decided to include another framework. This framework, apart from being well suited for novice users, needs to test another important aspect of any Java EE web project, the UI.

In this case the UI is web-based which narrowed down the possibly candidates. Ultimately, Selenium is chosen as the second framework because of its ease of use and simple integration to existing code base.

Example on other aspects to a Java EE web project that is not covered by these two frameworks are testing underlying frameworks such as OSGi or CDI and testing of database driven projects. To these aspects, there exist other testing frameworks such as Pax Exam and DbUnit, respectively.

Finally, the different test cases created to evaluate both frameworks lead to the conclusion that these two frameworks are, in fact, well suited for this thesis project. Together, they achieve the first goal set for this report, namely that they need to be user-friendly and simple enough so that they can be learned by the students within the timeframe of the course *IV1201 – Design of Global Applications*.

6.3.2 Tutorials

Ultimately, having the tutorial evaluated and having listened and acted upon the evaluations, the tutorial is now at the point where it is user-friendly enough so that a student with no experience about software testing can follow it.

Possibly the most important fact derived from the evaluations is that a student, not only acquires basic knowledge about testing in general but also about a specific testing framework, in less than 30 minutes. The student reaches a point where he or she can start testing a Java EE code base just by applying the information learned in the tutorial. This is one of the goals set for this thesis report and it is accomplished.

7 RECOMMENDATIONS AND FUTURE WORK

In this chapter, the ethics of the work is taken into account and future work in this field is presented.

7.1 Recommendations for a sustainable future

The point with automated unit testing is to give a quick answer to whether or not a change in the source code has broken the logic of the code in any way. In other words, Unit testing yields, if implemented properly and effectively in the beginning, robust code of high quality that is easy to maintain and develop further. The benefits may not be visible in the short term, especially in minor projects and if no standards are in place. However, if the project is supposed to have a longer life time and if the testing process is standardized, the development is sped up a lot and the maintenance of the code becomes easier and cheaper.

Automated testing can be an economically viable option for a sustainable future of a software company. Good unit tests leads to less bugs in the source code which, in turn, cuts the maintenance costs **[22]**. The slow start is compensated and eventually surpassed by the benefits of testing the software since it finds problems in an early stage of development. Also, because maintenance becomes easier, the number of testers can be lowered. This reduces salary costs and developers can spend more time actually developing than testing the code.

One aspect to unit testing and integration testing that is not addressed in this thesis project is the benefit of begin testing early on in the development of the project. More concretely, this can mean testing elements that might depend on other elements that have yet to be completed.

The goal is to find errors as early as possible so that it is easier to correct. If the code base is already finished, implementing unit testing involves refactoring code for testability, which can lead to more bugs. Another problem is that the tests themselves can become bias, meaning that they test the implementation and not the requirement.

Furthermore, testing after the code project has been finished makes the process more difficult. At that point, the code base is less flexible and requires more work to change. From a sustainability point of view, this problem might be the most damning. By slowing things down, salary costs increases due to the extra hours of development the project needs **[23]**.

The notion of testing early on, during development is an important aspect in a developing methodology called *Test-Driven Development (TDD)* **[8]**. The methodology is used in conjunction with *Extreme Programming (XP)* **[3]**, a software development methodology. TDD focuses testing early on with a slow start but with a high returned investment in the form of robust and flexible code. This, in turn, leads to fewer development costs **[5]**.

In conclusion, if the possibility of developing the Java EE Web project again presented itself, testing of it would begin much earlier and we would try to implement TDD. Even though the benefits of TDD may not be so apparent due to the small scale of the project, it would still be worth it mostly for the learning experience. We realize that using TDD for a project such as the recruitment system may not be doable because of the timeframe provided for testing when attending the course IV1201.

Such a thing would require a major restructure of the course itself and deter from the focus of designing of global web applications, which is the purpose of the course. With that said, it is

something we would like for course responsible to think about. The benefits of TDD are too many to ignore. We recommend that an appendix of no more than 4 pages about TDD should be handed out to the students early on in the course. This way, they can start thinking about unit testing and integration testing before beginning with the code project.

7.2 Future work

In the beginning of the project, a fourth unit testing framework called Pax Exam was looked into. This framework is specifically designed for In-container testing of OSGi **[12]**, Java EE and CDI. In this case, the capabilities of Pax Exam are viewed upon from a Java EE perspective. This include, without completely focusing on, some testing of OSGi and CDI.

Pax Exam offers the programmer the ability to take control of the OSGi framework, the test framework (in this case, JUnit) and the Java EE application under test conditions at the same time.

In order to understand the advantages and disadvantages of Pax Exam, a good understanding of what OSGi and CDI is and what these frameworks are used for is needed.

OSGi together with CDI are two comprehensive frameworks because they offer a lot of services to a Java EE project. This also means that they require a significant amount of reading in order to understand these frameworks enough to properly analyze the capabilities and limitations of Pax Exam as a testing framework.

This would take too long and, for this reason, it is decided to exclude Pax Exam from the scope of the project. Instead, it is encouraged that anyone interested in the field of unit testing to take the time and learn about Pax Exam.

It is an extensive set of tools for taking control over OSGi and CDI which are both important parts of a Java EE web project. The knowledge gathered here can be very valuable, not only when it comes to Pax Exam but also for other unit testing frameworks.

8 REFERENCES

In this chapter, all references are presented as a numbered list in alphabetic order from A to Z.

[1] A. Orso. “Integration Testing of Object-Oriented Software”, Polytechnic University of Milan, 1999, 119 pages

[2] artima developer, “How Much Unit Test Coverage Do you Need? – The Testivus Answer”. See http://www.artima.com/forums/flat.jsp?forum=106&thread=204677. Post number 95 by Alberto Savoia. Posted 2007-05-04. Accessed 2014-05-14.

[3] Extreme Programming (XP), "Extreme Programming: A gentle introduction", See http://www.extremeprogramming.org/. Last updated 2009-05-10. Accessed 2014-05-26.

[4] International Business Machines Corporation (IBM), “Error Logging”. See http://publib.boulder.ibm.com/infocenter/aix/v7r1/index.jsp?topic=%2Fcom.ibm.aix.kernelext%2Fdoc%2Fkernextc%2Ferror_log.htm. Accessed 2014-04-28.

[5] M. Müller and F. Padberg, "About the return on investment of Test-Driven Development, Universität Karlsruhe, Germany. Accessed 2014-05-26.

[6] M. Subbarao. “Is Code Coverage Important?”. See http://java.dzone.com/articles/is-code-coverage-important. Published 2008-10-20. Accessed 2014-05-14.

[7] MCDOWELL, “JSF: mocking FacesContext for Unit tests”. See http://illegalargumentexception.blogspot.se/2011/12/jsf-mocking-facescontext-for-unit-tests.html. Published 2011-12-27. Accessed 2014-02-05.

[8] Microsoft Developer Network, "Guidelines for Test-Driven Development". See http://msdn.microsoft.com/en-us/library/aa730844%28v=vs.80%29.aspx. Published 2005-05. Accessed 2014-05-26.

[9] Microsoft Developer Network, “Integration Testing”. See http://msdn.microsoft.com/en-us/library/aa292128%28v=vs.71%29.aspx. Accessed 2014-05-23.

[10] mockito – simpler & better mocking, “FAQ”. See https://code.google.com/p/mockito/wiki/FAQ. Last updated 2013-11-03. Accessed 2014-05-23.

[11] monkey island – about software, “expect-run-verify... Goodbye!”. See http://monkeyisland.pl/2008/02/01/deathwish/. Published 2008-02-01. Accessed 2014-05-23.

[12] N. Bartlett, “OSGi In Practice”. See https://github.com/njbartlett/njbartlett.github.com/raw/master/files/osgibook_preview_20091217.pdf. Published 2009-12-17. Downloaded 2013-09-16.

[13] Open Web Application Security Project (OWASP), “Error Handling, Auditing and Logging”. See https://www.owasp.org/index.php/Error_Handling,_Auditing_and_Logging. Last modified 2013-05-12. Accessed 2014-04-28.

[14] Oracle Corporation and/or its affiliates, "ExternalContext (Java EE 6)". See http://docs.oracle.com/javaee/6/api/javax/faces/context/ExternalContext.html. Published 2011-02-10. Accessed 2014-04-11.

[15] Oracle Corporation and/or its affiliates, "FacesContext (Java EE 6)". See http://docs.oracle.com/javaee/6/api/javax/faces/context/FacesContext.html. Published 2011-02-10. Accessed 2014-04-11.

[16] Oracle Corporation and/or its affiliates, "release (Java EE 6)". See http://docs.oracle.com/cd/E17802_01/j2ee/javaee/javaserverfaces/2.0/docs/api/javax/faces/context/FacesContext.html#release%28%29. Accessed 2014-04-21.

[17] R. Osherove, The Art of Unit Testing. 2 edition. New York: Manning Publications Co., 2014.

[18] S. Borate, "Importance of logging in web development". See http://www.codediesel.com/software/logging-in-web-development/. Published 2010-10-04. Accessed 2014-04-28.

[19] S. Cornett. "Minimum Acceptable Code Coverage". See http://www.bullseye.com/minimum.html. Updated 2013. Accessed 2014-05-14.

[20] S. Lindberg and F. Strandberg. "The Development and Evaluation of a Unit Testing Methodology", Karlstad University, 2006, 161 pages.

[21] S. Mishra, "Integration testing". See http://www2.informatik.hu-berlin.de/~hs/Lehre/2004-WS_SWQS/20041126_Ex_Integration-testing.pdf. Published 2004-11-26. Downloaded 2013-12-10.

[22] S. Vaaraniemi. "The benefits of automated unit testing". See http://www.codeproject.com/Articles/5404/The-benefits-of-automated-unit-testing. Published 2013-11-08. Accessed 2014-05-23.

[23] S. Walter, "Test-after Development is not Test–Driven Development". See http://stephenwalther.com/archive/2009/04/08/test-after-development-is-not-test-driven-development. Published 2009-04-08. Accessed 2014-05-26.

[24] Selenium Project, "Introduction". See http://docs.seleniumhq.org/docs/01_introducing_selenium.jsp. Last updated 2014-04-21. Accessed 2014-04-29.

[25] SMARTBEAR, "Why Automated Testing?". See http://support.smartbear.com/articles/testcomplete/manager-overview/. Published 2014. Accessed 2014-05-23.

APPENDIX A: MOCKITO UNIT TESTING TUTORIAL

This appendix is a tutorial over how to implement Mockito Unit testing/mocking framework. It also contains a code example of a simple test created exclusively for this tutorial.

The goal with this tutorial is to show how to implement and use the Mockito testing framework. Mockito is a testing framework implemented as an extension to JUnit, a testing framework itself for Java. Mockito allows for mocking of objects. Mocked objects are used in automated unit testing with Mockito. A mock simulates the behavior of an object in order to test another object that is dependent on it. The advantage of this is that the behavior of a mock can be controlled very precisely in a test environment and dependencies between different objects are easily set-up in a separate container.

Before starting the tutorial, it is assumed that Netbeans 7.0.1 or later has been installed and that the user has access to our Java EE Web project, The Recruitment System. Also the project must be imported and implemented in NetBeans. This tutorial has only been tested on a PC running Windows 7. It has not been verified to work on other Operating systems but there should not be any major differences since this tutorial focuses on NetBeans IDE.

A.1 Downloading the necessary files

Visit the Mockito official website at: https://code.google.com/p/mockito/downloads/list
Download the latest stable build jar file called mockito-all-x.x.x.jar. This file only contains a single jar file for importing the Mockito source files.

A.2 Implementing Mockito to your Java EE Web project

Import the library to the project as described below.
Start NetBeans.
Right-click on the folder *Test Libraries* located inside your Java EE Web project tree. Your Java EE project tree can be found under *Projects* in NetBeans (Figure 1).

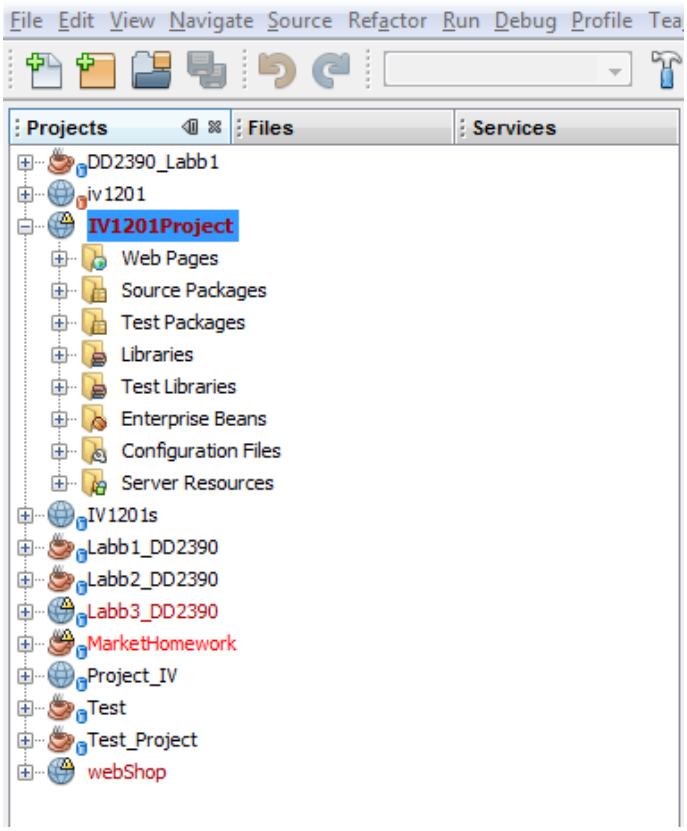

Figure 1 – Project tree in NetBeans IDE.

Left-click *Add Jar/Folder* and then navigate on your PC to the folder which contains the Mockito library that you downloaded earlier from the Mockito website. Choose the Mockito library jar file called *mockito-all-x.x.x.jar*.

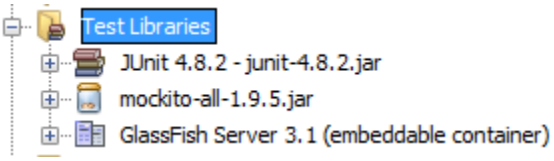

Figure 2 – Mockito framework imported into test libraries.

Once Mockito has been imported it will be visible as a jar file in the folder *Test Libraries* at your Java EE Web project tree (Figure 2). You are now able to use Mockito as a testing tool.

A.3 Setting up test environment for Mockito

Right click on your Java EE Web project and navigate to *New* and then to *Other…*
Choose the category *JUnit* on the left hand side under *Categories* and choose the file type *Test for existing class* (Figure 3).
NOTE: In later NetBeans builds the category name has been changed to *Unit Tests*.

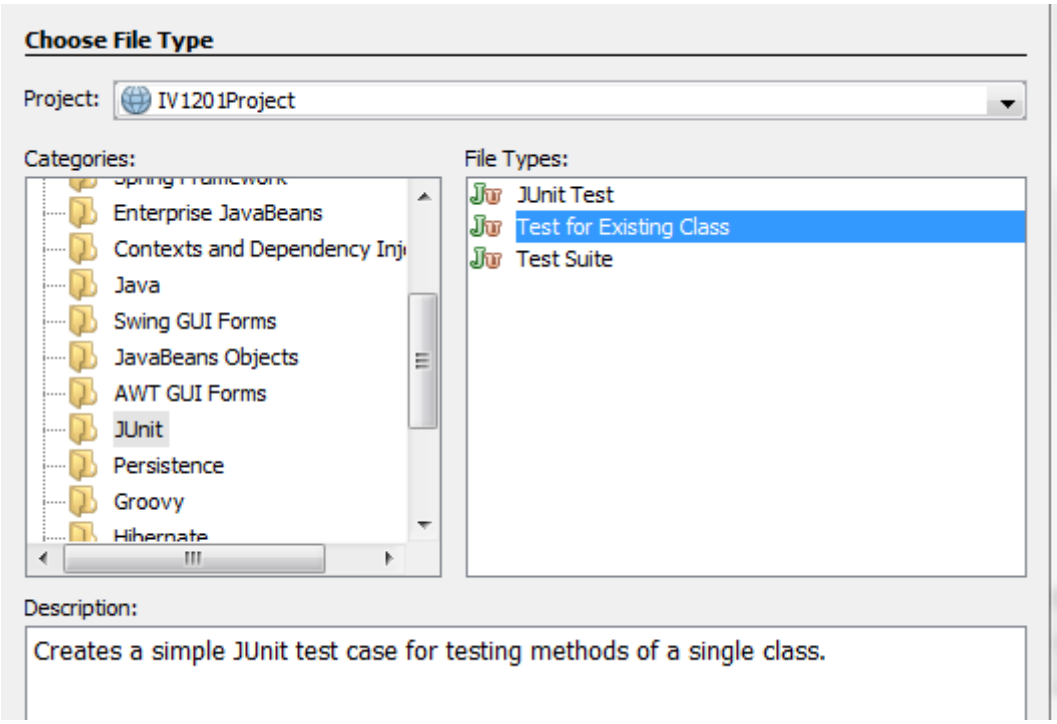

Figure 3 – How to create a new test for existing java class in NetBeans IDE.

Choose the class you want to test from your Java EE Web project either by typing the class name or by browsing to it with the *Browse...* button. For this example, we have chosen the class *DAOFacade.java*. It is recommended that you do the same for easier learning. *DAOFacade* is located in the project source package *controller*.

Click *finish*.

You will now be presented with a new test class called *DAOFacadeTest.java* which contains around 150 lines of automatically generated test code. For this example, please delete this code but leave the imports at the top of the class intact. This new test class, *DAOFacadeTest*, will be located under a new folder called *Test Packages*. The name of test package will be the same as the name for the source package where the tested class resides. For instance, *DAOFacade* is located in the source package *controller* and its test class, *DAOFacadeTest* is found in the test package with the same name, *controller*. Their locations are illustrated in Figure 4 where the project tree is shown. The classes are highlighted in blue.

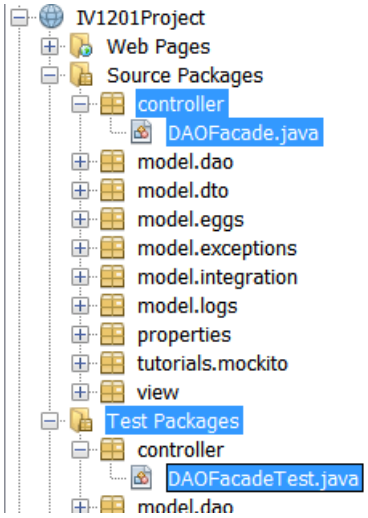

Figure 4 – *DAOFacade.java* and *DAOFacadeTest.java* in their respective locations.

A.4 Writing a simple test with Mockito

At this point, *DAOFacadeTest.java* should only contain the imports. It is now time to write the test in it. The Mockito API provides several methods for many different testing scenarios. For this example, the focus will be on one common method called *verify()*. Before writing the test, it must be defined and, to this end, some context must be given first.

DAOFacade is the controller for the IV1201 web project and it acts as a facade over the lower layers of the system, such as the source package *model*. All calls from the upper layers, for instance the source package *view*, must pass through the controller. Its job is to validate the calls and to delegate them down to the right location in the lower layers.

DAOFacadeTest will test one of those delegations and check that the right method was called in another class, at the lower layers. The method that will be subject to test is called *login()*. This method receives two parameters from the upper layers and passes them to a lower-layer class called *Logic.java* by calling the method *login()* in this class. To clarify this, the method that will be tested in *DAOFacadeTest* is shown in Figure 5.

```java
@Stateless
public class DAOFacade
{

    @EJB
    private Logic logic;

    /*
     * Injection point for Logic. Instead of accessing it as an EJB
     */
    @Inject
    void setLogic(final Logic logic)
    {
        this.logic = logic;
    }

    /**
     * Checks if the username and password are correct.
     *
     * @param username
     * @param password
     * @return 0 if success
     */
    public int login(String username, String password)
    {
        return logic.login(username, password);
    }
```

Figure 5 – Login method in *DAOFacade.java* passes parameters to method of the same name in class *Logic.java.*

The class *Logic* returns to *DAOFacade* an integer 0 if the set parameters are correct. These parameters have to do with the login information and are provided by the user when trying to log in to the system. Now that the test is defined, it is time to give a short description of the methods used in the test at *DAOFacadeTest*.

The method *verify()* is often used to make sure that a certain behavior happened or not. The method also allows for high granularity. For instance, the test can verify that a certain method was called at least three times or that the call is done with a specific set of parameters. In this case, the test will verify that the method in *Logic* was called exactly one time with the correct login parameters.

This implies that there exist dependencies between *DAOFacade* and *Logic*. Since the test is run in a separate container than the project itself, the dependencies must be set up somehow. Thankfully, these can be mocked out using Mockito. This is what mocking is used for and why Mockito was created, to simplify mocking for the user. The tutorial will come back to the properties of Mockito concerning mocks in section 1.6 but for now, it is only shown how it is done. Figure 6 illustrates the mocking of *Logic*, the class for which *DAOFacade* is dependent on.

```
/*
 * Set-up that runs Before the test.
 * Specified with JUnit annotation @before
 */
@Before
public void setUp()
{
    // Mock Logic in order to set up dependencies
    mockedLogic = mock(Logic.class);
    // Pass mocked Logic to DAOFacade
    daof.setLogic(mockedLogic);
}
```

Figure 6 – Set-up stage of test in *DAOFacadeTest.java.* This shows how to mock *Logic.java.*

Please observe the *@Before* annotation. This is a JUnit annotation and it specifies that the method *setUp()* must be executed before the test itself. *@Before* is usually used when there is more than one test method and they share resources such as mocked objects. It is therefore good practice to mock out the required dependencies during the set-up stage of the test instead of directly in the test method, although this is also a viable option. In Figure 7, *DAOFacadeTest* is shown in its entirety. Please use this as a template for your own test class.

```java
package controller;

import model.dao.Logic;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import static org.mockito.Mockito.*;
import org.mockito.runners.MockitoJUnitRunner;
// Needed for Mockito Annotations such as @Mock. Not used in this test class.
@RunWith(MockitoJUnitRunner.class)
public class DAOFacadeTest{
    // Variable instantiations
    DAOFacade daof = new DAOFacade();
    Logic mockedLogic;

    @Before
    public void setUp(){
        mockedLogic = mock(Logic.class);
        daof.setLogic(mockedLogic);
    }
    /*
     * Test of login method in DAOFacade.java.
     * Test verifies that method delegates to right method in lower-layer
     * class called Logic.java exactly once with correct set of paramters.
     */
    @Test
    public void testLogin() throws Exception{
        // Run method login in DAOFacade.java
        daof.login("Mustafa", "Mus");
        // Make verification on method login in Logic.java
        verify(mockedLogic, times(1)).login("Mustafa", "Mus");
    }
}
```

Figure 7 – *DAOFacade.java*. This can be used as a template.

With the test class provided, it is now time to run the test. *DAOFacadeTest* can also be accessed directly at its test package called *controller*.

A.5 Executing a test

To execute the test, right click on the test class at the project tree and choose *Run File* (Figure 8). Alternatively, all test classes under the folder *Test Packages* can be run simultaneously by right-clicking on the project instead and chose *Test*.

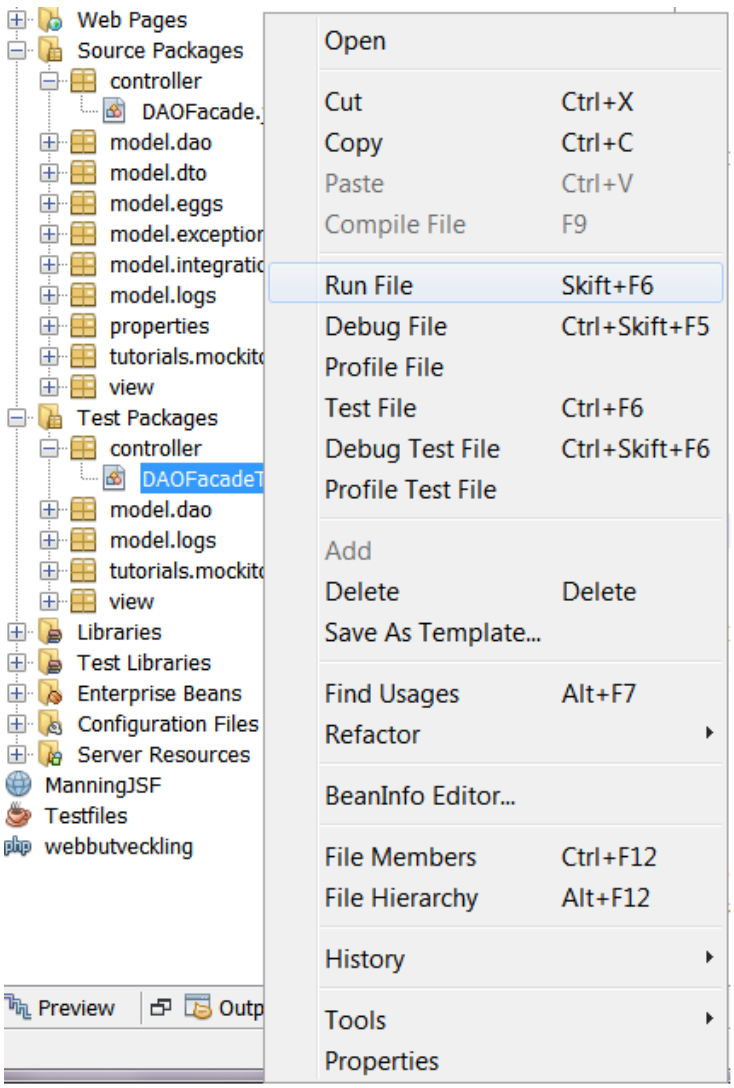

Figure 8 – How to run a test class in NetBeans IDE.

A new window will appear showing the test results at the lower left in NetBeans (Figure 9).

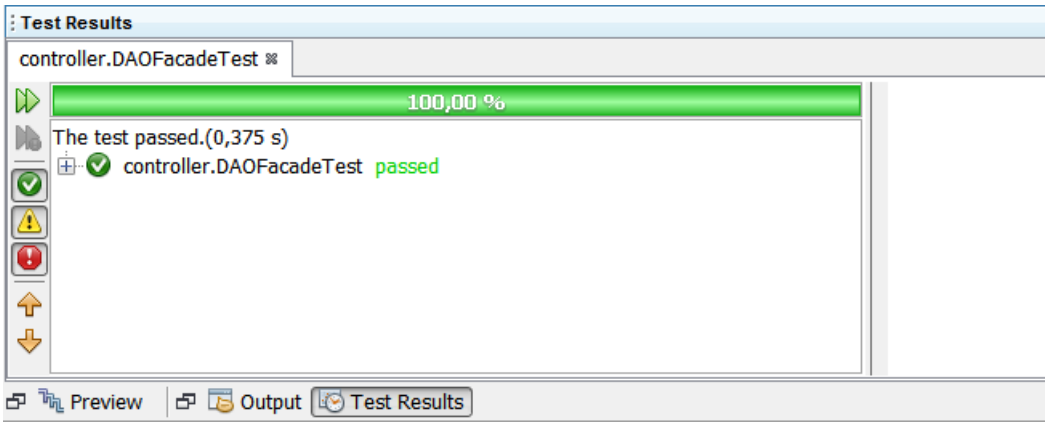

Figure 9 – Results from a passed test.

A.6 The Mockito API

Please disregard any project specific naming such as class names and package names in the following example. This example was coded for the sole purpose of showing the Mockito API in a tutorial. The goal with this example is to familiarize the reader with the structure of the code more and to see how Mockito is being used. The example also uses the method *verify()* which should now be familiar to the reader. This is important because this section focuses more on the theory behind the Mockito API and establishes the terminology used when talking about unit testing in general.

The test example illustrated in Figure 10 below, uses a method from the Mockito framework called *verify()*. As mentioned in section 1.4, *verify()* can be used whenever a verification of some sort needs to be done, however *verify()* is often used to check that a certain method call invokes the right method in another class. Dependencies between classes and method calls between them are verified. This type of testing is called integration testing. During such a test, the interactions between modules are tested. A module can be different things depending on the nature of the test. In this case, a module is just a java class. More specifically, it is *ClassA.java*.

This class is dependent of *ClassB.java* in such a way that a the only method found in *ClassA*, *methodA()* delegates the work to the one method in *ClassB* called *methodB*. *ClassA* is essentially a facade over *ClassB*, much like the case of *DAOFacade.java* and *Logic.java* in section 1.4. The example here, just as before, verifies that the method in *ClassB* was actually called from the method in *ClassA*.

In order to make such a test, dependencies between the two classes need to be set up. This is where Mockito is used. With Mockito, the dependencies can be mocked by creating what is called a mocked object of a class for which the tested class is dependent upon. In the example, *ClassA* is dependent on *ClassB* because its method invokes a method in *ClassB*. This is why *ClassB* is mocked. It is recommended that the comments in the code is read and understood (Figure 10).

```java
public class ClassATest
{
    /*
     * Mock out dependency to ClassB using Mockito. If static imports are used,
     * it is enough with just typing mock instead of Mockito.mock.
     */
    ClassB b = mock(ClassB.class);

    /*
     * Create object of ClassA that receives an object of classB. In this case,
     * a mocked object.
     */
    ClassA a = new ClassA(b);

    /*
     * Test run of methodA, of ClassA. Test verifies different things about
     * methodB in classB. This is done by mocking dependencies to ClassB.
     */
    @Test
    public void testMethodA1() throws Exception
    {
        // Call methodA in ClassA.
        a.methodA("anyString");

        // Verify that methodB was actually invoked with the correct
        // parameter when methodA was called.
        verify(b).methodB("anyString");
    }
}
```

Figure 10 – Complete source code for test class.

The test class above tests the only method found in the SUT. SUT stands for System under test and it is an abbreviation used to refer to whatever is being tested. As a consequence, an SUT can mean different things depending on the test case. In this case however, the SUT refers to the java class that is being tested, namely *ClassA*. Furthermore, in section 1.4 the SUT would be *DAOFacade*.

The method in *ClassB* does nothing. The SUT, *ClassA*, has a constructor that receives an object of *ClassB* which is used to access the method found in *ClassB*. In the test class, instead of an object of the real *ClassB*, a mocked object is sent to *ClassA* using Mockito in order to mock the dependencies between *ClassA* and *ClassB*. It is important to realize that unit tests are run in a separate container to avoid contamination of the project. This way, the behavior of the project is in no danger of being affected by the tests.

Figure 11 illustrates the constructor found in the SUT, **(a)** and how this is used by the test class to mock out the dependencies in the test container, **(b)**.

```
/**
 * Constructor of ClassA receives
 * an object of ClassB
 * @param b
 */
ClassA(ClassB b)
{
    this.b = b;
}
```

(a)

```
/*
 * Mock out dependency to ClassB
 * using Mockito. If static imports
 * are used, it is enough with just
 * typing mock instead of
 * Mockito.mock.
 */
ClassB b = mock(ClassB.class);
/*
 * Create object of ClassA that
 * receives an object of classB.
 * in this case a mocked object.
 */
ClassA a = new ClassA(b);
```

(b)

Figure 11 – (a) shows the constructor in the SUT. (b) Shows the test class using the constructor to pass a mocked object to the SUT.

In order for the test to run, dependencies to *ClassB* need to be mocked first. There are several ways to do this. One way is shown in Figure 12.

```
ClassB b = mock(ClassB.class);
```

Figure 12 – One way to mock when static imports are used. If not, the Mockito class must be explicitly referenced.

Another way to mock is to use the Mockito annotation for mocking *@Mock* (Figure 13).

```
@Mock
private ClassB b;
```

Figure 13 – One way to mock using annotations.

To enable Mockito specific annotations, the code must specify how to run the test class. This is done using the JUnit annotation *@RunWith* (Figure 14). The annotation is put on top of the class definition.

```
@RunWith(MockitoJUnitRunner.class)
public class DAOFacadeTest
{
```

Figure 14 – JUnit Annotation *@RunWith* to enable Mockito annotations.

Please observe that no expectation was set up before the test. This is done after the fact and it is a feature unique to Mockito. Usually, an expectation is set up before the method is called, where the expected outcome is specified.

This structure is known as the expect-run-verify pattern and it is one that does not need to be followed when using Mockito, making the test more intuitive. In the case above, the test only verifies that *methodB* was invoked with the correct parameter, *"anyString"*, when *methodA* was called. This was the expected outcome of the test and Mockito offers many options to tailor the verification.

For example, the test can check that a specific method was invoked a specified number of times. Using the same example classes, such verification would look like this (Figure 15).

```
verify(b, Mockito.times(1)).methodB("anyString");
```

Figure 15 – Verify that *methodB* was invoked exactly one time with the parameter "*anyString*".

Several verifications can be made in the test method by simply typing verify again with new logic, as seen in Figure 16.

```
verify(b).methodB("anyString");
verify(b, Mockito.times(1)).methodB("anyString");
```

Figure 16 – Several verifications in succession.

A test class can have more than one test method. This is accomplished by adding the JUnit annotation *@Test* right above the intended test method. This is illustrated by Figure 17.

```
@Test
public void testMethodA1() throws Exception
{
    // Call methodA in ClassA.
    a.methodA("anyString");

    // Verify that methodB was actually invoked with the correct
    // parameter when methodA was called.
    verify(b).methodB("anyString");
}

@Test
public void testMethodA2 ()throws Exception
{
    // Call methodA in ClassA.
    a.methodA("anyString");

    // Verify that methodB was actually invoked with the correct
    // parameter when methodA was called.
    verify(b, Mockito.times(1)).methodB("anyString");
}
```

Figure 17 – Several test methods using the JUnit annotation @Test.

The way of passing a mocked object as illustrated in Figure 11 can become problematic when the code base becomes more complex and data structures such as EJB's are used. In some cases, like in Java EE Web projects with the MVC-model-structure, constructors for passing class objects may not be a viable option due to the need of encapsulating and having low coupling between the different layers. To solve this issue, an injection point that sets the EJB to be the mocked object can be used to pass it to the SUT when running a test.

This means altering the SUT by adding this ability. In the context of testing, this violates the rule of not altering the SUT just for the sake of testing it. However, by adding this ability to the SUT, neither its logic nor the overall structure is changed. It is therefore, an acceptable solution to the problem and it is how it is done in *DAOFacadeTest* in section 1.4. Figure 18 shows such a case, where dependencies between two classes at different layers of the system are set up using EJB's and not by passing objects of classes using constructors.

```
/**
 * JSF Managed bean used to check and store the user identity.
 *
 * @author admin
 */
@ManagedBean
@SessionScoped
public class AuthenticationBean implements Serializable {

    @EJB
    private DAOFacade daof;
```

Figure 18 – *AuthenticationBean.java* in the upper layer of the project has access to methods in *DAOFacade* in the next, lower layer.

In this case, a class called *AuthenticationBean.java*, located in the upper layer of the Java EE Web project, passes all methods to the lower layers of the system via the facade, *DAOFacade*. *DAOFacade* is the controller and it is located one layer down, in the source package *controller*. *AuthenticationBean* does not see the lower layers. Instead, it sees only a facade over it which it interacts with, much like what the SUT *ClassA* does with *ClassB*. The difference here is that, instead of having a constructor in *AuthenticationBean* that receives an object of the facade, the facade is set as an EJB in *AuthenticationBean* with the *@EJB* annotation (Figure 18 again). This yields better encapsulation of the code layers and lower coupling between them, some of the goals with the MVC-model.

Now, a test is created under these circumstances. The SUT is now *AuthenticationBean* and the method in it that will be tested is called *login()*. This method passes login information to the lower layers by calling a method in *DAOFacade* and not any methods directly from the lower layers. Those methods are hidden from *AuthenticationBean*. Since *DAOFacade* is an EJB in *AuthenticationBean* this can be done, even though *AuthenticationBean* does not have constructor for *DAOFacade* as in the case of *ClassA* in Figure 11 - (a). Figure 19 illustrates the portion of the code from the method that shows how the work is delegated to *DAOFacade*.

```
public String login()
{
    if (daof.login(username, password) == 0)
    {
```

Figure 19 – login method in the new SUT, *AuthenticationBean.java*, passing on to *login()* in *DAOFacade.java*.

The test verifies that the method *login()* in *DAOFacade* was actually called exactly once with two specific strings a parameters. This is shown in Figure 20.

```
/*
 * Tests the method used when admins try to login.
 */
@Test
public void TestAdminLogin()
{
    authBean.login();
    verify(mockedDAOFacade, times(1)).login("Terminator",
                                            "c00lk1ll3rb0y#96");
}
```

Figure 20 – The test for the SUT.

The depending class, *DAOFacade*, is mocked as usual in the test class (Figure 21).

```
@Before
public void setUp()
{
    // Mock the class DAOFacade
    mockedDAOFacade = mock(DAOFacade.class);

    // Pass the mocked object to the SUT.
    authBean.setDAOFacade(mockedDAOFacade);
```

Figure 21 – *DAOFacade.java* mocked as in a previous test example (Figure 6).

As mentioned before, the difference lies in the SUT. In order for the test to pass, the SUT must be able to receive a mocked object somehow. For this reason, an injection point is added to the SUT that explicitly sets an object of type *DAOFacade* equal to the EJB for *DAOFacade* in the SUT (Figure 22).

```
public void setDAOFacade(final DAOFacade daof) {
    this.daof = daof;
}
```

Figure 22 – The added code in the SUT.

In the test class, after mocking *DAOFacade*, the mocked object is set to be equal to the EJB in the SUT through this injection point. This way, the SUT uses the mocked object instead of the EJB. If this injection point is not present, the SUT tries to pass on to the method in the real *DAOFacade* instead of using the mocked object. This results in a null pointer exception since the EJB is not defined within the context of the test. The dependencies have not been passed to the SUT, even though *DAOFacade* is mocked.

This example showed how to mock dependencies to EJB's. Mockito offer more ways of mocking other types of dependencies such as different types of contexts but examples on how to do this is not covered in this tutorial.

End of tutorial.

APPENDIX B: SELENIUM FRAMEWORK TUTORIAL

This appendix is a tutorial about implementing the Selenium framework for black-box testing at user level. It also contains code examples on how to use Selenium.

The goal with this tutorial is to show how to implement and use the Selenium testing framework. Selenium is a black-box testing framework that focuses on testing the web-based user interface of a system without the need of learning a scripting language. It accomplishes this in different ways and some of these are brought up in this tutorial.

Before using this tutorial, it is assumed that NetBeans 7.0.1 or above has been installed together with Mozilla Firefox web browser 26.0 or above. The user should have access to the Java EE Web project, The Recruitment System.

This tutorial has only been tested on a PC running Windows 7. It has not been verified to work on other operating systems but there should not be any major differences since this tutorial focuses on NetBeans and Mozilla Firefox.

B.1 Downloading the necessary files

Visit the Selenium official website at: http://docs.seleniumhq.org/download/
Download and install the latest Selenium IDE release, called selenium-ide-x.y.z.xpi. A link to a direct download and install is available in the main page. Firefox will prompt to restart. Do so. When downloading the plug-in, Firefox may ask to allow installing of third party software as shown in Figure 1. Click *accept*.

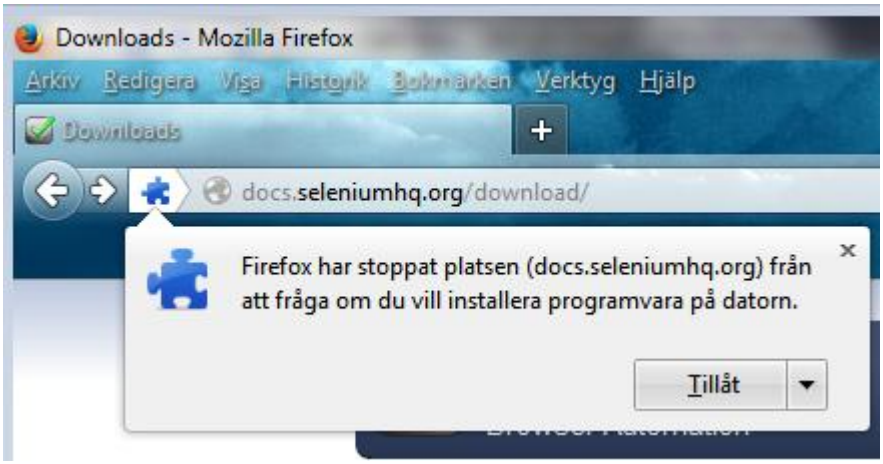

Figure 1 – The pop up that needs to be confirmed for the Selenium plug-in to be installed.

Also, download the zip file for the Java language bindings found under *Selenium Client & WebDriver Language Bindings* further down on the same download page (Figure). The file is called selenium-java-w.xy.z.zip. Save the zip-file to a location of your choice, preferably on the desktop.

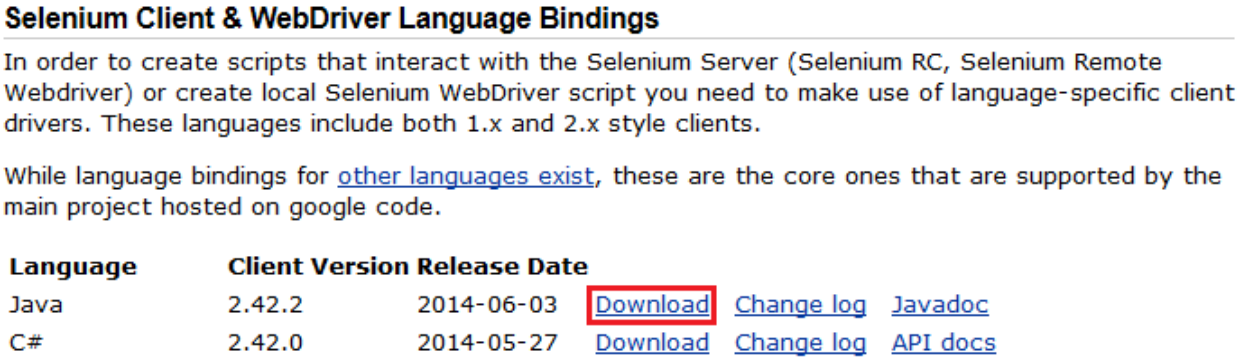

Figure 2 – Download ink to Java language bindings zip file.

Extract the contents of the zip file.

The Selenium IDE plug-in is for Firefox and it will be installed and used within the Firefox container. The zip file contains several files. Among them two jar folders which will be imported into the web project in NetBeans IDE later on. These will give you access to the Selenium testing framework in NetBeans.

B.2 Implementing Selenium to Firefox and NetBeans

There are different ways to test your Java EE web project interface with Selenium. We have chosen to focus on three of them. Only one of these methods described below should be followed to avoid confusion.

The first method is to use Selenium IDE through the Firefox plug-in to record or script your own interactions with the website. The recording or script can be run in the plug-in directly and this will constitute your test. To implement this plug-in, simply download the Selenium IDE and allow Selenium to install the plug-in to the Firefox browser. Once the Firefox plug-in is downloaded, it is installed and you are prompted to restart the browser. Once restarted, it has access to Selenium. These steps are done in activity B.1 - *Downloading the necessary files*. We will come back to the recording tool in activity B.3.

The second method is to use the Selenium framework through NetBeans IDE by exporting a recording or written script as Java code and run the test in NetBeans instead of the Firefox plug-in. This method is explained from activity B.2.2.

The third method is to implement a Selenium test solely in NetBeans. With this alternative, you have the ability to code your own test case and not depend on the recording tool provided through Selenium IDE. This is explained in activity B.2.1. All three courses of action in this tutorial are presented in a bullet list below. They are also illustrated in Figure 3.

- If you want to record and play back a test solely using the Selenium IDE plug-in and not worry about manually coding the test, please jump to activity B.3 Recording with Selenium IDE plug-in

- If you want to export a recording to Java code in order to execute the test in NetBeans IDE, please jump to activity B.2.2 Exporting recording to Netbeans IDE.

- If you want to manually code your own Selenium test solely in Java, please jump to section B.2.1 Creating test through Netbeans IDE.

Figure 3 illustrates the different paths that you can take when implementing Selenium with this tutorial. The boxes describe the methods and the circles are the different chapters, called activities, to follow for each of the three methods. Each path is also color-coded so that they are easier follow. For example, at the end of activity B.3 – *Recording with Selenium IDE plug-in* you can either run the recording in the Selenium IDE plug-in or export it to Java code and execute it in NetBeans IDE. This depends on which method you choose and it is illustrated by the two differently colored lines coming out of the activity circle B.3. It is emphasized once again that only one method should be followed to avoid confusion.

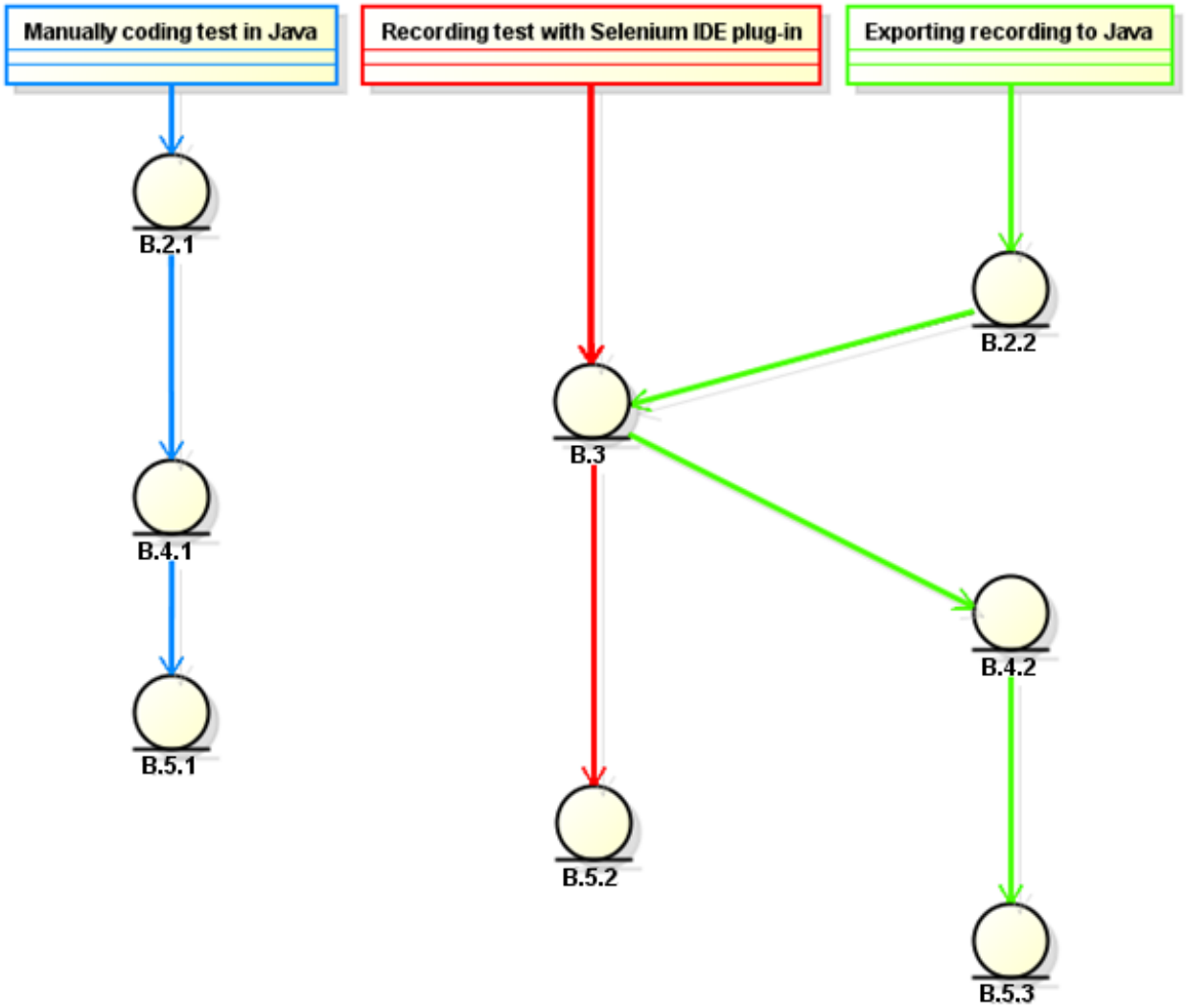

Figure 3 – Possible implementation paths in this tutorial.

B.2.1 Creating test through Netbeans IDE

This activity explains how to import the Selenium framework into the NetBeans IDE in order to be able to run manually written tests. It also shows how to create a new Java project for the test case. The main issue here is that a java class with a main method must be coded manually but more on this later. Here, it is only explained how to enable Selenium as a testing tool in a new Java project. Follow the directions as described below.

Start NetBeans.

Click on *File* and then click on *New Project…*
Choose the category *Java* on the left hand side under *Categories* and choose project *Java Application* (Figure 4).

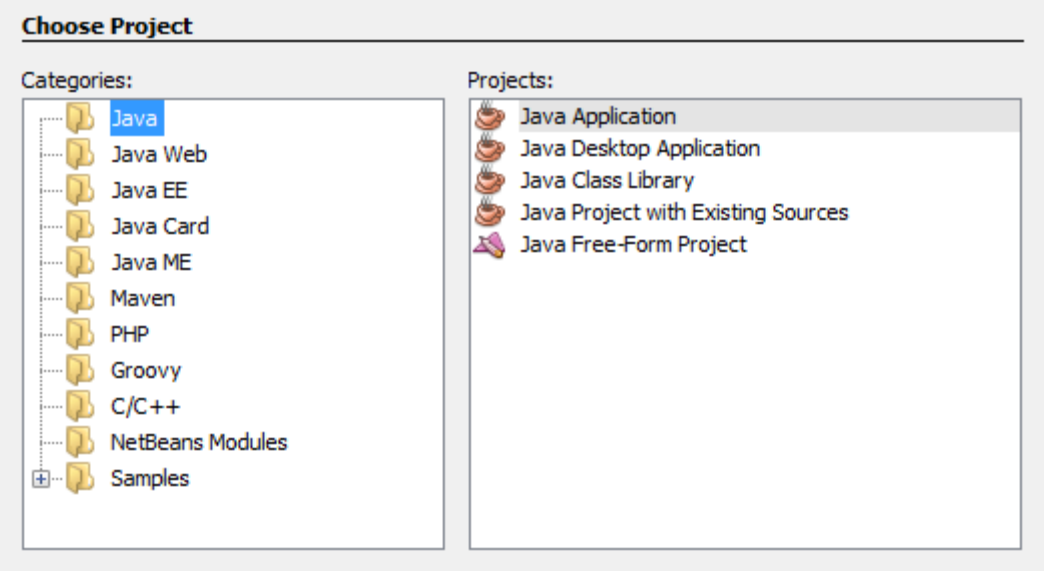

Figure 4 – Creating a new Java Application.

Click *Next* >.

Call your new Java project *"seleniumFirstMethod"* and click *Finish*.

Netbeans IDE will now show your new Java project with a Java class, containing a main method. To access your newly created Java project go to your *Projects*-tree usually located on the left hand side of NetBeans IDE.

Once the java project has been located, right-click on the folder *Libraries* found in the newly created Java Application (Figure 5) and choose "*Add Jar/Folder"* (also shown in Figure 5).

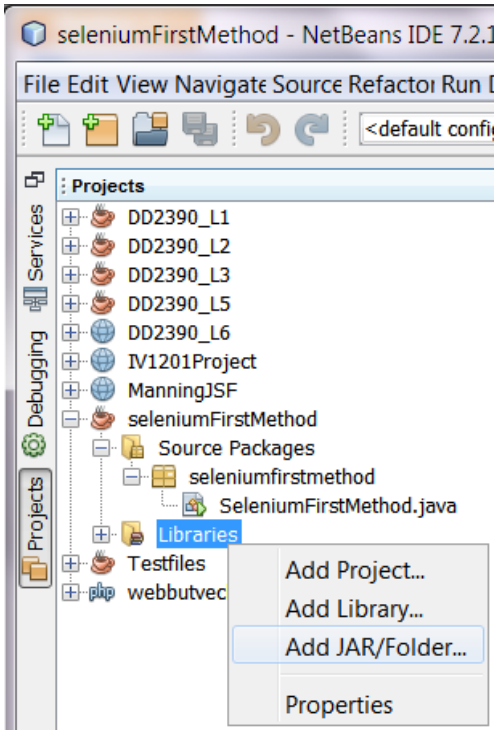

Figure 5 – The new Java Application.

Navigate on your PC to the folder which contains the extracted files from the Selenium library zip file. Choose a file by left-clicking it and then click *Open*. Repeat this process for each file to be added. The names of the files that should be imported are listed below:

- *selenium-java-w.xy.z-srcs.jar*
- *selenium-java-w.xy.z.jar*
- All jar files in the subfolder called *libs*. Do this quickly by left-clicking on the first jar file and then press *Ctrl +A* to select all of them.

If imported correctly, the *libraries* folder should look something like this (Figure 6):

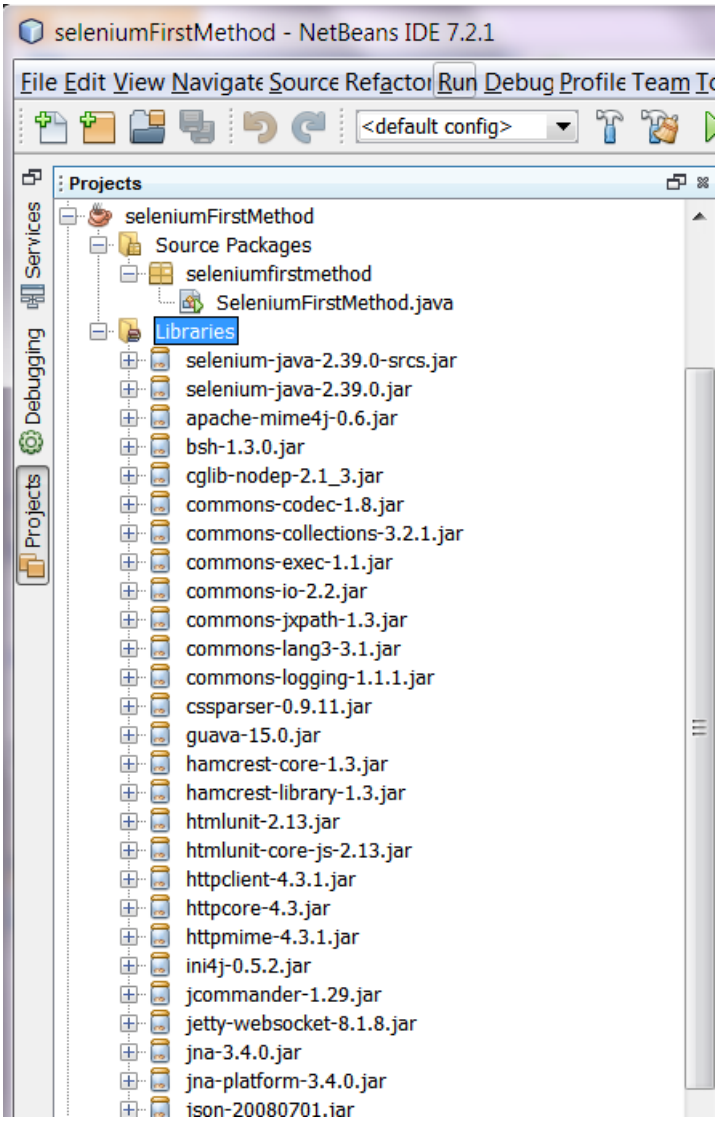

Figure 6 – The jar files for Selenium added under libraries

You are now able to use Selenium as a testing tool through the Netbeans IDE. You can now jump to section B.4.1 Guidelines for a manually coded test.

B.2.2 Exporting recording to Netbeans IDE

In this activity, a new Java project is created. Here, the Selenium framework is imported as a testing framework into NetBeans IDE. Also, a java class with a main method is not needed when creating a new Java project. This way of importing the Selenium Framework as a testing framework is required in order to later be able to export a recording and run it in NetBeans IDE.

To import the Selenium library into NetBeans IDE, follow the directions as described below.

Start NetBeans.

Click on *File* and then click on *New Project…*

Choose the category *Java* on the left hand side under *Categories* and choose project *Java Application* (Figure 7).

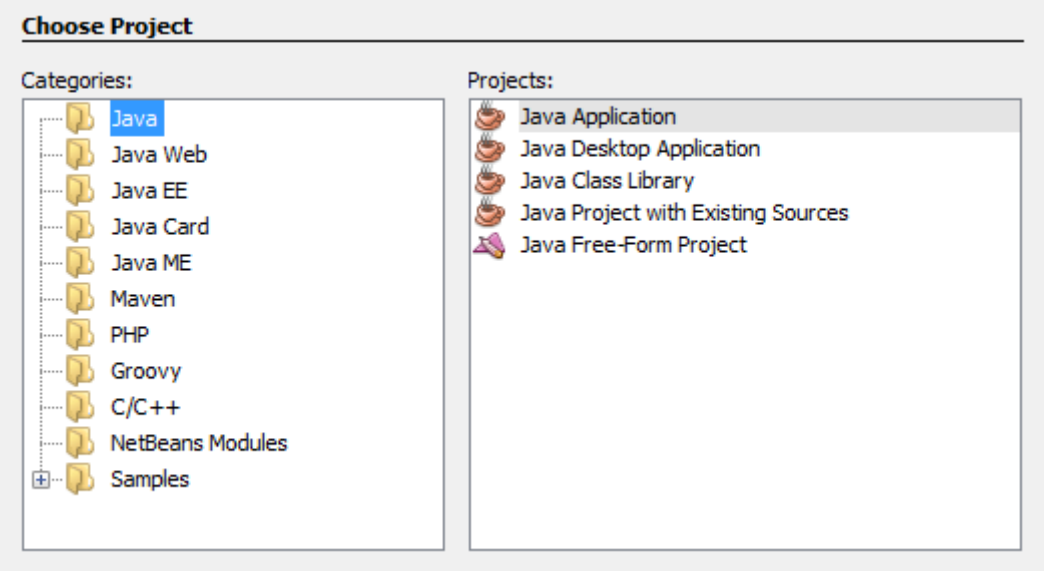

Figure 7 - Creating a new Java Application.

Click *Next >*.

Name your project *"seleniumSecondMethod"*

Since a main class is not needed, please uncheck the option for it as shown in Figure 8 below.

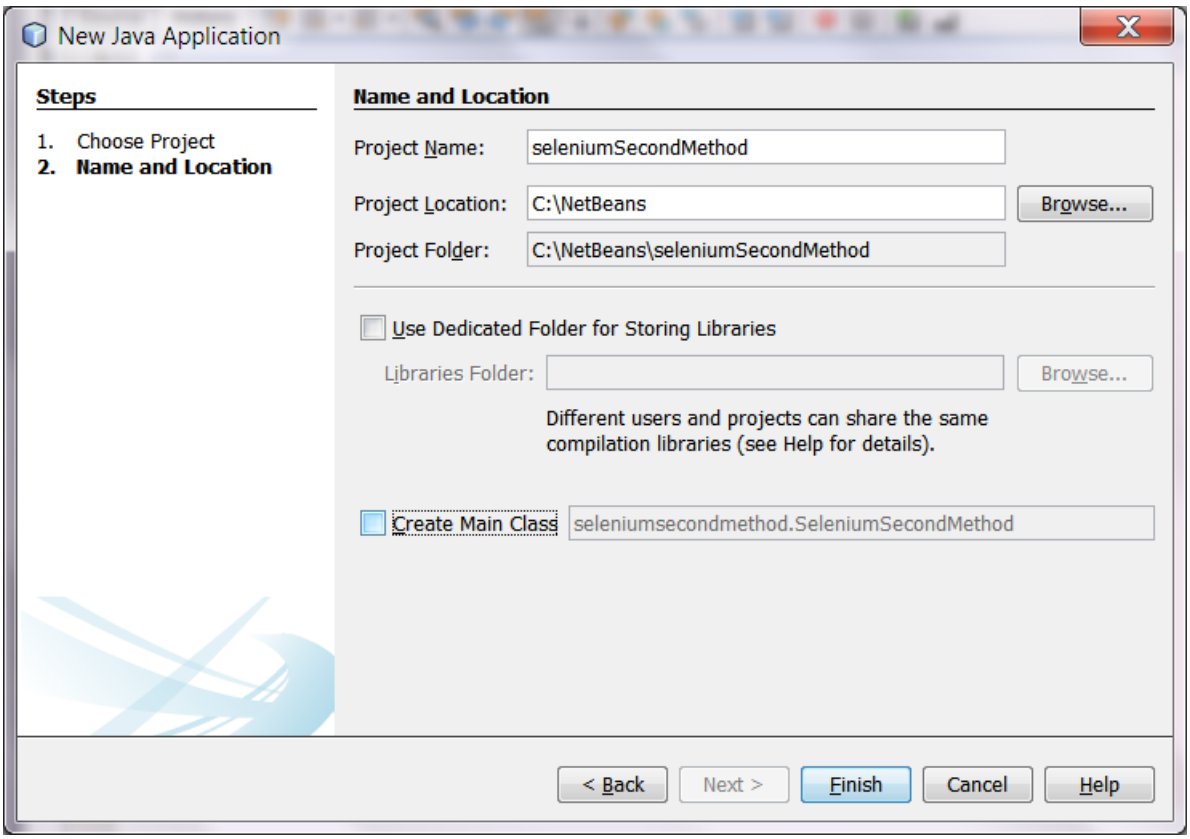

Figure 8 – uncheck the checkbox called *Create Main Class*.

Click *Finish*.

Netbeans IDE will now show your new Java project tree. Since there is no main class, the source packages is empty.

Now, it is time to create a unit test class in order to import the Selenium framework as a testing framework. Right-click on your Java project, at the root of the project tree and navigate to *New* and then click *Other…*

Choose the category *Unit Tests* on the left hand side under *Categories* and choose the file type *JUnit Test* and click *Next >*. On earlier Netbeans builds the category name is called *JUnit.*

Call your new unit test class *"SeleniumUnitTest"* and click *Finish*. Notice the warning displayed that you should not add your java classes in the default package. It is good practice to follow this advice. However, not doing so will not affect the examples in this tutorial (Figure 9).

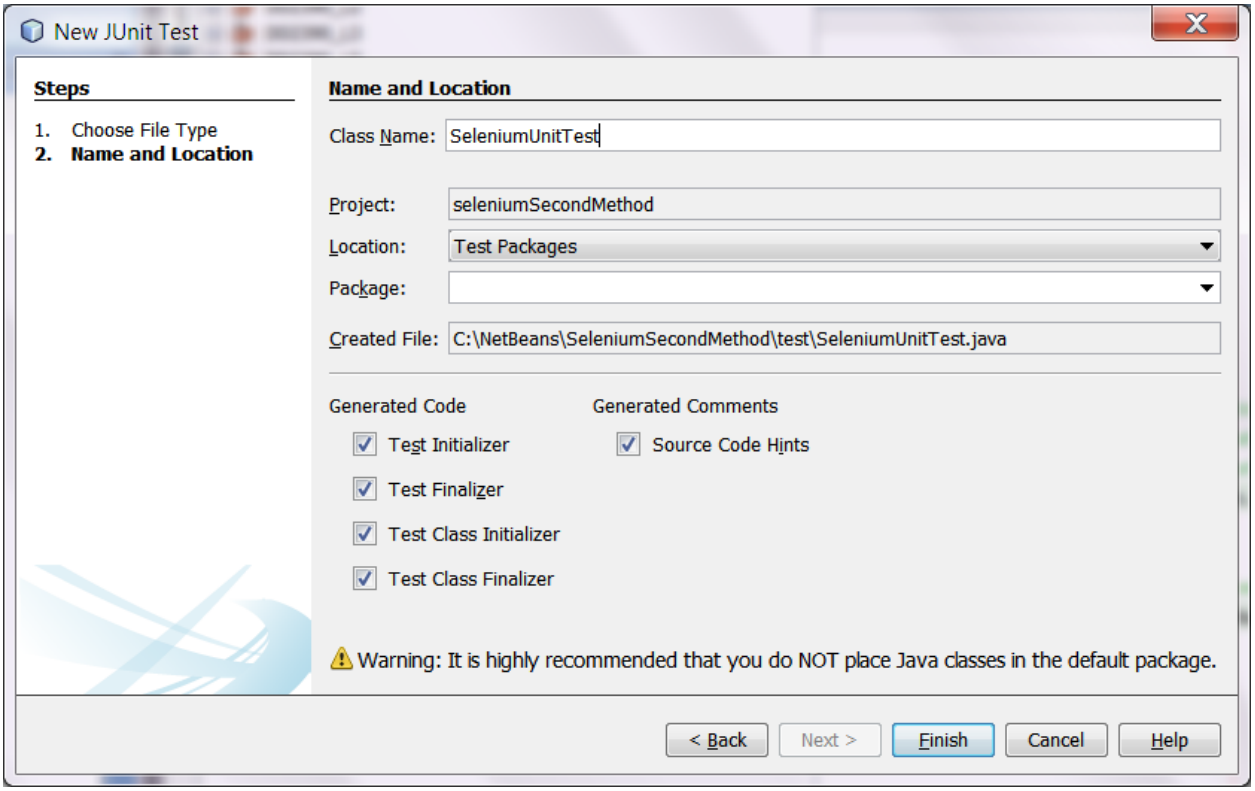

Figure 9 – Creating a JUnit test class. Ignoring the warning will not affect the tutorial.

If more than one version of the JUnit testing framework is available, NetBeans will ask which one to use when creating the unit test class (Figure 10). For this tutorial JUnit 4.x was used. It has not been tested with earlier versions of JUnit.

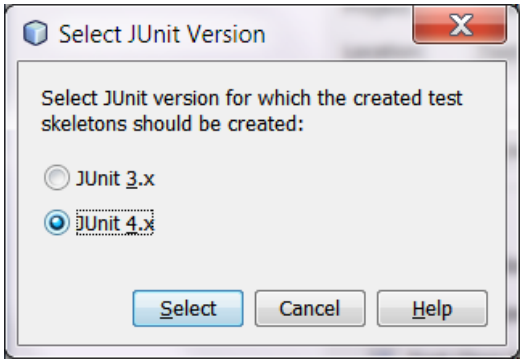

Figure 10 – Selecting JUnit version. Window will only show if more than one version is available.

Once the java project has been located under *Projects*, right-click on the folder *Test Libraries* located in the newly created Java Project (Figure 11). Notice that the *Source packages*-folder is empty and a new Java folder called *Test packages* has appeared. This folder was created when a new Unit test class for the JUnit testing framework was created (Figure 11).

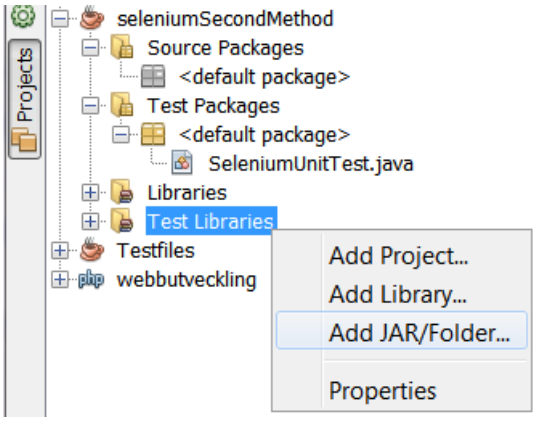

Figure 11 - The new Java Application.

Click *Add Jar/Folder* (also shown in Figure 11) and then navigate on your PC to the folder which contains the extracted files from the Selenium library zip file. Choose a file and click Open. Repeat this process for each file to be added. The names of the files that should be imported are listed below:

- *selenium-java-w.xy.z-srcs.jar*
- *selenium-java-w.xy.z.jar*
- All jar files in the subfolder called *libs*. Do this quickly by left-clicking on the first jar file and then press *Ctrl +A* to select all of them.

If imported correctly, the *Test Libraries* folder should look something like this (Figure 12):

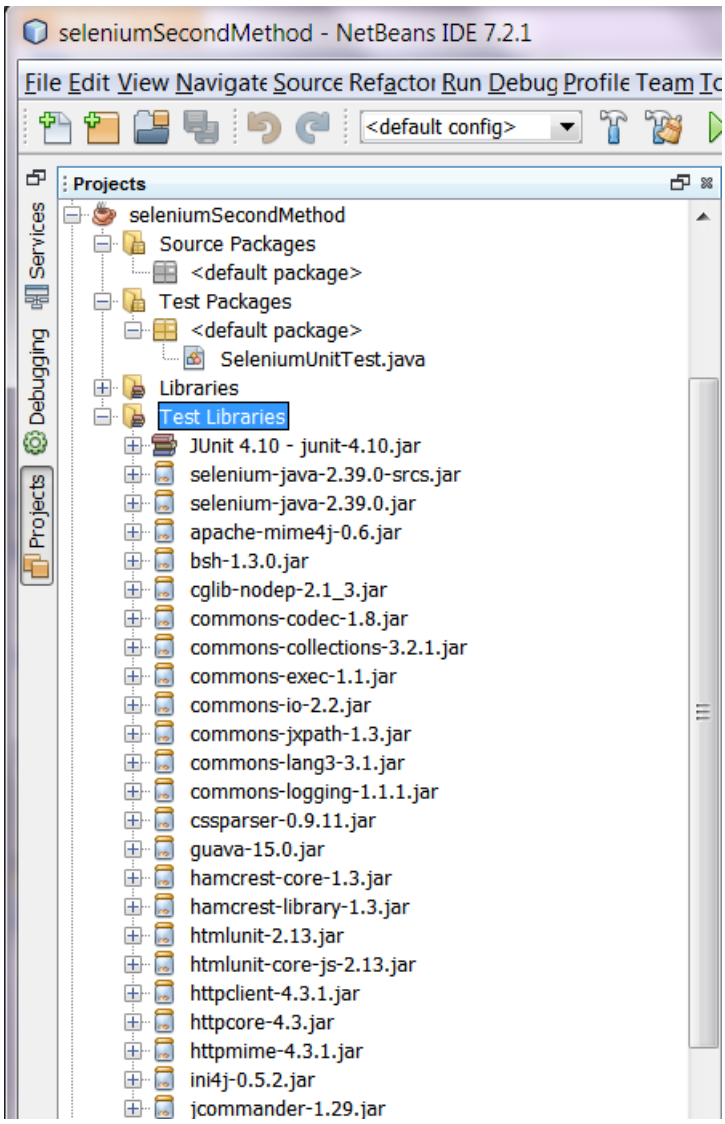

Figure 12 – The jar files for Selenium added under libraries

You are now able to use Selenium as a testing tool. Please continue to section B.3 - Recording with Selenium IDE plug-in.

B.3 Recording with Selenium IDE plug-in

In this activity, it is shown with examples how to do a black-box test with Selenium. Make sure to deploy your Java EE Web project through NetBeans IDE to be able to access the website of the project.

Open up the Mozilla Firefox browser and go to the website of your Java EE Web project, usually found at *http://localhost:8080/Projectname/* where "*Projectname*" is the name of your Java EE Web project.

Click on the Selenium IDE plug-in icon on the top right corner the Firefox browser window (Figure 13).

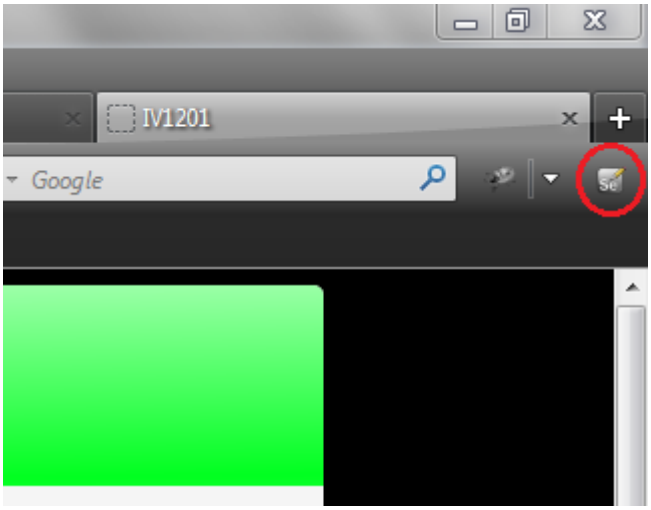

Figure 13 – The Selenium IDE icon.

The code examples are based off of the Java EE Web project interface that is included in the source files. The web interface is built with JSF pages and can be found under the source folder *Web Pages*. As mentioned earlier, this tutorial will focus on two ways to test with Selenium, through Mozilla Firefox and NetBeans IDE.

When clicking on the Selenium IDE plug-in icon the program is executed and new window is presented in a new Firefox browser instance (Figure 14). Here, it is possible to manually enter a variety of commands or record an interaction with the website.

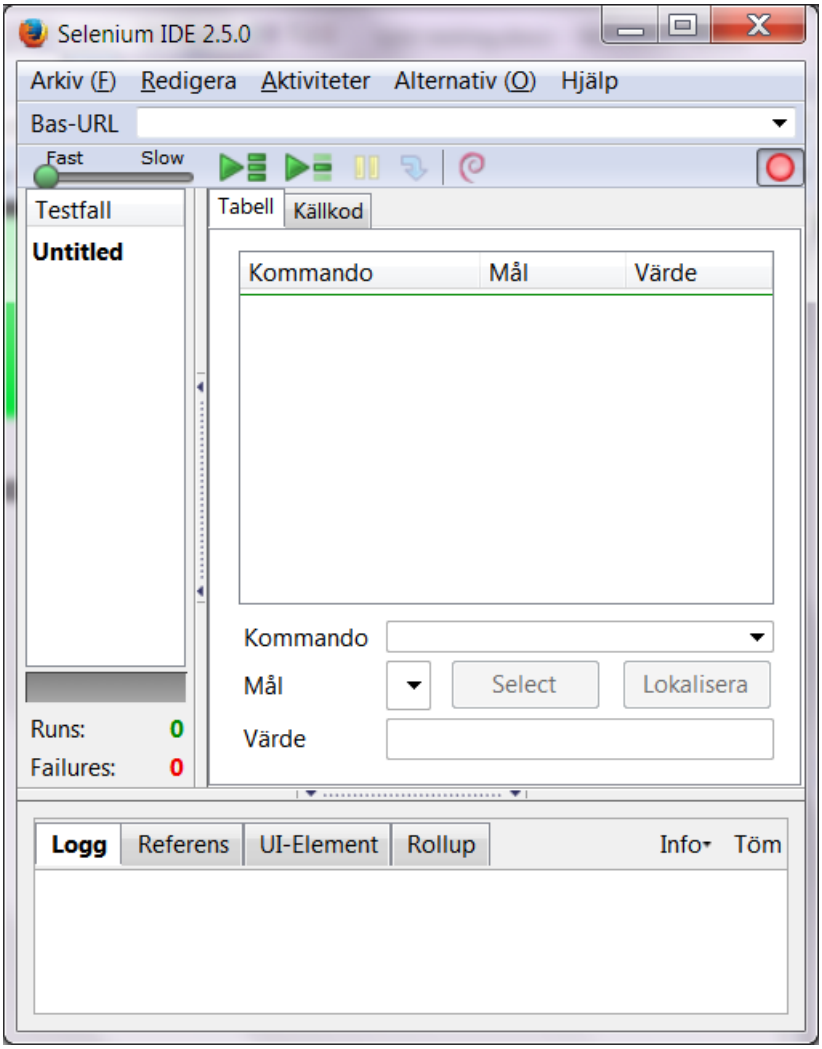

Figure 14 – Selenium IDE window through Firefox. Notice that recording is on (red circle, top right corner).

For this tutorial, a number of interactions with the project website are recorded. Firstly, the language of the website is changed from English to Swedish. Secondly, a login to the admin window is recorded by clicking on the admin link in the menu to the left. Once at the admin window, a username and password is entered. The recording ends once *Login* has been clicked.

A recording is started by clicking on the red circle on the top right corner on the Selenium IDE window. This activates a new recording (Figure 15).

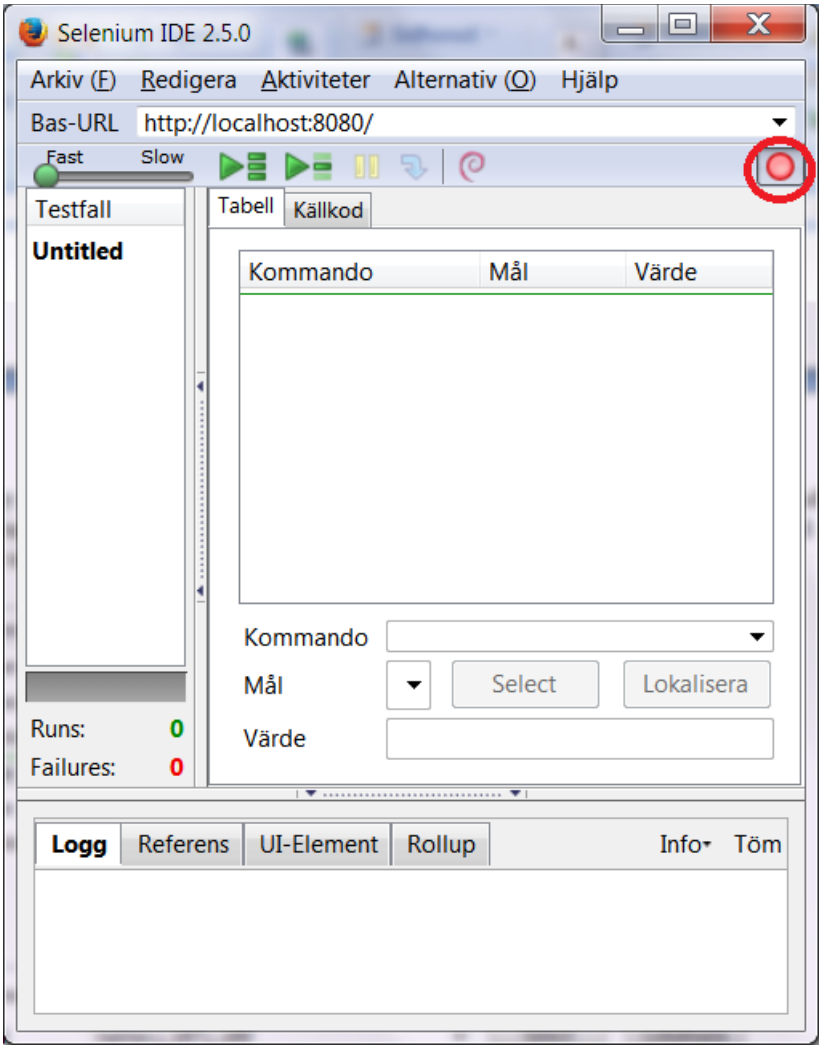

Figure 15 – Start recording

Anything that is clicked on the website from now on is recorded and entered to the script. After the interactions described above are made, the recording is stopped by clicking on the red circle.

Figure 16 shows the commands after recording the interactions.

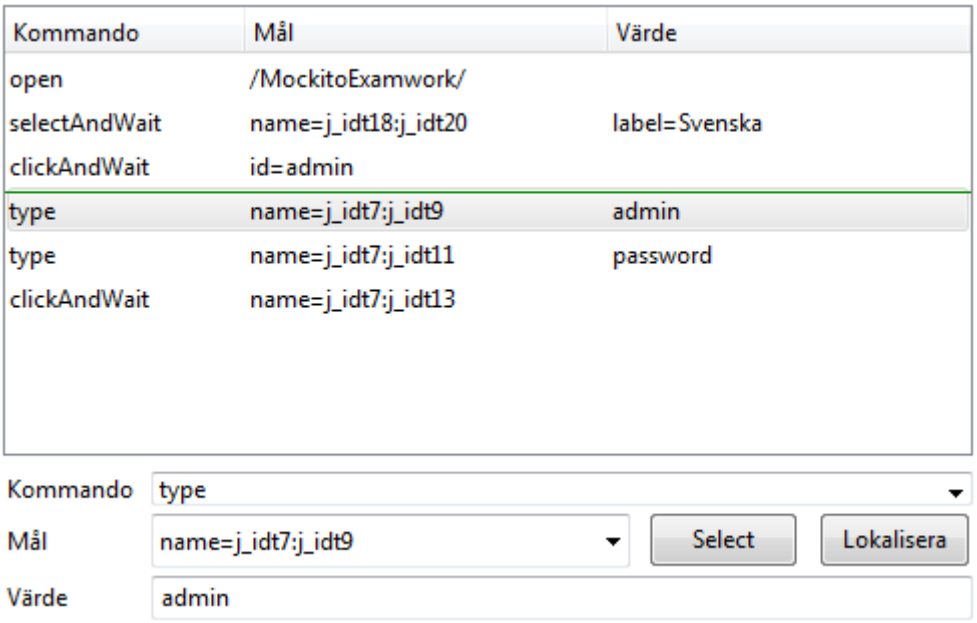

Figure 16 – The result from recording the interactions with the website.

The recording can be played back through the plug-in directly or exported as Java code to be executed in NetBeans IDE. To run the recording in the Selenium IDE plug-in jump to activity B.5.2 – *Executing recording in Selenium IDE (red-colored path in Figure 3).* If you wish to export the recording to Java code in order to execute in NetBeans instead, continue to activity *B.4.2 – Implementing a test through NetBeans IDE (green-colored path in Figure 3).*

B.4 Implementing a test through Netbeans IDE

In this activity, it is explained how to create a Selenium test solely with Netbeans. It is also explained how to export a recording done in the Selenium IDE plug-in for Firefox in order to run the test in NetBeans IDE.

There is more than one way to execute a Selenium black-box test with NetBeans IDE. This example will focus on two methods. Firstly, by manually coding a test case and secondly, by exporting the code we recorded using the Selenium IDE Plug-in through Firefox (shown in activity B.3).

B.4.1 Guidelines for a manually coded test

This activity shows how to manually code a test case with the Selenium framework, keeping in mind that any illustrations of code are just pseudo code. The reason for not sharing the full code base is because is it very specific to the website and will not be applicable to other Java EE web project interfaces. Instead, see these illustrations as guidelines on how to go about when creating a test. The code does the same procedures as during the recording. It is divided into parts in order to be explained in detail.

In the following examples, Selenium works by referring to an element in the HTML code through either the element name or ID. It then simulates an action on this element. In order to code a test, the element names or IDs must be known. To find an element name or ID, Firefox has an option called "*Inspect element"*. This option will show the corresponding HTML code for an element on a given website. Once the HTML code is shown, the ID or name of the element can be extracted.

To use this option, it is as simple as right-clicking on a specific object on the website and left-clicking on *"Inspect element"* (Figure 17).

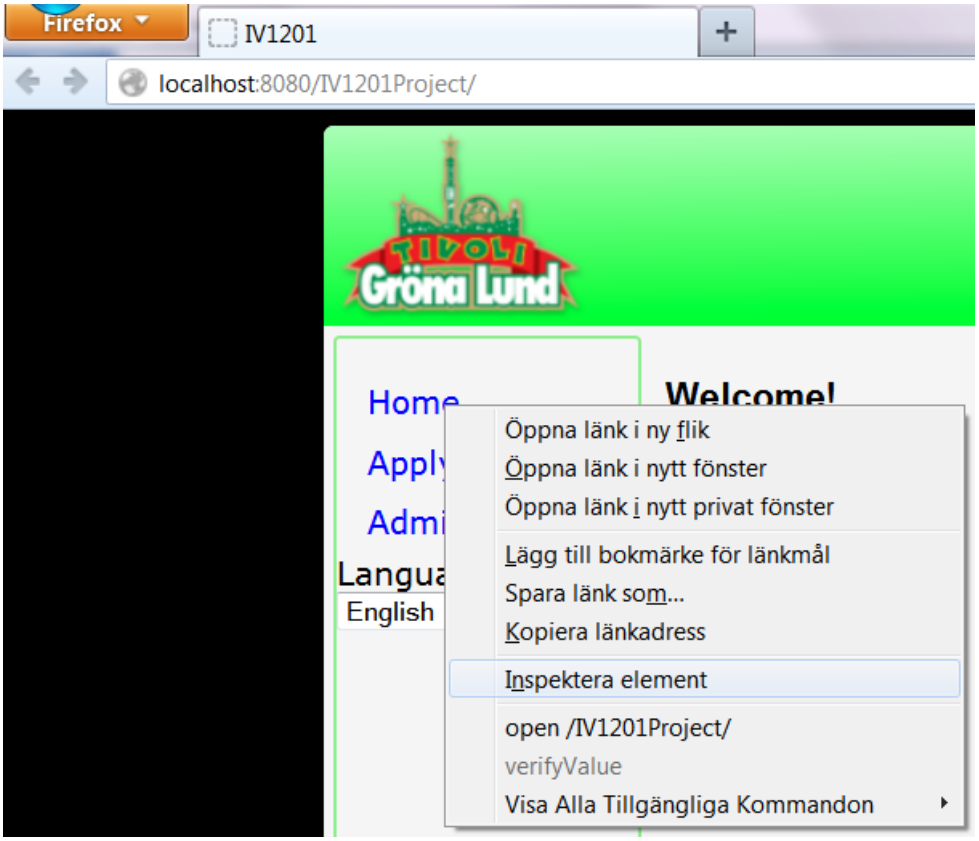

Figure 17 – right-clicking on *Home* to find its element name or ID by inspecting it.

Upon inspecting an element, a new window will appear at the bottom showing the corresponding HTML code for the chosen element. It is here where the element name or ID is found (Figure 18).

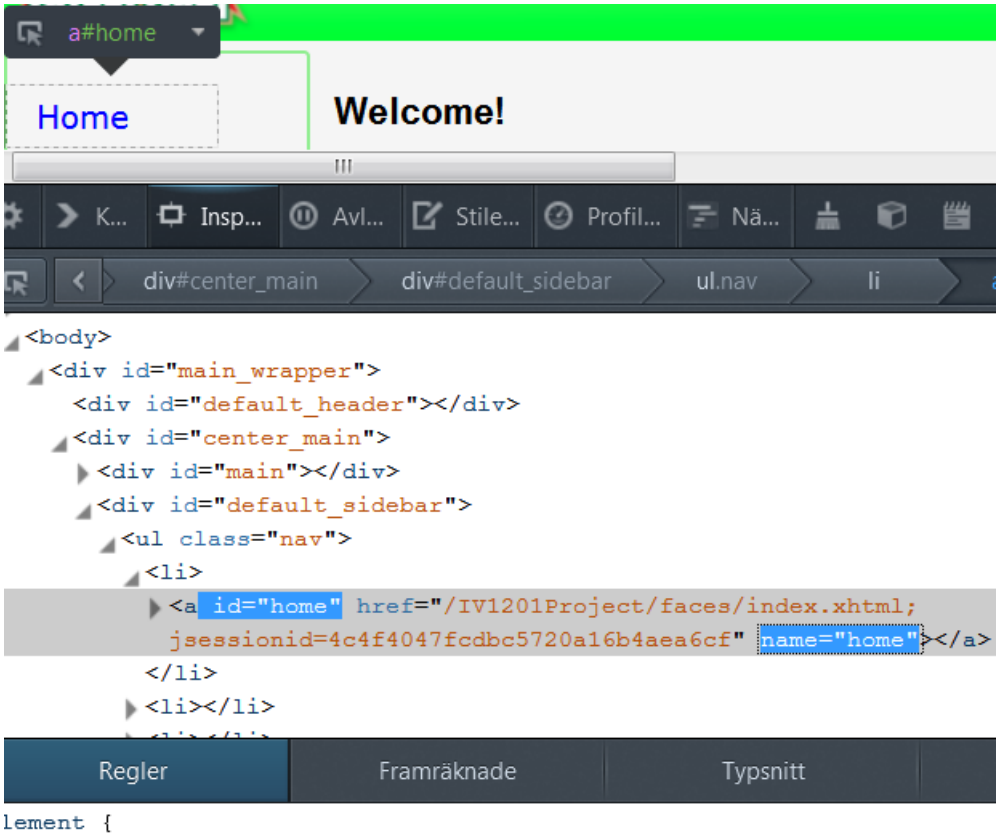

Figure 18 – Inspecting element *Home*. Its ID is *"home"* and its name is *"home"*.

Knowing how to find the names and IDs we can proceed with examples. The following pseudo code in Figure 19 is the initialization of the HtmlUnitDriver. This driver helps with automated testing of the website. The following method, *get*, loads the website that is submitted for testing, in this case, the web project website.

```
public class X {
    public static void main(String[] args) {
        // Create a new instance of the html unit driver.
        WebDriver driver = new HtmlUnitDriver();

        // Open up the website of the project.
        driver.get("https://localhost:8181/MockitoExamwork");
```

Figure 19 – Initialize the driver & load the website.

The pseudo code in Figure 20 is an example on how to interact with the website. In the first method, the focus is set to the menu bar with the different language options. The method will simulate a click on the menu bar and then select the option labelled "*Svenska*". This will change the language from English to Swedish.

```
// Find the element of the dropdown list and change the language to
// swedish, by choosing the text "Svenska".
new Select(driver.findElement(By.name("j_idt14:j_idt16")))
        .selectByVisibleText("Svenska");

// Find the element to the link and click on it.
WebElement adminLink = driver.findElement(By.id("admin"));
adminLink.click();

// Click on the username field and enter "admin" as username
WebElement usrname = driver.findElement(By.name("j_idt7:j_idt9"));
usrname.sendKeys("admin");

// Click on the password field and enter "password" as password
WebElement pword = driver.findElement(By.name("j_idt7:j_idt11"));
pword.sendKeys("password");

// Find the element for the submit button and click it.
WebElement submitButton = driver.findElement(By.name("j_idt7:j_idt13"));
submitButton.submit();
```

Figure 20 – Simulated interaction of the interface but in NetBeans.

The second method in Figure will find the element in the HTML code for which ID is set to "*admin*". This ID is found in the JSF-page called *masterLayout.xhtml*. The JSF-page is found in the source code folder *Web pages* for the Web project. The method will simulate a click on said element once found. In other words, this method simulates the interaction of entering the admin window by clicking on *Admin* in the menu to the left of the website.

The third method shown in Figure has the same procedure as the method above it but describes a different interaction. It will find an element, but in this case it will be found through the name of this element and not the ID. This is only to show that there is more than one way to refer to an element using the Selenium framework.

When the element has been found, a string of characters will be send to it, "*admin*". This represents the interaction of clicking on the text field in the admin window and entering a username. The interaction of entering the password is similar to this, which is the fourth method.

The fifth and last method in Figure simulates the interaction of clicking on the *Login* button. It follows the same structure as the second method.

The pseudo code in Figure 21 is to confirm if the test succeeded, in other words, if the login was successful. There are various ways to check if the test passes. One way is to check if there is an element with ID "*logout*" that is visible at the website. The reasoning behind this is that, upon a successful login, a link called *Logout* will appear. If not, then this link will not be visible for the user, meaning that the login failed. Depending on if this element is visible or not, a message will be printed to the NetBeans IDE console.

```
// A confirmation.
WebElement x = driver.findElement(By.id("j_idt25:logout"));

// The "Logga ut" link visible means that it logged in to the adminpage.
if (x.isDisplayed())
    System.out.println("Login success!");

else
    System.out.println("No success!");
```

Figure 21 – One of many ways to confirm if the test has passed.

This test case was manually coded in the NetBeans IDE and not recorded through the plug-in. This test case works behind the scenes meaning, it is not possible to follow the execution of the test like it is with the Selenium IDE plug-in in Firefox. The whole test is executed without a Firefox window. It is now time to execute the test. Please jump to section Executing a test.

B.4.2 Implementing an exported recording

Now, it is time explain how to export the code from the Selenium IDE plug-in recording. When done recording a test, it must be exported as Java code. How to do this is explained below.

Click on *File*, and then hover on *Export Test Case As…* until a new window appears.

Click on *Java / JUnit4 / WebDriver*.

Save the file with the same name as the java class created in activity B.2.2 (Figure 9). Figure 22 shows how to export a recording.

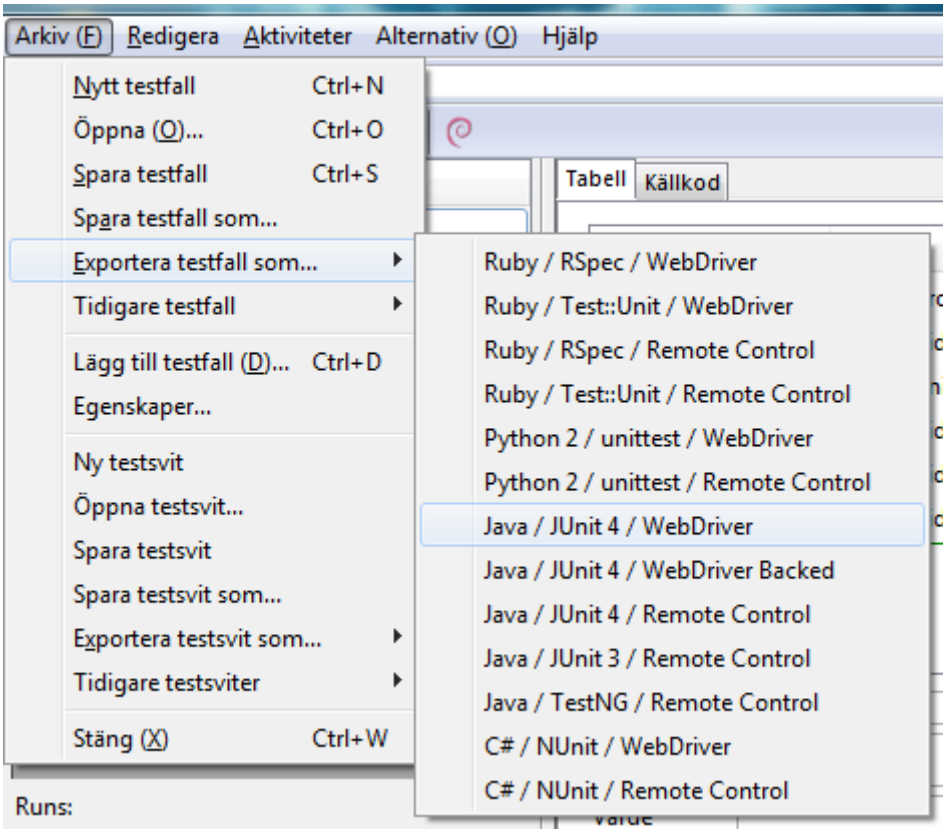

Figure 22 – Exporting the recorded interaction to java code.

This file contains the recorded interaction in java code. Next step is to open the test class that was created in activity B.2.2 (Figure 9). You will now be presented with a new test class that contains around 150 lines of automatically generated test code. For this example, please delete this code.

This class will be located under a new folder called *Test Packages* in your new project called *"seleniumSecondMethod"*.

Copy the java code from the exported recording and paste it in the JUnit test class created in activity B.2.2 (Figure 9). Remove package name declaration found in line 1 and replace it with the package name for where the java class resides.

There are some unused additional methods in this example and can be removed if desired. These are all the methods from *isAlertPresent* and after. Also, unused variables and imported libraries can be removed if desired. Let us now go over the code.

Once the test class is set to run, a couple of things need to happen before the actual test code is executed. Before the test can execute, the driver needs to be initiated and the website loaded. That is done by using the JUnit annotation *@Before* as shown in Figure 23.

```
@Before
public void setUp() throws Exception {
  driver = new FirefoxDriver();
  baseUrl = "http://localhost:8080/";
  driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
}
```

Figure 23 – Initialization of driver.

The Firefox driver is initialized, similar to the initialization in Figure 19. The difference here is that the Firefox driver will be initialized, which means that the test will be executed on a new Firefox browser instance. This also means that you will be able to follow the progress of the execution while the test is running.

The *@Test* annotation specifies that the following method is a test case.
The pseudo code in Figure 24 shows the object references to the different elements in the HTML web pages

```
@Test
public void testAdminLogin() throws Exception {
  driver.get(baseUrl + "/MockitoExamwork/");
  new Select(driver.findElement(By.name("j_idt14:j_idt16"))).
          selectByVisibleText("Svenska");
  driver.findElement(By.id("admin")).click();
  driver.findElement(By.name("j_idt7:j_idt9")).clear();
  driver.findElement(By.name("j_idt7:j_idt9")).sendKeys("admin");
  driver.findElement(By.name("j_idt7:j_idt11")).clear();
  driver.findElement(By.name("j_idt7:j_idt11")).sendKeys("password");
  driver.findElement(By.name("j_idt7:j_idt13")).click();
}
```

Figure 24 – The test case.

The *@After* annotation in Figure 25 specifies what happens after the test has been executed. This can be removed if desired. This method will close the Firefox browser once the test case has finished executing.

```
@After
public void tearDown() throws Exception {
  driver.quit();
  String verificationErrorString = verificationErrors.toString();
  if (!"".equals(verificationErrorString)) {
    fail(verificationErrorString);
  }
}
```

Figure 25 – After the test has been executed.

B.5 Executing a test

In this activity, it is explained how to execute a test through the Selenium IDE plug-in or NetBeans IDE.

B.5.1 Executing a manually coded test

The manually coded test case is executed just like any other java class that contains a main method. Right-click on the class found under *Source Packages* and simply click on the option *Run File*. This is shown in Figure 26.

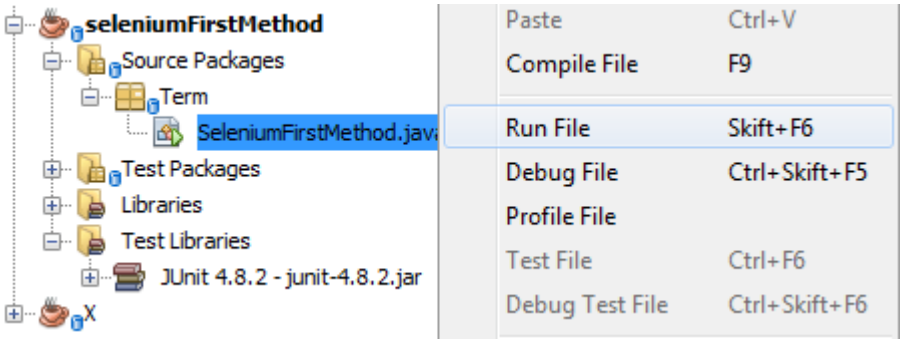

Figure 26 – Executing a test with a main method.

There is no method of confirmation if the test passes or fails. Because of this, a simple if-case was created to check if the logout element is visible or not. Depending on if it is visible or not, a message will be printed to the NetBeans IDE console. See Figure 21 in activity B.4.1 for the code.

If the logout element is visible, the print will be "*Login success!*" as shown in Figure 27.

Figure 27 – The "confirmation" that the test passed.

B.5.2 Executing recording in Selenium IDE

Figure 28 shows the Selenium IDE action bar. You can control the recorded test using this bar by pressing play, pause, stop, choose the execution speed etc.

When done recording your interactions with the Selenium IDE plug-in, it is possible to choose how fast the recording should run. It is recommended to lower the speed to lowest since it may not be possible to see the interactions at higher speeds.

Figure 28 – Action bar used to control the test.

To play the recording from the beginning, press the play symbol. In this example, the website will open up, the language will be changed to Swedish and the script will click on the admin link. Finally, entering username, password and clicking on the login button. You will be able to follow the steps during the execution either by watching the Firefox browser or by following the output to the log window in the Selenium IDE plug-in (Figure 29).

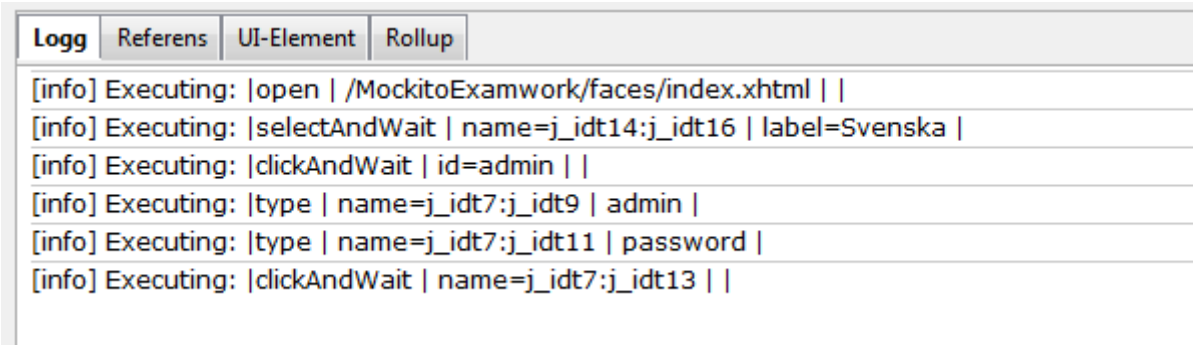

Figure 29 – Output when playing a recorded interaction in the plug-in window of Selenium IDE.

B.5.3 Executing exported recording

To execute the exported Selenium IDE test, right-click on the test class at the project tree and choose *Run File* or *Test File* (Figure 30). A less common error that might occur when running the test is that Selenium cannot find the binary path to Firefox. To solve this issue, make sure that the Mozilla Firefox folder is located at your PC's *Program Files*-folder.

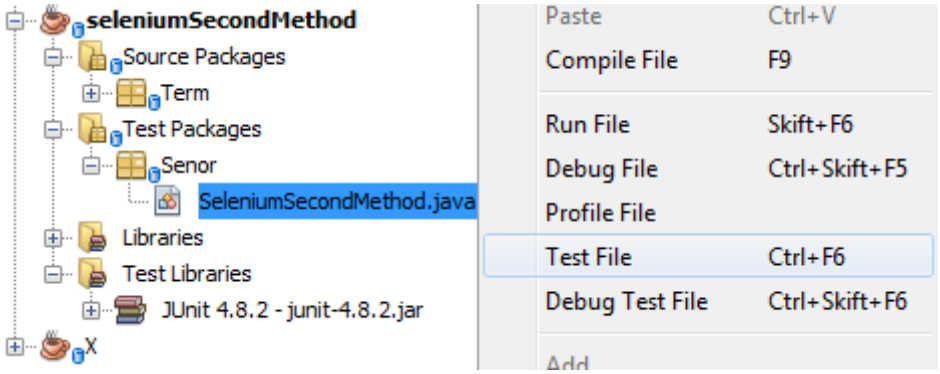

Figure 30 – Executing the exported recording from the Selenium IDE plug-in.

A new window will appear showing the test results at the lower left in NetBeans IDE (Figure 31).

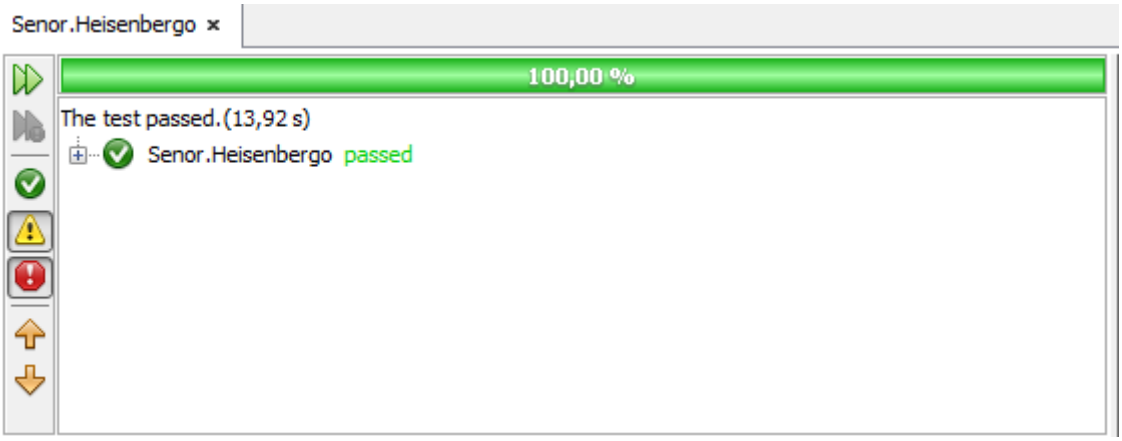

Figure 31 – Test result showing the test passed.

End of tutorial.

TRITA-ICT-EX-2014:55

www.kth.se