Automated Testing of Java Web Applications

Master of Science Thesis in Software Engineering and Technology

MÄRT KALMO

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, June 2009

Automated Testing of Java Web Applications

MÄRT KALMO

Examiner: JOACHIM VON HACHT

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden, June 2009

Abstract

The thesis analyses the profitability of doing automated testing on web applications. It starts with laying out the problems with web application testing and suggests some solutions to overcome them. These ideas about addressing web application testing are combined into a notion of testing framework for web application development. This framework is realized as a part of the thesis and its actual effectiveness is evaluated through using the framework in an experiment. The essence of this experiment is building a sample application twice – once with the support of testing framework and once without it. The performance of both trials will be measured and the findings analyzed. The thesis concludes with the discussion about whether the data gathered from the experiment supports the goal of the thesis.

1 Background

As software development altogether, testing as well, is immature field. There are a lot of open questions, and it is likely that testing is even more controversial than software development in general. This is based on the observation that software producers usually think much more about the software architecture or development process than about testing activities. One possible reason for this is the fact that building software and testing it are separate activities. Developers don't care much about testing and testers can't do much because they get the artifacts too late [EM07]. By the time testers get some software to test the options, how to test it, are quite limited. Another complication with testing is deciding what should be tested and how much it should be tested. Optimal testing practice for different applications could be very different and different applications need different amounts of testing.

One special case of applications is web applications. Major parts of web applications' functionality deals with presentation which is inherently hard to test automatically because it is meant for human as opposed to machine consumption. Furthermore, web applications have many dependencies like web servers, web frameworks or databases. These properties make writing unit tests for web applications a complicated practice. Quite often web applications consist of many simple methods that contribute something for composing the output page. Many of these methods do quite trivial things as for example taking data from data transfer object [Fow02] and put it for the framework specific form object. Writing unit tests for such methods would be wasteful because it takes as much time to write the test as it takes to write the method itself, big amounts of test code would be hard to manage and the test itself would not be very valuable as there is not much that can go wrong with a code that simple. The other problem is that many of these simple methods have a dependency to for example database, web server or web framework. Satisfying all of these dependencies requires a lot of mock code and this complicates the tests considerably. And of course even if we have passing test for each method, it does not mean that the end to end functionality works.

Just as there are problems with unit testing, there are also problems with functional approach. Most functional testing tools for web applications require that the application is deployed into web server before it can be tested. The deployment of an application takes noticeable time and thus wastes the developers time and breaks the so called "development flow". Test Driven Development (TDD) practice describes that tests should be executed about once in five minutes while developing and even more often while refactoring the code [SW07]. If the deployment of an application takes one minute then doing TDD this way would be inconvenient to say the least. Furthermore, when the application under test (AUT) is deployed into the web server and executed there, the only thing for evaluation is the Html output, which

is too little on most cases. This is not so important for the final test code because the Html output usually shows whether the program behaved correctly or not, but it is valuable while developing and debugging the application. If the output Html is not what we expected, then we would like to inspect the intermediate states of the program and get the necessary information for debugging.

Besides the problems mentioned above there is also a difficulty in evaluating the Html code produced by the application under test. The code that extracts the information from the Html pages is difficult to write, it is ugly and hard to read and besides that it is fragile as well. Most common methods for extracting information from Html pages are regular expression and XPath queries. Both of these queries depend on the structure of the document they operate on and if this structure is complicated the usage of the queries gets complicated. If a developer has to come up with different XPath query or different regular expression for each piece of important data on web page, writing only the extraction code could take a lot longer than writing the underling functionality that builds the page we are testing. Writing regular expressions in Java is inconvenient and requires a lot of boilerplate code. And of course such code is not clear and easy to read. As one of the functions that automated tests have is to document the application code, tests that extract information from a Html document by using regular expression do not serve this purpose.

But of course testing web applications automatically is not impossible, only the start up cost is higher than on testing stand alone code with well defined output. It is possible to mock all the dependencies and it is possible to evaluate the Html output too, but providing the scaffolding that makes it possible for every test is too expensive.

Hence the idea for a framework for automatically testing of web applications that provides all the scaffolding and makes writing individual tests easier. Of course this is not entirely novel idea and there are a lot of tools that try to solve the same set of problems. Most of these tools are very general purpose and are meant for testing a wide range of different web applications. That fact entails that these tools are suboptimal for most individual cases. The inspiration for another testing framework comes from the idea of software factories which in essence are sets of artifacts that enable building a certain restricted kind of software more efficiently than doing it in conventional way. So, the idea is to provide a framework that makes writing automated tests for specific set of web applications easier. The functionality and the other restricting factors of those applications are provided in chapter 6.2. It seems logical that by restricting the set of testable applications, it is possible to provide a framework that enables writing simpler and better tests.

At the same time it is hard to prove the effectiveness of such a framework as it is hard to prove that some software development methodology is good or bad. For example, for 30 years a lot of software companies thought that waterfall development model is the right way to build software. It is only in the last decade that the adversaries of the model are gaining more and more supporters [Lar03]. There are quite many studies that compare for example test-first and test-last development but very little studies that try to evaluate the effectiveness of automated testing. The proponents of agile methodologies consider automated testing essential and self evident practice but for a lot of developers this is not at all so.

2 Aim

The aim of this thesis is to demonstrate that writing automated tests for web applications streamlines the application development process and thus makes producing automated tests a profitable practice. In order to accomplish this higher goal several sub goals must be achieved first. The first sub goal is to propose an experiment setting that allows comparing the development process with automated testing and the one without them. This setting should also have clear measurement criteria which allow gathering empirical data about both development processes and thus benchmarking them against each other. The second sub goal is to build a supporting framework that would alleviate the problems with testing web applications that were mentioned above. The framework must enable end to end testing of web applications outside of web server, so that running such a test would take just a few seconds and thus make Test Driven Development a viable development practice. The framework should also support web application debugging and development overall by providing convenience methods that enable inspecting intermediate state of the application execution. The third sub goal is to propose a system of special marking elements in the applications Html output. These markers should eliminate the problem with the fragile test code that relies on the structure of Html documents. The framework should be able to take advantage of these markers and provide evaluation methods for querying the Html output of the application under test. These methods should enable writing test code that is concise, easy to write and serve as documentation for the application.

3 Limitations

The results found in this thesis apply only to certain class of web applications that are described in section 6.2. It is also required that there is a supporting framework which helps writing automated tests. Such framework is described in section 6.3.

4 Method

The aim of this thesis is to demonstrate that writing automated tests for web applications is profitable practice at least on special cases. The aim of the thesis will be realized thorough an experiment which involves building sample application twice. Once it will be built without writing tests at all and relying only on manual testing. On the other time testing framework will be used and thus automated tests will be written during the application development. The first trial is the one with automated tests and the second without them. The trials are ordered this way because of the fact that when writing the same application for the second time it tends to take less time as the developer has already solved some problems during the first trial. This way the order of the two trials is not in favor of the aim of the thesis and the results from the experiment are more reliable.

The experiment will be conducted by only one developer (the author of the thesis) and this is definitely not a very common practice. Usually such experiments have at least a dozen test subjects but in this case there are some other conditions that alleviate this shortcoming. The first positive factor is that each trial takes longer time than the experiment on most similar studies in which it lasts just one or two days. The other advantage is that the performance of the development will be measured hourly as opposed to just measuring the completion time of the whole experiment. What the measuring involves exactly is described next.

4.1 Measuring the Outcome

A first criterion of measuring a development performance is application deployment count during its development. That is how many times the sample application was built and deployed into the web server for inspection. This is quite important criterion because the application deployment takes considerable amount of time and this interruption breaks the flow of the development. The other thing is that such a deployment is usually followed by a manual inspection and testing session which also takes time. It would be really tedious and error prone to count the deployments manually but fortunately this process can easily be automated. Namely, by parsing web server logs, it is possible to get a deployment count for any given period.

The other criterion for measuring development performance is to measure the time that is spent on different activities during the development. Of course it is possible to find a dozen different development activities but on the light of this thesis only three categories are important. These three categories are development, debugging and writing tests. And of course it is quite clear that two distinct trials of the experiment have different amounts of them. For example, the second trial does not involve writing tests at all and if the tests are at all useful the development of first trial should involve less manual testing. There are two hypothesizes that the aforementioned data could confirm. The first hypothesis is that

automated tests reduce the debugging time compared to the development time. If this hypothesis proves to be valid it would of course be positive considering the aim of this thesis. The other hypothesis is that writing a test code takes considerable time compared writing the application code. If this would be true, then it would of course be negative factor. As there is not an easy way to automate the gathering of the data about the different development activities, the following method will be used. The development of sample application will be divided into use cases which are described in the chapter 13. After every hour of the development the developer estimates how much time was spent on the different activities. Of course this method is not very accurate but making notes more often started to influence development flow too much. Among other things this method requires that the developer is experienced in developing such applications because otherwise the development and debugging activities would be mixed into one and could not be estimated separately.

5 Technical background

Automated testing means that the AUT is manipulated programmatically and the response of the application is evaluated automatically. A good automated test should not produce much information when it passes. It should just indicate that it passed. Further information is only necessary when there is a failure, so that the cause of the failure can be easily found. This property makes running tests very cheap – developer just executes the test set and sees “yes” or “now” answer or red or green bar as it is usually described. Unfortunately, it is not always possible to have such tests for every possible application due to many different factors. Some of these factors will be described later in this chapter. Furthermore, automated testing does not replace the need for manual testing entirely [Wei08]. It can just reduce it to a certain decree. How much exactly depends on the quality of automated tests. In a sense an automated test just says when the software is stable enough for manual testing [HT99].

Automated tests have the following lifecycle [Mes07]:

- Setup AUT
- Exercise AUT
- Evaluate the results
- Cleanup

Setup phase puts the AUT into such a state that it is possible to execute the behavior that the test should examine. Then the test is executed and its results are compared with the expected values. This phase decides whether the test succeeded or not. And finally the cleanup phase removes all the traces about the test run. This could involve resetting global variables,

deleting files, deleting database entries or many other things. The cleanup is very important here as it ensures that the tests are independent from each other. This is that the order of how tests are executed does not change the outcome.

5.1 Economy of Automated Testing

Most common conception about automated testing is that it consumes resources while the application is initially developed but starts to pay of in some later stage of software lifecycle. The gain comes from the reduced testing expenses when the software is modified and extended. First, executing automated tests costs almost nothing as opposed to manual testing. The other thing is that if an application needs considerable changes and it doesn't have automated tests and no one from the original development team is available, then it very likely that new team decides to develop new application instead of extending the old one. Automated tests serve as a documentation and safety net when extending existing software. Developing new software instead of extending the existing one could be much more expensive and unpredictable.

The Internet changes very quickly and unpredictably as new technologies and customer needs emerge. This entails that web applications change in the same fashion. The experience shows that there are number of web applications that are not going to live more than five years and are changed very little after the initial development. Then according to the conception presented above, it would be wasteful to write automated tests for these applications. Writing automated tests takes time and there will never be the time when this effort pays off. Thus, when considering such applications, there is either no point using automated testing at all, or we would have to find different concept than the one presented above.

Some authors claim that writing automated tests for applications, does not have to be more expensive than not doing it [DRP99]. They claim that writing automated tests during the development makes the development more efficient and reduces the final testing expenses enough to cancel out the time spent on writing the tests. It is easy to believe, that this holds for easily testable applications and definitely it can be imagined that it holds for certain web applications. As already stated in the beginning of this thesis, showing this about web applications is the main goal of this thesis.

5.2 Purposes of testing

5.2.1 Quality Assurance

The most common use of testing is quality assurance (QA). The testing is done mostly after finishing the implementation of the project. There is usually special team of testers and s comprehensive testing plan. If the plan is executed and all the errors found corrected, then it should convince the project stakeholders that the software is ready for deployment. This is a time honored way of developing software and it works at least that much that it ensures that the application works (to a certain decree of course) when it is deployed. But ensuring the quality of a software is certainly not the only goal of a project and testing should not be constrained only for achieving the quality.

5.2.2 Regression Testing

Regression Testing means that software is tested more than once. If the software would not change there would be no need for regression testing. But of course most of the software changes during its lifetime. For regression testing it is very desirable to have automated tests because when there is not then significant changes to the software would require considerable costs on testing. And this is clearly not cheap. If the QA phase of the project produced automated tests then these tests can be used as automated regression tests during the development of a new version of the software in question. But the development of a new version of some application is definitely not the only point when the application changes. It is usual that an application changes quite a lot during the development. This is especially true for agile development methods and this is the main reason why more traditional methods are being criticized [Lar03]. So, if we are going to write automated tests during the QA phase anyway, why not consider writing them earlier and thus render them more useful.

5.2.3 Test Driven Development

Test Driven Development (TDD) is a practice which prescribes that test code should be written before the application code. TDD is very strict about how the development should be done. It says that all the coding should be done in the following faces. First step, write test method and execute it. It should fail or even not compile as there is no application code yet. Second step, write the simplest possible implementation for the test to pass. Third step, refactor the application code to remove all duplication and make it cleaner. Actually TDD is not only about testing but also about designing the software. Some proponents of the method say that the name is a misnomer and suggest behavior driven or sample driven development instead. TDD helps to design application architecture because it forces the developers to think about how clients would use the functionality. Proponents of the TDD believe that the method

produces better API-s. Good thing about TDD is that this way tests are used maximally. Besides just ensuring the quality, the tests provide a safety net for doing refactoring, help debugging while developing, provide developers confidence to the code and more. It is not clear whether the strict adherence to these rules is necessary considering that the studies about this show different results [BN06] [GSM04] [MW03]. Nevertheless, it seams very reasonable to write tests during the development of an application rather than after it is done.

5.3 Testing methods

Of course there are a lot more of testing methods around than will be discussed here, but the intent here is not to be exhaustive but rather to describe methods that are relevant to this thesis. It was clear from the beginning that the framework, which will be developed for this thesis, is not going to fit under just one testing method. The reasoning here is, that these methods in their pure form are not optimal for developing web applications. The methods and their shortcomings will be described next.

5.3.1 Unit testing

Unit testing is the most fine-grained testing method. Most commonly a unit test tests just one method and not the interaction of several methods. This implies that the methods that we are going to test can be executed separately and it is possible to provide all the dependencies for the methods under test. Furthermore those dependencies should behave in a controlled fashion, so that we can predict the outcome of method execution. The other implication is that we will have a lot of test code. If method has many dependencies, then the test that tests this method could be much longer that the method itself. This is because the test must provide all the plumbing necessary for the method execution.

The main advantage of unit testing is that, if the test catches an error, then it is easy to find what causes it. This is because these tests examine very small part of the application. Unit tests are usually written by programmers and not testers. Furthermore unit tests are written before the application code or right after it. It is hard to write unit tests for the application that is written without unit testing in mind. It can be argued that unit tests are not very suitable for testing web applications mostly because of two reasons. The first reason is that web applications usually have a lot of methods that do quite a little work – fetch something from the underling database and put it somewhere for the view. If we trust the database code (this is a different issue and not related only to web applications) then there is not much to test. It is not common for example to test accessor methods of a Java bean. Testing such simple code would only produce a lot of test code and we still can't be sure that the end to end functionality of the application works. The second issue is satisfying all the dependencies that

web applications usually have. Listing 5-1 depicts a servlet implementation that is just about the simplest possible Java web application.

```
public class HelloServlet extends HttpServlet {
  public void doGet (HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException {

    PrintWriter out = res.getWriter();
    out.println("Hello, world!");
    out.close();
  }
}
```

Listing 5-1

But unit testing its `doGet` method is not trivial at all. First we need to provide implementations for the method parameters which are usually provided by a servlet container. The second problem is that this method has void as its return type and we must evaluate its result some other way.

5.3.2 Functional testing

The most common way to perform functional testing on a web application is to deploy the application to web server and access it over the Http protocol. There are several good tools for this task. Some of them simulate real browser, the others drive a real browser like Mozilla Firefox. There is mainly two ways of writing the tests. The first possibility is to put the tool in a recording mode and exercise the application under the test. The tool records all the actions and they can be played back. The second method is to navigate the application under the test by driving the underling browser programmatically. The first option is very fragile because the slightest change in the interface of the application that is being tested causes the tests to break. The second method is a little less fragile and the tests are quite easy to write – point the browser to the right URL and check the Html returned. But still there are some problems with this approach. The first problem is the fact that the application needs to be deployed to a web server before it is possible to test it. If the build is entirely automated (as it should be) then it is not that big of a trouble when we do it once. Building the application takes one or two minutes and then it is possible to run all the test methods. The trouble is that when we are doing debugging or Test Driven Development we would like to run some particular test often. In this case one or two minutes is too much. The other problem with functional testing is also related to debugging. When some test fails then we have quite a little information in our

disposal. There is only the Html code that is output from the application. If something is wrong we would have to modify the application by inserting “print” or “log” statements, deploy the application again and start over.

5.3.3 Acceptance testing

Acceptance testing is little different than other forms of testing because the people who write the tests are not testers but customers. The customers should evaluate whether the application behaves the right way. At this stage the application should be in a state that it does not crash or behave in some other way which is obviously wrong. Of course developer should fix these defects before – there is no point asking customer whether crashing is the right behavior. Acceptance testing used to be hard to automate but the situation has improved thanks to some good tools. Some of the tools even enable writing test code the form which resembles ordinary English. The testing framework that is the basis for this thesis is meant for programmers, so how is acceptance testing relevant to this thesis? Although the framework does not provide support for executing customer tests, the attempt was made to make evaluation methods as readable as possible. This way it is easier to translate use cases into automated test cases and the other way around. It also makes test cases are more understandable to the developers and even the customers. The clarity and understandability of the tests becomes even more important after some time has passed from writing them.

5.4 Testing different types of software

5.4.1 Software libraries

Software libraries are by definition meant for reuse. Because of this reuse requirement they have certain properties that make them easier to test. Of course all the software libraries are not the same (graphic processing libraries are not easy to test) but the following applies on the most cases. First, problems that the libraries usually solve are better understood than the problems on the average. Also, the libraries usually provide good abstractions and clear interfaces. Usually methods of the library have well-defined input and output. For example a sorting method takes an unsorted list and produces a sorted list and it does not have any side effects. Second, libraries usually don’t have many dependencies and if they have some, there are clean and well defined interfaces for those dependencies. Because of that there is not much setup involved - testing code just calls the library code and evaluates the result that the call returns. Third, as libraries are meant for reuse and often for people other than the creators of the library, they should be well tested. These properties make software libraries good candidates for automated testing and it is most likely that in the case of library code automated testing is profitable practice.

5.4.2 Highly dependent software

Dependent means here that the code is not stand alone entity but uses some external resources. Of course most of the applications have some external dependencies but on some occasions using automated testing is especially hard. The example comes from a tele-communication company, which developed a program that provided electronic TV guide for digital TV. The code ran on a big and very expensive machine. The company had only one those and it was in a live use. In this case it was hard to do any testing at all, let alone automated testing. The only option was to review the code long enough to be sure that it works and then just deploy it. And of course it happened that for some short periods the people who were reading TV guide from their TV-s could see something that was not entirely correct.

The other example comes from the same company. In this case there was not any special hardware involved but the system itself was one big dependency. It was not especially badly designed but there was only one copy of the system and it was in live 24x7. The system was a web portal which ran on multiple machines and which had gradually grown into its current size. The developers at the company thought that it was not profitable to maintain development a environment and so, all the development was done in live. And of course, these conditions make automated testing very difficult.

The third example comes from much more “normal” situation. Usually bigger systems are divided into modules and subsystems which are developed separately in separate teams.
Even in the case that these subsystems have well defined interfaces that connect them, mocking all the dependencies for testing subsystems separately could not be wise. Much more common approach in this case is to rely only on unit testing on module level and do functional end to end testing for the whole system together. In a word, it does not pay of to write comprehensive set of automated tests for every module.

5.4.3 Graphical User Interfaces

Graphical User Interfaces (GUI) are hard to test because they are meant for the interaction with humans and not computers. As GUI-s are less stabile than the underling functionality of the application testing through GUI is usually more fragile. When the appearance of the application changes the tests will brake. The tests could even brake when the application is run on a different environment, for example a different screen resolution. Humans adapt to a different resolution even without noticing it but it presents a problem for the machines. Even more difficult than the input is the output part of the interaction. Some output is even completely impossible to test automatically, for example how to test that the images meant for

separating computers from humans (CAPTCHA) are legible for humans. Besides this human versus computer difficulty, GUI code is usually quite complex. GUI applications have at least one extra thread dedicated to communicate with user interface and the communication is event based.

5.4.4 Web applications

Web applications are another class of applications that have some properties that makes testing them different from many other applications and unfortunately web applications are often most untested [WB07]. The first and foremost of those properties, that makes testing harder, is the fact that a big part of web application code usually deals with the presentation and the page flow. It could be that the whole web application serves just as a user interface for some other program. One such example could be an Internet bank. Internet bank is just an interface for underling banking system that was created without knowing that it would be accessed over the Internet in the future. The fact that web applications have a lot of presentation logic make them harder to test automatically as mentioned above where GUI applications were discussed. Fortunately, web applications have GUI-s that are a little different from the GUI-s of desktop application and this somewhat alleviates the problem. It is quite easy to simulate user input by just generating a query string that represents user clicking on a link or pressing a button. The evaluation of the output is also somewhat easier because it is in a standardized form – the Html code. Another issue with web applications is the abundance of dependencies that they usually have.

Usually request to a web application goes through roughly the following faces:

- Interpret a query and select the action
- Access the database
- Do processing
- Access the database
- Put the data for the view to be found
- Select the right view
- Render the output

From those faces only “Do processing” is a good fit for unit testing. As web applications usually have a little business logic compared to other parts it is clear that relying only on unit testing in the case of web applications is not wise. The other steps have different problems which will be discussed next. The query interpretation has usually dependencies to the web server (HttpServletRequest object in Java servlets case) and is also not very complicated. The code just needs to select the right action based on the query string. The database access could

(and should from the design point of view) be factored behind a clear interface and thus can be tested separately. Database testing is a specific field and should not be mixed with testing web applications in general. “Put the data for the view to find” is by authors view too trivial to test on its own. It is not common to test for example accessor methods which is basically the same thing. “Select the right view” is very dependant to the code of the underling framework and thus hard to test independently. And finally, the rendering of output the Html is usually done by some other language than Java. Most common way for Java web applications to render the Html output is to use JSP templates and custom tags. And of course unit testing this behavior is complicated. Considering all the above, it seams quite clear for the author that in case of web applications end to end testing should be preferred to unit testing.

5.5 Existing Testing Tools

At the current time Open Source Testing[1] website lists 89 testing frameworks and 67 Java unit testing tools. Next, short overview of these will be given. The purpose of this overview is to show which tools already exist and why there would be a need for another testing tool. It also shows which components could be used as the parts of the new testing framework.

5.5.1 Testing Frameworks

These frameworks can be roughly categorized into few distinct groups:

GUI testing tools are tools that enable testing Java applications with a graphical user interface. These tools achieve their coals either by manipulating graphical framework (e.g. Swing) objects directly or by generating system events like keystrokes.

Web testing tools that use HTTP enable querying application under test over the HTTP protocol. They either simulate a web browser or drive the actual browser like Mozilla Firefox. They allow navigating the pages of the application under test and enable to parsing and querying the returned Html contents.

Script languages that are legible for other people than software professionals are mainly intended for making testing scripts more understandable for customers. This allows customers to either write the test cases or at least validate the ones written by testing professionals. The main variations are table based test cases and test cases that resemble plain English.

[1] http://www.opensourcetesting.org/

Performance testing tools allow benchmarking throughput, scalability, resource consumption and other performance aspects.

Special purpose testing tools are projects that are intended for testing some narrower scope of applications e.g. Flash or Ajax applications.

5.5.2 Java Testing Tools

These tools are the basis for many frameworks mentioned above. Description of the most important of them is presented next.

JUnit is a very basic tool that allows executing tests written in Java. It enables scanning classes that contain unit tests. It identifies methods that are marked as test methods and executes these methods. JUnit also provides some primitive evaluation methods that can be used in tests. It also enables the construction of test suits which consist of arbitrary number of test classes. A many tools use JUnits functionality or provide extensions for it.

JMock is a representative of the tools that allow creating mock objects. It generates an implementation for an interface provided while creating mock object. Generated implementation can then be passed to the methods that expect the real implementation of a given interface. It is possible to specify expected return values of certain methods. Mock objects can also record method call sequence that was made against its methods and provide this information for later inspection. JMock is also extensively used by other frameworks and tools.

DBUnit is an extension of JUnit which simplifies database testing. It provides convenient means to populate database with sample data. It also provides many different methods for querying database for changes after executing test code.

Cactus allows executing web applications unit tests in their final environment – in a web container. It consists of a server and a client part. The server code is deployed to the web container with the web application. After deploying the application it is possible to ask client API to execute some unit test. Client contacts the server which executes the test methods and returns the result.

Cobertura allows to measure test coverage. That is how much of the application code is executed from the test code. It modifies the application classes' byte code and adds callback methods to it. So, the tool can see which paths of the application code were run when the test was executed.

6 Implementation

6.1 Technology used in the project

6.1.1 Java Web Applications

The frontend of Java web applications is usually implemented by using Java servlets. There is a standard for Java servlets and this means that a servlet that follows this standard works on every web server that conforms to this standard. It is quite easy to write a servlet. The only thing that we should do for following the standard is to implement the interface `javax.servlet.Servlet`. And even easier way is to extend the `javax.servlet.http.HttpServlet` which already provides some functionality and default implementations for all the methods needed. Besides extending `HttpServlet` we should override one of its methods for adding our own behavior. If the servlet is deployed to the web server then the request sent to the web server executes this method and provides the query as a parameter. The web server also provides an output stream for writing the response for the request. Our servlet can then interpret the request and write an appropriate response to it. Using servlets in this way is inconvenient on most cases. This approach works only with the very trivial applications because the API is too low level. Most web applications use some framework that simplifies the development. Usually such framework provides its own implementation for the `Servlet` interface and this implementation intercepts all the requests to the web application. In this case we don't write the implementation for the `Servlet` interface but extend some framework specific class. In essence it is quite the same thing but this way it is possible to use frameworks functionality. This functionality usually consists of some sort of template system, easier request dispatch, automatic form object population from the Http request and more.

6.1.2 Spring framework

Spring is Dependency injection framework. It allows centrally specifying which dependencies are set to which objects. Central configuration file describes all the objects that Spring framework manages and all the dependencies that these objects have. So, the frameworks main functionality is flexible object construction. When an application needs to instantiate an object that has many dependencies (aggregation relationships) then there is quite a lot of code involved for constructing the object – all the dependencies must be created and wired to the main object that we are creating. Further, this way all the dependencies are hardwired into the application code and the substitutions are troublesome and error prone. Spring enables specifying all the dependencies in an external configuration file and it creates all the dependent objects and wires them automatically on object creation onto the main object.

Applications built on a dependency injection framework are usually easier to test, because the framework promotes loose coupling (mostly by using dependencies through interfaces instead of concrete classes) and thus, different parts of the application can be tested separately. In addition to the fact that Spring framework makes testing easier by design, the developers of the framework have added many features that further simplify testing. Most useful of such features are the mock implementations of the interfaces `HttpServletRequest`, `HttpServletResponse` and `HttpSession`.

6.1.3 Spring MVC Web Framework

Spring has many extensions and one of them is its MVC web framework. Spring framework with this extension is the central part of the sample application used in this project. The most important parts from Spring MVC framework are described below.

Controller is a user provided class that extends framework specific class. Controller is the main connection point between the Spring MVC framework and a custom code which makes up the web application. Spring MVC framework calls controllers method which executes some custom code and returns the information for selecting and displaying the right view.

View is usually a template with static and dynamic parts which will be rendered as the output of a web application. Most common view technology for Java web applications is Java Server Pages (JSP). Such template consists of static Html and custom tags that enable programmatic behavior. It is also possible to embed pure Java code into JSP template. View code has access to objects prepared by the controller and thus is able to display dynamic information which the controller provides.

Form object is a simple Java Bean through which Spring MVC framework passes info to and from Html forms. When the Html form with prefilled data must be displayed the controller populates the form bean with necessary data. The framework makes this form also visible to the view that can render Html tags with the right values. When the Html form is submitted the frameworks populates the form bean from the POST request and makes this bean available to the controller which can access the data using the accessor methods.

When the web server passes a request to the Spring MVC framework the sequence depicted on Figure 6-1 will follow.

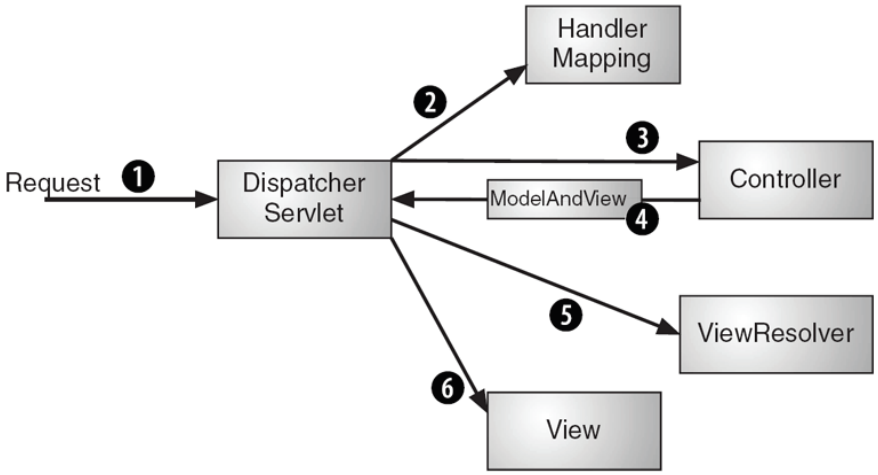

Figure 6-1

1. Web server passes a request to `DispatcherServlet` which intercepts all the requests for a web application.
2. `HandlerMapping` determines from the request which controller should service the request.
3. `Controller` executes applications functionality and creates `ModelAndView` object which contains the information about which view should be used. This object also contains all the data that the view needs for rendering itself.
4. `Controller` returns this `ModelAndView` object to the `DispatcherServlet`.
5. `DispatcherServlet` asks `ViewResolver` for the right view based on the `ModelAndView` object returned from the controller.
6. `DispatcherServlet` instructs the `View` to render itself. It passes the `ModelAndView` object to the `View` so that it can use the data provided by the `Controller`. The output rendered by the view is passed to back as response to the initial web request.

6.1.4 Freemarker

The most typical view technology for Java web applications is Java Server Pages (JSP) with its many tag libraries. As it is the most commonly used technology it would have been best to use this, but there are some problems using this. Namely there is no easy way to render JSP pages outside of web container. There are some open source tools that allow rendering JSP code outside of the web container but there is no integration with the Spring framework and they are not 100% conformant to the implementations provided by web containers (i.e. specification). So instead of spending too much time here the decision was made to use Freemarker templating framework instead of JSP. JSP enables to do more powerful things inside view layer as it is possible to write Java code inside JSP. But it is arguable whether it is a good thing or not. There should not be much processing in the view layer and Freemarker

has its own language that is quite impressive. Besides Freemarker lets you call Java methods from the templates. All in all, Freemarker seamed to be a better fit for the purpose of this thesis.

6.1.5 Apache Commons Validation

Commons Validation is the most common general purpose validation framework. Most well-known Java web frameworks provide integration with this validation framework for their web form validation. Commons Validation allows a declarative specification of which fields of which beans have which validation rules. Main advantage of the framework over ad hoc validation code is its declarative nature which decreases the possibility of errors. Although the validation language is not very powerful it is satisfactory for most of the validation problems. And it is possible to use Java code to realize more elaborate validation schemes.

6.1.6 JUnit

Testing framework described in chapter 6.3 (TF) is built on JUnit. As JUnit is meant for unit testing its evaluation methods (methods that enable to evaluate the out come of test execution) are very primitive. This is fine when testing methods that return primitive values, arrays of primitive values or objects with primitive values as fields. In TF case it is necessary to have more powerful evaluation methods. As many methods that will be tested return values with a common structure like a Html string it makes sense to use evaluation methods that are aware of the structure. So, instead of asking whether a string contains a substring “john”, it is possible to ask whether the page contains a form with a field named “username” with a value “john” with a simple method like `fieldValueIs(“username”, “john”)`.

6.1.7 HtmlUnit

HtmlUnit is a functional testing tool which allows browsing web applications over Http protocol and evaluating the content of the Html pages returned. The fact that it uses Http implies that the AUT must be deployed to a web server before it can be tested. As the intent is to do the testing outside the container, HtmlUnit can not be used the way it is intended to. Actually, its ability to query web applications is not important at all. Its main value in the context of this thesis is its ability to query the structure of the Html pages. HtmlUnit has very advanced functionality for inspecting Html pages and allows even XPath queries like //form[1]//input[@name=”john”]. This functionality makes it ideal for custom evaluation methods which evaluate Html strings.

6.1.8 The Standard Widget Toolkit

The Standard Widget Toolkit (SWT) is an open source widget toolkit for Java that is designed to provide efficient, portable access to the user interface facilities of the operating systems on which it is implemented. Among many others it has a component that renders a Html code from an input string provided to it. This functionality is very convenient for developing web applications. Sometimes it is much easier to just look at the rendered page and tell whether it is right, than it is to analyze its source code [HT99].

6.1.9 Apache Commons BeanUtils

BeanUtils is one of many Apache projects. BeanUtils makes reflective access of Java beans much easier than doing it manually. For example if the TF has a method like `getFormFieldValue("fieldName")`. This method accesses data from a Java object fields. The only way to realize this in Java is to use reflection and BeanUtils makes this much simpler than using reflection in more low level way.

6.1.10 Apache Maven

Maven is a build tool which has a philosophy "convention over configuration". It is intended mostly for building Java projects. The main purpose of Maven is the same as Apache Ant-s – to build a software project. It enables to compile a code, copy resources, package it together and deploy it. Maven differs from Ant mostly because it has many reasonable defaults for Java projects. One such convention is the directory structure of a project. So, all the tasks that run during the building of a project, already know where to find artifacts needed for their work. So, if we are satisfied with the standard then it is possible to get by with a very little of configuration. This convention also makes the project structure more understandable to the other people who already know the standard. The other good thing about Maven is its central repository. This functionality makes managing the dependencies (like spring.jar) of the project much easier. You just say that the project needs this library and Maven downloads it with all its sub dependencies and makes it available for the project.

6.2 Sample application

For testing the TF-s effectiveness a sample application will be used. The goal is to make this sample application a good representative of the applications that can be tested by the TF. It would seam that a real project would be the best candidate for the sample application. But the problem with real projects is that they usually have some features that are very specific and at the same time they usually lack some important features which are common among other similar projects. Because of the above the choice was made to use a sample application that

has some common functionality from several projects. The idea comes from the concept of software factories which define a family of software products which share a common core architecture. This core architecture has some well defined variability and extensibility points. This architecture dictates which products can be seen as a family. If a company gets an order for some product that differs too much from the products that their software factory is able to produce, then this offer is turned down or dealt separately without using the software factory. At the present case, the testing framework is intended to be used for building a certain set of products. If the intended application can not be realized by using predefined core functionality and provided variation points then it is likely that the TF does not provide much help for developing this application. For example TF expects that the application is built using the Spring or some other similar application framework. If this is not the case then it is quite likely that the TF does not pull its weight. Of course the TF could be modified to support for example something different like the Tapestry framework as well, but it is probably not profitable provided that the company is not going to do more Tapestry projects in the future. Considering the above the sample application will share most common features from some real projects from the past.

6.2.1 Application scope

The first limiting factor is technology. TF is meant for testing web applications written in Java and with the Spring application framework. There could be other technologies involved but in this case there must be clear interfaces between the Java/Spring parts and the others. The second limitation has to do with some implementation aspects. The TF requires that some Html elements are generated in a certain way (not using arbitrary string concatenation). This fact renders it mostly unusable for extending existing products. So, it is mostly usable for greenfield projects or projects where it is feasible to do refactoring. The third limitation is the complexity of the application. TF is intended for testing applications that do not carry out complicated calculations. The most common scenario is fetching some data from the database doing some transformations and presenting the data as Html. Actually, the framework is able to handle applications that do complicated computations but it does not add much to JUnit's functionality. Usually complicated computations have well-defined input and output and therefore JUnit satisfies all the needs for testing such computations. At the same time it probably does not provide much help for developing very simple, mostly static, web applications as well. In this case record and playback testing tools would be more suitable.

6.2.2 Feature selection criteria

As mentioned above the features of the sample application are chosen from the most common features in some real projects. Besides that, these are the features that are usually hard to test

automatically and take a lot of time while doing the testing manually. The third consideration was that it should be likely that the functionality brakes. There's not much sense testing trivial things. The sample application must have the following features to serve its purpose:

- **Authorization**. The main idea being that the application state must be preserved all the time while interacting with the application. If some link for form is missing authorization token then the link/form is broken.
- **Form data validation**
- **Internationalization**. If there is more than one interface language then it harder to write scripted functional tests. If the language is not English then the script that searches for button with a label "Cancel" does not work any more.
- **JavaScript**. Most application use JavaScript to some extent e.g. deletion confirmations. If functional testing framework is not able to handle it then it is not possible to test the functionality.
- **Multipage form or so called wizard**. It is time consuming to test this functionality by hand and it can easily brake.
- **List/Form view of a collection of domain objects with CRUD operations**. Almost all the authors past projects have had this functionality.

6.2.3 Final pick

One real project from the past covers most of the functionality that is described above. This is a part of Internet bank which allows clients to submit applications for loans, credit cards and the like. It is likely that TF could be helpful on this particular case. The specification of the application is presented in

Appendix C. The Specification of the Sample Application. Figure 6-2 shows a screenshot from the original application.

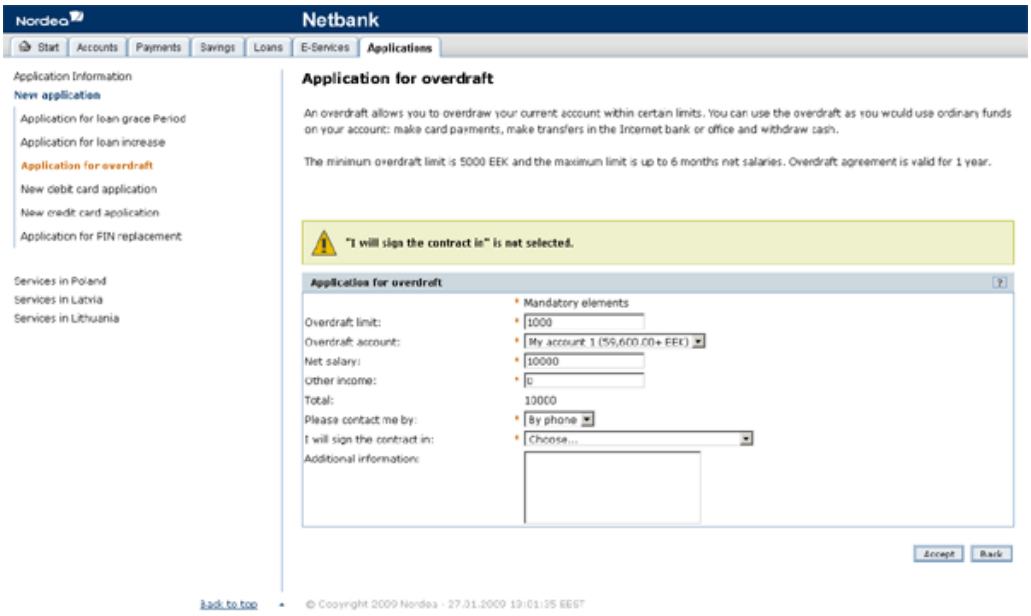

Figure 6-2

6.2.4 Architecture

The sample application is built on Spring MVC web framework. This choice dictates the main architecture for the application which is as the name suggests a Model-View-Controller (MVC) pattern [BMRSS96]. The view part is implemented by Freemarker templates and is managed by the framework. Also the model part of the application is quite nonexistent as the application just interfaces data between user and database and does not do much processing on this data. This leaves the controller which does the most of the work. The controllers of the applications are split into several classes and this architecture is presented next.

There are four different controllers: one for each use case. As there is quite a lot of overlapping functionality between those four controllers this functionality is pulled up into common superclass. Class diagram of the controllers is presented on the Figure 6-3.

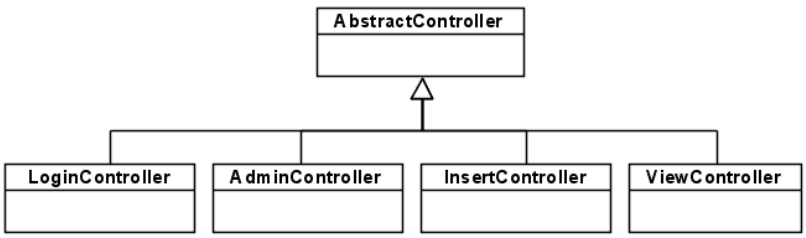

Figure 6-3

All the controllers except `LoginController` need to deal with three different application types. To avoid multiple switch statements and to simplify the controller code, the code that is different for each application type is factored out into polymorphic strategy [GHHV94] classes. So, there is a strategy for each application type. Each strategy class has its own form class through which Spring framework transports to and from Html form forms. This architecture is presented in class diagram Figure 6-4.

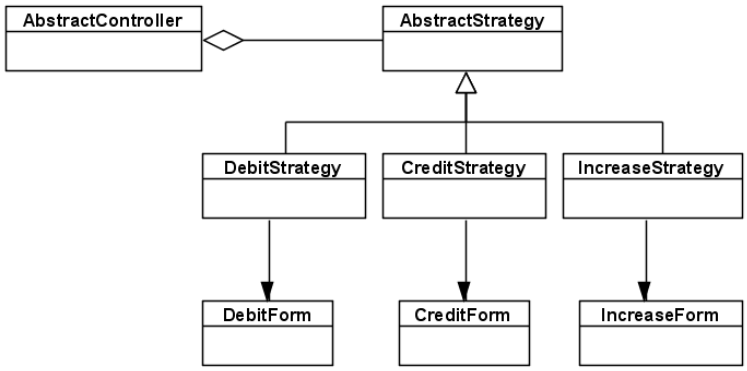

Figure 6-4

All the database access goes through Data Access Objects (DAO) [Noc03] interface. The implementation for this interface is injected to the controller by the Spring framework and this implementation can easily be replaced. Controller has now knowledge about underling datastore and accesses data by calling DAO interface methods and passing and receiving Data Transfer Objects (DTO) [Fow02]. This architecture is depicted on the class diagram on Figure 6-5.

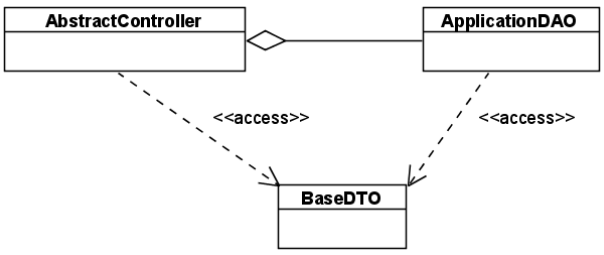

Figure 6-5

6.3 Testing Framework

6.3.1 Functionality of the Framework

Testing Framework allows executing test code without deploying the application

The framework enables instantiating Spring controllers so that they have all necessary dependencies. This way it is possible to just call controller's methods outside web server. This makes running individual tests much faster and makes Test Driven Development (TDD) more feasible. It also provides developers with the possibility to get debugging information much quicker. The framework provides also special methods for querying the intermediated state of the application under test. Also, just inserting "print" or "log" statements into AUT code works much better this way.

6.3.2 Special marker elements in Html code

To alleviate the problem of fragile tests the framework prescribes that developers use special marker elements in the Html that the AUT produces as output. If the developer follows this advice then the framework can take advantage of these special markers and produce tests that do not depend so much on the structure of the Html document. These special markers add additional structure to the Html pages. For example, let's say that there is a page with n tables and we would like to evaluate that one of them contains m links. If the page does not contain any additional markers we would have to rely on the order of the tables and to the fact that they are tables (rendered with the Html "table" tag). But if we surround the links we are looking for with additional tag, which has certain identifier, then it is possible to find those links without relying on the structure of the Html document. In this case the layout could change radically but if we still surround the links with this additional tag, then the tests don't break. These special markers could be rendered by macros which would allow turning of the rendering of the markers in a production environment where it is not needed.

6.3.3 Abstractions

The problem with a general purpose language like XPath is that it is meant for querying all possible XML documents. But the elements that we search for from the web applications' output are quite specific. It is logical that a general purpose tool is suboptimal for the very specific tasks. What we usually would like to check from the web applications output is that it contains form fields with certain values, it contains certain messages or that certain links are valid. But if we are interested in only these things it is possible to abstract away the structure of the Html almost completely and use evaluation methods that are much more human friendly. For example instead of using XPath query `//input[@name='amount']` to find an element and then evaluation the elements value attribute we could have a method like `fieldValueIs("amount", 7)` that would accomplice the same thing. This way the tests are much easier to write and are readable as well.

6.3.4 Usage

The intended usage of TF is very similar to the usage of HtmlUnit. Short example of using HtmlUnit is presented in Listing 6-1.

```
01. @Test
02. public void testTitle() throws Exception {
03.    WebClient webClient = new WebClient();
04.    HtmlPage page = webClient.getPage("http://localhost/test");
05.    assertEquals("HtmlUnit ", page.getTitleText());
06. }
```

Listing 6-1

- Line 3 creates a browser object.
- Line 4 requests a Url from web server and saves the result.
- Line 5 evaluates that the resulting page has a title “HtmlUnit”.

In both cases test code is organized as JUnit test cases but TF uses ControllerTester instead of WebClient and Result instead of HtmlPage. Both frameworks provide interface for executing application functionality and enable inspection of the result through special object. The difference is that TF executes application code outside of web container and thus does not require that the application is deployed before the execution of test code. Also TF-s result object provides much more feedback for evaluation than HtmlUnit’s HtmlPage. Listing 6-2 provides simple example of TF usage. A simple tutorial of the TF is presented in chapter 12.

```
01. @Test
02. public void presentForm() {
03.   ControllerTester ct = new ControllerTester("sample1.xml");
04.   Result r = ct.execute("/login.html");
05.   assertThat(r.getViewName(), is("loginForm"));
06.   r.div("loginForm").contains("login.label.username");
07.   assertThat(r.getFormFieldValue("username"), is("john"));
08.   assertThat(r.getHtmlFieldValue("username"), is("john"));
09.   assertThat(r.getModelField("accounts['2']"), is("Account 2"));
10. }
```

Listing 6-2

- Line 3 creates ControllerTester object. “sample1.xml” is Spring configuration file that contains applications configuration. All spring managed objects are described in this file.
- Line 4 executes the code of a controller that corresponds to the identifier “/login.html” in Spring configuration.
- Line 5 checks that the controller code selects the right view to be rendered.
- Line 6 checks that a certain part of the output Html contains a certain label.
- Line 7 checks that the form bean’s field “username“ has a value “john”.
- Line 8 checks that the Html contains form field named “username“ and this has a value “john”.
- Line 9 makes an evaluation about model objects structure.

6.3.5 Architecture and Design

The central classes of the framework are `ControllerTester` and `Result`. Those classes bind together different classes from Spring framework and HtmlUnit and add some custom functionality. Together these classes make up the interface for the TF. `ControllerTester` enables to compose and execute a query against a web application and Result provides evaluation methods for inspecting the outcome.

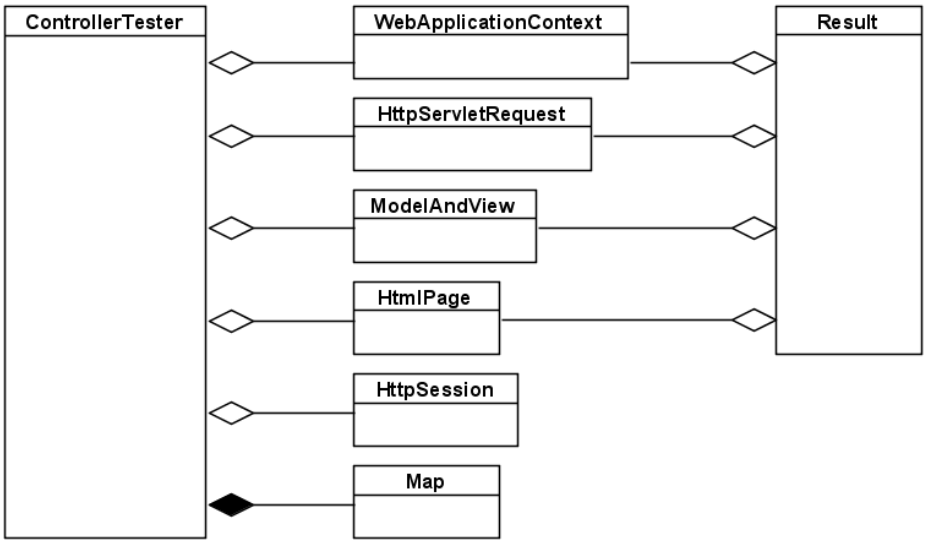

Figure 6-6

WebApplicationContext is a Spring class. This is the main point through which the Spring framework is accessed. When we need a Spring managed object we ask the `WebApplicationContext` to create it for us. `WebApplicationContext` consults its

configuration and then creates all the dependencies, creates the object requested and wires the dependencies to object requested. TF uses the `WebApplicationContext` to create controller objects. `WebApplicationContext` instantiates a controller class and it populates it with data access object, form bean object and other objects.

HttpServletRequest is actually populated with `MockHttpServletRequest` object which comes with spring distribution. This object is sent to controller as a parameter when the controller code is called. Also, this object can be inspected after the controller code has been executed. Among other things it is possible to get HttpSession object through HttpServletRequest.

ModelAndView is also Spring specific object. When controller code is called it returns ModelAndView object. This object contains information about which view to select and also all the dynamic data that controller wants view to render.

HtmlPage is HtmlUnit specific class. This class is not used in the execution of the test. Its purpose is to simplify html parsing code. HtmlPage is an abstraction of an Html document and it provides many methods for querying the document that it represents. Among others it provides the possibility to use XPath to fetch elements from Html document.

HttpSession is created when it is when the client code calls `ControllerTester` objects `startSession()` method. After this call `ControllerTester` object creates `Session` object and populates every `HttpServletRequest` that it sends with this session object. This enables testing functionality which depends on the possibility to store data in session. This could be used for example to test multipage entry forms where all the pages except the last collect data into the session and all the data is submitted to the permanent store when the whole multipage form is finally submitted.

Map is ordinary Java `Map` interface implementation. This is used for collecting the parameters that must be passed to controller. As `ControllerTester` takes care of the creation of `HttpServletRequest` object it must also populate this object with the right parameters. So, `ControllerTester` accumulates the parameters for some query into this map and populates `HttpServletRequest` with them if there is a time to call the controller code which requires `HttpServletRequest` as a parameter.

`ControllerTester.execute()` returns a `Result` object which provides access to request, session and model objects as well as html output. Most objects described here are shared between `ControllerTester` and `Request` as seen from the class diagram in the Figure 6-6. For example `HttpServletRequest` object is shared between `ControllerTester` and the `Result`.

This is because `ControllerTester` creates the `HttpServletRequest` and passes this to controller code. Controller code could add some attributes to `HttpServletRequest` object (`HttpSession` is also accessible through `HttpServletRequest` object) and after the controller code is finished we would like to see that the controller actually did the required modifications. So, it is possible to inspect the `HttpServletRequest` object through the `Result` object. `Result` provides several evaluation methods which simplify the inspection of Html output and the intermediate state of the execution.

7 Results

7.1 Measurement Data Analysis

The sample application described in section 6.2 was built twice - once with the help of TF and with automated tests and the second time without the TF and without writing any tests. The results gathered from the experiments are presented and analyzed below.

The first trial took 16 days[2] and the second trial took 8 days. This would suggest that the attempt was complete failure as developing application with automated tests took twice as long. But these numbers are misleading because of the reasons presented next. It was stated in section 4.1 that the development performance measuring criterion is deployment count and the time spent on the different development activities. Namely the time spent on coding, debugging and on writing tests. Measuring the deployment count succeeded entirely and produced following results:

	Deployments/day	Mode
With testing framework	0-8[3]	2
Without testing framework	8-21	14

This shows that the framework saved up to 21 minutes a day. And this is only the time that was spent on waiting for the application to be deployed. Other costs like the “flow” interruption and the time spent for manual inspection through a web browser are also considerable. Unfortunately these costs were not measured.

Unfortunately measuring the time spent on the different development activities was not very successful. The first problem was that during the first trial development two entirely new and

[2] A day means here 5 hours of development without pauses.

[3] The application was deployed 8 times a day only once and this was during framework improvement.

unanticipated activities emerged. One of those activities was improving the test framework which took about 25% of the total development time. Although during the development of the testing framework it was used in a sample development session, more real usage reveled many aspects that had to be changed or improved. It also turned out that some of the frameworks functions were not used at all.

The other problem that the first trial development session reveled was that the sample application had some unsolved design issues and solving these issues took about 12% of the total development time. As these design problems were solved during the first development trial, no time was spent on solving them during to the second trial. So, altogether 37% of the first trials development effort was spent on the activities that where not initially anticipated and that are not relevant for the comparisons of the two development trials.

It turned out that there was not much activity that could be clearly classified as debugging. Although during the first trial a whole day was spent on debugging the page flow of Use Case 3, this issue was finally solved through using a different design and this problem did not come up during the second trial because the better design was used from the beginning. Most of the development was done in small increments. And probably as the sample application was quite simple any sudden bugs, which would have called for a debugging session, did not show up.

Fortunately measuring the time spent for writing tests compared to overall development time turned out to be manageable. Writing a test for some piece of functionality took about 10-60% of the time spent on developing the functionality and on most occasions it was 15% or less. This is very good result as this invalidates one of the hypothesizes presented in section 4.1. So, the experiment showed that the hypothesis, “writing automated tests takes considerable time compared to time spend for developing the functionality that is being tested”, is untrue. Unfortunately it is not possible to say anything concrete about the second hypothesis, which was that automated tests reduce time spent on debugging. This is because the attempt to gather relevant empirical data failed. But besides that the experiment showed that the hypothesis is likely true because the testing framework made development much smoother and more pleasant, and as already said the development was somewhat mixed with the debugging.

7.2 Repeatability

It is very likely that the measurements of deployment count and the time spent for writing tests would give similar results on further attempts. The data gathered during the experiments was very consistent and did not show many fluctuations. As the experiment did not reveal

much information about the debugging activity it is not possible to say anything conclusive about that.

7.3 Quality of the Testing Framework

The quality of the TF was evaluated through the following changes to the application code which served as sabotage [HT99] to the application.

7.3.1 Rename Controller and Form Classes

Spring configuration contains the names of controller classes and controllers' factory methods refer to form class names. So, renaming these classes requires changes that Eclipse refactoring methods can't handle. If the class name is not changed everywhere, the application breaks. Although it is not difficult to do these changes without the support of TF using TF as a help is very convenient. By using TF it was possible to just rename the classes, execute test set, and make corrections in the places that the failing test suggested. Test set revealed all the errors and when all the tests passed the application was correct. In this case the contrast was not very big, because the developer knew all the changes required. But it would have been different when the initial developer and the one making the changes would have been different persons. Besides that, the application with the automated test set did not require manual testing after the refactoring to prove that it did not break anything.

7.3.2 Pull Up Methods

This refactoring means that some common functionality is moved to superclass. As compiler caches most of the errors that this refactoring can introduce the TF did not catch any errors. But it still saved some time because the absence of TF required some deployments and the absence of automated tests required manual testing after the refactoring was done.

7.3.3 Refactor Switch Statements into Polymorphism

This refactoring was very similar to the previous one but as it is more complicated required more deployments and more manual testing. Thus it showed the usefulness of the TF again. The refactoring did not introduce any serious errors.

8 Conclusion

Despite the fact that some measurements did not produce the expected data the overall outcome of the thesis is positive. The measurements of deployment count and time spent on

writing the tests support the goals stated in the aim of the thesis. The same can be said about the relation of the time spent on both trials. If we leave aside the time spent on improving the TF and time spent on some bigger design issues, then the difference between the two attempts is only 13% in favor of the trial without the automated tests. If we take into account the order of the experiments then the result is even better. Besides the data gathered from the measurements, there are also some other factors which can't be easily measured, but are nevertheless important. The tests were concise and easy to write and on most cases proved to be adequate for detecting errors in application code. The development with the support of the TF was pleasant and the tests gave confidence for doing refactorings and other changes. It is very likely that if the project would have been bigger and if there would have been more developers the results would have been better as these conditions introduce more complexity. Furthermore the use of TF promoted more formal development style. It was easier to stay focused on completing the use case at hand because there were no interruptions of deploying and manually testing the application. Also the written tests documented the work that was already done and helped focusing on the parts that were not.

The main problem with the experiment was that the TF was not entirely complete before the experiment started. Because of this substantial part of development time was spent on the development of TF instead of the sample application. Although these activities were easily separable, such distractions certainly influenced the outcome. It would have been possible to avoid this factor if before the real experiment the TF would have been used on yet another sample project, but this seemed too excessive. Another problem was the fact that some design decisions of the sample application were not analyzed well enough. The reason for this was probably the author's limited experience with the Spring MVC framework. The last problem was the fact that it takes less time to program the same application for the second time. This problem was known from the beginning, but there was not much that was possible to do about it. Of course it would have been possible to overcome this with for example having a dozen developers from whom half would have developed the application with automated tests and the other half without. But this setup would have required 10 days from dozen developers and this was certainly not an option.

9 Discussion

The outcome of the thesis was quite satisfactory. Even in the retrospect it is not easy to suggest such a change in a setup that would make it possible to get significantly better result without much bigger cost. On the other hand, the testing framework that was build as a part of this thesis proved to be very usable and thus worth further development.

Although the framework was initially planned only for using with the applications that use Spring MVC framework, it would be possible to tailor it to other frameworks like Struts as well. It could easily be used in a project that does not use any framework at all and is based only on plain servlets or JSP-s.

As the framework was only prototype it has some serious shortcomings that must be corrected before it can be used in everyday use. Most urgent improvement for the framework would be better error messages. During the experiment that was conducted as a part of this thesis the developer new the inner workings of the TF very well and thus could pinpoint any error just by looking at Java stack trace but of course this is not acceptable for commercial use.

The other improvement that could be made to TF concerns performance. At the moment every test class instantiates new ControllerTester object which further instantiates `WebApplicationContext`. But the instantiation of `WebApplicationContext` takes quite long time, that is about 1 second. I would be possible to use Singleton [GHHV94] pattern to instantiate only one instance of `WebApplicationContext` for running all the tests for whole application. This would provide considerable saving it there are a lot of test classes.

10 Bibliography

[BMRSS96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. Pattern-Oriented Software Architecture Volume 1: A System of Patterns. Wiley, 1996

[BN06] Thirumalesh Bhat and Nachiappan Nagappan. Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies. ACM Press, 2006

[DRP99] Elfriede Dustin, Jeff Rashka, John Paul. Automated Software Testing: Introduction, Management, and Performance. Addison-Wesley Professional, 1999

[EM07] Gerald D. Everett, Raymond McLeod Jr. Software Testing: Testing Across the Entire Software Development Life Cycle. Wiley-IEEE Computer Society Pr, 2007

[Fow02] Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley Professional, 2002

[GHHV94] Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1994

[GSM04] A. Geras, M. Smith, and J. Miller. A Prototype Empirical Evaluation of Test Driven Development. IEEE Computer Society, 2004

[HT99] Andrew Hunt, David Thomas. The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley Professional, 1999

[Lar03] Craig Larman. Agile and Iterative Development: A Manager's Guide. Addison-Wesley Professional, 2003

[Mes07] Gerard Meszaros. xUnit Test Patterns: Refactoring Test Code. Addison-Wesley, 2007

[MW03] E. Michael Maximilien and Laurie Williams. Assessing Test-Driven Development at IBM. IEEE Computer Society, 2003

[Noc03] Clifton Nock. Data Access Patterns: Database Interactions in Object-Oriented

[SW07] James Shore, Shane Warden. The Art of Agile Development. O'Reilly Media, Inc, 2007

[WB07] Craig Walls, Ryan Breidenbach. Spring in Action. Manning Publications, 2007 Applications. Addison-Wesley Professional, 2003

[Wei08] Gerald M. Weinberg. Perfect Software: And Other Illusions about Testing. Dorset House, 2008.

11 Appendix A. Abbreviations

CRUD – Create, Read, Update, Delete
API – application programming interface
AUT – application under test
QA – quality assurance
TF – Testing framework described in section 6.3.
JSP – Java Server Pages
TDD – Test Driven Development

12 Appendix B. Short Tutorial of the Testing Framework

This section describes a sample usage of the TF. It describes of how one would use the framework to implement login functionality of the sample application. This functionality is described in Use Case 1 in appendix #.

The first thing that we would like to implement is that if user goes to the address “/login.html” the application displays a login form. We start from a test. What this test needs to do is to simulate query to address “/login.html” and check that the application produces Html with certain elements on it. So we write the following test:

```
ControllerTester ct = new ControllerTester("classpath:sample1-servlet.xml");

01. @Test
02. public void presentForm() {
03.    Result r = ct.execute("/login.html");
04.    assertThat(r.getViewName(), is("loginForm"));
05.    r.div("loginForm").contains("login.label.username");
06.    r.div("loginForm").contains("login.label.password");
07.    r.div("loginForm").contains("login.button.login");
08.
09.    assertThat(r.getHtmlFieldValue("username"), is(""));
10.    assertThat(r.getHtmlFieldValue("password"), is(""));
11. }
```

Listing 12-1

- Line 3 simulates query to the address “/login.html” and gets a result object through which it is possible to evaluate whether the application behaved the right way.
- Line 4 checks that the right view was chosen. This does not check the actual output but only the fact that controller requests the right template for rendering the output. If this assertion passes we can go on and check that the rendered Html is also correct.
- Lines 5-7 check that the Html contains some labels.
- Lines 9-10 check that Html contains certain form elements with certain values.

This is quite a simple functionality. It just presents Html form and nothing more. But actually there is quite a lot that can go wrong. Identifier "/login.html" must be associated with the right controller, the controller must be instantiated, controller must select the right template, the template (and its sub templates) should not contain any errors. On the other hand, if this test passes, there could still be errors in this functionality. But the test covers the most important parts and of course it is not possible to test everything anyway.

Now, if the most simple functionality works we know that the controller is configured right and we can go on and start testing more interesting functionality. The next test checks that the form validation works and it produces the expected messages.

```
01. @Test
02. public void submitInvalidUserOrPassword() {
03.   ct.addParam("username", "user");
04.   ct.addParam("password", "123");
05.   ct.buttonPressed("loginButton");
06.   Result r = ct.execute("/login.html");
07.
08.   assertThat(r.getViewName(), is("loginForm"));
09.   r.div("messageBox").contains("FormLogin.password[regexp]");
10.   r.div("messageBox").contains("FormLogin.username[regexp]");
11. }
```

Listing 12-2

- Lines 3-4 add parameters to the query, as if the form was submitted with these values.
- Line 5 simulates pressing submit button of the form.
- Line 6 executes the query that was set up on lines 3-5.
- Line 8 checks that the controller selected the right template.
- Lines 9-10 check that the output Html contains message box with two messages.

Next test checks that if we provide username and password which pass form validation, but are not right username password combination, then we get different error message and we should not be logged into the system.

```
01. @Test
02. public void submitWrongUserOrPassword() {
03.    ct.startSession();
04.    ct.addParam("username", "user01");
05.    ct.addParam("password", "123456");
06.    ct.buttonPressed("loginButton");
07.    Result r = ct.execute("/login.html");
08.    assertThat(r.getViewName(), is("loginForm"));
09.
10.    r.div("messageBox").contains("FormLogin.passwordError");
11.
12.    assertThat(r.getFormFieldValue("username"), is("user01"));
13.    assertThat(r.getFormFieldValue("password"), is("123456"));
14.
15.    assertThat(r.getHtmlFieldValue("username"), is("user01"));
16.    assertThat(r.getHtmlFieldValue("password"), is(""));
17.
18.    assertThat(r.getFromSession(ConstAppl.KEY_USER), nullValue());
19. }
```

Listing 12-3

- Line 3 starts new session. Successful login attempt modifies the session and on this test we would like to check that the session is not modified. So, it is necessary to do the simulation with using the session so it is possible to check that the session was not modified.

- Line 10 checks that we get the expected error message.
- Lines 12-13 check that the information sent with the query ended up on the form bean.
- Line 15 checks that the username was echoed back on Html form.
- Line 16 checks that the password was not echoed back on Html form.
- Line 18 checks that certain session attribute is not set. That is, we did not get accidentally logged in.

The last test checks the behavior when we try to log in with the right credentials.

```
01. @Test
02. public void successfulLogin() {
03.   ct.startSession();
04.   ct.addParam("username", "username");
05.   ct.addParam("password", "password");
06.   ct.buttonPressed("loginButton");
07.   Result r = ct.execute("/login.html");
08.
09.   assertThat(r.getViewName(), is("select"));
10.
11.   DtoUser user = (DtoUser) r.getFromSession(ConstAppl.KEY_USER);
12.   assertThat(user.getName(), is("Dummy User"));
13. }
```

Listing 12-4

- Line 9 checks that we were redirected to another page.
- Line 11 fetches an object from the session
- Line 12 checks that this object from the session is the right one.

13 Appendix C. The Specification of the Sample Application

13.1 Use Cases

13.1.1 USE CASE 1: Log into the System

Description: User provides his/her credentials and is logged into the system

Primary actor: client/administrator (User)
Preconditions: none
Success guarantee: user is authenticated and authorized

Main scenario:
1. System presents form for user authorization
2. User enters his/her username and password
3. System validates provided information
4. System authorizes the User

Extensions:
4a. Either username or password is wrong
4a1. Back to Main scenario 1. Also error message is presented on the same form.

13.1.2 USE CASE 2: Review Entered Applications

Description: client reviews the applications he/she has submitted before

Primary actor: Client
Preconditions: Client is logged into the system
Success guarantee: Client has reviewed the application that he/she looked for

Main scenario:
1. System presents list with applications entered by the Client
2. Client selects an application

3. System presents application content in read-only mode. Application status is also presented on the same page

Extensions:

1a. There are not any applications entered by the Client

1a1. System presents information message instead of the list

13.1.3 USE CASE 3: Enter New Application

Description: client enters new application and system stores it

Primary actor: Client
Preconditions: Client is logged into the system
Success guarantee: new application is entered to the system

Main scenario:

1. System presents list with different application types to choose from
2. Client selects an application
3. System presents application form with some prefilled fields
4. Client fills the application form and submits it
5. System presents application in read-only view for review
6. Client confirms the application
7. System presents the final application with the success message
8. Client chooses to go back
9. System presents page with the applications entered by the client

Extensions:

3a. Some precondition is not satisfied for submitting this application

3a1. System presents information message instead of the form

3a2. Clients chooses to go back

3a3. Back to main scenario 1.

5a. Some fields are not correctly filled

5a1. System presents the form with the data entered by the client. It also informs the client about the validation problems

5a2. Back to main scenario 4.

7a. There is an error saving the application

7a1. System informs the client about the problem

7a2. Clients chooses to go back

7a3. Back to main scenario 1.

13.1.4 USE CASE 4: Process entered applications

Description: administrator reviews entered applications and changes their statuses or deletes them

Primary actor: User (administrator)
Preconditions: User is logged into the system. User has adequate rights

Flow of Events
Basic Flow – review and change application status:
1. System presents list of applications entered by the Clients
2. User selects an application
3. System presents the application in read only mode. The same page allows changing the applications status
4. User reviews the application and changes its status and confirms it
5. System saves the new status and presents the application list

Extensions:
1a. There are not any applications entered by the Clients
1a1. System presents information message instead of the list
4a. User does not change the status or does not confirm the changes
4a1. Back to main scenario 1.

Alternative Flow
Delete an application

1. System presents list of applications entered by the Clients. The same page has the option to delete the applications.
2. User chooses to delete an application
3. System asks for confirmation
4. User confirms his/her intent
5. System deletes the application

Extensions:

1a. There are not any applications entered by the Clients
1a1. System presents information message instead of the list
4a. User does not confirm his/her intent
2a1. Back to main scenario 1.

13.2 Additional information

13.2.1 Description of Validation identifiers

T{x,y} - minimum of x and maximum of y in a row of the type T
NUM - digits
ALFA – alphanumeric characters
CHECKED - checkbox is checked
NOT_DEFAULT_OPTION - valid value is selected. For example not the first value like "Choose..." which is there for information and is not meaningful choice
SELECTED - one of the radio button options must be chosen

13.2.2 Application types

This system supports the following applicatoin types: Debit Card Application, Credit Card Application and Application for loan increase.

13.2.3 Form Fields and Validation Rules

Debit Card Application form

- Card type: not editable
- Current account: select
- Card holder: not editable
- 24 h usage limit: text input (NUM{1,10})
- 24 h cash withdrawal limit: text input (NUM{1,10})
- I will take my card from: select
- I confirm that ... : checkbox (CHECKED)

Credit Card Application form

- Card type: select
- Current account: select
- Card holder: not editable

- Credit limit: text input (NUM{1,10})
- 24 h usage limit: text input (NUM{1,10})
- 24 h cash withdrawal limit: text input (NUM{1,10})
- I will take my card from: select (NOT_DEFAULT_OPTION)
- I confirm that ... : checkbox (CHECKED)

Application for Loan Increase form

- Number of loan agreement: select
- Desirable amount of the increase: text input (NUM{1,10})
- By increasing loan, I would like to: radio (SELECTED)
  - Keep the same loan term:
  - Increase loan term by # month(s): text input (NUM{1,2})
- I confirm that ... : checkbox (CHECKED)

Login form

- Username: text input (ALFA{6,10})
- Password: text input (ALFA{6,10})