Test-Driven Web Application Development in Java

Jens Uwe Pipka

Daedalos Consulting GmbH
Ruhrtal 5, 58456 Witten, Germany
Tel. +49 2302 979 0
jens-uwe.pipka@daedalos.com

Abstract. Today, software availability as well as adaptability has a strong impact on application development. Together with the success of the Internet technology, it is often necessary to offer a web-based software solution. For Java development, J2EE builds a bridge from traditional towards web application development. Java Server Pages and Servlets allow the implementation of web sites with dynamic content. Anyway, it is still possible to apply well-tried programming paradigms, e.g. the Model-View-Controller (MVC) paradigm.

With regard to software quality as well as adaptability, application development is affected more and more by advanced testing techniques. Especially unit testing provides a powerful technique to develop new functionality as well as to extend and reuse existing parts. This is also strengthened by the success of agile processes that force test-driven development. From there, application code grows up with unit tests from scratch on. For Java, JUnit has become the de-facto standard unit test environment.

Nevertheless, test-driven web application development requires broadening the unit test approach. Considering the special requirements of web application development, both the scope of the unit tests as well as the JUnit framework itself have to be extended to enable a test-driven development. In this paper, we present an approach to customize the unit test cycle as well as the JUnit framework towards a test-driven web application development.

1 Introduction

The flexibility and availability of the World Wide Web has great impact on software development. It is increasingly important that applications have to be delivered in good time. For web applications, it is essential to be highly customizable as well as adaptable to meet changing requirements.

Moreover, the usage of HTTP as a stateless protocol and HTML as the client language is quite different from traditional client as well as client/server development. The same applies to test processes and tools. So, it is necessary to adapt existing test processes to meet the special requirements of web application development. To go on,

an appropriate test environment including the right tool support has to be set up to enable a test-driven web application development.

In the following, we start with a short presentation how to implement web applications using J2EE. We present two different kinds of web application architecture: First, we give a broad overview of the JSP page centric approach. Second, we show how web applications can be implemented using the Model-View-Controller design paradigm, the so-called "MVC Model 2". This is based on Java Server Pages (JSP) combined with servlet technology. In this model, we distinguish between using a single servlet and using individual servlets for each client request.

Based on these variants, we present an approach for test-driven web application development. This includes suitable test strategies as well as test tools for the whole development cycle. Furthermore, we show how a test-driven design can be applied consequently through all development stages. Finally, we discuss how applicable the test-driven web application development is and how to handle larger projects using this approach.

2 "Best Practices" for Developing Web Applications using Java

In this chapter, we present today's most important best practices for developing web applications using Java. First, we consider the pros and cons of the JSP page centric approach, also known as "Model 1". Second, we introduce the "MVC Model 2". We distinguish between Single-Servlet architecture and using an individual servlet for each client request. Finally, we discuss why we have chosen the "MVC Model 2" to apply test-driven development.

2.1 Model 1: A Page-Centric Architecture

Since a couple of years, the client-server approach is used to built applications. This architecture is based on an application running on a client that needs to connect to a server-side application to provide its specific functionality. This simple two-tier model was adapted to implement web pages with dynamic content. In the beginning, the Common Gateway Interface (CGI) was the first tool to provide dynamic web content.

As dynamic web pages were more and more successful, it was necessary to adapt this concept to Java. With the introduction of servlets, it was also possible to implement those simple two-tier applications in Java. But this suffers from the fact that web pages have to be created by the servlet itself. So, HTML code was embedded directly inside the programming code. Java Server Pages (JSP) change that: Instead of embedding HTML in programming code, application code can be embedded into HTML pages [1].

So, JSPs can access for example resources like databases as well as other resources to answer to a client request directly. HTML code is enriched with Java code to implement certain business logic. Incoming client requests are intercepted and

processed inside the JSP directly. Furthermore, the JSP page is also responsible for the response that is sent back to the client. This page centric approach is also known as the "Model 1" programming approach [2].

The mixture between Java application and HTML presentation code leads to complex pages that are not maintainable any more. This gets even harder with the growing demands towards functionality and flexibility of web pages. The introduction of Java worker beans to reduce business logic and data access code inside the JSP code as well as JSP tag libraries solves only parts of this problem, because the main flow control still resides inside the JSP.

Since this page centric architecture usually leads to a significant amount of Java code embedded within the JSP page, it is only suitable for simple web applications. It is not desirable for complex implementations, especially if there is a significant amount of request processing to be performed. As we will show in the following, it is a better approach to combine JSP with other Java technologies.

2.2 Model 2: A Model-View-Controller Architecture

The main difference between the page centric architecture and the MVC architecture is the location at which the request processing is performed. As pointed out, in the page centric architecture the JSP page alone is responsible for processing incoming requests and replying them back to the client.

It is necessary to change the MVC paradigm, because web applications are based on another presumptions as traditional applications. Most important, it is not possible for the model to notify the view about changes, because connection between client and server is stateless. So, the web browser would have to re-query the server to keep informed about changes inside the model. Web applications are also based on a client browser: Triggered from a HTTP request created by the client browser, the web application has to respond with a HTTP response.

So, a slightly modified Model-View-Controller architecture also known as Model 2 for building web applications using Java technologies was introduced:

- *Model*: The model could be divided in two parts: First, it provides the business logic; second, it represents application state. The business logic updates the application state. The model knows nothing about the controller or view. In the context of web application development, the model can be implemented by using Java beans.
- *View*: The view provides the look and feel of the application. The view can only access the current state of the model; it knows nothing about the controller. Using Java web technology, the view is based on JSP pages.
- *Controller*: The controller receives the user input from the web browser and dispatches it. In a Java web application, servlets take the part of the controller.

Introducing this architecture, it is obvious that different technologies as well as concepts are used for implementation. Therefore, it is possible to implement the business logic using pure Java; the web presentation layer is implemented using JSP

based on HTML code. As we show later on, you can distinguish between the single servlet and the individual servlet architecture [3].

Properly applied, the MVC architecture forces the development of well-structured web application. The processing logic that is implemented using Java code is detached from the presentation layer that is based mainly on HTML code inside JSP pages.

As shown, it is possible to build a web application using the slightly modified MVC paradigm. In short, the Model 2 architecture is made of servlets processing incoming requests; Java beans providing the business logic and JSP pages handling the reply back to the client. An example for the Model 2 architecture is "Struts", an open source JSP and servlets framework to manage complexity in large web sites [4].

2.2.1 Single Servlet Architecture

The Model 2 can be applied using single servlet architecture. This means, that every request is processed by one and the same servlet. For each request, the relevant input as well as the initial flow control has to be prepared. Normally, this architecture is implemented using a command broker that dispatches the incoming requests to the specific command objects. Nevertheless, the servlet has to create the initial context.

For complex applications, this approach could be too restricted, because the initial flow control and context initialization lies within one servlet. In this case, it is better to use the individual servlet architecture.

2.2.2 Individual Servlet Architecture

In the individual servlet architecture, each client request is assigned to a specific servlet. For each view, a specific servlet exists that creates the processing context and provides the initial flow control. This approach is suitable for complex applications, especially with different processing logic. In this case, each concrete servlet implementation can decide what individual context information is relevant for processing a specific view.

2.3 Web Applications Development and Testing

In this paper, we focus on Web Application Development following the Model 2. In contrast to the Model 1, its structure is perfectly suited to support testing during all development stages. Furthermore, it is possible to adapt many well known testing techniques to support a test-driven development.

In the following, the specific testing techniques that are needed for web application development are presented with respect to the individual as well as to the single servlet architecture. As we will show, nearly the same testing approach as well as testing techniques can be applied for both variants.

3 Web Applications and Test-Driven Development

To support a test-driven design from scratch on, the implementation follows the Model 2. Therefore, a web application is divided into different parts according to the Model-View-Controller paradigm as presented in the previous chapter. This is done for both Model 2 variants, the single servlet as well as for the individual servlet architecture.

Using the single servlet architecture, the entry servlet is the crucial point for further processing and therefore a very important entity. It is essential to provide suitable and comprehensive unit tests for this servlet. Using the individual servlet architecture, the number of servlets is continuously growing during development. So, it is important that the servlet implementation grows up with the corresponding test cases. Consider for example that the response of a servlet should be redirected dependent from a specific input. Without existing unit tests, it is not possible to validate if the modification also affects other inputs.

According to the Model 2 architecture, it is possible to divide the test activities also in three different test scenarios:

- *Model Testing*: The model is implemented using Java beans. The implementation can be tested using traditional unit tests. These tests can be run independently from the rest of the web application.
- *View Testing*: The output of the JSP and HTML code has to be verified as whole. Additionally, the client interaction between browser and server side has to be tested. It is necessary to use functional tests on client side that simulates client request and check the server response.
- *Controller Testing*: Inside the servlet, the request processing including the flow control and the context initialization has to be tested. So, it is necessary to simulate the specific request behavior of the servlet.

With respect to a test-driven web application development approach, it is necessary to divide the development process in three different steps using a specific order:

1. First, the model is implemented, i.e. the state as well as the business logic. It is important to define a clear interface that can be used from the controller to call the business logic as well as from the view to access the state.
2. Based on this interface, the controller is implemented next. It provides the processing of the input as well as the basic flow control.
3. Finally, the view is implemented using the interface to the model to read and if necessary to display the application state.

It is obvious that each task require its own test strategy. Nevertheless, it is also important to supply a homogenous environment for each test scenario. In the Java world, JUnit has become the de-facto standard for traditional unit tests. Combined with HttpUnit and ServletUnit as additional extensions, JUnit is also capable to support the test-driven development of web applications.

In the following, we present at first our test environment for developing web applications in Java. Then we continue with the description of the different test scenarios as mentioned above.

3.1 Web Test Environment

As base for the test environment serves JUnit, the de-facto standard for unit testing in Java [5]. Implementing and running unit tests using the JUnit framework has many advantages. With JUnit, testing is closely integrated with development and test code is coupled tightly to application code. So, it is possible to build test suites incrementally that will help to focus on the development efforts as well as to identify errors. Furthermore, JUnit itself is implemented in Java and available as source code. Hence, it could be extended easily.

With respect to web testing, there exist several extensions for the JUnit Framework. The best-known extensions are HttpUnit/ServletUnit and Cactus [6]. Cactus is part of the Apache Jakarta project and enables JUnit to test JSP and servlet requests. The main difference between Cactus and ServletUnit is how they access servlets: Cactus enables to test a servlet in a real servlet container, i.e. the tests have to be deployed into the container along with the servlets. Instead, ServletUnit uses a simulated servlet container. So, the tests can be developed and executed on any client workstation. It is not necessary to deploy the test cases on the server. More, HttpUnit/ServletUnit test cases are run directly on the client but use the remote servlet resp. the generated http response. Therefore, all components between the request and response have to be accessed together with the actual test case. Compared with this, Cactus is the more unit-like test tool, because the test code are coupled more tightly to the application code. On the other hand, the HttpUnit/ServletUnit approach allows to extend the tests to html- as well as JSP output directly.

Cactus as well HttpUnit/ServletUnit are open source software and can be downloaded from [5] resp. [7]. In the following, we focus on JUnit together with HttpUnit and ServletUnit. We use these extensions here, because they provide an easy understandable but nevertheless powerful functionality to test both, JSP pages as well as servlets. Furthermore, the test environments is set up easily and the test code implementation is coupled very tightly with the application code implementation.

JUnit builds the base for all further test activities. First, it is used out of the box to test Java beans. Combined with HttpUnit, JUnit is used to test JSP pages. Furthermore, JUnit and ServletUnit serve to test servlets. By it, the whole test cycle is based on one common test environment. This simplifies the creation as well as the execution of automated test cases, since during the whole development process, similar testing techniques are applied.

3.2 Model Testing

First, we start with implementing the model providing the business functionality and the state of the application. If the model implementation is based on Java beans as we propose it in this paper, it can be tested using traditional unit tests. For test-driven development, it is essential to start with writing unit tests and to implement the corresponding application code afterwards. Enforced by this “Test First, by intention”, application code grew up with unit tests [8]. Applying this approach

consequently leads to build a stable base for the existing code, because all functionality is tested and it can be verified at any time.

Using the JUnit framework, this can be done easily: Tests are defined as subclasses from the basic JUnit class TestCase and implemented in pure Java. JUnit provides different TestRunner to run the tests and to display their results. Furthermore, different tests can be combined to a TestSuite. For an introduction to JUnit, see the JUnit documentation [5].

Simplified, the test definition and execution can be summarized as follows:

- Subclass junit.framework.TestCase.
- Implement the test methods, e.g. testMyBusinessLogic():
  - To check a value, call assertTrue(expression) and pass a boolean that is true if the test succeeds. Or, call assertEquals(expected, actual) and pass the expected and compare it with actual result.
  - Run the unit tests using a JUnit TestRunner. In the case of an unsuccessful test JUnit reports the failed tests.

Using the test first approach, clear interfaces are defined during the development cycle. The tests design the interface to the functionality. The business logic is called from within the tests as it will be called in the application code later on.

This procedure is well known from traditional application development. With the test first approach, the system grows up with unit tests and its functionality can be tested at any time.

This approach is also applicable during the integration of external applications: For each interface of the external application that is used from within the application, corresponding unit tests have to be defined. This makes it possible to verify if the expected result is the actual result. Furthermore, a modified version of an external application can be integrated without the risk that this leads to a changed system behavior.

3.3 Controller Testing

After implementing the specific business and state logic, we can go on with implementing the controller. The controller implementation relies on a specific subclass from javax.servlet.http.HttpServlet.

To test a specific servlet, we use JUnit combined with ServletUnit. ServletUnit is no real unit test tool, but it provides an interface to access a servlet directly. Hence, ServletUnit supports unit tests by providing a simulated servlet container. It is possible to handle a request in steps. The result of a request is not a final response but an invocation context. It includes the request and response objects of the servlet. Therefore, it is possible to call the servlet's methods directly. Furthermore, the current request and response objects as well as the session context are accessible. This is important to initialize a specific request as well as to test the current state of the processing.

To define and execute a servlet test, the following steps are necessary:

- Subclass junit.framework.TestCase.
- Implement the test methods, e.g. testMyServlet():
  - Instantiate com.meterware.servletunit.ServletRunner, the simulated servlet container, and register your servlets.
  - Create your web request and set your request parameter, e.g. inputs.
  - Ask the client for an invocation context.
  - Test your servlet including the request and response objects as well as the session context, using the standard unit test methods assertTrue(expression) and assertEquals(expected, actual).
- Run the unit tests using a JUnit TestRunner. In the case of an unsuccessful test JUnit reports the failed tests.

It is obvious that the structure of a servlet test is similar to a traditional unit test. Despite of the servlet specific test actions, the steps are identical to a common unit test. This simplifies the process of web application test. Everyone who is familiar with writing unit tests can write servlet tests quite easily.

It is also possible to implement the tests before writing the specific servlet processing. To force this, we introduce the following servlet structure: First, the doGet(request, response) and doPost(request, response) methods both call performTask(request, response). This method dispatches the incoming requests and is implemented as follows:

```
public void performTask(
    HttpServletRequest request,
    HttpServletResponse response) {
  ServletContext sc;
  RequestDispatcher rd;
  try {
    String targetURL =
      this.startProcessing(request, response);
    sc = getServletContext();
    rd = sc.getRequestDispatcher(targetURL);
    rd.forward(request, response);
  } catch (Throwable theException) {…}
}
```

Using this structure, it is possible to test the servlet behavior itself. It is not necessary to take care of forwarding or error handling. Instead, this is part of the functional tests we will introduce in chapter 3.4 View Testing. However, it is also possible to enhance the servlet test itself if necessary.

The main entry point is the startProcessing(request, response) method. It serves as the specific dispatcher for the servlet. Based on the client input, the corresponding servlet action is called. Inside the servlet action, the relevant input is extracted from the parameters, the application context is initialized and the business logic is called. After processing of the business logic is finished, the current state is interpreted and the appropriate response for the client is initialized.

The results of the individual servlet actions can be verified in our servlet test. So, it is possible to define a specific action in the tests first, initializing the test input for the request and comparing the expected response with the actual one provided by the servlet. After implementing the test, the action itself can be defined and verified instantly with the existing test.

3.4 View Testing

Finally, the view and with it a JSP page has to be tested. This is done by functional or black box testing of the web application. Compared to the previous test steps, it is typical that these functional tests cannot follow the test first approach any more. Nevertheless, this means no restriction to our test-driven approach, because the JSP pages are restricted to present the response. Inside the JSP page, there is no further request processing, model manipulation or flow control.

To test the view representation of a web application, the HttpUnit extensions to JUnit are used. Despite of its name, HttpUnit is no test tool but a programmable web client. Using the HttpUnit interface, it is possible to define common actions that normally occur when working with a client web browser. So, it is possible to define several different orders of events to test a web application using common unit test functionality.

The definition and execution of this test scenario is quite similar to the previous ones:

- Subclass junit.framework.TestCase.
- Implement the test methods, e.g. testMyWebPage():
  - Instantiate com.meterware.httpunit.WebConversation. This is the center of HttpUnit that is similar to a browser session.
  - To use it, a WebRequest has to be created. Then, the WebConversation is asked for the WebResponse.
  - The response can be used to verify the tests; this can be done again the basic unit test methods assertTrue(expression) and assertEquals(expected, actual). HttpUnit provides a complete interface to access the response including the complete HTML structure as well as individual elements, e.g. tables or forms.
  - Furthermore, it is possible to access and fill web forms, to follow links and to use cookies. So it is possible to create and verify entire test sequences over several web pages.
  - Create your web request and set your request parameter, e.g. inputs.
- Run the unit tests using a JUnit TestRunner. In the case of an unsuccessful test JUnit reports the failed tests.

It is important, that every JSP page that is introduced in a web application has to be tested. Furthermore, this test should be automated to repeat it every time something is changed. It is essential that the definition of a new JSP page lead to a new view test. Applying this consequently, no part of the web application remains untested. It can be

verified at any time, if an extension or a change inside the web application has side effects on a view.

Nevertheless, it is also important that the quality and quantity of view tests grow up with the web application. At the beginning, the view tests concentrate on a specific view rather than on sequences over several web pages. With the introduction of new views that are connected with other views, it is essential to define additional view tests that verify the interaction between the web pages.

4 Test-Driven Web Application Development in Practice

As an example for the proposed test-driven web application development, we have implemented a web-based user authentication as known from various websites. The use-case for this example is quite simple:

- Starting point is a web site that asks for the username and password. The input can be submitted or reset.
- If the input is correct and the user can be authenticated, a welcome page with some user related information is shown.
- If the user is not found or a wrong password is given, a specific error page should appear. If the user wants to try again, he can go back to the starting point.

Now, we implement this example. We start with dividing our web application in different parts following the MVC pattern using a single servlet. For our web application, the application is divided as follows: A static HTML page takes the user input and send it as a request to a controller servlet. The controller servlet processes the input and forward it to the model. There, the authentication itself is realized using classical Java worker beans. Dependent from the processing, the output is finally generated by a specific view that is provided as a JSP page.

Test-Driven Model Implementation

The underlying model is an authentication component. For simplicity, we assume that the class Login encapsulates the authentication specific implementation, e.g. access to a corporate group and user management. We start with defining the appropriate unit tests that define the interface to perform the user authentication. Therefore, we assume that the method validate(username, password) returns true if the login is correct and false otherwise. Accordingly, the JUnit test case looks as follows:

```
public class LoginTest extends junit.framework.TestCase{
  private Login login;
  public LoginTest(String name) {
    super(name);
  }
  public void setUp() {
    login = new Login();
  }
```

```
    public void testValidLogin() {
      Login login=new Login();
      this.assertTrue(login.validate("nod", "secret"));
    }
    public void testInvalidLogin() {
      Login login=new Login();
      this.assertTrue(login.validate("nod", "")==false);
      this.assertTrue(login.validate("foo", "")==false);
    }
  }
```

After implementing this test case, the Login class itself can be defined. As already pointed out, this class simply encapsulates an existing component, so we renounce to present the source code in detail here. After all unit tests are successful, we can go on with the implementation of the controller servlet.

Test-Driven Controller Implementation

Next, we are going to implement the controller and according to it the controller servlet. To perform the authentication, we design a HTML page that contains a form to take the username and the password. The LoginServlet processes the request from this HTML page, to extract the input values and pass them to the authentication bean. Finally, dependent from the result, the output is set.

Again, we start with writing the appropriate unit test. Because now we have to test a HttpServlet, it is necessary to use JUnit together with ServletUnit. The implementation is straightforward: We start with creating a new ServletRunner that provides the simulated servlet container and register our servlet. Next we create a test-specific WebRequest that contains the parameters username and password. Then, the WebRequest is invoked together with the LoginServlet inside a ServletUnitClient. As a result, the InvocationContext includes the current state of the LoginServlet instance after processing this WebRequest. It is now possible to verify the state of the LoginServlet instance, e.g. to call servlet specific methods or to check the servlet state. For the unit tests, the same JUnit functionality can be used as before for traditional test cases.

In this example, the return value is the String “/welcome.jsp” if the login is successful. Otherwise, the string “/error.jsp” is returned. Accordingly to the unit tests before, it is possible to define test cases for the different login scenarios. In the example shown below, we implement the test case for a successful login, i.e. username and password are correct. Using the performTaks(request, response) method as presented in the previous chapter, we expect that a method validateLogin (request) is called that performs the user authentication and sets the output. Because we know that the login is successful, we check if the return value of validateLogin is “/welcome.jsp”. To save space, we only show the implementation for this case. Consider that this is only an extract from the complete implementation; it is also necessary to test the other test variants.

```
import com.meterware.httpunit.*;
import com.meterware.servletunit.*;
```

```
public class LoginServletTest
  extends junit.framework.TestCase {
  …
  public void testLogin() throws Exception {
    ServletRunner sr = new ServletRunner();
    sr.registerServlet(
      "/servlet/LoginServlet",
      LoginServlet.class.getName());
    ServletUnitClient sc = sr.newClient();
    WebRequest request = new PostMethodWebRequest(
      "http://myhost/servlet/LoginServlet");
    request.setParameter("username", "nod");
    request.setParameter("password", "secret");
    InvocationContext ic = sc.newInvocation( request );
    LoginServlet login = (LoginServlet)ic.getServlet();
    this.assertEquals(
      "/Welcome.jsp",
      login.validateLogin(ic.getRequest()));
  }
}
```

After defining the unit test, the LoginServlet class can be implemented. We begin we implementing the method startProcessing(request, response) as mentioned before with respect to the specific behavior of the LoginServlet. As already defined by our test case, it is necessary to check the user authentication functionality. Accordingly, the validateLogin(request, response) method is called. The implementation is as follows:

```
public class LoginServlet extends HttpServlet {
  public void performTask(
     HttpServletRequest request,
     HttpServletResponse response) {…}
  public String startProcessing(
     HttpServletRequest request,
     HttpServletResponse response) {
    return validateLogin(request);
  }
}
```

Next, the validateLogin(request) method has to be defined. The implementation is quite easy: The parameters are extracted from the request and passed to the Login class. If the authentication is valid, the return value is set accordingly. Also, the user specific information is retrieved and saved in the session context.

```
public String validateLogin(HttpServletRequest request){
    String name = request.getParameter("username");
    String password = request.getParameter("password");
    Login login = new Login();
    if (login.validate(name, password)) {
      User userInformation = login.getUserInformation();
      this.setUserInformation(request, userInformation);
      return "/Welcome.jsp";
    }
    return "/error.jsp";
  }
}
```

Now the test is run and successful. So the next step towards our web application is done. The login functionality of the servlet is already tested. So, we can go on with developing the view component.

Test-Driven View Implementation

Finally, we want to test the view. In our example, we focus on the successful login. In this case, on the welcome page a specific message showing the complete user name should be displayed in the title as “Welcome, <complete user name>!”. The complete name is retrieved from the Login class based on the username. So, we start again with defining our test case.

For the view test, we use HttpUnit that enables the functional test of the generated output. Furthermore, the view test can be extended to test whole request-response sequences as rather a single response. This is anyway necessary, because the dynamic content of a specific page is in most cases dependent from the previous processing.

The use of HttpUnit is quite similar to the one of ServletUnit. We start with creating a new WebConversation instance that takes the place of a browser talking to a single site and maintains session context. Next, a WebRequest for the Login HTML page is created and the WebConversation asked for a response. This response is modified with respect to our test case, i.e. the input form is identified and the input fields for username and password are set accordingly with the values for a valid login. This modified WebRequest is sent to the WebConversation again. The response can now be used to verify the processing status. As you see in the example code below, the standard JUnit functionality can be used as before to define the tests.

In our example, the response should be the welcome page that contains the specific title “Welcome, NetObjectDays User!” if the implementation works correctly. Consider, that this is again only an extract from the original source code that just presents the test for the successful login.

```
import junit.framework.*;
import com.meterware.httpunit.*;
public class LoginPageTest extends TestCase {
   …
   public void testValidLogin() throws Exception {
    WebConversation wc = new WebConversation();
    WebRequest request = new GetMethodWebRequest(
      "http://myhost/Login.html");
    WebResponse response = wc.getResponse( request );
    WebForm form = response.getForms()[0];
    request = form.getRequest();
    request.setParameter("username", "nod");
    request.setParameter("password", "secret");
    response = wc.getResponse( request );
    this.assertEquals(
      "Welcome, NetObjectDays Guest!",
      response.getTitle());
   }
}
```

For simplicity, we only test the page title in this example. Despite from this, HttpUnit also enables to work with forms and frames as well as to examine and follow links or to inspect tables. Furthermore, the whole content of the inspected web site is available as plain text or as a domain object model (DOM) tree. So, almost every kind of content can be tested.

After defining our test case, it is now the time to implement our welcome page. Because this is a dynamic web page that contains user specific information, we implement it as a JSP page.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<% User user = (User)request.getAttribute("user"); %>
<HTML>
<HEAD>
   <TITLE>Welcome, <%=user.getName()%></TITLE>
</HEAD>
<BODY>
…
</BODY>
</HTML>
```

Again, this test case runs successfully. So, the test-driven implementation of the web-based user authentication is finished. Clearly, we could only present a small extract from the possibilities of test-driven development here. Nevertheless, this example provides a good start to extend the test cases as well as the implementation. Most important, it is always the test case implementation that is the starting point to define functionality. The same approach is applied if a defect is detected: First, an appropriate regression test case is defined. Only afterwards, the implementation itself is changed.

5 Conclusion

As seen, web application development is quite different from traditional application development. Extended unit test strategies as well as appropriate test tools are essential to support a test-driven design.

In this paper, we show how test-driven web application development in Java is possible. For this, we introduce the Model 2 architecture for web application development following the MVC paradigm. Based on this architecture, we present specific techniques and tools to enable unit testing in each development stage. Applying these techniques, a test-first approach and therefore a test-driven web application development is enforced.

If the test first approach is applied consequently, you get a fully testable as well as tested web application. Every part of the application can be tested automatically at any time, no matter if you want to test model, view or controller. By it, the whole web application becomes more maintainable, extendable and stable. Changes can be made without the risk of side effects. Furthermore, the interfaces of the different components are separated clearly, i.e. business logic, servlet and JSP page are used in a specific manner. For that reason, the interfaces are defined clearly. Furthermore, it is

possible to test each part and with it the implementation becomes more reliable in the end.

The proposed testing techniques rely on JUnit, the de-facto standard for unit testing in Java. So, a developer can still use a familiar unit test environment. Furthermore, existing application as well as test code can be integrated easily.

The only gap between proposed test-driven web application and the suggested techniques lies within view testing. Currently, there are no appropriate tools as well as strategies to develop views using a test first approach. So, only functional testing is supported yet. As shown, this actually does not have any serious restrictions. Nevertheless, our future work is to fill this gap and to enable a continuously test-driven development of JSP pages including appropriate tools.

References

1. Java Servlet Technology, see http://java.sun.com/products/servlet/whitepaper.html, 2002.
2. Malks, D. et al.: Professional JSP. Wrox Press Ltd., 2001.
3. Hettel, J.: “Best-practice”-Architekturen für Web-Anwendungen, NetObjectDays 2001, Erfurt.
4. Davis, M.: Struts, an open-source MVC implementation. See http://www-106.ibm.com/developerworks/library/j-struts/index.html, 2001.
5. JUnit test framework, http://www.junit.org.
6. Cactus test framework for unit testing server-side java code, see http://jakarta.apache.org/index.html.
7. HttpUnit/ServletUnit, http://httpunit.sourceforge.org.
8. Jeffries R., Anderson A., Hendrickson C.: Extreme Programming Installed. Addison Wesley, 2000.