# Fast Feedback Loop: Unit Testing Strategies for Tapestry

BC Holmes

Director, Architecture and Technology
Intelliware Development, Ltd.
http://www.intelliware.ca

Session TS-7354

## Goal of This Talk

Learn how to test Tapestry pages outside of the web container, resulting in better tests and better Tapestry applications.

## Agenda

Quick Primer on Background Info

Testing and Test-Driven Development

Basic Tests

Tapestry Test Support

Analyzing Page Output

Stubbing/Mocking Dependencies

Putting It All Together

## Agenda

Quick Primer on Background Info

Testing and Test-Driven Development

Basic Tests

Tapestry Test Support

Analyzing Page Output

Stubbing/Mocking Dependencies

Putting It All Together

## Things That (I Hope) Everyone Knows

Background information for this session

- Tapestry is a framework for creating web applications
- Tapestry applications usually don’t use JavaServer Pages™ (JSP™ page), relying instead on its own “Page Components”
- Testing Tapestry applications is usually accomplished in the same fashion as other web applications
    - Use a testing framework to send HTTP requests at the Java™ 2 Platform, Enterprise Edition (J2EE™ platform) server

## Tricky Things About Testing Tapestry

- It’s easy to test a page method
  - Especially in Tapestry 5, because the pages are no longer abstract
- It’s harder to test that the page renders correctly
  - Rendering is accomplished as a negotiation between the Tapestry Framework, the HTML “template”, and various configuration options in the Page class
  - There is no single class that has responsibility for the entire render process for a particular page

## Typical Web Application Testing Frameworks

- Commonly-used testing frameworks
  - HttpUnit
  - HtmlUnit
  - Canoo
  - IeUnit (more rarely)
- Each of these frameworks performs “in server” testing
  - The web application is first deployed to the server
  - The framework emulates getting pages, and filling in forms

## Agenda

Quick Primer on Background Info

**Testing and Test-Driven Development**

Basic Tests

Tapestry Test Support

Analyzing Page Output

Stubbing/Mocking Dependencies

Putting It All Together

## Is XP Testing About Testing?

A quotation

“When XP uses the words ‘test’ and ‘testing’, they are (increasingly) bound up with different assumptions [than QA testers]. XP testing is about facilitating the act of programming, about process rather than product. XP tests provide value because they speed and smooth the programming process by helping people think through what they’re about to do and giving them feedback as they do it.”

Source: Brian Marrick, Front Royal Mailing List

## Saying It a Different Way

- Agile testing is not the same type of activity as QA testing
  - Different goals
    - e.g., promoting courage to allow a developer to make a radical change to a design
- Having said that, there are still insights that can be gleaned from the experts in QA-style software testing

## Also Worth Noting
### Test-Driven Development

“**Test-Driven Development** (TDD) is a software development technique that involves repeatedly first writing a test case and then implementing only the code necessary to pass the test…

Practitioners emphasize that test-driven development is a **method of designing software, not merely a method of testing.**”

Source: Wikipedia entry on Test-Driven Development

## Thesis

The case for out-of-container testing

- In-server testing strategies do not support test-driven development
- The practice of creating in-server tests exists to enable change detection
  - As such, in-server testing exists as a regression-testing technique, not as a design technique
- Out-of-container testing is far more viable with Tapestry than most other web frameworks

## Pop

### Testing for failure

- Cem Kaner coined the term “Pop” to describe a quality of good tests
  - Refers to the ability of a test to reveal things about our basic assumptions
  - Named after the philosopher, Karl Popper
- Karl Popper asserted that good conjectures allow us to imagine cases in which the conjecture fails
  - Kaner, then, argues that it’s more important to test for the purpose of making the product fail, than to make the product work

## Compare Testing Styles

Based on practices I’ve observed

- JUnit/test-driven development
  - Write the test
  - Make it compile
  - Run to get the red bar
  - Write code to get the green bar
  - Repeat
- In-server testing
  - Write the artifacts (pages, web.xml, etc.)
  - Build and/or deploy
  - Start the server
  - Tweak until it works
  - Create a test case to ensure that future changes are detected

## Five Weaknesses of In-Server Tests

- They’re slower to execute
  - Because they’re slower, developers don’t run them as often
- They’re not as thorough
  - “That’s probably good enough…”
- The test is less isolated
  - more “integration” and less “unit”
- They’re more fragile
- They’re harder to troubleshoot

## Test Characteristics Affect Behaviour

Comparison of test characteristics on real projects

| | JUnit | In-Server |
|---|---|---|
| **Project One** | | |
| Number of tests | 1035 | 87 |
| Time to run | ~ 2m 37s | ~ 5m 28s |
| **Project Two** | | |
| Number of tests | 495 | 20 |
| Time to run | ~ 59s | ~ 5m |

Source: Two recent projects

## Prerequisite Factors

- Developers require fast feedback in order to use that feedback to make design decisions

# Features of Good Tests

- Power
- Valid
- Value
- Credible
- Representative
- Non-redundant
- Motivating
- Performable
- Maintainable
- Repeatable
- Pop
- Coverage
- Easy to evaluate
- Supports troubleshooting
- Appropriately complex
- Accountable
- Cost
- Opportunity cost

Source: Cem Kaner and James Bach

## Agenda

Quick Primer on Background Info

Testing and Test-Driven Development

**Basic Tests**

Tapestry Test Support

Analyzing Page Output

Stubbing/Mocking Dependencies

Putting It All Together

## Two Basic Tests

- Objective: to use testing to help eliminate all the basic typo problems
  - You can spend a lot of time fussing with getting the JWCIDs
- These tests are “structural”

## First Basic Test

- Create a test that:
  - Walks through your source code directory
  - Finds all Tapestry page classes
  - Finds corresponding .html files
  - Parse the .html files, and extract all JWCIDs
  - Reflect the page class, and assert that each JWCID has a corresponding component
- Part of the appeal of this test is that you write it once for your project, and you can keep getting value out of it each time you add a page

## Typical Implementation

```
interface Assertion {
  public void performAssertion(
      File file, Document html, Class pageClass)
      throws Exception;
}

public void testJwcids() throws Exception {
  iterateOverAllPages(new Assertion() {

    public void performAssertion(
        File file, Document html, Class pageClass)
        throws Exception {

      assertAllJwcidsHaveCorrespondingClassProperties(
        file, root, pageClass);
    }
  });
}
```

## Second Basic Test

- Similarly, use reflection to ensure that OGNL property paths are correct
  - If you do a lot of refactoring, it’s easy to get OGNL expressions that no longer resolve correctly
  - A bit trickier to implement, but still quite viable

## But What Does This Buy You?

Is this type of testing really worthwhile?

- As described, it’s hard to see the overall value
- Usually, the “problems” they detect are problems that any competent person would notice while trying to start up the application
  - The test is probably faster to run, but that’s just splitting hairs
  - And there’s the standard regression argument, but we’ve already covered that
- Once in place, it’s easy to elaborate this type of test to include other “structural” checks

## Improving Your Basic Tests

Extending to detect the “Hard Stuff”

- Consider:
  - Localization
    - It’s surprisingly easy to forget to add appropriate bindings to Submit buttons

```
@Component(type="Submit",
      bindings={"value=message:login"})
public abstract Submit getLoginSubmit();
```

## Also…

Tests document design decisions

- Tests like this keep all members of the team on the same page, as far as design decisions go
  - Less worry about a new team member not being informed of a particular expectation or requirement

## Agenda

Quick Primer on Background Info

Testing and Test-Driven Development

Basic Tests

**Tapestry Test Support**

Analyzing Page Output

Stubbing/Mocking Dependencies

Putting It All Together

## Invoking the Render Cycle

Using JUnit tests to call up and render pages

- In order to test functionality on pages, we need to emulate sending requests at the Tapestry Servlet
  - But first we need to emulate the start-up and initialization of servlets
- We must consider the “unit” under consideration to be the combination of the Tapestry framework and the page
- One of the few frameworks for this type of test support is the ServletUnit framework
  - Made by the same folk who make HttpUnit

## A Mock Web Container

- If you don’t want to use ServletUnit, you can accomplish much the same thing by creating your own “stubs” (or “mocks”) of some key javax.servlet interfaces
  - HttpServletRequest
  - HttpServletResponse
  - HttpSession
  - HttpServletContext
  - HttpServletConfig
  - Perhaps also the filter equivalents

## Initializing the Tapestry Environment

```
private void setUpApplicationServlet() throws ... {
  if (servlet == null) {
    servletContext = new MockServletContext();
    servletContext.setWebRootDirectory(
        new File("./WebRoot"));

    setUpSpringEnvironment(servletContext);
    ApplicationServlet temp = new ApplicationServlet();
    MockServletConfig config =
        new MockServletConfig("app", servletContext);
    temp.init(config);

    // only assign this after the object
    // has been initialized
    servlet = temp;
  }
}
```

## Sending a Mock Request

```
MockHttpServletRequest httpRequest = createRequest(path);
MockHttpServletResponse httpResponse =
    new MockHttpServletResponse();

getApplicationServlet().service(
    httpRequest, httpResponse);

assertOk(httpResponse);
```

## Interesting Discoveries

Things you learn while testing

- The Tapestry environment for Tapestry 4.x takes about 6 to 7 seconds to initialize in our out-of-container test environment
  - That's a lot of time, all things considered
  - We only want to incur that cost once per suite
    - In our environment, we cache the Tapestry Application Servlet and reuse it across multiple requests
- External links tend to be more conducive to testing than direct links
  - In practice, this finding influenced how we tended to build applications

## Agenda

Quick Primer on Background Info

Testing and Test-Driven Development

Basic Tests

Tapestry Test Support

**Analyzing Page Output**

Stubbing/Mocking Dependencies

Putting It All Together

## Making Sense of the Output

### Interpreting the HTML in the output

- Typically, you want to be able to write meaningful assertions on the HTML output

## Example

```
public void testAttemptInvalidLogin() throws Exception {
  // set up a case where the password fails
  ...

  Form form = getForm(renderSimplePage());
  form.setInput("usernameTextField", "wrong");
  form.setInput("passwordTextField", "bad");

  post(form);

  assertTitle(root, "Login Page");
  assertFormValue("usernameTextField", "wrong");
  assertFormValue("passwordTextField", "");
  assertHasMessage(INVALID_PASSWORD);
}
```

## What Do You Need to Do This?

### Parse the Response HTML

- To make sense of the output, you probably want to parse the HTML into nodes of some sort
  - Standard XML Parser (if the output is XHTML)
  - CyberNeko HTML Parser
    - Used by HttpUnit and ServletUnit
  - JTidy (tries to “tidy” the HTML in addition to parsing)
- Create some utilities to perform basic traversal of nodes, and wrap these up as custom assertions
  - At this point, this strategy is essentially the same as the strategy of various in-server testing frameworks

## Agenda

Quick Primer on Background Info

Testing and Test-Driven Development

Basic Tests

Tapestry Test Support

Analyzing Page Output

**Stubbing/Mocking Dependencies**

Putting It All Together

## Spring Injection

### Stubbing out dependencies

- Tapestry allows you to inject an object directly into a page from Spring
  - These dependencies often provide important functionality that’s tested separately
  - You don’t want to include that functionality in your testing unit

## Spring Injection

### Sample InjectObject Annotation (Tapestry 4.x)

```
@InjectObject("spring:ReportService")
public abstract ReportService getReportService();
```

### Sample Inject Annotation (Tapestry 5)

```
@Inject("spring:ReportService")
private ReportService reportService;
```

### Corresponding Spring Definition

```
<bean id="ReportService"
  class="com.example.service.impl.ReportServiceImpl">
  ...
</bean>
```

## Mocking Out Dependencies

### Loading a “Mock” Spring file

```
XmlWebApplicationContext spring =
    new XmlWebApplicationContext();
spring.setServletContext(servletContext);
spring.setConfigLocations(new String[] {
    "/mock-spring.xml" });
spring.refresh();

servletContext.setAttribute(
    ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, spring);
```

### Mock Spring Definition

```
<bean id="ReportService"
  class="com.example.service.impl.MockReportServiceImpl">
  ...
</bean>
```

## Hand-Cranked Mock Objects?

### Stubbing out dependencies

- Here we've set up an alternate (mock) version of the ReportService in a "mock" application context file
    - But really, this is what cool tools like jMock are for
- It's hard to use jMock to inject objects into a Tapestry page because Tapestry controls the page lifecycle
- Idea
    - Use Spring's idea of a FactoryBean to help

## Revised Spring File

```
<bean id="ReportService"
    class="com.example.util.spring.JMockBeanFactory">
  <property name="primaryInterface"
    value="com.example.service.ReportService" />
</bean>
```

## JMockBeanFactory

```
public class JMockBeanFactory implements FactoryBean {

  private static final Map<String,Object> MAP =
      synchronizedMap(new HashMap<String,Object>());

  private Class primaryInterface;

  ...

  // This method is invoked when Spring initializes
  public Object getObject() throws Exception {
    return Proxy.newProxyInstance(
      this.primaryInterface.getClassLoader(),
      new Class[] { this.primaryInterface },
      new MockHolder());
  }

  ...
```

## Invocation Handler (Inner Class)

```
class MockHolder implements InvocationHandler {

  // this method is invoked when a page calls a method
  // on the dependency
  public Object invoke(Object proxy,
      Method method, Object[] args)
      throws Throwable {
    try {
      return method.invoke(getMockObjectFromMap(), args);
    } catch (InvocationTargetException e) {
      throw e.getCause();
    }
  }
}
```

## Test Code

```
public void testRenderNoReports() throws Exception {

  Mock mock = mock(ReportService.class);
  mock.expects(once())
      .method("getReportCount")
      .will(returnValue(0));

  ...

  // register the mock object in the factory
  JMockBeanFactory.register(
      ReportService.class,
      mock.proxy());
```

## Other Thoughts

Additional tricks for testing

- You could use some simple utility to auto-convert the normal applicationContext.xml file into a “mock” version using JMockBeanFactory entries
  - The key trick is being able to correlate the registered name of the bean with its primary interface
- You could also use jMock to mock the entire WebApplicationContext interface
  - Can be a bit tricky

## Agenda

Quick Primer on Background Info

Testing and Test-Driven Development

Basic Tests

Tapestry Test Support

Analyzing Page Output

Stubbing/Mocking Dependencies

**Putting It All Together**

## Last Little Details

### Handling state

- Tapestry stores important state in the HttpSession object
  - If we’ve created a stub implementation, then we can manipulate the state
  - Most common example, “the logged in state”

## Emulating a Logged-In User

```
protected void emulateUser(
    MockHttpServletRequest httpRequest, User user) {

  HttpSession session = httpRequest.getSession(true);
  UserContext userContext = new UserContext();
  userContext.setCurrentUser(user);
  session.setAttribute(
      "state:app:userContext", userContext);
}
```

## DEMO

Test Suite

## Summary

- We've discussed the way in-server testing often constrains the way developers use testing
- We've discussed some structural types of tests that can be used to handle fussy elements of dealing with Tapestry development
- We've show strategies for invoking Tapestry application pages outside of the container

## Q&A

# Fast Feedback Loop: Unit Testing Strategies for Tapestry

BC Holmes

Director, Architecture and Technology
Intelliware Development, Ltd.
http://www.intelliware.ca

Session TS-7354