Sample Chapter

JUnit Recipes

Practical Methods for Programmer Testing

J. B. Rainsberger

with contributions by Scott Stirling

MANNING

JUnit Recipes
Practical Programmer Testing Methods

by J.B. Rainsberger
Chapter 9

Part 2

Testing J2EE

Most "serious" Java development these days revolves around J2EE: enterprise components for Java. This part of the book explores testing techniques for the core J2EE technologies: servlets, JavaServer Pages, Velocity templates, Enterprise JavaBeans, and Java Messaging Service. It also covers concepts common to web application platforms: page flow, dynamic content, distributed objects, as well as the separation of application logic, business logic, and persistence. The recipes in this part of the book generally fall into two categories: refactoring towards testing J2EE designs, and dealing with legacy J2EE designs, where *legacy* generally means "you are afraid to change it." From this point forward, recipes build on the foundations in part 1, so you will see numerous references here to earlier recipes. In a way, the goal of the remaining recipes is to reduce every testing problem to a smaller number of recipes from part 1. Enjoy.

Designing J2EE applications for testability

We originally conceived this part as "JUnit and Frameworks," because there are a few general guidelines that govern how to test code that lives in a framework. With J2EE, frameworks are everywhere: servlets, EJBs, JMS message listeners—these are all objects we code within a component framework and execute in the context of containers. This raises two main issues: performance and dependency; and a testable design is one that manages both effectively.

The performance problem

The performance issue stems from the containers or the application server. Executing code inside a container incurs overhead that affects the execution speed of your tests. This has a negative impact on the testing experience, which Kent Beck mentions briefly in *Test-Driven Development: By Example:* "Tests that run a long time won't be run often, and often haven't been run for a while, and probably don't work." Ron Jeffries provides some interesting commentary on how we might behave if we can someday test our entire system, be sure those tests mean the system works, and all in less time than it takes to think about it.[1] One of the key benefits of testing is the refactoring safety net—the ability to make changes confidently because you can always execute the tests to see if your last change broke anything. One of the key practices in refactoring is to execute the tests after every change, to be sure that you have not injected a defect.[2] Because many refactorings involve several changes—say ten or more—a slow test suite discourages you from executing the tests frequently enough to realize most of the benefit of the refactoring safety net. This is all rather a roundabout way of saying that slow tests are more than a nuisance, and their productivity impact is more than just the extra time to execute the tests. It is worth investing considerable amounts of time in making tests fast enough that you will feel free to execute them at any moment. Many of the recipes in the chapters that follow address this issue by guiding you through refactorings that move your code away from the container, allowing you to execute most of your tests without incurring the overhead of an application server. This performance issue is related to the way your J2EE components depend on their environment.

---

[1] www.xprogramming.com/xpmag/expUnitTestsat100.htm

[2] Martin Fowler describes this in Chapter 4 of *Refactoring: Improving the Design of Existing Code*. See "The Value of Self-testing Code."

The dependency problem

A framework is based on the Hollywood Principle—"don't call us; we'll call you." Framework designers expect you to build components that depend directly on their code: your classes implement their interfaces, or extend their classes. They say, "You worry about the business logic and let us take care of persistence (or a web interface, or transactions)." It looks good on paper, but it leads to testing problems: in particular, if your business logic is built right into a servlet, then you cannot execute that logic without running the application server, which leads us to the performance problems of the previous paragraph.[3] There are two main approaches to this problem: mock objects and reducing dependency. It turns out that this is a source of some controversy among JUnit practitioners.

Mock objects—palliative care

We loosely define a *mock object* as follows: an object that you can use in a test to stand in place of a more expensive-to-use or difficult-to-use object. When people ask about testing and databases, most people offer this advice: "mock the database." That is, rather than use a real database in your tests, you should substitute a mock version of the database objects and let your business logic operate on those. They could use files for persistence, or they could ignore persistence altogether. The benefit is avoiding the expensive, external resource and the test execution speed problems that come with it. We use the term *mock objects approach*, then, to mean a general approach to testing that relies on mock objects to avoid the pain of testing against the real thing. Throughout this book—but especially when discussing J2EE components—we recommend the mock objects approach to test those parts of your application that integrate with expensive, external resources. We do not, however, recommend mock objects as the first and final solution to this problem. (For more about mock objects, see the essay "The mock objects landscape." There is more than meets the eye.)

NOTE *Terminology*—Members of the Programmer Testing community have been trying to converge on a narrow, accurate definition of mock object, as the term seems to mean different things to different people. They have coined the term *testing object* to refer to any object you would use for a test that you would not use in production code, roughly equivalent to the meaning of *mock object* we have just introduced. Using the community's definition, a mock object, then, would be a special kind of testing object—

[3] Altogether accidental alliteration.

one that allows the programmer to set expectations on how it is used and can verify itself against those expectations with a simple method invocation. This matches better with the concept as it was introduced in the Mock Objects paper.[4] The definition we have presented here merely matches the current common usage with which you may already be familiar, but we rather like the term *testing object*, as it more directly describes the underlying concept. We will try to use the two terms appropriately, but will invariably revert to long-standing habits—calling testing objects "mock objects"—from time to time.

Reducing dependency—the cure

Rather than reach for a mock object right away, we recommend reducing your dependency on the frameworks around you. Briefly, code your objects in such a way that you can "plug in" a J2EE-based implementation of the services it provides: persistence, web interface, transactions, and so on. If you store customer information in a database, then create a `CustomerStore` interface and have your business logic depend only on this interface. Now create the class `JdbcCustomerStore`, which implements `CustomerStore` and provides JDBC-specific services, interacting with the database. By separating your business logic from the persistence framework, you make it easy to test the business logic with predictable data and those tests are *fast*. Just provide an in-memory implementation of `CustomerStore`—what some might call a mock object, but we disagree[5]—and test your business logic using the `InMemoryCustomerStore`. If your business logic cannot tell the difference between the two implementations of `CustomerStore`, then by and large it does not matter which implementation you choose for your tests! As always, the devil is in the details, and we get into those details throughout the rest of this book.

It takes an investment in time and effort to make a J2EE design easy to test. The good news is that once you know the general principles, and especially if you let tests drive your work, then it becomes quite natural to build J2EE-independent components, and then integrate them with J2EE later. The goal is to make the integration with J2EE as thin as possible to minimize the number of tests you need to execute against the live container. This maximizes the benefit of testing your code with JUnit. Unfortunately, there are times when this is simply not an option.

---

[4] Steve Freeman, Steve Mackinnon, Philip Craig, "Endo-Testing: Unit Testing with Mock Objects," www.connextra.com/aboutUs/mockobjects.pdf.

[5] If you want the sordid details, then scan the archives of the Test-Driven Development Yahoo! group at groups.yahoo.com/group/testdrivendevelopment/messages. Start at message 5068.

Testing legacy J2EE components

As we wrote previously, we define *legacy* in this context as "code you are afraid to change." In this case "you" might actually be you, or your team lead, your manager, or some other person with a vested interest in keeping you from changing existing code. There are a number of reasons for this, including (but not limited to, as lawyers like to say):

- "We need features; don't waste my time working on stuff that's already done!"
- "It works and we don't know why, so don't touch it."
- "Didn't you get it right the first time?!"
- "If you change John's code, he gets angry, so just leave it alone."
- "You can't change Mary's code without her approval, and she's on vacation this week."

We could go on, but as fun as that is, we need to figure out how to test code that we are not in a position to change. Refactoring towards a more testable design is therefore not an option open to us. What do we do? Recognizing that this situation is more common than we would hope, we have devoted some recipes in this part to testing legacy J2EE components: servlets, JSPs, and EJBs. There are really only two general techniques that you can use in this situation: test from end to end or resort to trickery to substitute mock objects.

End-to-End Tests—tests that execute the end-user interface and involve the entire application—are important, as they establish that the application actually works; however, these tests are expensive to execute, to write, and to maintain. They are slow, because they involve the entire application and all the expensive, external resources it uses. They are difficult to write, because you have to understand how all the components interact in order for the test to verify exactly what you want to verify. It is also difficult to re-create certain obscure error conditions, making it difficult to write tests for how well you handle those error conditions—things such as "disk full" and "graphical resources low."[6] End-to-End Tests are difficult to maintain, because insignificant changes can break them—different screen layout, different button names, translating into German—all these things can lead to changes in the End-to-End Tests. This is the negative feedback loop that drives many development shops to overspend on testing resources rather than use that

[6] J. B. worked on a project that spent three weeks hunting down a defect that only occurred when graphical resources were low. It took fifteen minutes of opening windows to recreate the conditions for each test.

money (and time) to eliminate the problem through design improvement. Relying on End-to-End Tests to verify components is a losing proposition.

An alternative is to use virtual machine and bytecode magic to substitute mock objects where you need them. We have always been leery of this approach, mostly because we do not know much about it; however, a recent development in this area looks promising: Virtual Mock Objects (www.virtualmock.org). Their site states, "VirtualMock is a Java unit testing tool which supports the Mock Objects testing approach. Through the use of Aspect-Oriented Programming, it is designed to address some of the limitations of existing Mock Object tools and approaches." Because this is relatively new, much of the advice we offer in this book ignores it. Once we have had the chance to digest it, perhaps our approach will change: the trade-off between mock objects and reducing dependency will certainly change and we will have to change with it. We think that Virtual Mock Objects has the potential to open up new options for people who need to write Object Tests for legacy code—at least in Java. But even so, we recommend reducing dependency over using mock objects. The result is code that is more flexible in addition to being easier to test. It is not a question of disliking mock objects; but in the spirit of the Agile Manifesto (www.agilemanifesto.org), we value reducing dependency over mocking collaborators.

Post script

As we came closer to releasing the book, we discovered that the jMock project (www.jmock.org), together with the CGLib package (http://cglib.sourceforge.net), make it relatively easy to substitute mock implementations of classes (not just interfaces) at runtime in tests. This achieves some of the same goals as Virtual Mock, and so it will be interesting to see how these projects progress in the coming months and years. We strongly recommend you look into both jMock and Virtual Mock to see which approach will best help you apply the mock objects approach to legacy code.

The Coffee Shop application

The Coffee Shop application is a run-of-the-mill online store that we will use to provide examples throughout the remainder of this book. Although we will refer to it frequently in various recipes, it is not the goal of this book to build towards a working online coffee shop. To do that, we would need to make certain design decisions, such as whether to use JavaServer Pages (JSPs), Velocity templates, or

XSL transformations as our presentation engine. Rather than decide on one technology, we will use examples from all these technologies in different recipes to illustrate the techniques they describe. We use the Coffee Shop application simply to provide a context for our J2EE-related recipes, rather than try to talk about testing J2EE applications in abstract terms.

To understand the recipes, it is enough to understand the *requirements* for the Coffee Shop application, so you will not see any architecture or design diagrams here. Throughout part 2 we will introduce those details as we need them. To illustrate a recipe, we will say, "Suppose the Coffee Shop application needs to do *this*; then you might have a design that looks like *that*, and here is how to use JUnit to test it." With this out of the way, we will describe the essential requirements for the Coffee Shop application.

In general terms, this is an online store, in the spirit of amazon.com, although not nearly as rich in features. The users of the store are generally shoppers purchasing coffee beans by the kilogram. The store uses the shopcart[7] metaphor, allowing the shopper to collect different kinds of coffee beans into his cart before completing his purchase. The store has an online catalog of coffee beans that the shopper can browse; the kinds and prices of the coffee are set by the administrator.

As we have already mentioned, we will propose additional features as needed to illustrate the techniques in our recipes. You can assume that the Coffee Shop application always has the very basic features we have mentioned here. Now that we have set the context for part 2, it is time to get to some recipes!

[7] Or "shopping cart," if you prefer. Both terms are relatively common, and it just happens that we use "shopcart" more often.

9 Testing and XML

This chapter covers

- Comparing XML documents with XMLUnit
- Ignoring superficial differences in XML documents
- Testing static web pages with XMLUnit
- Testing XSL transformations with XMLUnit
- Validating XML documents during testing

XML documents are everywhere in J2EE: deployment descriptors, configuration files, XHTML and XSL transformations, Web Services Description Language, and on and on. If you are going to test J2EE applications, you are going to need to write tests involving XML documents. Notice the title of this chapter is "Testing and XML," and not "Testing XML." It is not our mission here to test XML parsers, but rather to describe how to write tests involving XML documents and other XML-related technologies, such as XSL transformations. Given that XML documents are everywhere, it looks to us that this is an important skill to have, and we *can* summarize the entire chapter in two words: use XPath.

The XPath language defines a way to refer to the content of an XML document with query expressions that can locate anything from single XML attribute values to large complex collections of XML elements. If you have not seen XPath before, we recommend Elizabeth Castro's *XML for the World Wide Web*, which describes XML, XSL, and XPath in detail.[1] The overall strategy when testing with XML documents is to make assertions about those documents using XPath to retrieve the actual content. You want to write assertions like this:

```
assertEquals("Rainsberger", document.getTextAtXpath("/person/lastName"));
```

This assertion says, "the element `lastName` inside element `person` at the root of the document should have the text `Rainsberger`." Following is an XML document that satisfies this assertion.

Listing 9.1 A simple XML document

```
<?xml version="1.0?">
<person>
    <firstName>J. B.</firstName>
    <lastName>Rainsberger</lastName>
</person>
```

This is perhaps the simplest and most direct way to use XPath in your tests. When testing Java components whose output is an XML document, you want to treat the XML document as a return value and write assertions just as we described back in chapter 2, "Elementary tests." The key difference is that you need some additional tools to help you manipulate XML documents as Java objects, and that is where

[1] Also consider Elliotte Rusty Harold's *Processing XML with Java* (Pearson Education, 2002). Castro's book is a more general introduction, whereas Harold's deals directly with Java. Harold's book is available online at http://cafeconleche.org/books/xmljava/.

XPath comes in. You can use XPath to obtain the text of an element, to obtain the value of an attribute, or to verify whether an element exists. For example, if you have an XML document with many `person` elements and want to verify that there is a `Rainsberger` among them, you can use the same technique but with a different XPath statement:

```
assertFalse(document.getNodesAtXpath(
    "//person[lastName='Rainsberger']").isEmpty());
```

Here, the method `getNodesAtXpath()` works rather like a database query: "find all the `person` elements having a `lastName` element with the text `Rainsberger` and return them as a collection." The assertion says, "The collection of `person` elements with the `lastName` of `Rainsberger` should not be empty." These are the two kinds of XPath statements you will use most often in your tests.

Up to this point we have said little about where to find methods such as `getNodesAtXpath()` and `getTextAtXpath()`, because they are certainly not part of JUnit. There is an XPath API that you can use to execute XPath queries on a parsed XML document. The most widely used implementations of the XPath API are found in Xalan (http://xml.apache.org/xalan-j) and jaxen (http://jaxen.sourceforge.net).[2] You could use this API directly to make assertions about the structure and content of XML documents, but rather than reinventing the wheel, we recommend using XMLUnit (http://xmlunit.sourceforge.net). This package provides `XMLTestCase`, a base test case (see recipe 3.6, "Introduce a Base Test Case") that adds custom assertions (see recipe 17.4, "Extract a custom assertion") built on XPath. To use XMLUnit, create a test case class that extends `org.custommonkey.xmlunit.XMLTestCase`, then write your tests as usual. In your tests you will use the various XMLUnit assertions to verify that XML elements exist, to verify their value, and even to compare entire XML documents. Let's take an example.

Consider XML marshalling, which is the act of turning Java objects into XML documents (and the other way around). Web applications that use XSL transformations (rather than page templates, such as JSPs) as their presentation engine typically need to marshal Java beans to XML so that the transformation engine can present that data to the end user as a web page. Other systems marshal data to and from XML to communicate with other computers in a heterogeneous environment—one involving many computing platforms or programming languages, such as that created by web services. Suppose we are testing a simple XML marshaller

[2] Our experience is with Xalan, rather than with jaxen, so when we discuss XPath throughout this book, we are referring to the Xalan implementation.

that creates an XML document from a Value Object. The "person" document in listing 9.1 could have been produced by our XML marshaller operating on a `Person` object with the `String` attributes of `firstName` and `lastName`. We ought to be able to write out the XML document corresponding to a `Person` object and verify its contents. Enough words; it is time for a test.

Assume that the `XmlMarshaller` constructor takes the class object corresponding to our Value Object class and a `String` that you would like it to use as the name of the XML root element. The remaining XML elements are named according to the attributes of the Value Object. Listing 9.2 shows the resulting test.

Listing 9.2 Testing an XML object marshaller

```
package junit.cookbook.xmlunit.test;

import java.io.StringWriter;
import junit.cookbook.xmlunit.*;
import org.custommonkey.xmlunit.XMLTestCase;

public class MarshalPersonToXmlTest extends XMLTestCase {
    public void testJbRainsberger() throws Exception {
        Person person = new Person("J. B.", "Rainsberger");
        XmlMarshaller marshaller = new XmlMarshaller(
            Person.class, "person");
        StringWriter output = new StringWriter();
        marshaller.marshal(person, output);

        String xmlDocumentAsString = output.toString();

        assertXpathExists("/person", xmlDocumentAsString);

        assertXpathEvaluatesTo(
            "J. B.",
            "/person/firstName",
            xmlDocumentAsString);

        assertXpathEvaluatesTo(
            "Rainsberger",
            "/person/lastName",
            xmlDocumentAsString);
    }
}
```

One thing you will notice about this test is that it makes no mention at all of files. We are very accustomed to thinking of XML documents as files, because we usually use XML documents in their stored format, and we usually store XML in files. Nothing about XML *requires* that documents be processed using files. In fact, web services generally send and receive XML documents over network connections without

ever writing that data to file. To keep everything simple, our `XmlMarshaller` writes data to a `Writer`—any `Writer`—and the most convenient kind to use for this kind of test is a `StringWriter`: this way we do not have to load the resulting XML document from disk before parsing and verifying it. It is already in memory!

The assertions we make in this test are the custom assertions that XMLUnit provides. The first we use is `assertXpathExists()`, which executes the XPath query and fails only if there are no nodes in the XML document matching the query. Here we are using this custom assertion to verify the existence of a `person` root element. The other custom assertion we use is `assertXpathEvaluatesTo()`, which executes an XPath query and compares the result to an expected value; the assertion fails if the result and the expected value are different. XPath queries generally return either a list of nodes (in which you can search for the one you expect) or a `String` value (corresponding to the text of an element or attribute).

We could simplify this test, at the risk of making it more brittle, by making an assertion on the entire XML document which we expect the XML marshaller to produce. XMLUnit provides the method `assertXMLEqual()`, which checks whether two XML documents are equal. The way that XMLUnit defines equality in this case, though, requires some explanation. With XMLUnit, documents may be *similar* or *identical*. Documents are identical if the same XML elements appear in exactly the same order. Documents are similar if they represent the same content, but perhaps with certain elements appearing in a different order. Sometimes it matters, and sometimes it does not. As XMLUnit's web site says, "With XMLUnit, you have the control." Referring to listing 9.1, the order in which the `<firstName>` and `<lastName>` elements appear does not affect the meaning of the document. If we compare the document in listing 9.1 to a copy with `<lastName>` before `<firstName>`, we would expect them to be equal, based on the way we interpret the documents. This way of interpreting whether documents are equal–viewing them as data–is common in the kinds of applications we build, so `assertXMLEqual()` compares documents for similarity, rather than identity. As an example, here is a test for our XML object marshaller. We are verifying that the data is marshalled correctly, so we can build the test around `assertXMLEqual()` without having to resort to multiple XPath-based assertions (we have highlighted the differences between this test and the previous one in bold print):

```
public void testJbRainsbergerUsingEntireDocument()
    throws Exception {

    Person person = new Person("J. B.", "Rainsberger");
    XmlMarshaller marshaller =
        new XmlMarshaller(Person.class, "person");
```

```
    StringWriter output = new StringWriter();
    marshaller.marshal(person, output);

    String expectedXmlDocument =
        "<?xml version=\"1.0\" ?>"
            + "<person>"
            + "<firstName>J. B.</firstName>"
            + "<lastName>Rainsberger</lastName>"
            + "</person>";

    String xmlDocumentAsString = output.toString();

    assertXMLEqual(expectedXmlDocument, xmlDocumentAsString);
}
```

Rather than write an assertion to verify each part of the document that interests us, this second test creates the entire document we expect and compares it with the actual document the XML marshaller gives us. This is how to use XMLUnit in its simplest form. There is, however, one warning to go with this technique: what *you* think of as white space differences are not really mere white space differences in XML.

An XML document consists of a structure of elements—a tag, its attributes, and its content. The content of an XML element can either be text, more elements, or a combination of the two. It is the "combination of the two" that creates problems when comparing XML documents with one another. To illustrate this important point, let us consider what happens when we change the expected XML document in what we believe to be a purely cosmetic way. Another programmer has decided that failure messages would be easier to read if the expected XML document were formatted with line breaks and tabs, so she changes the test to the following:

```
public void testJbRainsbergerUsingEntireDocument()
    throws Exception {

    Person person = new Person("J. B.", "Rainsberger");
    XmlMarshaller marshaller =
        new XmlMarshaller(Person.class, "person");
    StringWriter output = new StringWriter();
    marshaller.marshal(person, output);

    String expectedXmlDocument =
        "<?xml version=\"1.0\" ?>\n"
            + "<person>\n"
            + "\t<firstName>J. B.</firstName>\n"
            + "\t<lastName>Rainsberger</lastName>\n"
            + "</person>\n";

    String xmlDocumentAsString = output.toString();

    assertXMLEqual(expectedXmlDocument, xmlDocumentAsString);
}
```

When she executes this test, she fully expects it to continue to pass, so she does so almost absentmindedly.[3] Much to her surprise, she sees this failure message:

```
junit.framework.AssertionFailedError: org.custommonkey.xmlunit.Diff [dif-
  ferent] Expected number of child nodes '5' but was '2' - comparing <per-
  son...> at /person[1] to <person...> at /person[1]
```

Five nodes? By our count there are two: `firstName` and `lastName`. What's the problem? The issue is that what you treat as white space, and therefore not a node, an XML parser treats as a text node with empty content. When you look at the following XML document, you see a `person` element with two elements inside it:

```
<?xml version="1.0" ?>
<person>
    <firstName>J. B.</firstName>
    <lastName>Rainsberger</lastName>
</person>
```

When the XML parser looks inside the `person` element, it finds an empty text element (thanks to the carriage return/linefeed character or characters), a `firstName` element, another empty text element, a `lastName` element, and another empty text element—that makes five. That explains the failure message, although it is not exactly the kind of behavior we want. We would like to ignore those empty text elements altogether. Fortunately, XMLUnit provides a simple way to ignore these kinds of white space differences. We invoke one extra method in our test, telling XMLUnit to ignore white space (we have highlighted the new method invocation in bold print):

```
public void testJbRainsbergerUsingEntireDocument()
    throws Exception {

    Person person = new Person("J. B.", "Rainsberger");
    XmlMarshaller marshaller =
        new XmlMarshaller(Person.class, "person");
    StringWriter output = new StringWriter();
    marshaller.marshal(person, output);

    String expectedXmlDocument =
        "<?xml version=\"1.0\" ?>\n"
            + "<person>\n"
            + "\t<firstName>J. B.</firstName>\n"
            + "\t<lastName>Rainsberger</lastName>\n"
            + "</person>\n";
```

[3] Do not be critical of her for this. Other programmers have concluded in similar cases, "It's a superficial change, so I'm sure it works. I'll just check in my change and go on vacation." At least she ran the tests!

```
        String xmlDocumentAsString = output.toString();

        XMLUnit.setIgnoreWhitespace(true);
        assertXMLEqual(expectedXmlDocument, xmlDocumentAsString);
    }
```

This tells XMLUnit to ignore all those extra elements containing only white space when comparing XML documents. Spaces in an element's content are not affected by this setting. If you need to do this for entire suites of tests, then move this statement into your test case class's `setUp()` method, as it is part of those tests' fixture.

There are other kinds of superficial differences that you may want to ignore on a test-by-test basis. You may, for example, want to compare only the structure of two documents without worrying about the values of attributes and the text. XMLUnit allows you to decide which differences are significant by implementing your own `DifferenceListener`. See recipe 9.3, "Ignore certain differences in XML documents," for details.

As we mentioned previously, many web applications transform XML documents into XHTML using XSLT (http://www.w3.org/TR/xslt) as their presentation engine. If so, you will want to test your XSL transformations in isolation, just as you would test your JSPs or Velocity templates in isolation. See recipe 9.6, "Test an XSL stylesheet in isolation," and also see chapter 12, "Testing Web Components," for recipes addressing JSPs and Velocity templates. The next most common use of XML documents in testing concerns the various J2EE deployment descriptors. There are many cases in which making an assertion on a deployment descriptor is more effective than testing the underlying deployed component. See the other chapters in this part of the book for details on when, how, and why to use the various J2EE deployment descriptors during testing. Finally, it is possible to treat HTML documents as though they are XML documents by writing XHTML (http://www.w3.org/TR/xhtml1/). You can test web pages in isolation in one of two ways: either write them in XHTML and use the techniques we have previously described, or use an HTML-tolerant parser that converts web pages into easy-to-verify XML documents. See recipe 9.5, "Test the content of a static web page," for an example of how to do this.

Validating XML documents provides a way to avoid writing some JUnit tests for your system. We like JUnit, but whenever there is an opportunity to do something even simpler, we take advantage of it. You can validate XML documents using either a DTD or an XML schema, and although this book is not the place to describe how to do that, see recipe 9.7, "Validate XML documents in your tests," for a discussion on making XML document validation part of your testing environment.

9.1 Verify the order of elements in a document

◆ Problem

You want to verify an XML document whose elements need to appear in a particular order.

◆ Background

You have code that produces an XML document, and the order in which the elements appear in the document changes the "value" the document represents. Consider, for example, a DocBook document: the order in which chapters, sections, and even paragraphs appear determines the meaning of the book. If you are marshalling a `List` of objects out to XML (see the introduction to this chapter for a discussion of XML marshalling) then you will want the corresponding XML elements to appear in the same order as they were stored in the `List` (otherwise, why is it a `List`?). If you have tried using `assertXMLEqual()` to compare documents with this sensitivity to order, then you may have seen XMLUnit treat certain unequal documents as equal, and this is not the behavior you want. You need XMLUnit to be stricter in its definition of equality.

◆ Recipe

To verify the order of elements in a document you may need to check whether the actual document is *identical* to the expected document, rather than *similar*. These are terms that XMLUnit defines by default, but allows you to redefine when you need to (see recipe 9.3). By default, documents are identical if their node structures are the same, elements appear in the same order, and corresponding elements have the same value. If you ignore white space (see the introduction to this chapter) then XML documents are identical if the only differences between them are white space.[4] If the elements are at the same level of the node's tree structure, but sibling elements *with different tag names* are in a different order, then the XML documents are not identical, but similar. (See the Discussion section for more on this.) You want to verify that documents are identical, whereas `assertXMLEqual()` only verifies whether they are similar.

Verifying whether documents are identical is a two-step process, compared to just using a customized assertion. First you "take a diff" of the XML documents, then

[4] Specifically ignorable white space as XML defines the term. This includes spacing of elements, but not white space inside a text node's value. Node text "A B" is still different from "A B".

make an assertion on that "diff." If you are familiar with CVS or the UNIX toolset, then you know what we mean by a "diff": the UNIX tool computes the differences between two text files, whereas XMLUnit's class `Diff` (`org.custommonkey.xmlunit.Diff`) computes the differences between two XML documents. To take a diff of two XML documents with XMLUnit, you create a `Diff` object from the respective documents, after which you can ask the `Diff`, "Are the documents similar? Are they identical?" Let us look at a simple example.

Consider a component that builds an article, suitable for publication on the web, from paragraphs, sections, and headings that you provide through a simple Java interface. You might use this `ArticleBuilder` as the model behind a specialized article editor that you want to write.[5] As you are writing tests for this class, you decide to add a paragraph and a heading to an article and verify the resulting XML document. Listing 9.3 shows the test using `assertXMLEqual()`.

Listing 9.3 Comparing documents with `assertXMLEqual()`

```
public class BuildArticleTest extends XMLTestCase {
    public void testMultipleParagraphs() throws Exception {
        XMLUnit.setIgnoreWhitespace(true);

        ArticleBuilder builder =
            new ArticleBuilder("Testing and XML");

        builder.addAuthorName("J. B. Rainsberger");
        builder.addHeading("A heading.");
        builder.addParagraph("This is a paragraph.");

        String expected =
            "<?xml version=\"1.0\" ?>"
                + "<article>"
                + "<title>Testing and XML</title>"
                + "<author>J. B. Rainsberger</author>"
                + "<p>This is a paragraph.</p>"
                + "<h1>A heading.</h1>"
                + "</article>";

        String actual = builder.toXml();
        assertXMLEqual(expected, actual);
    }
}
```

[5] Ron Jeffries explores building a specialized article editor in "Adventures in C#" (http://www.xprogramming.com/) as well as his book *Extreme Programming Adventures in C#* (Microsoft Press, 2004).

Here we have "accidentally" switched the heading and the paragraph in our expected XML document: the heading ought to come before the paragraph, not after it. No problem, we say: the tests will catch that problem—but this test passes! It passes because the expected and actual documents are similar, but not identical. In order to avoid this problem, we change the test as shown in listing 9.4 (the change is highlighted in bold print):

Listing 9.4 Testing for identical XML documents

```
public void testMultipleParagraphs() throws Exception {
    XMLUnit.setIgnoreWhitespace(true);

    ArticleBuilder builder =
        new ArticleBuilder("Testing and XML");

    builder.addAuthorName("J. B. Rainsberger");
    builder.addHeading("A heading.");
    builder.addParagraph("This is a paragraph.");

    String expected =
        "<?xml version=\"1.0\" ?>"
            + "<article>"
            + "<title>Testing and XML</title>"
            + "<author>J. B. Rainsberger</author>"
            + "<p>This is a paragraph.</p>"
            + "<h1>A heading.</h1>"
            + "</article>";

    String actual = builder.toXml();

    Diff diff = new Diff(expected, actual);
    assertTrue(
        "Builder output is not identical to expected document",
        diff.identical());
}
```

First we ask XMLUnit to give us an object representing the differences between the two XML documents, then we make an assertion on the `Diff`, expecting it to represent identical documents—specifically that the corresponding elements appear in the expected order. This test fails, as we would expect, alerting us to our mistake. So if we run the risk of this kind of problem, why does `assertXMLEqual()` behave the way it does? It turns out not to be a common problem in practice.

While looking for an example for this recipe, we asked programmers to show us examples of XML documents with a particular property. We wanted to see a

document with sibling tags (having the same parent node) with different names, where changing the order of those elements changes the document's meaning. For the most part, they were unable to come up with a compelling example, which surprised us. Far from a proof, let us look at some reasons why.

Consider XML documents that represent books or articles. These documents have sections, chapters, paragraphs—structure that maps very well to XML. In an HTML page, paragraphs merely follow headings, but the paragraphs in a section really *belong* to that section. It makes more sense to represent a section of a document as its own XML element *containing* its paragraphs, as opposed to the way HTML does it. This is the approach that DocBook (http://www.docbook.org/) takes: a section element contains a title element followed by paragraph elements. The paragraph elements are siblings in the XML document tree, the order of the paragraphs matters, and the elements all have the same name. On the other hand, when we use XML documents to represent Java objects, we often render each attribute of the object as its own element. Those elements have *different* names, and most often the order in which those elements appear in the document does not affect the value of the object the document represents. So there appears to be a correlation here:

- If sibling elements have different names, then the order in which they appear likely does not matter.
- If the order in which sibling elements appear matters, then they likely have the same name.

Based on these simple observations, we can conclude that `assertXMLEqual()` behaves in a manner that works as you would expect, most of the time. It may not be immediately obvious, but we thought it was neat once we reasoned it through.

◆ Discussion

When you use `assertXMLEqual()`, XMLUnit ignores the order in which sibling elements with different tag names appear. This is common when marshalling a Java object to XML: we typically do not care whether the first name appears before or after the last name, so long as both appear in the resulting XML document. We emphasize "different tag names" because XMLUnit preserves (and checks) the order of sibling elements with the same tag name, even when checking documents for similarity. If you need to marshal a `Set`, rather than a `List`, to XML, then you will likely represent each element with its own tag and those tags will have the same name, such as `item`. When comparing an expected document with the

actual marshalled one, you want to ignore the order of these `item` elements. In order to ignore this difference between the two documents, you need to customize XMLUnit's behavior, which we describe in recipe 9.3.

- ***Related***
  - 9.3—Ignore certain differences in XML documents
  - DocBook (http://www.docbook.org/)

9.2 Ignore the order of elements in an XML document

- ***Problem***

  You want to verify an XML document and the order of certain XML elements within the document does not matter.

- ***Background***

  In recipe 9.1, "Verify the order of elements in a document," we wrote about a correlation between the names of XML elements and whether it matters in which order the elements appear. We claimed that when there are many tags with the same name, the order in which they appear tends to be important. One exception to this rule that we have encountered is the web deployment descriptor. Servlet initialization parameters, request attributes, session attributes, and other parts of the servlet specification are all essentially `Map`s of data. In particular, a servlet's initialization parameters are stored in a `Map` whose keys are the names of the parameters (as `String`s) and whose values are the values of those parameters (also as `String`s). The web deployment descriptor—or `web.xml` as you may know it—represents servlet initialization parameters with the XML element `<init-param>`, which contains a `<param-name>` and a `<param-value>`. It treats the parameters as a `List` of name-value pairs, but those name-value pairs do not logically form a `List`; they form a `Set` instead. The order of the parameters in the deployment descriptor generally does not affect the meaning of those parameters.[6] Accordingly, when we test for the existence of and the correctness of a servlet's initialization parameters, we either have to pay attention to the order in which we specify the parameters in the deployment descriptor or change the test so that it does not take their order into account.

[6] If your servlet initialization code depends on the order in which the web container hands you those parameters, then you are in for a surprise one day. Consider yourself warned.

◆ *Recipe*

The most direct way to solve this problem, which we recommend, is to traverse the DOM (Document Object Model) tree for both XML documents. In general, if you are unconcerned about the order of a group of sibling elements with the same tag name, then they represent items in either a `Set` or a `Map`.[7] For that reason, we recommend you collect the data into a `Set` (or a `Map`) using the XPath API, then compare the resulting objects for equality in your test. XMLUnit does not do quite as much for us in this case as we would like, but no tool can be all things to all people. As an example, consider the servlet initialization parameters in a web deployment descriptor for a Struts application, although it could be for any kind of web application. We just chose Struts because we like it. Listing 9.5 shows such a sample:

Listing 9.5 A sample web deployment descriptor

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

<web-app>
    <display-name>Struts Taglib Exercises</display-name>

    <!-- Action Servlet Configuration -->
    <servlet>
        <servlet-name>action</servlet-name>
        <servlet-class>
            org.apache.struts.action.ActionServlet
        </servlet-class>
        <init-param>
            <param-name>application</param-name>
            <param-value>
                org.apache.struts.webapp.exercise.ApplicationResources
            </param-value>
        </init-param>
        <init-param>
            <param-name>config</param-name>
            <param-value>/WEB-INF/struts-config.xml</param-value>
        </init-param>
        <init-param>
            <param-name>debug</param-name>
            <param-value>2</param-value>
        </init-param>
        <init-param>
            <param-name>detail</param-name>
            <param-value>2</param-value>
```

[7] We looked hard—honestly, we did—for a counterexample and could not find one.

```
        </init-param>
        <load-on-startup>2</load-on-startup>
    </servlet>

    <!-- Additional settings omitted for brevity -->
</web-app>
```

We want to focus on the elements in bold print, gather them into `Map` objects in memory, and then compare the corresponding `Map` objects for equality. This makes for rather a simple looking test:

```
public void testActionServletInitializationParameters()
    throws Exception {

    File expectedWebXmlFile =
        new File("test/data/struts/expected-web.xml");
    File actualWebXmlFile = new File("test/data/struts/web.xml");

    Document actualDocument = buildXmlDocument(actualWebXmlFile);
    Document expectedDocument =
        buildXmlDocument(expectedWebXmlFile);

    Map expectedParameters =
        getInitializationParametersAsMap(expectedDocument);

    Map actualParameters =
        getInitializationParametersAsMap(actualDocument);

    assertEquals(expectedParameters, actualParameters);
}
```

The good news is that the test itself is brief and to the point. In words, it says, "get the initialization parameters from the `expected` and `actual` documents and they ought to be equal." The bad news is that, as always, the devil is in the details. Rather than distract you from your reading, we have decided to move the complete solution—which is mostly XML parsing code—to solution A.3, "Ignore the order of elements in an XML document." There you can see how we implemented `getInitializationParametersAsMap()` and `buildXmlDocument()`, the latter of which uses a nice convenience method from XMLUnit.

NOTE *Network connectivity and the DTD*—Notice that the web deployment descriptor in listing 9.5 declares that it conforms to a remotely accessible DTD. XMLUnit tests will attempt to load the DTD from the remote location at runtime, requiring a network connection. If XMLUnit does not find the DTD online, it will throw an `UnknownHostException` with a message reading "Unable to load the DTD for this document," which does not clearly describe the real problem in context. One way to avoid this problem is to execute the tests on a machine that has access to the remote site providing

the DTD. Perhaps better, though, is to make the DTD available locally by downloading and storing it on the test machine. Include the location of the local copy in the DTD declaration as follows in bold print:

```
<!DOCTYPE web-app
   PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
   "file:///C:/test/data/struts/web-app_2_2.dtd">
```

This also has the pleasant effect of avoiding the remote connection in the first place, increasing the execution speed of your tests. Of course, now your tests are slightly more brittle, as they depend on files on the local file system. We describe the trade-offs involved in placing test data on the file system in both chapter 5, "Managing Test Data," and chapter 17, "Odds and Ends."

◆ Discussion

When we first tried to write this deployment test, we compared the actual web deployment descriptor against a Gold Master we had previously created[8] (see recipe 10.2, "Verify your SQL commands," for a description of the Gold Master concept):

```
public void testStrutsWebDeploymentDescriptor()
    throws Exception {

    File expectedWebXmlFile =
        new File("test/data/struts/expected-web.xml");
    File actualWebXmlFile = new File("test/data/struts/web.xml");

    Document actualDocument = buildXmlDocument(actualWebXmlFile);
    Document expectedDocument =
        buildXmlDocument(expectedWebXmlFile);

    assertXMLEqual(expectedDocument, actualDocument);
}
```

We decided that it was simpler to just compare the entire document, so that is what we did. When we executed this test, all was well until we started generating the web deployment descriptor, rather than handcrafting it. The generator wrote the parameters to XML in a different order than when we wrote the document by hand. We were not concerned about the order of these tags, and when that order changed, our test failed *unnecessarily*. This is when we decided that we needed to change the test to do what this recipe recommends.

Next we tried solving this problem with an XMLUnit `DifferenceListener` (see recipe 9.3), but were not able to do it, at least not always. This is another place

[8] The Gold Master technique is only as good as the correctness of the master copy itself. Be careful! Make sure that the master is absolutely correct and is under strict change control.

where a human's interpretation of the document differs from the software's interpretation. The XMLUnit Diff engine sees four `<init-param>` tags in both the expected and actual documents and says, "Same names and the same number of them, so they are the same." It does not consider the nontext content of the `<init-param>` tags when comparing them to one another, so it does not notice (yet) that they are different. Only when the Diff engine proceeds to compare the `<param-name>` and `<param-value>` tags does it notice a difference, and by that point, it interprets the difference as "`<param-name>` nodes have different text," rather than the much more benign "`<init-param>` nodes are out of order." The former is a failure, but the latter is not, leading to a false negative result in our tests. We could not think of a way to *safely* ignore these differences, short of tracking all the values we have seen so far and comparing them against one another after the Diff engine has processed all the `<init-param>` nodes. That is just too much work. At that point, we thought we may as well just process the nodes in the first place, which is why we recommend the solution in this recipe.

This may not always be a problem, so try using `assertXMLEqual()` before embarking on writing all this extra code. This problem is caused by the document itself: the fact that the differences between the `<init-param>` nodes are found deeper in each node's subtree, and not right at the same level as the `<init-param>` nodes themselves. If, for example, the `<init-param>` nodes had ID attributes and those ID attributes were out of sequence, the Diff engine would have detected that and reported the difference as a *sequence* difference, which we could easily ignore. See recipe 9.3 for details on how to use the `DifferenceListener` feature to ignore those kinds of differences.

◆ Related

- 9.1—Verify the order of elements in a document
- 9.3—Ignore certain differences in XML documents
- A.3—Ignore the order of elements in an XML document (complete solution)

9.3 Ignore certain differences in XML documents

◆ Problem

You want to verify an XML document, but need to ignore certain superficial differences between the actual and expected documents.

◆ Background

We generally prefer comparing an expected value to an actual value in our tests, as opposed to decomposing some complex value into its parts and then checking each corresponding part individually. For this reason, we prefer to use `assertXMLEqual()` over individual XPath-based assertions in a test, but sometimes we run into a situation where we want to compare, say, 80% of the content of two XML documents and ignore the rest. We might need several dozen XPath-based assertions, when really all we want to do is to ignore what we might determine to be "superficial" differences between the two documents. We want to ignore a few values here and there, or the order of certain attributes, but compare the rest of the documents for similarity without resorting to a mountain of assertions.

In our Coffee Shop application there is an unpleasant dependency between our tests and the product catalog. Our choice of XSL transformation for the presentation layer prompts us to convert Java objects from the business logic layer into XML suitable for presentation. In particular, we need to convert the `ShopcartModel`, which represents the items in the current shopcart. When the user places a quantity of coffee in her shopcart, the system adds `CoffeeQuantity` objects to the `ShopcartModel`. When it comes time to display the shopcart, the system needs to display the total cost of the items in her shopcart, for which it must consult a `CoffeeBeanCatalog`. The catalog provides each coffee product's unit price from which the system computes the total cost. In summary, we need to convert a `ShopcartModel` into an XML document with prices in it, but in order to do this we need to prime the catalog with whatever coffee products we want to put in the test shopcart. That does not seem right. If we just ignored the prices, trusting that our business logic computes them correctly,[9] we could avoid the problem of having tests depend on a specific catalog.

◆ Recipe

The good news is that XMLUnit provides a way to ignore certain differences between XML documents. The even better news is that XMLUnit allows you to change the meaning of "different" from test to test. You can "listen" for differences as XMLUnit's engine reports them, ignoring the superficial differences that do not concern you *for the current test*. To achieve this you create a custom *difference listener*—that is, your own implementation of the interface `org.custommonkey.xmlunit.DifferenceListener`. To use your custom difference listener, you ask XMLUnit for the differences between the actual and expected XML documents,

[9] The business logic has comprehensive tests, after all.

then use your listener as a kind of filter, applying it to the list of differences between the documents in order to ignore the ones that do not concern you. We can solve our problem directly using a custom difference listener.

Looking at the `DifferenceConstants` interface of XMLUnit, we can see the way XMLUnit categorizes differences between XML documents. One type of difference is a *text value difference* (represented in `DifferenceConstants` as the constant `TEXT_VALUE`)— that is, the text is different for a given XML tag. We can therefore look for differences of this type, examine the name of the tag, and ignore the difference if the tag name is `unit-price`, `total-price` or `subtotal`. Another type of difference is `ATTR_VALUE`—that is, the values are different for a given tag attribute. We can look for differences of this type, examine the name of the attribute and the element that owns it, and ignore the difference if both the element name is `item` and the attribute name is `id`. (Product IDs depend on the catalog, too.) We now have enough information to write our `IgnoreCatalogDetailsDifferenceListener`. We warn you: the DOM API is rather verbose, so if you are not accustomed to it, read slowly and carefully. First, let us look at the methods we have to implement from the interface `DifferenceListener`, shown here in listing 9.6:

Listing 9.6 An implementation of `DifferenceListener`

```
public class IgnoreCatalogDetailsDifferenceListener
    implements DifferenceListener {

    public int differenceFound(Difference difference) {
        int response = RETURN_ACCEPT_DIFFERENCE;

        int differenceId = difference.getId();
        if (DifferenceConstants.TEXT_VALUE_ID
            == differenceId) {

            String currentTagName =
                getCurrentTagName(difference);

            if (tagNamesToIgnore.contains(currentTagName)) {
                response =
                    RETURN_IGNORE_DIFFERENCE_NODES_SIMILAR;
            }
        }
        else if (DifferenceConstants.ATTR_VALUE_ID
            == differenceId) {

            Attr attribute = getCurrentAttribute(difference);

            if ("id".equals(attribute.getName())
                && "item".equals(
                    attribute
                        .getOwnerElement()
                        .getNodeName())) {
```

```
                response =
                    RETURN_IGNORE_DIFFERENCE_NODES_SIMILAR;
            }
        }

        return response;
    }

    public void skippedComparison(
        Node expectedNode,
        Node actualNode) {

        // Nothing to do here
    }
}
```

XMLUnit invokes `differenceFound()` for each difference it finds between the two XML documents you compare. As a parameter to `differenceFound()`, XMLUnit passes a `Difference` object, providing access to a description of the difference and the DOM `Node` objects in each document so that you can explore them further. Our implementation looks for the two kinds of differences we wish to ignore: text value differences and attribute value differences.

When our difference listener finds a text value difference, it retrieves the name of the tag containing the text, and then compares it to a set of "tags to ignore." If the current tag name is one to ignore, then we respond to XMLUnit with "Ignore this difference when comparing for similarity." (There is another constant we can return to say, "Ignore this difference when comparing for identity.") We declared the `Set` of tag names to be ignored as a class-level constant.[10]

```
public class IgnoreCatalogDetailsDifferenceListener
    implements DifferenceListener {

    // Remaining code omitted

    private static final Set tagNamesToIgnore = new HashSet() {
        {
            add("unit-price");
            add("total-price");
            add("subtotal");
        }
    };
}
```

[10] The coding technique here is to create an anonymous subclass with an instance initializer. It sounds complicated, but the benefit is clear, concise code: one statement rather than four. It looks a little like Smalltalk. See Paul Holser's article on the subject (http://home.comcast.net/~pholser/writings/concisions.html).

We also provided a convenience method to retrieve the name of the "current tag"—the tag to which the current `Difference` corresponds. The method `getTagName()` handles `Text` nodes differently: because a `Text` node does not have its own name, we are interested in the name of its parent node:

```
public class IgnoreCatalogDetailsDifferenceListener
    implements DifferenceListener {

    // Remaining code omitted

    public String getCurrentTagName(Difference difference) {
        Node currentNode =
            difference.getControlNodeDetail().getNode();

        return getTagName(currentNode);
    }

    public String getTagName(Node currentNode) {
        if (currentNode instanceof Text)
            return currentNode.getParentNode().getNodeName();
        else
            return currentNode.getNodeName();
    }
}
```

When our difference listener finds an attribute value difference, it retrieves the current attribute name and the name of the tag that owns it, compares it against the one it wants to ignore, and if there is a match, the difference listener ignores it. Here is the convenience method for retrieving the current attribute name:

```
public class IgnoreCatalogDetailsDifferenceListener
    implements DifferenceListener {

    // ...

    public Attr getCurrentAttribute(Difference difference) {
        return (Attr)
            difference.getControlNodeDetail().getNode();
    }
}
```

If `differenceFound()` does not find any differences to ignore, then it responds "accept this difference," meaning that XMLUnit should count it as a *genuine* difference rather than a *superficial* one. If there remain genuine differences after the superficial ones are ignored, then XMLUnit treats the documents as dissimilar and `assertXMLEqual()` fails. Listing 9.7 shows an example of how we used this difference listener (the lines of code for the difference listener are in bold print):

Listing 9.7 Using the `DifferenceListener`

```
package junit.cookbook.coffee.model.xml.test;

import java.util.Arrays;
import junit.cookbook.coffee.display.*;
import junit.cookbook.coffee.model.*;
import org.custommonkey.xmlunit.*;
import com.diasparsoftware.java.util.Money;

public class MarshalShopcartTest extends XMLTestCase {
    private CoffeeCatalog catalog;

    protected void setUp() throws Exception {
        XMLUnit.setIgnoreWhitespace(true);
        catalog = new CoffeeCatalog() {
            public String getProductId(String coffeeName) {
                return "001";
            }

            public Money getUnitPrice(String coffeeName) {
                return Money.ZERO;
            }
        };
    }

    public void testOneItemIgnoreCatalogDetails() throws Exception {
        String expectedXml =
            "<?xml version='1.0' ?>\n"
                + "<shopcart>\n"
                + "<item id=\"762\">"
                + "<name>Sumatra</name>"
                + "<quantity>2</quantity>"
                + "<unit-price>$7.50</unit-price>"
                + "<total-price>$15.00</total-price>"
                + "</item>\n"
                + "<subtotal>$15.00</subtotal>\n"
                + "</shopcart>\n";

        ShopcartModel shopcart = new ShopcartModel();
        shopcart.addCoffeeQuantities(
            Arrays.asList(
                new Object[] { new CoffeeQuantity(2, "Sumatra")}));

        String shopcartAsXml =
            ShopcartBean.create(shopcart, catalog).asXml();

        Diff diff = new Diff(expectedXml, shopcartAsXml);

        diff.overrideDifferenceListener(
            new IgnoreCatalogDetailsDifferenceListener());

        assertTrue(diff.toString(), diff.similar());
    }
}
```

In the method `setUp()` we faked out the catalog so that we would not have to prime it with data. Every product costs $0 and has ID `001`.

If the documents are not similar, in spite of ignoring all these differences, then the failure message lists the remaining differences. You can then decide whether to change the difference listener to ignore the extra differences or to fix the actual XML document.

◆ Discussion

An alternative to this approach is to build a custom Document Object Model from the XML documents, an approach we describe in recipe 9.2, "Ignore the order of elements in an XML document," by loading servlet initialization parameters into a `Map`.[11] It is then easy to compare an expected web deployment descriptor document object against the actual one, because the `Map` compares the servlet entries the way you would expect: ignoring the order in which they appear. We think that this approach is simpler; however, if you doubt us, then as always, try them both and measure the difference.

Remember the two essential approaches to verifying XML documents: using XPath to verify parts of the actual document, or creating an entire expected document and comparing it to the actual document. When working with Plain Old Java Objects, we will generally go out of our way to use the latter approach by building the appropriate `equals()` methods we need. Our experience tells us to expect a high return on investment in terms of making it easier to write future tests. With XML documents the trade-off is less clear.

Building a complex difference listener can take a considerable amount of work, which mostly comes from the difficulty in figuring out exactly which differences to ignore and which to retain. This is not a criticism of XMLUnit, but the way its authors have categorized differences may not map cleanly onto your mental model of the differences between two documents. This complexity is inherent to the problem of describing the difference between two structured text files.[12] From time to time, depending on the complexity of what "similar" and "identical" XML documents mean in your domain, you may find yourself spending an hour trying to build the correct difference listener. If this happens, we recommend you stop, abandon the effort, and go back to using the XPath-based assertions. We also strongly recommend sticking with XPath-based assertions if you find yourself wanting to ignore 80% of the actual document and wanting to check "just this part

[11] You do not need to build a custom DOM for the entire document, just the parts you care about.

[12] How many times has your version control system reported the difference between your version of a Java source file and the repository's version "in a strange way?" That is the nature of the problem.

here." It may be more work to describe the parts of the document to ignore than simply to write assertions for the part you want to examine. In that case, you can combine the approaches: use XPath to extract the document fragment that interests you, then compare it with the fragment you expect using `assertXMLEqual()`.

- ***Related***
  - 9.2—Ignore the order of elements in an XML document

9.4 Get a more detailed failure message from XMLUnit

- ***Problem***

  You want a more detailed failure message from XMLUnit when documents are different.

- ***Background***

  The XMLUnit Diff engine stops reporting differences after it finds the first difference, just as JUnit stops reporting failed assertions once the first assertion fails in a test. Although this is consistent with JUnit's core philosophy, it limits the amount of information available to diagnose the causes of the defects in your code. It would certainly be helpful if you had more information from XMLUnit about the differences between the document you have and the one you expect.

- ***Recipe***

  We recommend placing *all the differences* between the expected and actual XML documents in the failure message of your XMLUnit assertions. To do that, create a `DetailedDiff` object from the original `Diff` then include it in your failure message. Here is an example (our use of `DetailedDiff` is in bold print):

```
public void testMultipleParagraphs() throws Exception {
    XMLUnit.setIgnoreWhitespace(true);

    ArticleBuilder builder =
        new ArticleBuilder("Testing and XML");

    builder.addAuthorName("J. B. Rainsberger");
    builder.addHeading("A heading.");
    builder.addParagraph("This is a paragraph.");

    String expected =
        "<?xml version=\"1.0\" ?>"
            + "<article>"
```

```
            + "<title>Testing and XML</title>"
            + "<author>J. B. Rainsberger</author>"
            + "<p>This is a paragraph.</p>"
            + "<h1>A heading.</h1>"
            + "</article>";

    String actual = builder.toXml();

    Diff diff = new Diff(expected, actual);
    assertTrue(new DetailedDiff(diff).toString(), diff.identical());
}
```

This `DetailedDiff` lists all the differences between the two documents, rather than just the first difference. In this case, the paragraph and heading elements are mixed up in the test, although we do not realize that right now. (Sometimes we make a mistake when writing a test.) Here is the information that XMLUnit gives us when this assertion fails:

```
[not identical] Expected sequence of child nodes '2' but was '3'
≫  - comparing <p...> at /article[1]/p[1] to <p...> at /article[1]/p[1]

[not identical] Expected sequence of child nodes '3' but was '2'
≫  - comparing <h1...> at /article[1]/h1[1] to <h1...> at /article[1]/h1[1]
```

This tells us that the test expects the `<p>` tag to appear in position 2 and the `<h1>` tag to appear in position 3, relative to the `<article>` tag that contains them both. That does not make sense! The heading ought to come before the paragraph! This is the detailed information we need to see that this time it is the test, and not the production code, that needs fixing. Once we reverse the order of the lines in the expected XML document, the test passes:

```
public void testMultipleParagraphs() throws Exception {
    XMLUnit.setIgnoreWhitespace(true);

    ArticleBuilder builder =
        new ArticleBuilder("Testing and XML");

    builder.addAuthorName("J. B. Rainsberger");
    builder.addHeading("A heading.");
    builder.addParagraph("This is a paragraph.");

    String expected =
        "<?xml version=\"1.0\" ?>"
            + "<article>"
            + "<title>Testing and XML</title>"
            + "<author>J. B. Rainsberger</author>"
            + "<h1>A heading.</h1>"
            + "<p>This is a paragraph.</p>"
            + "</article>";
```

```
        String actual = builder.toXml();

        Diff diff = new Diff(expected, actual);
        assertTrue(new DetailedDiff(diff).toString(), diff.identical());
    }
```

That's much better.

◆ Discussion

We recommend building a custom assertion (see recipe 17.4, "Extract a custom assertion") called `assertXMLIdentical(String expected, String actual)` with this specialized failure message. We are a little surprised that no one has added it to XMLUnit yet—or perhaps by now they have! That is the essence of open source: if the product is missing something you need, add it yourself, then later submit it for inclusion into the product. You do not need to be stuck with code that falls short of what you need, even if only a little bit. Of course, XMLUnit is an excellent library and we are not out to criticize it, but all software needs improvement.

◆ Related

- 17.4—Extract a custom assertion

9.5 Test the content of a static web page

◆ Problem

You want to test the content of a web page, but your web pages are not written in XHTML, so they are not valid XML documents.

◆ Background

We love XHTML, but it has one fatal flaw: no web browser on the planet enforces it. Browsers are *very* lenient when it comes to HTML, which is why very few people—programmers, web designers, hobbyists—are motivated to write their web pages in XHTML. It is more work for them to do it and, unless they need to parse their web pages as XML, they benefit nothing from the effort. If web design tools were to create XHTML by default (and some at least give you the option to do so) then the story might be different, but as it stands, very few people write XHTML. As a result, unless *you* write every part of every web page you need to test, *you* will have to work a little harder to use the testing techniques we have introduced in this chapter. The alternative is to use another tool to help turn loosely written HTML into well-formed XML.

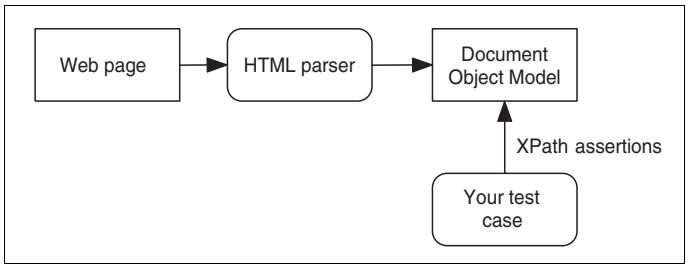

Figure 9.1 Testing a web page by converting it to XML

This recipe works best for verifying static web pages. If you want to verify the content of a dynamically generated web page, see chapter 12, "Testing Web Components," and chapter 13, "Testing J2EE Applications." The former discusses testing web page templates in isolation and the latter describes how to test generated web pages by simulating a web browser. HtmlUnit, the tool of choice for so-called "black box" web application testing, uses the techniques in this recipe to provide a rich set of custom assertions for verifying web page content. This recipe explains some of the machinery behind HtmlUnit, in case you wish to, or need to, get along with HtmlUnit.

There are HTML parsers that you can use to convert HTML documents into equivalent, well-formed XML. These parsers present web pages as DOM trees which you can inspect and manipulate as needed. The two most well-known parsers are Tidy (http://tidy.sourceforge.net/) and NekoHTML (http://www.apache.org/~andyc/neko/doc/html/). Although one can generally use either parser, we favor NekoHTML, as it handles a wider range of badly formed HTML. The general strategy is to load your web page into the HTML parser, which then creates a DOM representation of the page (see figure 9.1). You can then apply the techniques in the preceding recipes to analyze the DOM and verify the parts of it you need to verify. We will use this technique to verify the welcome page for our Coffee Shop application.

Following is the web page we would like to verify. As you can see, it is not quite XHTML compliant: the `link` and `input` start tags do not have corresponding end tags and there is text content without a surrounding paragraph tag.

```
<html>
<head>
<link href="theme/Master.css" rel="stylesheet" type="text/css">
<title>Welcome!</title>
</head>
<body>
<form name="launchPoints" action="coffee" method="post">
Browse our <input type="submit" name="browseCatalog" value="catalog">.
</form>
```

```
</body>
</html>
```

The test we would like to write verifies that there is a way to navigate from this welcome page to our product catalog. We are looking for a `form` whose `action` goes through our `CoffeeShopController` servlet and has a `submit` button named `browseCatalog`. If those elements are present, then the user will be able to reach our catalog from this page. In our test we need to configure the NekoHTML parser, parse the web page, retrieve the DOM object, and use XMLUnit to make assertions about the content of the resulting XML document. Listing 9.8 shows the test we need to write:

Listing 9.8 WelcomePageTest

```
package junit.cookbook.coffee.web.test;

import java.io.FileInputStream;

import org.custommonkey.xmlunit.XMLTestCase;
import org.apache.xerces.parsers.DOMParser;
import org.cyberneko.html.HTMLConfiguration;
import org.w3c.dom.Document;
import org.xml.sax.InputSource;

public class WelcomePageTest extends XMLTestCase {
    private Document welcomePageDom;

    protected void setUp() throws Exception {
        DOMParser nekoParser =
            new DOMParser(new HTMLConfiguration());

        nekoParser.setFeature(
            "http://cyberneko.org/html/features/augmentations",
            true);

        nekoParser.setProperty(
            "http://cyberneko.org/html/properties/names/elems",
            "lower");

        nekoParser.setProperty(
            "http://cyberneko.org/html/properties/names/attrs",
            "lower");

        nekoParser.setFeature(
            "http://cyberneko.org/html/features/report-errors",
            true);

        nekoParser.parse(
            new InputSource(
                new FileInputStream(
                    "../CoffeeShopWeb/Web Content/index.html")));
```

```
        welcomePageDom = nekoParser.getDocument();
        assertNotNull("Could not load DOM", welcomePageDom);
    }

    public void testCanNavigateToCatalog() throws Exception {
        assertXpathExists(
            "//form[@action='coffee']"
                + "//input[@type='submit' and @name='browseCatalog']",
            welcomePageDom);
    }
}
```

The test itself is very simple: it uses a single XPath statement to look for a form with the expected action that also contains the expected submit button. Our web application maps the URI `coffee` to the servlet `CoffeeShopController`, which explains why we compare the form's action URI to `coffee`. We could have written separate assertions to verify that the expected form exists, that it has the expected action URI, that it contains a submit button, and that it contains the expected submit button, as shown here:

```
public void testCanNavigateToCatalog() throws Exception {
    assertXpathExists("//form", welcomePageDom);

    assertXpathEvaluatesTo(
        "coffee",
        "//form/@action",
        welcomePageDom);

    assertXpathExists(
        "//form[@action='coffee']//input[@type='submit']",
        welcomePageDom);

    assertXpathEvaluatesTo(
        "browseCatalog",
        "//form[@action='coffee']"
            + "//input[@type='submit']/@name",
        welcomePageDom);
}
```

There are a few differences with this more verbose test. First, because each assertion verifies only one thing, it is easier to determine the problem from a failure. If the second assertion fails, you can be sure of one of two causes: there is more than one form in the web page *or* the first form on the page has the wrong action. Next, these assertions are more precise: if there are other forms on the page or other buttons in the form, these assertions may fail. Whether this last difference is a benefit or excessive coupling depends on your perspective. We generally prefer to make the weakest assertion that can possibly verify that we have done something

right, rather than make the strongest assertion that can possibly eliminate everything we have done wrong. The latter kind of assertion tends to make tests overly brittle.

We should mention the way we have configured NekoHTML for this test. We highlighted in bold print the classes that we imported, because we are not actually using the NekoHTML parser, but rather a Xerces DOM parser with NekoHTML's HTML DOM configuration. This allows us to assume in our tests that all tag names and attribute names are lowercase (the XML standard) even though the web page may not be written that way. This configuration minimizes the disruption that web-authoring tools may introduce into a web design environment. Many tools automatically "fix up" web pages, making them conform to whatever conventions the tool uses when it generates HTML in WYSIWYG mode. These include trying to balance some tags, converting all tag names to uppercase, converting all attribute names to lowercase, and so on. You want your tests to be able to withstand these kinds of changes. We therefore use the NekoHTML `HTMLConfiguration` object to create a Xerces `DOMParser`, which allows us to set various NekoHTML-supported features and properties on the parser, including "convert all tag names to lowercase" and "convert all attribute names to lowercase." We recommend that you consult the NekoHTML documentation for a complete discussion of the available features and properties.

◆ Discussion

There are some important configuration notes about NekoHTML, which its web site discusses in detail. First, be sure to put `nekohtml.jar` on your runtime class path *before* your XML parser. Next, you only need `nekohtml.jar`, and not the XNI version of the parser. The most important item, however, has to do with how your web pages are written: specifically, whether the HTML tag names are uppercase or lowercase. It is very important to get this setting right, otherwise none of your XPath-based assertions will work. It is standard practice in HTML for tag names to be uppercase and attribute names to be lowercase, such as in this HTML page:

```
<HTML>
<HEAD>
<LINK href="theme/Master.css" rel="stylesheet" type="text/css">
<TITLE>Welcome!</TITLE>
</HEAD>
<BODY>
<FORM name="launchPoints" action="coffee" method="post">
Browse our <INPUT type="submit" name="browseCatalog" value="catalog">.
</FORM>
</BODY>
</HTML>
```

By default, NekoHTML is configured to expect web pages written this way; so if you write an XPath-based assertion that expects the tag `input`, the assertion fails, because the tag name is `INPUT`. The `DOMParser` object you want for your tests depends on how the web pages have been written. If they follow the HTML DOM standard of uppercase tag names and lowercase attribute names, then simply use Neko's parser with the default `HTMLConfiguration` settings. (No need to set any features.) This means that your XPath-based assertions must use uppercase for tag names and lowercase for attribute names. Listing 9.9 demonstrates:

Listing 9.9 How *not* to use `HTMLConfiguration` for NekoHTML

```
package junit.cookbook.coffee.web.test;

import org.custommonkey.xmlunit.XMLTestCase;
import org.cyberneko.html.parsers.DOMParser;
import org.w3c.dom.Document;

public class WelcomePageTest extends XMLTestCase {
    private Document welcomePageDom;

    protected void setUp() throws Exception {
        DOMParser nekoParser = new DOMParser(new HTMLConfiguration());

        nekoParser.parse(
            new InputSource(
                new FileInputStream(
                    "../CoffeeShopWeb/Web Content/"
                        + "index-DOMStandard.html")));

        welcomePageDom = nekoParser.getDocument();
        assertNotNull("Could not load DOM", welcomePageDom);
    }

    // Tests must expect tag names in UPPERCASE
    // and attribute names in lowercase to work
    // with this configuration of NekoHTML
}
```

It is important to note that if you instantiate the NekoHTML parser this way *you will not be able to set the various DOM parser features or properties*. The NekoHTML site (http://www.apache.org/~andyc/neko/doc/html/) says more about this, so if you need to know more, we suggest visiting the site. The good news is that NekoHTML provides you with a sensible default behavior (matching the HTML DOM standard), and it gives you the control you need to change that behavior when needed. If you want to use NekoHTML to verify that your web pages comply with the XHTML standard, then follow these steps:

1. Create the DOM Parser with the NekoHTML configuration, as we did in this recipe.
2. Change the property configurations for `names/elems` and `names/attrs` to `match`, rather than `lower`.
3. Write your tests to expect all tag names and attribute names to be lowercase.

If you configure your parser this way, then your XPath-based assertions will only pass if the web pages themselves have lowercase tag names and attribute names, per the XHTML standard.

As we were writing this, Tidy is not supported on Windows, although you *can* obtain unsupported binaries and try it out yourself. If you would like to use Tidy outside Java—after all, it is a useful tool on its own—then you need to explore the Tidy web site to examine your options. In spite of its unsupported status, there is a thriving user community around Tidy, so if you have questions, we are confident you can find the help you need. You may simply have to be a bit more patient. Also be aware that as of this writing JTidy had not released new code since August 2001, so you may be better off choosing NekoHTML. That said, please consult NekoHTML's site for its own limitations and problems, one of the most important being *you cannot use Xerces-J 2.0.1 as your XML parser.* At press time, the latest version of NekoHTML does not work with this particular version of the popular XML parser.

NOTE *JTidy is alive!*—Just before we went to press a reviewer brought to our attention that there is activity on the JTidy project. For their development releases—the first ones in about 18 months—the JTidy folks are concentrating on adding tests, which is always a good sign. So far, they have managed to get 63 of their 185 tests to pass. Naturally, we support their efforts and look forward to improved versions of JTidy in the future!

We hope that XHTML grows in popularity, because it is much easier to test web pages written in XHTML than web pages written in straight HTML. However, we are not holding our breath, because what is truly important to users is whether their browser can render a web page. Those browsers are *very* forgiving, and as long as they can process horrendous examples of HTML then we will need solutions such as Tidy or NekoHTML. We suspect that nothing will change until browsers simply stop rendering HTML in favor of XML with cascading stylesheets. Once again, we are not holding our breath.

- ***Related***
  - 9.1—Verify the order of elements in a document
  - 9.2—Ignore the order of elements in an XML document
  - 9.3—Ignore certain differences in XML documents
  - NekoHTML (http://www.apache.org/~andyc/neko/doc/html/)
  - JTidy (http://sourceforge.net/projects/jtidy)
  - Chapter 12—Testing Web Components
  - Chapter 13—Testing J2EE Applications

9.6 Test an XSL stylesheet in isolation

- ***Problem***

  You want to verify an XSL stylesheet outside the context of the application that uses it.

- ***Background***

  Despite the proliferation of JSPs in the J2EE community, it is not the only way to build a web application's presentation layer. One strong alternative is XSL transformation, which takes data in XML format and presents it as HTML to the end user. We have worked on projects that used this technology and we find that it has one great benefit over JSPs: XSL transformation does not require an application server. We like this a great deal, because it means we can test such a presentation layer entirely outside any kind of J2EE container. All we need are parsers and transformers, such as Xerces and Xalan (http://xml.apache.org/).

  If you are working on such a project, you may feel tempted not to test your XSL stylesheets in isolation. You reason, as we have reasoned in the past, "We will test the XSL stylesheets when we test the application end to end." You want to avoid duplicating efforts, which is a laudable goal, so you decide not to treat your XSL stylesheets as independent units worthy of their own tests. Or, you may simply not know how to do it![13] We can tell you from direct, painful, personal experience that this is an error in judgment. Transforming XML is a nontrivial operation fraught with errors. If you need convincing, look at the size of Michael Kay's excellent reference work *XSLT: Programmer's Reference*—it's slightly more than 1,000 pages. That tells us that XSL transformations are complex enough to break, and

[13] That was *our* excuse. Get your own.

when they *do* break, you will be glad to have isolated tests that you can execute to help you determine the cause of the defect. We strongly recommend testing XSL stylesheets in isolation, and this recipe describes two approaches.

◆ Recipe

Testing an XSL stylesheet is about verifying that your templates do what you think they do. The general approach consists of transforming a representative set of XML documents and verifying the result, so the techniques we use here are the XML comparison techniques used in the rest of this chapter. We hard code simple XML documents in our tests—usually as `Strings`—then apply the XSL transformation and make assertions about the resulting XML document. As with any other XML document tests, XMLUnit provides two main strategies for verifying content: XPath-based assertions on parts of the actual document, or comparing an expected document against the actual document. Fortunately, XMLUnit provides some convenience methods to help us do either. To illustrate this technique, we return to our Coffee Shop application and test the XSL stylesheet that displays the content of a shopper's shopcart.

The structure of the "shopcart" document is a simple one: a shopcart contains items and a subtotal. Each shopcart item describes its own details: the coffee name, quantity, unit price, and total price. Listing 9.10 shows a sample XML document showing three items in the shopcart:

Listing 9.10 A shopcart as an XML document

```
<?xml version="1.0" encoding="UTF-8"?>

<shopcart>
    <item id="762">
        <name>Special Blend</name>
        <quantity>1</quantity>
        <unit-price>$7.25</unit-price>
        <total-price>$7.25</total-price>
    </item>
    <item id="903">
        <name>Huehuetenango</name>
        <quantity>2</quantity>
        <unit-price>$6.50</unit-price>
        <total-price>$13.00</total-price>
    </item>
    <item id="001">
        <name>Colombiano</name>
        <quantity>3</quantity>
        <unit-price>$8.00</unit-price>
        <total-price>$24.00</total-price>
```

```
    </item>
    <subtotal>$44.25</subtotal>
</shopcart>
```

NOTE *Effective application design with XSLT*—You may notice that the data in this document is preformatted—the currency values are formatted as such—and that there is derived data that we could have calculated on demand. We recommend delivering data to the presentation layer already formatted and fully calculated. The presentation layer ought not do anything other than decide how to present data; calculations are business logic and formatting currency is application logic. The more we separate responsibilities in this way, the easier it is to write isolated tests.

Now to our first test. We need to verify that our stylesheet correctly renders an empty shopcart. Before we provide the code for this test, we ought to mention that these tests verify *content*, and are not meant to examine the look-and-feel of the resulting web page. In general, nothing is as effective as visual inspection for ensuring that a web page looks the way it should.[14] Moreover, if our tests depend too much on the layout of web pages, then they become overly sensitive to purely cosmetic changes. We are wary of anything that unnecessarily discourages us from changing code. Listing 9.11 shows the test, which expects to see an HTML table for the shopcart, no rows representing shopcart items, and a $0.00 subtotal.

Listing 9.11 DisplayShopcartXslTest

```
package junit.cookbook.coffee.presentation.xsl.test;

import java.io.*;
import javax.xml.transform.*;
import javax.xml.transform.stream.StreamSource;
import org.custommonkey.xmlunit.XMLTestCase;
import org.custommonkey.xmlunit.Transform;
import org.w3c.dom.Document;

public class DisplayShopcartXslTest extends XMLTestCase {
    private String displayShopcartXslFilename =
        "../CoffeeShopWeb/Web Content/WEB-INF/template"
            + "/style/displayShopcart.xsl";

    private Source displayShopcartXsl;
```

[14] Sure, there are automated ways to verify some aspects of a window's layout, but in our experience, the return on investment is low compared to spending the time to just look at the page.

```
    protected void setUp() throws Exception {
        displayShopcartXsl =
            new StreamSource(
                new FileInputStream(displayShopcartXslFilename));
    }

    public void testEmpty() throws Exception {
        String shopcartXmlAsString =
            "<?xml version=\"1.0\" ?>"
                + "<shopcart>"
                + "<subtotal>$0.00</subtotal>"
                + "</shopcart>";

        Document displayShopcartDom =
            doDisplayShopcartTransformation(shopcartXmlAsString);

        assertXpathExists(
            "//table[@name='shopcart']",
            displayShopcartDom);

        assertXpathEvaluatesTo(
            "$0.00",
            "//table[@name='shopcart']//td[@id='subtotal']",
            displayShopcartDom);

        assertXpathNotExists(
            "//tr[@class='shopcartItem']",
            displayShopcartDom);
    }

    public Document doDisplayShopcartTransformation(
        String shopcartXmlAsString)
        throws
            TransformerConfigurationException,
            TransformerException {

        Source shopcartXml =
            new StreamSource(
                new StringReader(shopcartXmlAsString));

        Transform transform =
            new Transform(shopcartXml, displayShopcartXsl);

        return transform.getResultDocument();
    }
}
```

The method `doDisplayShopcartTransformation()` performs the transformation we need to test. It uses the XMLUnit class `Transform` to simplify applying the transformation and to retrieve the resulting document.

The test builds an empty shopcart XML document as a `String`, applies the transformation, then makes XPath-based assertions on the resulting document. In particular, it expects the following things:

- A `table` representing the shopcart, which the test finds by examining its name.
- A table data cell (`td`) inside the `table` containing the shopcart subtotal amount, which the test finds by examining its ID.
- No table rows (`tr`) inside the `table` using the stylesheet class `shopcartItem`, which is how the test detects the existence of shopcart items on the page.

NOTE *Identifying the interesting parts of a web page*—You may have noticed that we use IDs, names, and other identifying text to help the test find the specific part of the web page it needs to verify. Labeling the various "interesting" parts of a web page is a core web-testing technique. Without it, tests would need to verify content by examining the relative positioning of HTML tags on the page. This kind of *physical* coupling makes tests sensitive to layout changes, which makes testing more expensive. Instead, we add a small amount of *logical* coupling by identifying the parts of the web page on which the tests need to do their work. At worst, someone changing the web page needs to make sure that these identifiers continue to identify the same information on the page. This requires some extra attention on the part of the web designer, but is more than worth the effort.

Because XSL stylesheets are generally quite long, we prefer not to distract you by showing the one we use to display the shopcart in its entirety. Refer to solution A.4, "Test an XSL stylesheet in isolation" to see a more complete suite of tests and an XSL stylesheet that makes them pass. As we add more tests, we will begin to repeat some of our XPath-based assertions. We recommend that you be aware of the duplication and extract custom assertions for the ones you use most often. See recipe 17.4, "Extract a custom assertion," for an example of this technique.

◆ Discussion

We mentioned an alternative technique, which involves comparing an expected web page against the actual web page that the transformer creates. These tests are easier to write, but can become difficult to maintain, as purely cosmetic changes to the presentation layer require you to update the expected web pages. The tests generally consist of extracting data from a web page, building objects that represent that data, then using `assertEquals()` to compare the expected data against the actual data. Because this amounts to parsing a web page, any change in its layout is likely to affect the parsing code. It is essential that you extract this parsing

code to a single place in your class hierarchy, as it is likely to change. Over time, if you refactor the XPath-based assertions mercilessly, you will find yourself building a small library of XPath queries that handle HTML elements such as tables, forms, and paragraphs. If you continue in this direction, you will eventually build another version of HtmlUnit, so once you recognize that you are heading in that direction, we recommend you simply start using HtmlUnit. See chapter 13, "Testing J2EE Applications," for recipes involving this web application-testing package.

In the process of preparing the complete solution for this recipe, we ran into a nasty problem: our XML document was too long to be all on one line. Because we built the XML document as a `String`, we did not pay attention to line breaks, as we would if we were writing a text file. At some point, some component the test uses (maybe XMLUnit, maybe the transformer; we did not bother to find out) began to truncate data. It turns out that we could solve the problem by adding line breaks to the XML document in our test. Very annoying. The good news is this: it might have taken hours and hours to narrow down the problem without all these tests. As it was, it still took over 15 minutes, but at least we found and solved the problem relatively quickly.

◆ Related

- Chapter 13—Testing J2EE Applications
- 17.4—Extract a custom assertion
- A.4—Test an XSL stylesheet in isolation (complete solution)

9.7 Validate XML documents in your tests

◆ Problem

You want your tests to validate the XML documents your application uses.

◆ Background

Most of you began wanting to validate XML documents after being bitten by recurring defects related to invalid XML input to some part of your system. Some teams, for example, validate their Struts configuration file (`struts-config.xml`) in order to avoid discovering configuration errors at runtime. For some configuration errors, the only recovery step is to fix the configuration error and *restart the application server*. During development and testing, restarting the application server is a time-consuming annoyance, and during production it may not be possible

until a predetermined time of day, week, or month! If you are in this position yourself, then you can appreciate the desire to prevent these configuration errors before they have the opportunity to adversely affect the system.

◆ Recipe

You can either add validation to your JUnit tests or perform validation somewhere else. As this is a book about JUnit, we will describe the second strategy briefly and focus on the first. There are two broad classes of XML documents that you may be using in your application: configuration documents and data transfer documents. We recommend that you validate configuration documents as part of your build process. We also recommend that you validate data transfer documents in your tests, so that you can safely disable validation during production. Let us explain what we mean by each of these recommendations.

A configuration document is generally a file on disk that your system uses to configure itself. The Struts web application framework's `struts-config.xml` is an example of a configuration document. You edit this document to change, for example, the navigation rules of your application, then Struts uses this document to implement those navigation rules. A configuration document typically changes outside the context of your system's runtime environment—that is, the system does not create configuration documents, but merely uses them. It is wise to validate configuration documents against a Document Type Definition (DTD) or XML schema, if one is available, as this helps you avoid discovering badly formed or syntactically incorrect configuration files before your system attempts to use them. For example, if you build and test a Struts-based application using Ant, add a target such as this to your Ant buildfile:

```
<target name="validate-configuration-files">
    <xmlvalidate warn="false">
        <fileset dir="${webinf}" includes="struts-config.xml,web.xml"/>
        <dtd publicId="-//Apache Software Foundation//
        ➢ DTD Struts Configuration 1.1//EN"
             location="dtd/struts-config_1_1.dtd" />
       <dtd publicId="-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
             location="dtd/web-app_2_3.dtd" />
    </xmlvalidate>
</target>
```

This target validates both the Struts configuration file and the application's web deployment descriptor against their respective DTDs. We recommend having your "test" target depend on this one, to ensure that whenever you execute your tests, you also validate these important configuration files. If the configuration files are

incorrect, then your test run may mean nothing, as it may attempt to test an incorrectly configured system. Just because you have a testing framework does not mean that that is the *only* place to do testing. Save yourself some irritation and validate configuration files *before* you run your tests.

A data transfer document is a way to transfer data from one tier of your application to another. We commonly use data transfer documents to build an XSLT-based presentation layer. The model executes business logic and returns Java objects to the controller, which then formats that data as an XML document and submits it to the presentation layer. The presentation layer transforms the document into a web page to display to the end user. The system generates data transfer documents at runtime, some of which share a specific, predictable structure. If you find your project having a problem with garbage-in, garbage-out in this part of your application, then we recommend that you add tests that validate that each data transfer document conforms to the structure you expect. You can use either DTDs or XML schemas for this purpose, although there is always the risk of overusing XML schemas. See the Discussion section for details. To validate data transfer documents, locate the XML parsers in your system and make it possible to configure them to validate incoming documents. This generally requires some refactoring on your part.

For example, if you use XSLT as your presentation engine, then some object somewhere is responsible for performing the transformation—it is either your Controller or some object that it uses. In the Coffee Shop application, we can configure the `CoffeeShopController` servlet to perform XSL transformations. We need the ability to enable validation on this XSL transformation service so that, when we execute our tests, we can validate the incoming XML document against its DTD or declared XML schema. This can be as simple as adding a method named `setValidateIncomingDocument()` which, when invoked, causes the underlying service (in this case, our XSL transformation service) to validate the incoming XML document before passing it through the XSL transformer. Implementing this feature involves nothing more complex than creating an XML parser, enabling validation, and parsing a document, but for an example implementation, see solution A.5, "Validate XML documents in your tests." The key part, from our perspective, is enabling this feature in our tests. To do that, we simply make it part of our test fixture. (See recipe 3.4, "Factor out a test fixture," and recipe 3.5, "Factor out a test fixture hierarchy," for details on managing test fixtures.) Here is the test fixture method[15] that performs an XSL translation with document validation enabled:

[15] A method can be part of a fixture, just as a variable is part of a fixture. We may eventually refactor the fixture and move such a method into a production class, but sometimes it really is just a tool for the test.

```
public Document doDisplayShopcartTransformation(
    String shopcartXmlAsString)
    throws Exception {

    TransformXmlService service =
        new TransformXmlService(displayShopcartXslReader);

    service.setSourceDocumentValidationEnabled(true);

    DOMResult documentResult = new DOMResult();

    service.transform(
        new StreamSource(
            new StringReader(shopcartXmlAsString)),
        documentResult);

    assertTrue(
        "Incoming XML document failed validation: "
            + service.getValidationProblems(),
        service.isSourceDocumentValid());

    return (Document) documentResult.getNode();
}
```

We used this technique in our tests for presenting shopcart contents. The data transfer document in our tests did not specify the product IDs for the products in the shopcart, which would cause problems for the end user. When we executed the tests with validation enabled, this is the error message we received:

```
junit.framework.AssertionFailedError: Incoming XML document
failed validation: [org.xml.sax.SAXParseException: Attribute
"id" is required and must be specified for element type "item".]
```

This message is certainly more helpful than finding out about the problem at runtime, where the symptom is less obvious: the "Buy!" button on the shopcart page (which uses the product ID in its HTML element name) would not work because there is no way to identify the brand of coffee that the shopper wants to buy. When we added an ID to the `item` tag in the data transfer document, the tests all passed. This recipe provides a way to make problems obvious, which is always a good idea.

NOTE *Make problems obvious*—Rather than infer the cause of a defect from a secondary symptom, write your tests in such a way that the problem becomes obvious. Without validating the shopcart data transfer document, all we would know is that one of our end-to-end tests fails because the "Buy!" button does not work. There are a few reasons this could fail: the XSL stylesheet could be wrong, the "Add Coffee to Shopcart" action could be

wrong, there could be data corruption, or there might be no coffee matching that particular product ID. By validating the data transfer document in the tests, the cause of the defect becomes obvious and, most importantly, inexpensive to fix.

Now whenever we execute our tests, we add another layer of problem isolation. If the XML document we pass as input to our test is invalid, then document validation fails before the object under test tries to process it, making it clear whether the problem is bad input or an incorrectly behaving XML-processing component.

◆ Discussion

Bear in mind that validating XML is expensive, particularly if doing so forces you to parse the same XML document more than once. If your system is passing data transfer documents to other components in your system, then you can feel safe turning validation off in production. After all, your tests will catch the vast majority of problems that you might have with those documents (at least the problems you know about). If, instead, you are receiving data transfer documents from a component outside your control—either from another group in your organization or someone outside your organization—then we recommend leaving validation enabled, even in production. If nothing else, it quickly settles the question of who is responsible for a problem: you or them.[16]

There is one trap to avoid when validating data transfer documents against XML schemas. The power of the XML schema is its expressiveness: it can validate structure *and* data, leveraging the power of regular expressions. As in any similar situation, you need to be ever aware of the power you have available and be careful not to overstep your bounds. In particular, it becomes tempting to validate every little thing you can in your XML schemas. For example, you may be tempted to verify that the shipping charge in a Purchase Order document is less than $10 when the order contains over $300 worth of goods. After all, XML schemas allow you to do this, so why not? The problem is simple: that is a business rule, and a data transfer document—a simple data structure that your system passes between layers—is the wrong place to validate business rules. Why? Because changes in business rules ought not to affect the system's ability to generate XML documents from a `PurchaseOrder` object! This is a clear sign that the design needs work.

[16] We are not concerned with assessing blame, but it is important to assess responsibility, because someone has to fix it. Better it be the programmer whose component is actually broken.

Validate business rules by testing your business rules; stick to just validating structure and formatting in your data transfer document tests.[17]

- ***Related***
  - 3.4—Factor out a test fixture
  - 3.5—Factor out a test fixture hierarchy
  - A.5—Validate XML documents in your tests

[17] Of course, if you process business rules using an XML-based engine then you may perform XML schema validation in your tests. It is better to separate the tests rather than have one try to do two things.

JAVA

JUnit Recipes Practical Methods for Programmer Testing

J. B. Rainsberger with contributions by **Scott Stirling**

FOREWORD BY
Robert C. Martin

When testing becomes a developer's habit good things tend to happen—good productivity, good code, and good job satisfaction. If you want some of that, there's no better way to start your testing habit, nor to continue feeding it, than with *JUnit Recipes*. In this book you will find one hundred and thirty-seven solutions to a range of problems, from simple to complex, selected for you by an experienced developer and master tester. Each recipe follows the same organization giving you the problem and its background before discussing your options in solving it.

JUnit—the unit testing framework for Java—is simple to use, but some code can be tricky to test. When you're facing such code you will be glad to have this book. It is a how-to reference on all issues of testing, from how to name your test case classes to how to test complicated J2EE applications. Its valuable advice includes side matters that can have a big payoff, like how to organize your test data or how to manage expensive test resources.

What's Inside

- How testing *saves* time
- Recipes for servlets, JSPs, EJBs, database code ...
- Difficult-to-test designs, how to fix them
- The right JUnit extension for the job: HTMLUnit, XMLUnit, ServletUnit, EasyMock, and more!

J. B. Rainsberger is a developer and consultant who has been a leader in the JUnit community since 2001. His popular online tutorial *JUnit: A Starter Guide* is read by thousands of new JUnit users each month. Joe lives in Toronto, Canada

"No other unit testing book [offers] as much wisdom, knowledge, and practical advice .... a remarkable compendium ..."

—Robert C. Martin
from the Foreword

"Study this book! It will zoom you along a paved road to expertise."

—Brian Marick, author,
The Craft of Software Testing

"... a compelling argument for how testing increases productivity and quality."

—Michael Rabbior, IBM

"JB's approach: been there, done that, don't do it please."

—Vladimir Ritz Bossicard
JUnit Development Team

"... a 'pattern reference'—it will have stuff to mine for years to come ..."

—Eric Armstrong
author of *JBuilder2 Bible*
consultant for Sun Computing


Ebook edition

www.manning.com/rainsberger

ISBN 1-932394-23-0

$49.95 US/$69.95 Canada