❶ You're making an out-of-JVM call to the container, so you need to specify the JNDI properties.

❷ Perform a JNDI lookup to retrieve the `PetstoreEJB` home instance.

At this stage, you have the deployment working and the test case written. If you try running the tests by typing **`ant test`**, you'll discover that it still fails. You still need to adjust the JNDI names: You must match the JNDI names you use to look up objects with the JNDI names under which JBoss publishes the objects.

### 12.9.5 Fixing JNDI names

The JNDI names used in the application are defined in `JNDINames.java`, as shown in listing 12.23.

**Listing 12.23 JNDI names used to look up the J2EE objects**

```
package junitbook.ejb.util;

public abstract class JNDINames
{
    public static final String QUEUE_CONNECTION_FACTORY =
        "ConnectionFactory";
    public static final String QUEUE_ORDER =
        "queue/petstore/Order";
    public static final String ORDER_LOCALHOME =
        "ejb/petstore/Order";
    public static final String PETSTORE_HOME =
        "ejb/petstore/Petstore";
}
```

In order to publish the objects under these names, you need to define a JBoss-specific `jboss.xml` file (see listing 12.24) that matches the JNDI names defined in `JNDINames.java`.

**Listing 12.24 jboss.xml with JNDI names matching JNDINames.java**

```
<jboss>
  <enterprise-beans>

    <session>
      <ejb-name>Petstore</ejb-name>
      <jndi-name>ejb/petstore/Petstore</jndi-name>
      <resource-env-ref>
        <resource-env-ref-name>
        →   jms/queue/petstore/Order</resource-env-ref-name>
        <jndi-name>queue/petstore/Order</jndi-name>
      </resource-env-ref>
```

```
    </session>

    <entity>
      <ejb-name>Order</ejb-name>
      <local-jndi-name>ejb/petstore/Order</local-jndi-name>
    </entity>

    <message-driven>
      <ejb-name>OrderProcessor</ejb-name>
      <destination-jndi-name>
 →    queue/petstore/Order</destination-jndi-name>
    </message-driven>

  </enterprise-beans>
</jboss>
```

You must also modify the `ejbjar` Ant target so that it picks the new JBoss-specific files you have added (changes are in bold):

```
<target name="ejbjar" depends="compile">
  <jar destfile="${target.dir}/ejb.jar">
    <metainf dir="${conf.dir}">
      <include name="ejb-jar.xml"/>
      <include name="jbosscmp-jdbc.xml"/>
      <include name="jboss.xml"/>
    </metainf>
    <fileset dir="${target.classes.java.dir}"/>
  </jar>
</target>
```

### 12.9.6 Running the tests

At last, you're ready to run the tests. They should now execute fine. Typing **`ant test`** generates the results shown in figure 12.7.

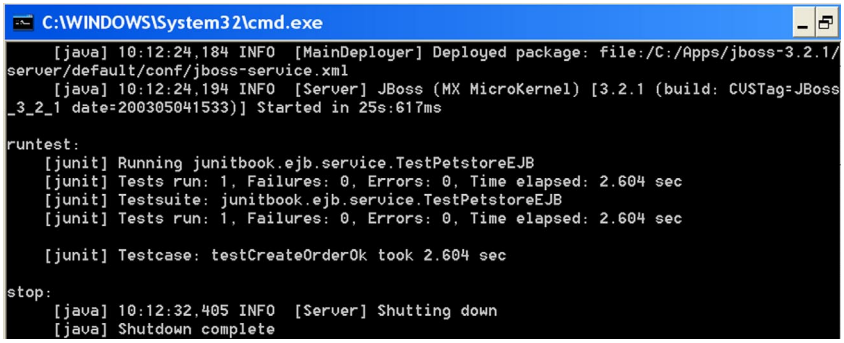


**Figure 12.7 Successful execution of pure JUnit tests calling the running container remotely**

## 12.10 Using Cactus

Let's now use Cactus to run some EJB unit tests. Cactus has several advantages over a pure JUnit solution:

- Cactus lets you unit-test Enterprise Beans using local interfaces, because Cactus tests run inside the container. For example, you have not been able to perform integration unit tests for the `Order` CMP entity bean (which uses a local interface). We'll demonstrate how to do this using Cactus.
- The Ant scripts were a bit complex because you had to script the container's start and stop (JBoss, in this case). Cactus provides an Ant integration that simplifies this operation. In addition, this Cactus Ant task supports several containers out of the box (JBoss, Tomcat, Resin, Orion, WebLogic, and so on), making it easy to run the tests on any container.

At this time of this writing, Cactus doesn't yet provide EJB Redirectors you can use to directly write tests against EJBs as you have done for servlets, taglibs, and filters in previous chapters.[3] Thus, you can't yet perform fine-grained integration tests, such as testing exceptions cases.

### 12.10.1 Writing an EJB unit test with Cactus

The current Cactus solution consists of transparently using the Cactus servlet Redirector so that the tests are executed within the context of the web container. For the pure JUnit solution, the tests perform a lookup on the EJB to unit-test and call its method to test. The difference is that this lookup is performed from the web container context and thus also works for local interfaces.

Let's demonstrate this on the `OrderEJB` CMP entity bean (listing 12.25).

**Listing 12.25 OrderEJB unit test as a Cactus ServletTestCase**

```
package junitbook.ejb.domain;

import java.util.Date;

import javax.naming.InitialContext;

import junit.framework.TestCase;
import junitbook.ejb.util.JNDINames;

public class TestOrderEJB extends ServletTestCase
```

[3] The addition of EJB redirectors is scheduled for Cactus 1.6 or later (it is on the todo list: http://jakarta.apache.org/cactus/participating/todo.html).

```
{
    public void testEjbCreateOk() throws Exception
    {
        OrderLocalHome orderHome =
            (OrderLocalHome) new InitialContext().lookup(
                JNDINames.ORDER_LOCALHOME);

        Date date = new Date();
        String item = "item 1";

        OrderLocal order = orderHome.create(date, item);

        assertEquals(date, order.getOrderDate());
        assertEquals(item, order.getOrderItem());
        assertEquals(new Integer(date.hashCode()
            + item.hashCode()), order.getOrderId());
    }
}
```

❶ **Lookup executed from inside container**

Compared to listing 12.22, you don't need to set up JNDI properties before calling `new InitialContext()` (❶) because the test runs inside the container.

Let's now run this test using Ant.

### 12.10.2 Project directory structure

Figure 12.8 defines the directory structure for the Cactus tests. Compared to the directory structure from figure 12.4, you add the `conf/cactus/` directory, which contains configuration files for running Cactus tests, and the `src/test-cactus/` directory, which contains the Cactus unit tests. You also add an Ant `build-cactus.xml` buildfile that contains the Ant targets to automatically execute the Cactus tests using the Cactus/Ant integration. (For more details on the Cactus/Ant integration, refer to section 11.5.2 from chapter 11.)

You'll use the same Ant script introduced in listing 12.16 to package the application as an ear file. However, you need to add some targets to package the Cactus tests and execute them.

### 12.10.3 Packaging the Cactus tests

Because the Cactus tests execute in the web container, you need to package them in a war inside the ear. Fortunately, Cactus provides a `cactifywar` Ant task that makes the creation of this war easy:

```
<cactifywar version="2.3" destfile="${target.dir}/cactus.war"
    mergewebxml="${conf.dir}/cactus/web.xml">
  <classes dir="${target.classes.cactus.dir}"/>
</cactifywar>
```

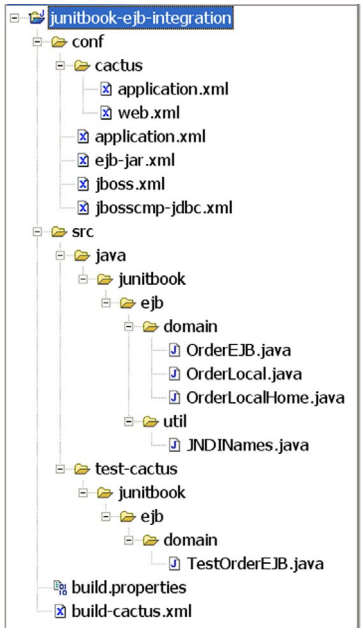


**Figure 12.8**
**Full directory structure, including configuration files and Ant build files for Cactus tests**

The `version` attribute specifies that you build a war for the Servlet API 2.3. Notice the `mergewebxml` attribute. This is needed because `TestOrderEJB` is called from a web context and is calling an EJB. Thus, as the J2EE specification mandates, you need an `<ejb-local-ref>` in the war `web.xml` file, as shown in listing 12.26.

**Listing 12.26 conf/cactus/web.xml containing ejb-local-ref entry to call OrderEJB**

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
   <ejb-local-ref>
      <ejb-ref-name>Order</ejb-ref-name>
      <ejb-ref-type>Entity</ejb-ref-type>
      <local-home>junitbook.ejb.domain.OrderLocalHome</local-home>
      <local>junitbook.ejb.domain.OrderLocal</local>
      <ejb-link>Order</ejb-link>
   </ejb-local-ref>
</web-app>
```

The last packaging step is to package the war in the ear:

```
<ear update="true" destfile="${target.dir}/ejb.ear"
    appxml="${conf.dir}/cactus/application.xml">
  <fileset dir="${target.dir}">
    <include name="cactus.war"/>
  </fileset>
</ear>
```

You update the application ear by overwriting the `application.xml` deployment descriptor with the special Cactus one. This step is necessary in order to add the Cactus war in `application.xml`, as shown in listing 12.27. This step is required only because your production ear doesn't container any war. Had it contained one, you could have reused this war by cactifying it. In that case, you wouldn't have needed to provide an additional `application.xml` file containing Cactus-specific definitions.

**Listing 12.27 Special application.xml file containing the Cactus war definition**

```
<?xml version="1.0"?>
<!DOCTYPE application PUBLIC
  '-//Sun Microsystems, Inc.//DTD J2EE Application 1.2//EN'
  'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>
<application>
  <display-name>ejb</display-name>
  <description>EJB Chapter Sample Application</description>
  <module>
    <ejb>ejb.jar</ejb>
  </module>
  <module>
    <web>
      <web-uri>cactus.war</web-uri>
      <context-root>test</context-root>
    </web>
  </module>
</application>
```

The full Ant script is shown in listing 12.28.

**Listing 12.28 Cactification of the application ear**

```
<?xml version="1.0"?>

<project name="Ejb" default="test" basedir=".">
[...]
  <property name="src.cactus.dir"
      location="${src.dir}/test-cactus"/>
[...]
  <property name="target.classes.cactus.dir"
      location="${target.dir}/classes-test-cactus"/>
[...]
  <target name="compile.cactus" depends="compile">
    <mkdir dir="${target.classes.cactus.dir}"/>
    <javac destdir="${target.classes.cactus.dir}"
        srcdir="${src.cactus.dir}">
      <classpath>
        <pathelement location="${target.classes.java.dir}"/>
        <pathelement location="${cactus.jar}"/>
        <pathelement location="${j2ee.jar}"/>
      </classpath>
    </javac>
  </target>

  <target name="ear.cactify" depends="compile.cactus,ear">

    <taskdef resource="cactus.tasks">
      <classpath>
        <pathelement location="${cactus.ant.jar}"/>
        <pathelement location="${cactus.jar}"/>
        <pathelement location="${logging.jar}"/>
        <pathelement location="${aspectjrt.jar}"/>
        <pathelement location="${httpclient.jar}"/>
      </classpath>
    </taskdef>

    <cactifywar version="2.3" destfile="${target.dir}/cactus.war"
        mergewebxml="${conf.dir}/cactus/web.xml">
      <classes dir="${target.classes.cactus.dir}"/>
    </cactifywar>

    <ear update="true" destfile="${target.dir}/ejb.ear"
    appxml="${conf.dir}/cactus/application.xml">
      <fileset dir="${target.dir}">
        <include name="cactus.war"/>
      </fileset>
    </ear>

  </target>

</project>
```

### 12.10.4 Executing the Cactus tests

You execute the Cactus tests using the Cactus-provided `cactus` task. This is the nice part, compared to the pure JUnit approach from section 12.9, because the `cactus` task does everything for you: It deploys the ear, starts the container, executes the tests, and stops the container. The `cactus` task extends the `junit` JUnit Ant task and thus inherits from all its features. Listing 12.29 demonstrates how to use the `cactus` task to run the `TestOrderEJB` test.

**Listing 12.29 Running Cactus tests automatically with the cactus task**

```
<?xml version="1.0"?>

<project name="Ejb" default="test" basedir=".">
[...]
  <target name="test" depends="ear.cactify">

    <cactus earfile="${target.dir}/ejb.ear" fork="yes"
        printsummary="yes" haltonerror="true"
        haltonfailure="true">
      <containerset>
        <jboss3x dir="${cactus.home.jboss3x}"
            output="jbossresult.txt"/>
      </containerset>
      <formatter type="brief" usefile="false"/>
      <test name="junitbook.ejb.domain.TestOrderEJB"/>
      <classpath>
        <pathelement location="${target.classes.java.dir}"/>
        <pathelement location="${target.classes.cactus.dir}"/>
      </classpath>
    </cactus>

  </target>

</project>
```

**Run Cactus tests in JBoss container**

Executing the tests by typing **`ant -f build-cactus.xml test`** yields the result shown in figure 12.9.

```
C:\WINDOWS\System32\cmd.exe

C:\junitbook\ejb\integration>ant -f build-cactus.xml test
Buildfile: build-cactus.xml

compile:
    [mkdir] Created dir: C:\junitbook\ejb\integration\target\classes
    [javac] Compiling 11 source files to C:\junitbook\ejb\integration\target\classes

compile.cactus:
    [mkdir] Created dir: C:\junitbook\ejb\integration\target\classes-test-cactus
    [javac] Compiling 1 source file to C:\junitbook\ejb\integration\target\classes-test-cactus

ejbjar:
      [jar] Building jar: C:\junitbook\ejb\integration\target\ejb.jar

ear:
      [ear] Building ear: C:\junitbook\ejb\integration\target\ejb.ear

ear.cactify:
[cactifywar] Building war: C:\junitbook\ejb\integration\target\cactus.war
      [ear] Updating ear: C:\junitbook\ejb\integration\target\ejb.ear

test:
   [cactus] -------------------------------------------------------------
   [cactus] Running tests against JBoss 3.2
   [cactus] -------------------------------------------------------------
   [cactus] Running junitbook.ejb.domain.TestOrderEJB
   [cactus] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.741 sec
   [cactus] Testsuite: junitbook.ejb.domain.TestOrderEJB
   [cactus] Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 0.741 sec

   [cactus] Shutdown complete

BUILD SUCCESSFUL
Total time: 31 seconds
C:\junitbook\ejb\integration>_
```

**Figure 12.9 Result of executing Cactus EJB tests with the Cactus/Ant integration**

## 12.11 Summary

EJBs are complex and powerful beasts. Unit-testing doesn't have to be difficult. This chapter has demonstrated several techniques for handling EJB unit tests: mock objects for out-of-the-container testing of any kind of EJBs (session beans, entity beans, message-driven beans) and integration unit testing for testing Enterprise Beans when they run inside the container. We demonstrated integration unit testing with two tools: pure JUnit tests that call the EJBs remotely; and Cactus, which runs the tests from inside the container and lets you unit-test local interfaces.

When you're performing integration unit tests, writing the tests isn't even half the story. The hard part, which isn't specifically related to unit testing, is about automating the packaging of the application, its deployment and test execution, and the start/stop of containers. We've demonstrated several techniques using Ant, including using some Cactus custom-made Ant tasks that have helped in this endeavor.

# A The source code

***This appendix covers***

- Installing the book source code
- Software versions required
- Directory structure conventions

This appendix gives an overview of the book's source code, where to find it, how to install it, and how to run it. When we were writing, we decided to donate all of the book's source code to the Apache Software Foundation because we've used a lot of frameworks from there in the making of this book. Thus we have made our source code available as open source on Sourceforge at http://junitbook.sourceforge.net/.

We're also committed to maintaining this source code and fixing it if bugs are found, as a standard open source project. In addition, a Sourceforge forum has been set up for discussing the code at http://sourceforge.net/forum/forum.php?forum_id=291665.

## A.1 Getting the source code

There are two possibilities for getting the source code on your local machine:

- Download a released version from http://sourceforge.net/project/showfiles.php?group_id=68011 and unzip it somewhere on your hard drive.
- Use a CVS client and get the source from CVS HEAD. Getting the source from CVS is explained at http://sourceforge.net/cvs/?group_id=68011.

Either way, place the source code in a local directory named `junitbook/` (for example `c:\junitbook` on Windows or `/opt/junitbook` on UNIX).

## A.2 Source code overview

Once you put the source code in the `junitbook/` directory, you should have the directory structure shown in figure A.1. Each directory represents the source code for a chapter of the book (except the `repository/` directory, which contains external jars required by the chapter projects). The mapping between chapter names and directory names is listed in table A.1.

Each directory maps directly to a project. A project is a way to regroup Java sources, test sources, configurations files, and so on under a single location. A project also has a build, which lets you perform various actions such as compiling the code, running the tests, and generating the Javadoc. We have used different build tools (Ant and Maven) for the different projects, as explained in the chapter matching each project.


**Figure A.1**
**Directory structure for the source code, shown here in Windows Explorer. (Note that the directories are colored by the TortoiseCVS CVS client.)**

**Table A.1 Mappings between chapter names and source directory names**

| Chapter name | Directory name |
|---|---|
| Chapter 1: JUnit jumpstart | `junitbook/jumpstart/` |
| Chapter 2: Exploring JUnit | `junitbook/exploring/` |
| Chapter 3: Sampling JUnit | `junitbook/sampling/` |
| Chapter 4: Examining software tests | `junitbook/examining/` |
| Chapter 5: Automating JUnit | `junitbook/automating/` |
| Chapter 6: Coarse-grained testing with stubs | `junitbook/coarse/` |
| Chapter 7: Testing in isolation with mock objects | `junitbook/fine/` |
| Chapter 8: In-container testing with Cactus | `junitbook/container/` |
| Chapter 9: Unit-testing servlets and filters | `junitbook/servlets/` |
| Chapter 10: Unit-testing JSPs and taglibs | `junitbook/pages/` |
| Chapter 11: Unit-testing database applications | `junitbook/database/` |
| Chapter 12: Unit-testing EJBs | `junitbook/ejb/` |

## A.3 External libraries

You may have noticed a directory named `repository/` in figure A.1. It contains the different external libraries (jars) that all the other projects need in order to compile and run. As a convenience, we're making them readily available to you to prevent you from having to fish for them all over the Net.

The directory structure of `repository/` is of the format `<library name>/jars/<library name>-<version>.jar`, as shown in figure A.2.

**NOTE** We have chosen this directory layout because it is the one needed to make the `repository/` project a remote Maven artifact repository (see chapter 5 for a presentation of the Maven repositories). When you install Maven the first time, its remote repository is configured to point to http://www.ibiblio.org/maven/, which is the official Maven repository containing hundreds of open source jars. Maven supports having several remote repositories, so adding yours is as easy as adding the following in your `build.properties`.

On Windows:

```
maven.repo.remote =
➔ http://www.ibiblio.org/maven/,file://c:/junitbook
➔ repository/
```

On UNIX:

```
maven.repo.remote =
➔ http://www.ibiblio.org/maven/,file:///opt/junitbook
➔ repository/
```

With this configuration, Maven will look for any dependency on `ibiblio` first and then in the `junitbook` repository in your filesystem.

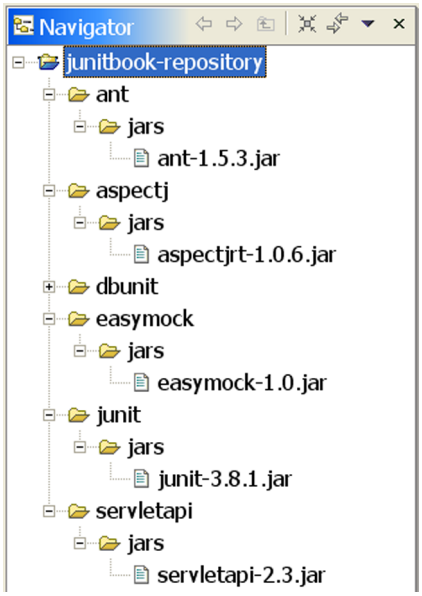


**Figure A.2**
**Some jars from the `repository/` directory, shown in the Eclipse navigator view**

## A.4 Jar versions

Table A.2 lists the versions of all external jars and applications used in the projects. We recommend using these versions when you try the book examples.

**Table A.2 External jar/application versions (sorted in alphabetical order)**

| External project name | Version | Project URL |
|---|---|---|
| Ant | 1.5.3 | http://ant.apache.org/ |
| AspectJ | 1.0.6 | http://eclipse.org/aspectj/ |
| Cactus | 1.5 | http://jakarta.apache.org/cactus/ |
| Commons BeanUtils | 1.6.1 | http://jakarta.apache.org/commons/beanutils.html |
| Commons Collections | 2.1 | http://jakarta.apache.org/commons/collections.html |
| Commons HttpClient | 2.0 | http://jakarta.apache.org/commons/httpclient/ |
| Commons Logging | 1.0.3 | http://jakarta.apache.org/commons/logging.html |
| DbUnit | 1.5.5 | http://dbunit.sourceforge.net/ |
| EasyMock | 1.0 | http://easymock.org/ |
| Eclipse | 2.1 | http://eclipse.org/ |

*continued on next page*

**Table A.2 External jar/application versions (sorted in alphabetical order)** *(continued)*

| External project name | Version | Project URL |
|---|---|---|
| HttpUnit | 1.5.3 | http://httpunit.sourceforge.net/ |
| Jakarta Taglibs / JSTL | 1.0.2 | http://jakarta.apache.org/taglibs/ |
| JBoss | 3.2.1 | http://jboss.org/ |
| Jetty | 4.2.11 | http://jetty.mortbay.org/ |
| JUnit | 3.8.1 | http://junit.org/ |
| Maven | 1.0 beta 10 | http://maven.apache.org/ |
| MockObjects | 0.09 | http://www.mockobjects.com/ |
| MockMaker plugin for Eclipse | 1.12.0 | http://www.mockmaker.org/ |
| Servlet API | 2.3 | http://www.ibiblio.org/maven/servletapi/jars/ |
| Tomcat | 4.1.24 | http://jakarta.apache.org/tomcat/ |

## A.5 Directory structure conventions

For each project, we have followed the directory conventions listed in table A.3.

**Table A.3 Directory structure conventions**

| Directory name | Explanation |
|---|---|
| `<project name>/src/java` | Java runtime sources. |
| `<project name>/src/test` | Java test sources. |
| `<project name>/src/test-cactus` | Java Cactus test sources. |
| `<project name>/src/webapp` | Web app resources (JSPs, `web.xml`, taglibs, and so on). |
| `<project name>/conf` | Configuration files (if any). |
| `<project name>/target` | Directory created by the build process (Ant or Maven) to store generated files and temporary files. It can be safely deleted, because it's re-created by the build. |

# B Eclipse quick start

***This appendix covers***

- Installing Eclipse
- Setting up the book source code in Eclipse
- Running JUnit tests in Eclipse

In this appendix, you will learn how to install Eclipse (http://eclipse.org/) and how to run the book's source code from within the IDE. This appendix is meant as a quick start to get you up and running quickly with Eclipse and with the integrated JUnit.

## B.1 Installing Eclipse

Installing Eclipse is very simple; the process consists of downloading Eclipse from http://eclipse.org/ and then unzipping it to somewhere on your hard drive. We recommend downloading Eclipse 2.1 or greater. In the remainder of this appendix, we'll assume Eclipse is installed in `[ECLIPSE_HOME]` (for example, `c:\eclipse-2.1`).

## B.2 Setting up Eclipse projects from the sources

The good news is that it's extremely easy to set up an Eclipse project, because we have provided the Eclipse project files with the book's source code distribution. Please refer to appendix A ("The source code") for directory structure organization and project names.

The first Eclipse project to import corresponds to the `junitbook/repository/` directory. It contains all the external libraries (jars) required by all the other projects. All the other Eclipse projects for this book depend upon this `repository` project for their classpath, which is why you need to import it first.

To import this project, select File→Import and then select Existing Project into Workspace. Point the Project Content to the `junitbook/repository/` directory on your hard disk, as shown in figure B.1.

Repeat the same process for all the projects you wish to see in your Eclipse workspace. If you import all the projects, you should end up with the workspace shown in figure B.2.

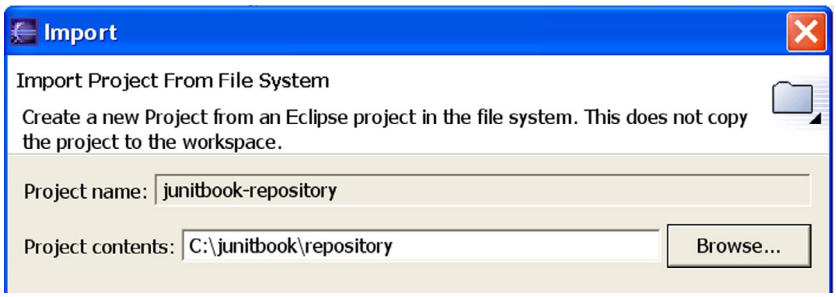


**Figure B.1 Importing the `junitbook-repository` project into Eclipse**

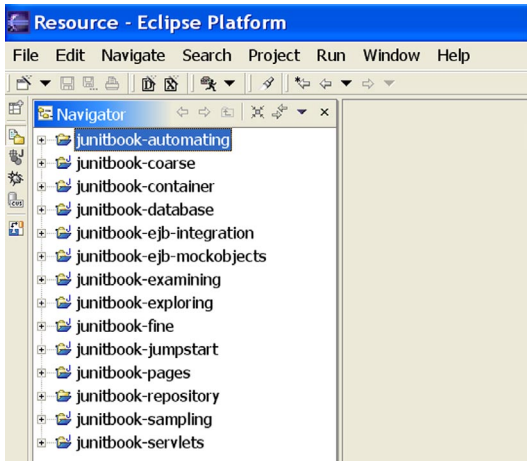


**Figure B.2**
**Eclipse workspace when all the book projects have been imported**

## B.3 Running JUnit tests from Eclipse

To run a JUnit test in Eclipse, select the Java perspective ( ), click on the test class to execute, click the Run As icon arrow ( ), and select JUnit Test. Figure B.3 shows what you'll get if you run the `TestAccount` test case found in the `junitbook-examining` Eclipse project from chapter 4.

For full details on how to run JUnit tests from Eclipse, please see the integrated Eclipse Help: Click Help→Help Contents. Then, in the Help browser, select the

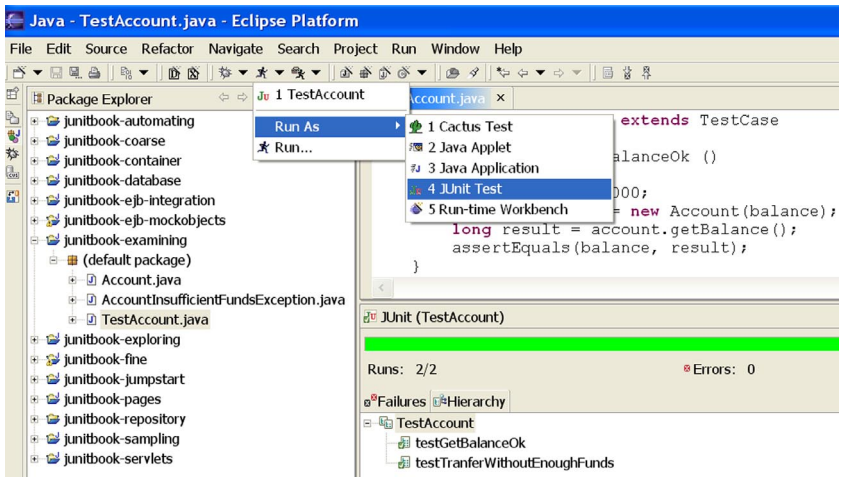


**Figure B.3 Running the `TestAccount` test case in Eclipse using the built-in JUnit plugin**

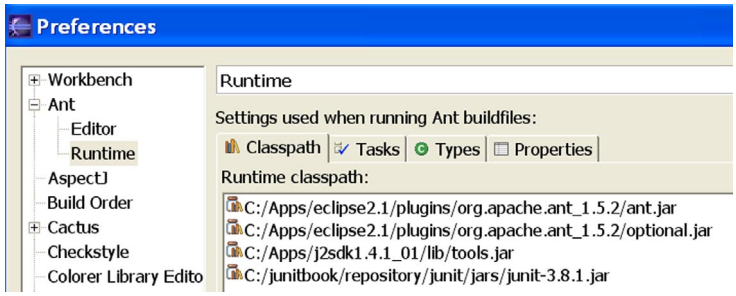


**Figure B.4 Adding the JDK `tools.jar` and the JUnit jar to the Ant classpath in Eclipse**

following topic: Java Development User Guide→Getting Started→Basic Tutorial→Writing and Running JUnit tests.

## B.4 Running Ant scripts from Eclipse

Before running Ant scripts, make sure you've added the JDK `tools.jar` library to your Ant classpath (it's needed by the Ant `javac` task). In addition, you also need to add the JUnit jar to the Ant classpath. To do so, select Window→Preferences, choose Ant→Runtime in the Preferences dialog box, and add the jars as shown in figure B.4.

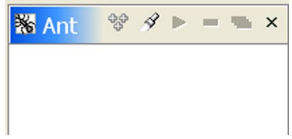


**Figure B.5 Eclipse displays the Ant view.**

To execute a target from an Ant buildfile, first tell Eclipse to display the Ant view by clicking the Window→Show View menu entry and selecting Ant. Figure B.5 shows the result.

Then, click the icon to add a buildfile to the Ant view. For example, add the `build.xml` file from the `junitbook-sampling` project. The Ant view now lists all the Ant targets it has found in the `build.xml` file, highlighting the default target (see figure B.6).

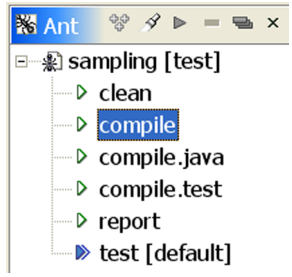


**Figure B.6 The Ant view displays all the Ant targets found in `build.xml`.**

To execute a target, select it and click the button. Figure B.7 shows the result of executing the `compile` target. Note that Eclipse captures the Ant output and displays it in the console view at the bottom right of the figure.

For full details on how to run Ant scripts from Eclipse, please see the integrated Eclipse Help: Click Help→Help Contents. Then, in the Help browser,

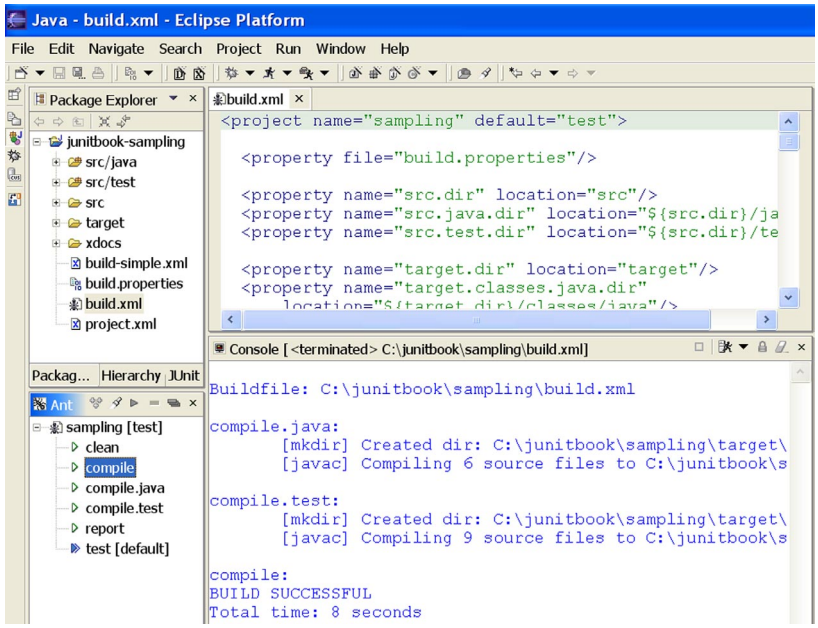


**Figure B.7 Result of executing the `compile` Ant target for the `junitbook-sampling` project**

select the following topic: Workbench User Guide→Getting Started→Ant & External Tools Tutorial→Eclipse Ant Basics.

## B.5 Running Cactus tests from Eclipse

Executing a Cactus test involves several steps: packaging the application to run as a war file, deploying it to the container, starting the container, and launching the tests using a JUnit runner. Launching a JUnit runner is easy, as demonstrated by the previous section. However, the other steps are harder to perform. The Cactus project provides two solutions to help:

- A Jetty integration, which can be run from any IDE (including Eclipse). This integration is described in detail in chapter 8 ("In-container testing with Cactus").
- A Cactus plugin for Eclipse, which lets you run Cactus tests within several containers (Resin, WebLogic, Tomcat, Orion, and so on). However, this plugin is still experimental (at the time of this writing). Full information about using the Eclipse plugin is available on the Cactus web site at http://jakarta.apache.org/cactus/integration/eclipse/index.html.

# references

### Bibliography


Alur, Deepak, John Crupi, and Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Upper Saddle River, NJ: Prentice Hall, 2001.

Beck, Kent. *Smalltalk Best Practice Patterns*. Upper Saddle River, NJ: Prentice Hall, 1996.

———. *Extreme Programming Explained: Embrace Change*. Reading, MA: Addison-Wesley, 1999.

———. *Test Driven Development: By Example*. Boston: Addison-Wesley, 2003.

Earles, John. "Frameworks! Make Room for Another Silver Bullet." http://www.cbd-hq.com/PDFs/cbdhq_000301je_frameworks.pdf.

Fowler, Martin. "The New Methodology." http://www.martinfowler.com/articles/newMethodology.html.

———. *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley, 2003.

———. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999.

Fowler, Martin, and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Reading, MA: Addison-Wesley, 2000.

Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

Hatcher, Erik, and Steve Loughran. *Java Development with Ant*. Greenwich, CT: Manning, 2003. http://www.manning.com/hatcher/.

Jeffries, Ron. On the TestDrivenDevelopment mailing list: http://groups.yahoo.com/group/testdrivendevelopment/message/3914.

Johnson, Ralph, and Brian Foote. "Designing Reusable Classes." *Journal of Object-Oriented Programming* 1.5 (June/July 1988): 22–35. http://www.laputan.org/drc/drc.html.

Marick, Brian. "How Many Bugs Do Regression Tests Find?" http://www.testingcraft.com/regression-test-bugs.html.

Potapov, Roman. "The Origin of Murphy's Law." http://www.geocities.com/murphylawsite/.

Rainsberger, J. B. "Refactoring: Replace Subclasses with Collaborators." http://www.diasparsoftware.com/articles/refactorings/replaceSubclassWithCollaborator.html.

Sisson, Derek. "Types of Tests." http://www.philosophe.com/testing/tests.html.

Walls, Craig, and Norman Richards. *XDoclet in Action*. Greenwich, CT: Manning, 2003. http://books.manning.com/walls/.

### *Software directory*

The software packages listed here are covered by the main text. Appendix A also provides a detailed list of the software packages and versions used by the book's source code.

**Table R.1 Software directory**

| Name | Web site | Quick description |
|---|---|---|
| Ant | http://ant.apache.org/ | Build tool |
| AspectJ | http://eclipse.org/aspectj/ | AOP framework |
| Cactus | http://jakarta.apache.org/cactus/ | J2EE unit-testing framework |
| Clover | http://www.thecortex.net/clover/ | Test coverage tool |
| Commons BeanUtils | http://jakarta.apache.org/commons/beanutils/ | Reflection and introspection utilities for working on JavaBeans |

*continued on next page*

**Table R.1 Software directory** *(continued)*

| Name | Web site | Quick description |
|---|---|---|
| Commons Collections | http://jakarta.apache.org/commons/collections.html | Complements the Java Collections API with other powerful data structures |
| Commons Httpclient | http://jakarta.apache.org/commons/httpclient/ | HTTP client |
| Commons Logging | http://jakarta.apache.org/commons/logging.html | Logging façade to other logging systems |
| DbUnit | http://www.dbunit.org/ | Database unit-testing framework |
| EasyMock | http://easymock.org/ | Mock objects generation framework |
| Eclipse | http://www.eclipse.org/ | Tools platform and Java IDE |
| HttpUnit | http://httpunit.sourceforge.net/ | JUnit extension for testing web applications |
| JBoss | http://www.jboss.org/ | J2EE container |
| Jester | http://jester.sourceforge.net/ | Tool to verify quality of unit tests |
| Jetty | http://jetty.mortbay.org/ | Servlet/JSP container |
| JMeter | http://jakarta.apache.org/jmeter/ | Load-testing tool |
| JSTL | http://java.sun.com/products/jsp/jstl/ | Standard JSP tag libraries |
| JUnit | http://junit.org/ | Unit-testing framework |
| JUnitBook | http://sourceforge.net/projects/junitbook/ | Source code for *JUnit in Action* |
| JUnitPerf | http://www.clarkware.com/software/JUnitPerf.html | JUnit extension for measuring performance and scalability |
| Maven | http://maven.apache.org/ | Project comprehension build tool |
| MockObjects | http://www.mockobjects.com/ | Mock-objects framework |

*continued on next page*

**Table R.1 Software directory** *(continued)*

| Name | Web site | Quick description |
|---|---|---|
| MockMaker plugin for Eclipse | http://www.mockmaker.org/ | Static mock-objects generation framework |
| Taglibs | http://jakarta.apache.org/taglibs/ | Jakarta's implementation of JSTL |
| Tomcat | http://jakarta.apache.org/tomcat/ | Servlet/JSP container |
| xPetstore | http://xpetstore.sf.net/ | Sample Petstore application |

### *Software licenses*

- The source code created for this book is provided under the Apache Software License (http://apache.org/LICENSE).
- JUnit is provided under the Common Public License (http://oss.software.ibm.com/developerworks/oss/license-cpl.html).

# index

## Numerics



## A



## B

## C

## F



## G



## H

## I



## J

## K



## L



## M

## T

## U



## V



## W



## X



## Y

JAVA


Green Bar

# JUnit IN ACTION

Vincent Massol with Ted Husted

Developers in the know are switching to a new testing strategy—unit testing—in which coding is interleaved with testing. This powerful approach results in better-designed software with fewer defects and faster delivery cycles. Unit testing is reputed to give developers a kind of "high"—whenever they take a new programming step, their confidence is boosted by the knowledge that every previous step has been confirmed to be correct.

***JUnit in Action*** will get you coding the new way in a hurry. As inevitable errors are continually introduced into your code, you'll want to spot them as quickly as they arise. You can do this using unit tests, and using them often. Rich in real-life examples, this book is a discussion of practical testing techniques by a recognized expert. It shows you how to write automated tests, the advantages of testing a code segment in isolation from the rest of your code, and how to decide when an integration test is needed. It provides a valuable—and unique—discussion of how to test complete J2EE applications.

## What's Inside

- Testing in isolation with mock objects
- In-container testing with Cactus
- Automated builds with Ant and Maven
- Testing from within Eclipse
- Unit testing
  - Java apps
  - Servlets
  - JSP
  - Taglibs
  - Filters
  - EJB
  - DB apps

**Vincent Massol** is the creator of the Jakarta Cactus testing framework and an active member of the Maven and MockObjects development teams. He is CTO of Pivolis, a specialist in agile offshore software development. Vince lives in the City of Light—Paris, France.

"... captures best practices for effective JUnit and in particular J2EE testing. Don't unit test your J2EE applications without it!"

—Erich Gamma, IBM OTI Labs
Co-author of JUnit

"Outstanding job... It rocks—a joy to read! I recommend it wholeheartedly."

—Erik Hatcher, co-author of *Java Development with Ant*

"Brings the mass of information out there under one coherent umbrella."

—J. B. Rainsberger, leader in the JUnit community, author

"Doesn't shy from tough cases ... Vince really stepped up, rather than side-stepping the real problems people face."

—Scott Stirling, BEA

**www.manning.com/massol**

AUTHOR ONLINE Author responds to reader questions

Ebook edition available



MANNING $39.95 US/$59.95 Canada

53995
9 781930 110991
ISBN 1-930110-99-5