```
        IDatabaseConnection connection =
            new DatabaseDataSourceConnection(new InitialContext(),
            "java:/DefaultDS");

        IDataSet dataSet = new FlatXmlDataSet(
            this.getClass().getResource(
            "/junitbook/database/data.xml"));

        try
        {
            DatabaseOperation.CLEAN_INSERT.execute(connection,
                dataSet);
        }
        finally
        {
            connection.close();
        }
    }
}
```

Now that you have factored the common database setup into a `DatabaseTestSetup` class, you can call it from the `TestJdbcDataAccessManagerIC` test case class, as shown in listing 11.21.

Listing 11.21 TestJdbcDataAccessManager using the DatabaseTestSetup class

```
package junitbook.database;

import junit.framework.Test;
import junit.framework.TestSuite;

import org.apache.cactus.ServletTestCase;

public class TestJdbcDataAccessManagerIC2 extends ServletTestCase
{
    public static Test suite()
    {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(TestJdbcDataAccessManagerIC2.class);
        return new DatabaseTestSetup(suite);
    }

    protected void setUp() throws Exception
    {
        // Database initialization for data that need to be
        // reset before each test.
    }

    public void testExecute1() throws Exception
    {
        [...]
    }

    public void testExecute2() throws Exception
    {
        [...]
    }

    public void testExecute3() throws Exception
    {
        [...]
    }
}
```

Although listing 11.21 includes three tests, the database will be initialized only once with the read-only data. The `suite` method is executed only once by the JUnit test runner.

11.6.2 Grouping tests in functional test suites

In practice, test cases often use similar sets of database data or perform a common set of operations. If you group these tests together, they can share resources, including the overhead of establishing the resources. For example, imagine you have several classes that are used to implement customer-related services (get customer details, get customer accounts, add new customers, and so forth). Instead of having one XML data file for each test case, you can have only one that you execute once before all the customer tests. If you have use cases that modify data, you can create several customers in the XML file so that one test will work on one customer and other tests can work on other customer records.

Listing 11.22 shows a class that is used only to group together test cases.

Listing 11.22 Grouping all customer-related test cases

```
package junitbook.database;

import junit.framework.Test;
import junit.framework.TestSuite;

public class TestCustomerAll
{
    public static Test suite()
    {
        TestSuite suite = new TestSuite(
            "Customer Related Tests");

        // Add tests from Customer related test cases
        suite.addTestSuite(TestCustomer1.class);
        suite.addTestSuite(TestCustomer2.class);
[...]
        suite.addTestSuite(TestCustomerN.class);

        return suite;
    }
}
```

11.6.3 Using an in-memory database

Most applications spend most of their time waiting for the database. The same is true for the database integration unit tests. If the test can access the database more quickly, the test will also run more quickly.

For running tests, an in-memory database can be accessed much more quickly (by several orders of magnitude) than a conventional database. As a result, your tests also run faster, by several orders of magnitude. Hypersonic SQL, among others, has an in-memory mode that is ideal for running tests.

Unfortunately, using a different database isn't a viable option for many applications. Often, the SQL is optimized for the target database. Several common needs are not standardized by SQL. For example, to return pages of data, Oracle has a `rowid` feature; but other databases implement this feature differently. Different vendors implement other commonly used database features, like triggers and referential integrity, differently. However, if your application uses SQL that can be used with an in-memory database as well as your target database, this can be an excellent optimization.

11.7 Overall database unit-testing strategy

You have seen two main approaches to testing database applications. One uses mock objects to perform unit tests in isolation. Another uses in-container testing (with a tool like Cactus) and preset database data (created with a tool like DbUnit) to perform integration tests.

At this point, you're probably wondering whether you need to write both types of tests. Is it enough to write only mock-object tests? Can Cactus tests be replaced by functional tests? Let's address these crucial questions.

11.7.1 Choosing an approach

There are two valid approaches:

- *Mock objects / functional testing*—Write mock-objects-style unit tests for the business logic (by separating the business logic from the database access layer). Also write mock-objects-style unit tests for the database access layer code. Finally, write functional tests to ensure the whole thing works when it's running in the deployed environment.

- *Mock objects / database integration unit tests / functional tests*—Write mock-objects-style unit tests for the business logic, as shown in section 11.6, by separating the business logic from the database access layer. Don't write mock-object unit tests for the database access layer code. Instead, write Cactus integration unit tests (possibly using some mocks to test failure conditions). Also write functional tests (as in the previous approach) for end-to-end testing.

The difference between the two approaches is that in the second, you use Cactus to perform integration unit tests. This approach provides better test coverage but requires more setup, because you need a database (or private dataspace) installed for each developer. You also need to think about builds and deployment from day one (but that can be a *good* thing).

11.7.2 Applying continuous integration

The Cactus approach pushes the integration issues into the development realm. This is helpful, because too often integration comes as a second phase in the project life cycle. In some projects, integration is treated almost as an afterthought!

Integration is when the real problems start to happen: Architectural decisions are changed, leading to rewriting of a portion of the application; deployment teams have not been trained beforehand and begin discovering how the application works—and of course the project becomes late. The cost of dealing with integration issues in a J2EE project is extremely high.

Our recommendation is to begin practicing packaging and deployment from day one (maybe after a first exploratory iteration) and improve the process continuously as the project progresses, by automating more and more. As you do this, you'll find that you gain increasing control over the configuration of your container, and that it takes less time to put a release into production. On a recent project, a complex system took two hours flat to deploy from development to preproduction, and we could put our system into production at any point in time.

The second approach using Cactus, discussed in the previous section, takes longer in execution time, but that shouldn't be an issue. We usually execute all the pure JUnit and mock-objects tests continuously from our development IDE and only execute the integration unit tests from time to time. For large projects, we always have a continuous build system located on a dedicated machine that runs continuously and executes all the tests all the time; it sends us an email if there is a build failure. This way, we benefit from the best of both worlds.

11.8 Summary

Most developers feel that it's difficult to unit-test applications that access a database. In this chapter, we've shown three types of simple unit tests that can test the data-access code calling the real database: *business logic unit tests* that test the business code in isolation from the database, *data access unit tests* that test the data-access code in isolation from the database, and *integration unit tests.*

Business logic tests and data access unit tests are easy to run as ordinary JUnit tests, with some help from mock objects. The MockObjects.com framework is especially useful, because it provides prewritten mocks for the SQL packages.

For integration unit tests, we have demonstrated the powerful Cactus/DbUnit combo. Not only have we succeeded in running database tests from inside the container, but we have also automated the execution of the tests using Ant. This combo paves the way for practicing *continuous integration.*

In chapter 9, we started down a path to completely unit-test a web application, including its servlets, filters, taglibs, JSPs, and database access. In this chapter, we completed that path by showing how easy it can be to unit-test database access. In the next chapter, we will introduce an EJB application that demonstrates unit-testing of EJBs.

12 Unit-testing EJBs

This chapter covers

- Unit-testing all types of EJBs: session beans, entity beans, and message-driven beans
- Using mock objects, pure JUnit, and Cactus
- Automating packaging, deployment, and container start/stop using Ant

"The time has come," the Walrus said, "To talk of many things: Of shoes—and ships—and sealing-wax—Of cabbages—and kings—And why the sea is boiling hot—And whether pigs have wings."

—Lewis Carroll, Through the Looking Glass

Testing EJBs has a reputation of being a difficult task. One of the main reasons is that EJBs are components that run inside a container. Thus, you either need to abstract out the container services used by your code or perform in-container unit testing. In this chapter, we'll demonstrate different techniques that can help you write EJB unit tests. We'll also provide guidelines and hints for helping you choose when to select one strategy over another.

12.1 Defining a sample EJB application

In previous chapters, we used a sample Administration application to demonstrate how to unit-test web components (servlets, filters, taglibs, JSPs) and database access code. In order to demonstrate the different EJB unit-testing techniques, we've chosen a simple order-processing application, which you'll implement using different types of EJBs components.

The inspiration for this example comes from the xPetstore application (http://xpetstore.sf.net/). xPetstore is an XDoclet version of the standard Java Pet Store demonstration application (http://developer.java.sun.com/developer/releases/petstore/). We've simplified the original xPetstore to focus on our topic of unit-testing different types of EJBs, as shown in figure 12.1.

As you can see from the figure, the sample Order application includes a message-driven bean (MDB), a remote stateless session bean (remote SLSB), and a local container-managed persistence (CMP) entity bean (local CMP EB). This chapter focuses on demonstrating how to unit-test each type of EJB. The example doesn't include stateful session beans (SFSB) because they are unit-tested in the same manner as SLSBs. In addition, we aren't covering bean-managed persistence (BMP) entity beans (BMP EB); they're the same as CMP EBs for which the

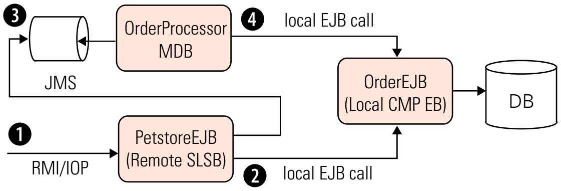


Figure 12.1
Sample Order application made of three EJBs: a remote stateless session bean, a local CMP entity bean, and a message-driven bean

persistence is performed manually, and we covered unit-testing database code in chapter 11.

The sample Order application has the following use case:

❶ The client application (whatever that is: another EJB, a Java Swing application, a servlet, and so on) calls the `PetstoreEJB` and calls its `createOrder` method.

❷ ❸ `PetStoreEJB`'s `createOrder` method creates an `OrderEJB` order and then sends a JMS message containing the Order ID to a JMS queue. With JMS you can process the different orders coming into the system asynchronously. It allows an immediate reply to the client application, indicating that the order is being processed.

❹ The JMS message is processed by the `OrderProcessorMDB` MDB, which extracts the order ID from the message in the queue, finds the corresponding `OrderEJB` EJB, and performs some business operation (you don't need to know the details of this business operation for the purpose of this chapter). In a production application, the `OrderProcessorMDB` could also send an email to the customer to tell them that their order has been processed.

We believe that the best way to demonstrate how to use the different techniques is by showing the code of this sample application as you would have written it without thinking about unit testing (or as it would have been written had it been a legacy application for which you were asked to write unit tests). Then, we'll show how the code needs to be refactored (or not refactored) to be testable, using different techniques.EJB:defining sample applications

12.2 Using a façade strategy

The *façade strategy* is an architectural decision that has the nice side effect of making your application more testable. This is the approach of designing for testability that we have recommended throughout this book.

Using this strategy, you consider EJBs as plumbing and move all code logic outside the EJBs. More specifically, you keep session beans or MDBs as a façade to the outside world and move all the business code into POJOs (plain old Java objects), for which you can then write unit tests easily. Yes, this is a strong architectural decision. For example, you won't use EJBs for persisting your data. Moreover, you'll need to cleanly separate any access to the container API (such as looking up objects in JNDI for obtaining data sources, JMS factories, mail sessions, and so on).

You must realize that this technique won't provide any means for unit-testing the container-related code you have separated. That code will be tested only during integration and functional tests.

This strategy will not work in the following cases:

- The application is a legacy application using EJBs that call each other (as in the `PetstoreEJB` SLSB calling the `OrderEJB` EB). Because the application is already written, it doesn't seem worthwhile to redesign it completely using a different architecture (POJOs).
- It has been decided that the application architecture has EJBs calling other EJBs.
- The application needs to access objects published in JNDI, such as data sources, JMS factories, mail sessions, EJB homes, and so on.

For all these cases, using this façade strategy won't help. For example, in the Order application, you would have to rewrite the `OrderEJB` EJB by transforming it into a class using JDBC (for example) in order to support this façade technique. But even then, you would still use JNDI to send the order ID in a JMS message. Thus, in this case, the façade technique isn't appropriate and would require too many changes.

To summarize, the façade strategy isn't a strategy to unit-test EJBs but rather a strategy to bypass testing of EJBs. You'll often read on the Web or in other books that unit-testing EJBs is a false problem and that EJBs should only be used as a pure façade, thus eliminating the need to unit-test them. Our belief is that EJBs are very useful in lots of situations and that there are several easy unit-testing techniques for exercising them, as demonstrated in the following sections.

12.3 Unit-testing JNDI code using mock objects

In this section and those that follow, we'll demonstrate how to use different mock-objects techniques to test session beans. We have chosen to use the DynaMock API from MockObjects.com to illustrate these techniques (see chapter 9 for an introduction to DynaMock).

Unit-testing EJB code that uses JNDI is the most difficult case to address in EJB unit testing. One reason is that the EJB homes are usually cached in static variables to enhance performance. Thus, the code to create a session bean instance usually consists of static methods.

Another complication is that you access JNDI objects by creating an `InitialContext` (`new InitialContext()`). In other words, the JNDI API doesn't follow the Inversion Of Control (IOC) pattern. Thus, it isn't easy to pass a mock `InitialContext` instead of the real one. However, as you'll see in one of the possible techniques, thanks to the Factory pattern followed by the JNDI API, it's possible to pass a mock `InitialContext` with a minimal amount of code.

We will now demonstrate three refactoring techniques you can use to unit-test JNDI code and, more specifically, to unit-test code that looks up EJBs (that is, that calls `InitialContext.lookup`):

- *Factory method*—Isolates the creation of domain objects in separate methods that can be overridden to return the mock instances
- *Factory class*—Isolates the creation of domain objects in factory classes, introduces setters for these factories, and provides mock factories for your tests
- *Mock JNDI implementation*—Sets up a mock JNDI implementation that returns mock objects whenever the `lookup` method is called

12.4 Unit-testing session beans

Let's follow the use case flow (see figure 12.1) and start by unit-testing the `PetstoreEJB` SLSB. You'll try each of the approaches listed in the previous section. The `PetstoreEJB` code, which is the bean implementation of the `Petstore` EJB, is shown in listing 12.1.

Listing 12.1 Initial PetstoreEJB.java

```
package junitbook.ejb.service;

import java.rmi.RemoteException;
import java.util.Date;

import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

import junitbook.ejb.domain.OrderLocal;
import junitbook.ejb.domain.OrderUtil;
import junitbook.ejb.util.JMSUtil;
import junitbook.ejb.util.JNDINames;

public class PetstoreEJB implements SessionBean
{
    public int createOrder(Date orderDate, String orderItem)         ❶
        throws RemoteException
    {
        OrderLocal order = OrderUtil.createOrder(orderDate,          ❷
            orderItem);

        try
        {
            JMSUtil.sendToJMSQueue(JNDINames.QUEUE_ORDER,                ❸
                order.getOrderId(), false);
        }
        catch (Exception e)
        {
            throw new EJBException(e);
        }
        return order.getOrderId().intValue();
    }

    public void setSessionContext(SessionContext sessionContext)
        throws EJBException, RemoteException {}
    public void ejbRemove()
        throws EJBException, RemoteException {}
    public void ejbActivate()
        throws EJBException, RemoteException {}
    public void ejbPassivate()
        throws EJBException, RemoteException {}
}
```

❶ For simplicity, you have only one method to unit-test in `PetStoreEJB`: `createOrder`.

❷ ❸ As you can see from the code, this method calls two static methods—`OrderUtil.createOrder` and `JMSUtil.sendToJMSQueue`—which, respectively, create an `OrderEJB` EB instance and send the order ID to a JMS queue (for later processing by the `OrderProcessorMDB`, as shown in figure 12.1).

Listing 12.2 shows the code for the `OrderUtil` helper class.

Listing 12.2 Initial OrderUtil.java

```
package junitbook.ejb.domain;

import java.util.Date;

import javax.ejb.CreateException;
import javax.ejb.FinderException;
import javax.rmi.PortableRemoteObject;

import junitbook.ejb.util.JNDINames;
import junitbook.ejb.util.JNDIUtil;

public class OrderUtil
{
    private static OrderLocalHome orderLocalHome;

    protected static OrderLocalHome getOrderHome()     ❶
    {
        if (orderLocalHome == null)
        {
            Object obj =
                JNDIUtil.lookup(JNDINames.ORDER_LOCALHOME);
            orderLocalHome =
                (OrderLocalHome) PortableRemoteObject.narrow(
                    obj, OrderLocalHome.class);
        }
        return orderLocalHome;
    }

    public static OrderLocal createOrder(Date orderDate,    ❷
        String orderItem)
    {
        try
        {
            return getOrderHome().create(orderDate, orderItem);
        }
        catch (CreateException e)
        {
            throw new RuntimeException("Failed to create ["
                + OrderLocal.class.getName() + "]. Reason ["
                + e.getMessage() + "]");
        }
    }

    public static OrderLocal getOrder(Integer orderId)    ❸
    {
        try
        {
            return getOrderHome().findByPrimaryKey(orderId);
        }
        catch (FinderException e)
        {
            throw new RuntimeException("Failed to find Order "
                + "bean for id [" + orderId + "]");
        }
    }
}
```

The `OrderUtil` class provides two public and static helper methods: `createOrder` (❷) and `getOrder` (❶). The `createOrder` method creates an instance of the `Order` EJB, whereas `getOrder` retrieves an `Order` EJB instance by its order ID. The `getOrderHome` (❶) method is protected and only used internally by the other two methods introduced; it is used to retrieve the `Order` EJB home instance.

In this section, the challenge is to be able to unit-test the `Petstore` EJB's `createOrder` method (listing 12.1). The main issue you'll have to overcome is replacing the JNDI `lookup` calls to return mock objects instead of the real objects. The static methods in listing 12.2 will be the main hurdle.

Listing 12.3 shows the code for the `JMSUtil` helper class.

Listing 12.3 Initial JMSUtil.java

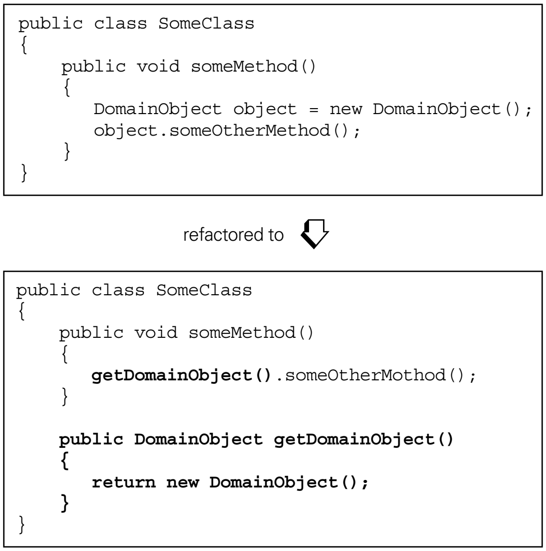
```
package junitbook.ejb.util;

import java.io.Serializable;
import javax.jms.JMSException;
import javax.jms.ObjectMessage;
import javax.jms.Queue;
import javax.jms.QueueConnection;
import javax.jms.QueueConnectionFactory;
import javax.jms.QueueSender;
import javax.jms.QueueSession;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class JMSUtil
{
    public static void sendToJMSQueue(String queueName,          ❶
        Serializable obj, boolean transacted)
        throws NamingException, JMSException
    {
        InitialContext ic = null;
        QueueConnection cnn = null;
        QueueSender sender = null;
        QueueSession session = null;

        try
        {
            ic = new InitialContext();

            Queue queue = (Queue) ic.lookup(queueName);

            QueueConnectionFactory factory =
                (QueueConnectionFactory) ic.lookup(
                JNDINames.QUEUE_CONNECTION_FACTORY);
            cnn = factory.createQueueConnection();
            session = cnn.createQueueSession(transacted,
                QueueSession.AUTO_ACKNOWLEDGE);

            ObjectMessage msg = session.createObjectMessage(obj);

            sender = session.createSender(queue);
            sender.send(msg);
        }
        finally
        {
            if (sender != null)
            {
                sender.close();
            }
            if (session != null)
            {
                session.close();
            }
            if (cnn != null)
            {
                cnn.close();
            }
            if (ic != null)
            {
                ic.close();
            }
        }
    }
}
```

❶ The JMSUtil class only defines a single static sendToJMSQueue, which sends a given object to a queue.

12.4.1 Using the factory method strategy

This is one of the easiest possible refactorings because it lets you introduce the mock objects by only impacting the class under test and not any of the dependent classes it calls. In the example at hand, it impacts only the PetstoreEJB class.

The strategy consists of finding all the places where your method to test creates domain objects. For each of these, you introduce a factory method that returns the domain object. See figure 12.2 for an example.



```
public class SomeClass
{
    public void someMethod()
    {
       DomainObject object = new DomainObject();
       object.someOtherMethod();
    }
}
```

refactored to

```
public class SomeClass
{
    public void someMethod()
    {
       getDomainObject().someOtherMothod();
    }

    public DomainObject getDomainObject()
    {
       return new DomainObject();
    }
}
```

Figure 12.2 Performing a factory method refactoring to decouple the creation of a domain object from the method using that object, in order to replace it later with a mock object

Note that depending on the use case, you can also write a variation where the `new DomainObject` happens only once and the resulting instance is saved in a private class variable.

Refactoring the PetstoreEJB class

For the sample Order application, this means transforming the `PetstoreEJB` class into the one shown in listing 12.4 (the changes have been marked in bold).

Listing 12.4 PetstoreEJB.java after a factory method refactoring

```
package junitbook.ejb.service;
[...]
public abstract class PetstoreEJB implements SessionBean
{
    public int createOrder(Date orderDate, String orderItem)
    {
        OrderLocal order = createOrderHelper(orderDate, orderItem);

        try
        {
            sendToJMSQueueHelper(JNDINames.QUEUE_ORDER,
                order.getOrderId(), false);
        }
        catch (Exception e)
        {
            throw new EJBException(e);
        }
        return order.getOrderId().intValue();
    }

    protected OrderLocal createOrderHelper(Date orderDate,
        String orderItem)
    {
        return OrderUtil.createOrder(orderDate, orderItem);
    }

    protected void sendToJMSQueueHelper(String queueName,
        Serializable object, boolean transacted)
        throws NamingException, JMSException
    {
        JMSUtil.sendToJMSQueue(queueName, object, transacted);
    }
[...]
```

You can now easily write a unit test for the `createOrder` method by creating a class that extends `PetstoreEJB` and overrides the `createOrderHelper` and `sendToJMSQueueHelper` methods.

Mocking factory methods

The behavior of the overridden methods is controlled by the unit test in exactly the same spirit as mock objects. Let's call `TestablePetstoreEJB` the extended class (see listing 12.5). Said differently, the `TestablePetstore` mocks the factory methods you have introduced.

Listing 12.5 TestablePetstoreEJB (mocks the factory methods)

```
package junitbook.ejb.service;

import java.io.Serializable;
import java.util.Date;

import javax.jms.JMSException;
import javax.naming.NamingException;

import junitbook.ejb.domain.OrderLocal;

public class TestablePetstoreEJB extends PetstoreEJB
{
    private OrderLocal orderLocal;
    private JMSException jmsExceptionToThrow;
    private NamingException namingExceptionToThrow;

    public void setupCreateOrderHelper(OrderLocal orderLocal)
    {
        this.orderLocal = orderLocal;
    }

    protected OrderLocal createOrderHelper(Date orderDate,
        String orderItem)
    {
        return this.orderLocal;
    }

    public void setupThrowOnSendToJMSQueueHelper(
        JMSException exception)
    {
        this.jmsExceptionToThrow = exception;
    }

    public void setupThrowOnSendToJMSQueueHelper(
        NamingException exception)
    {
        this.namingExceptionToThrow = exception;
    }

    protected void sendToJMSQueueHelper(String queueName,
        Serializable object, boolean transacted)
        throws NamingException, JMSException
    {
        if (this.jmsExceptionToThrow != null)
        {
```

Defines what createOrderBean method should throw when called

Defines what exception sendOrder method should throw when called

```
            throw this.jmsExceptionToThrow;
        }
        if (this.namingExceptionToThrow != null)
        {
            throw this.namingExceptionToThrow;
        }
    }

}
```

Writing the unit tests

You now have all the parts you need to write the unit tests (see listing 12.6).

Listing 12.6 TestPetstoreEJB using the factory method strategy

```
package junitbook.ejb.service;
[...]
public class TestPetstoreEJB extends TestCase
{
    private TestablePetstoreEJB petstore;
    private Mock mockOrderLocal;
    private OrderLocal orderLocal;

    protected void setUp()
    {
        petstore = new TestablePetstoreEJB();       ❶

        mockOrderLocal = new Mock(OrderLocal.class);            ❷
        orderLocal = (OrderLocal) mockOrderLocal.proxy();

        petstore.setupCreateOrderHelper(orderLocal);

        mockOrderLocal.matchAndReturn("getOrderId",         ❸
            new Integer(1234));
    }

    protected void tearDown()
    {
        mockOrderLocal.verify();    ❹
    }

    public void testCreateOrderOk() throws Exception    ❺
    {
        int orderId = petstore.createOrder(new Date(), "item1");
        assertEquals(1234, orderId);
    }

    public void testCreateOrderJMSException() throws Exception    ❻
    {
        petstore.setupThrowOnSendToJMSQueueHelper(          ❼
            new JMSException("error"));
        try
        {
            petstore.createOrder(new Date(), "item1");
            fail("Should have thrown an EJBException");
        }
        catch (EJBException e)
        {
            assertEquals("error",
                e.getCausedByException().getMessage());
        }
    }
}
```

❶ Instantiate the `TestablePetstoreEJB` class instead of `PetstoreEJB` as you would normally do for a standard test case.

❷ Create a mock `OrderLocal` bean and set up `TestablePetstoreEJB` to return it when its `createOrderHelper` method is called.

❸ Tell the `OrderLocal` mock to return the 1234 `Integer` whenever its `getOrderId` method is called. Note that if you used `expectAndReturn` instead of `matchAndReturn`, you would need to write the following two same lines (because `expect*` calls are expected, whereas `match*` calls return the specified return value whenever they match the call):

```
mockOrderLocal.matchAndReturn("getOrderId", new Integer(1234));
mockOrderLocal.matchAndReturn("getOrderId", new Integer(1234));
```

❹ In the `tearDown` method, you tell the `OrderLocal` mock to verify the expectations set on it (all the `expect*` calls you have set up).

❺ Write the first test, which verifies that the `createOrder` method works fine when you pass valid parameters to it.

❻ Verify that the `createOrder` method correctly throws an `EJBException` when a `JMSException` is thrown when sending the JMS order message.

❼ Use the mock to simulate the `JMSException`.

12.4.2 Using the factory class strategy

The general idea for this strategy is to add setter methods to introduce all the domain objects used by the methods to test. This then allows you to call these setters, passing them mock objects from the test case classes. Namely, this means introducing protected `setOrderUtil(OrderUtil)` and `setJMSUtil(JMSUtil)` methods in the `PetstoreEJB` class.

The following refactoring tasks are involved to transform the initial PetstoreEJB, OrderUtil, and JMSUtil classes:

- Create new OrderUtil and JMSUtil interfaces.
- Add two static private variables to PetstoreEJB.
- Add two methods to set the OrderFactory and JMSUtil domain objects.

Making the interfaces

First, you make OrderUtil and JMSUtil interfaces and rename the old OrderUtil and JMSUtil classes DefaultOrderUtil and DefaultJMSUtil. In addition, during this refactoring it becomes clear that the OrderUtil interface and DefaultOrderUtil class would be better named OrderFactory and DefaultOrderFactory. This renaming illustrates that performing refactorings also make you think about what you're doing and usually lets you enhance the code with small improvements. Of course, in the renaming case, you have to be careful not to break public APIs unintentionally. The refactoring yields the following for the OrderFactory class:

```
public interface OrderFactory
{
    OrderLocal createOrder(Date orderDate, String orderItem);
    OrderLocal getOrder(Integer orderId);
}

public class DefaultOrderFactory implements OrderFactory
{
    private static OrderLocalHome orderLocalHome;

    protected OrderLocalHome getOrderHome()
    {
        [...]
    }

    public OrderLocal createOrder(Date orderDate, String orderItem)
    {
        [...]
    }

    public OrderLocal getOrder(Integer orderId)
    {
        [...]
    }
}
```

Note that you drop the static modifiers on methods (see the initial listing 12.2) because you have moved from a unique OrderUtil static class to a normal DefaultOrderFactory class, allowing several instances to be created (especially a mock one).

The refactoring for the JMSUtil class gives the following:

```
public interface JMSUtil
{
    void sendToJMSQueue(String queueName, Serializable obj,
    boolean transacted) throws NamingException, JMSException;
}

public class DefaultJMSUtil implements JMSUtil
{
    public void sendToJMSQueue(String queueName, Serializable obj,
        boolean transacted) throws NamingException, JMSException
    {
        [...]
    }
}
```

Adding static private variables

Next, you add two static private variables to PetstoreEJB. These variables will hold the default OrderFactory and JMSUtil domain objects to be used by the createOrder method:

```
public abstract class PetstoreEJB implements SessionBean
{
    private static OrderFactory orderFactory =
        new DefaultOrderFactory();
    private static JMSUtil jmsUtil = new DefaultJMSUtil();
[...]
```

Adding methods to set the domain objects

Finally, add two methods to set OrderFactory and JMSUtil domain objects different from the default ones:

```
[...]
    protected void setOrderFactory(OrderFactory factory)
    {
        orderFactory = factory;
    }

    protected void setJMSUtil(JMSUtil util)
    {
        jmsUtil = util;
    }
```

Creating the test case

You can now use a standard mock-objects approach to create the TestCase, as shown in listing 12.7.

Listing 12.7 TestPetstoreEJB using the factory class strategy

```
package junitbook.ejb.service;
[...]
public class TestPetstoreEJB extends TestCase
{
    private PetstoreEJB petstore;
    private Mock mockOrderFactory;
    private Mock mockJMSUtil;
    private OrderFactory orderFactory;
    private JMSUtil jmsUtil;

    private Mock mockOrderLocal;
    private OrderLocal orderLocal;

    protected void setUp()
    {
        petstore = new PetstoreEJB() {};

        mockOrderLocal = new Mock(OrderLocal.class);
        orderLocal = (OrderLocal) mockOrderLocal.proxy();

        mockOrderFactory = new Mock(OrderFactory.class);              ❶
        orderFactory = (OrderFactory) mockOrderFactory.proxy();

        mockJMSUtil = new Mock(JMSUtil.class);
        jmsUtil = (JMSUtil) mockJMSUtil.proxy();

        petstore.setOrderFactory(orderFactory);          ❷
        petstore.setJMSUtil(jmsUtil);

        mockOrderFactory.expectAndReturn("createOrder",              ❸
            C.args(C.IS_ANYTHING, C.IS_ANYTHING), orderLocal);
        mockOrderLocal.matchAndReturn("getOrderId",
            new Integer(1234));
    }

    protected void tearDown()
    {
        mockJMSUtil.verify();
        mockOrderFactory.verify();
        mockOrderLocal.verify();
    }

    public void testCreateOrderOk() throws Exception
    {
        mockJMSUtil.expect("sendToJMSQueue",                       ❸
            C.args(C.IS_ANYTHING, C.eq(new Integer(1234)),
                C.IS_ANYTHING));

        int orderId = petstore.createOrder(new Date(), "item1");

        assertEquals(1234, orderId);
    }

    public void testCreateOrderJMSException() throws Exception
    {
        mockJMSUtil.expectAndThrow("sendToJMSQueue",             ❸
            C.args(C.IS_ANYTHING, C.eq(new Integer(1234)),
                C.IS_ANYTHING), new JMSException("error"));

        try
        {
            petstore.createOrder(new Date(), "item1");
            fail("Should have thrown an EJBException");
        }
        catch (EJBException expected)
        {
            assertEquals("error",
                expected.getCausedByException().getMessage());
        }

    }
}
```

❶ Now that you have `OrderFactory` and `JMSUtil` interfaces, you can create mock objects for them.

❷ Call the newly introduced setter methods (`setOrderFactory` and `setJMSUtil`) to set the mock objects on the `PetstoreEJB` class.

❸ Then, as usual, tell the mocks how to behave for the different tests.

Let's now see the third strategy to introduce mock objects for unit-testing the `PetstoreEJB` `createOrder` method: the mock JNDI implementation strategy.

12.4.3 Using the mock JNDI implementation strategy

This is a very logical strategy that consists of mocking the JNDI implementation. It lets you write unit tests without impacting the code under test, which can be quite nice if you're writing unit tests for a legacy application, for example.

This is possible because the JNDI API offers the possibility to easily plug any JNDI implementation. You can do this several ways (using a System property, using a `jndi.properties` file, and so on); but the best way in this case is to use the `NamingManager.setInitialContextFactoryBuilder` API, which lets you set up an initial context factory. Once it has been set, every call to `new InitialContext` will call this factory.

Mocking the JNDI implementation

Here is a compact solution (using inner classes) that returns a mock Context whenever new InitialContext is called:

```
NamingManager.setInitialContextFactoryBuilder(
    new InitialContextFactoryBuilder()
    {
        public InitialContextFactory createInitialContextFactory(
            Hashtable environment) throws NamingException
        {
            return new InitialContextFactory() {
                public Context getInitialContext(Hashtable env)
                    throws NamingException
                {
                    // Return the mock context here
                    return context;
                }
            };
        }
    }
);
```

The only limitation is that the setInitialContextFactoryBuilder method can be called only once during the JVM lifetime (otherwise it will throw an IllegalStateException). Thus you need to wrap this initialization in a JUnit TestSetup, as shown in listing 12.8.

Listing 12.8 Setting up the JNDI implementation in a JUnit TestSetup

```
package junitbook.ejb;
[...]
public class JNDITestSetup extends TestSetup
{
    private Mock mockContext;
    private Context context;

    public JNDITestSetup(Test test)
    {
        super(test);
        mockContext = new Mock(Context.class);
        context = (Context) mockContext.proxy();
    }

    public Mock getMockContext()
    {
        return this.mockContext;
    }

    protected void setUp() throws Exception
    {
        NamingManager.setInitialContextFactoryBuilder(
            new InitialContextFactoryBuilder()
            {
                public InitialContextFactory
                    createInitialContextFactory(
                    Hashtable environment) throws NamingException
                {
                    return new InitialContextFactory() {
                        public Context getInitialContext(
                            Hashtable env) throws NamingException
                        {
                            // Return the mock context here
                            return context;
                        }
                    };
                }
            }
        );
    }
}
```

Note that you can reuse this `JNDITestSetup` class as is and put it in your unit-testing toolbox. You're now ready to write the unit tests.

Writing the test case

The `TestCase` is shown in listing 12.9. Note that you factorize all mock setup in the JUnit `setUp` method. We did not do that right away when writing this example. As a practice, we begin by writing the first test and put all needed initialization in the first test method (`testCreateOrderOk`). However, once we started writing the second test (`testCreateThrowsOrderException`), we realized that there was lots of common setup code. So, we refactored the `TestCase` and moved the common bits into the `setUp` method.

JUnit best practices: refactor test setups and teardowns

As you write more tests in a given `TestCase`, always take time to refactor by factorizing out the common code needed to set up the tests (the same applies to `tearDown` methods). If you don't do this, you'll end up with complex tests that cannot be easily understood, will be very difficult to maintain, and, as a consequence, will break more often.

Listing 12.9 TestPetstoreEJB using the mock JNDI implementation strategy

```
package junitbook.ejb.service;
[...]
public class TestPetstoreEJB extends TestCase
{
    private static JNDITestSetup jndiTestSetup;

    private PetstoreEJB petstore;

    private Mock mockOrderLocalHome;
    private OrderLocalHome orderLocalHome;
    private Mock mockOrderLocal;
    private Mock mockQueue;
    private Queue queue;
    private Mock mockQueueConnectionFactory;
    private QueueConnectionFactory queueConnectionFactory;
    private Mock mockQueueConnection;
    private QueueConnection queueConnection;
    private Mock mockQueueSession;
    private Mock mockQueueSender;
    private Mock mockObjectMessage;

    public static Test suite()                                       ❶
    {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(TestPetstoreEJB.class);
        jndiTestSetup = new JNDITestSetup(suite);
        return jndiTestSetup;
    }

    protected void setUp() throws Exception     ❷
    {
        petstore = new PetstoreEJB() {};

        jndiTestSetup.getMockContext().reset();     ❸

        setUpOrderMocks();
        setUpJMSMocks();
        setUpJNDILookups();

        jndiTestSetup.getMockContext().matchAndReturn("close",
            null);
    }

    public void setUpOrderMocks()     ❷
    {
        mockOrderLocalHome = new Mock(OrderLocalHome.class);
        orderLocalHome =
            (OrderLocalHome) mockOrderLocalHome.proxy();

        mockOrderLocal = new Mock(OrderLocal.class);
        OrderLocal orderLocal = (OrderLocal) mockOrderLocal.proxy();

        mockOrderLocalHome.matchAndReturn("create", C.ANY_ARGS,
            orderLocal);
        mockOrderLocalHome.matchAndReturn("findByPrimaryKey",
            new Integer(1234), orderLocal);
        mockOrderLocal.matchAndReturn("getOrderId",
            new Integer(1234));
    }

    public void setUpJMSMocks()    ❷
    {
        mockQueue = new Mock(Queue.class);
        queue = (Queue) mockQueue.proxy();

        mockQueueConnectionFactory =
            new Mock(QueueConnectionFactory.class);
        queueConnectionFactory = (QueueConnectionFactory)
            mockQueueConnectionFactory.proxy();

        mockQueueConnection = new Mock(QueueConnection.class);
        queueConnection =
            (QueueConnection) mockQueueConnection.proxy();
        mockQueueConnection.matchAndReturn("close", null);

        mockObjectMessage = new Mock(ObjectMessage.class);
        ObjectMessage objectMessage =
            (ObjectMessage) mockObjectMessage.proxy();

        mockQueueSession = new Mock(QueueSession.class);
        QueueSession queueSession =
            (QueueSession) mockQueueSession.proxy();
        mockQueueSession.matchAndReturn("close", null);
        mockQueueSession.matchAndReturn("createObjectMessage",
            C.ANY_ARGS, objectMessage);

        mockQueueConnection.matchAndReturn("createQueueSession",
            C.ANY_ARGS, queueSession);

        mockQueueSender = new Mock(QueueSender.class);
        QueueSender queueSender =
            (QueueSender) mockQueueSender.proxy();
        mockQueueSender.matchAndReturn("close", null);
        mockQueueSender.matchAndReturn("send", C.ANY_ARGS, null);

        mockQueueSession.matchAndReturn("createSender",
            C.ANY_ARGS, queueSender);
    }

    public void setUpJNDILookups()    ❷
    {
        jndiTestSetup.getMockContext().matchAndReturn(
            "lookup", JNDINames.ORDER_LOCALHOME, orderLocalHome);
        jndiTestSetup.getMockContext().matchAndReturn(
            "lookup", JNDINames.QUEUE_ORDER, queue);
        jndiTestSetup.getMockContext().matchAndReturn(
            "lookup", JNDINames.QUEUE_CONNECTION_FACTORY,
            queueConnectionFactory);
    }

    protected void tearDown()
    {
        jndiTestSetup.getMockContext().verify();
        mockOrderLocal.verify();
        mockOrderLocalHome.verify();
        mockQueue.verify();
        mockQueueConnection.verify();
        mockQueueConnectionFactory.verify();
        mockQueueSender.verify();
        mockQueueSession.verify();
        mockObjectMessage.verify();
    }

    public void testCreateOrderOk() throws Exception
    {
        mockQueueConnectionFactory.expectAndReturn(
            "createQueueConnection", queueConnection);

        int orderId = petstore.createOrder(new Date(), "item1");

        assertEquals(1234, orderId);
    }

    public void testCreateThrowsOrderException() throws Exception
    {
        mockQueueConnectionFactory.expectAndThrow(
            "createQueueConnection", new JMSException("error"));

        try
        {
            petstore.createOrder(new Date(), "item1");
            fail("Should have thrown an EJBException");
        }
        catch (EJBException expected)
        {
            assertEquals("error",
                expected.getCausedByException().getMessage());
        }
    }
}
```

❶ Use the `JNDITestSetup` class to set up the mock JNDI implementation only once.

❷ Create all the mocks and define their expected behaviors common for all tests (the behavior specific to a given test is in the `test`*`XXX`* method).

❸ This is the tricky part in this listing. The call to `NamingManager.setInitialContextFactoryBuilder` must happen only once per JVM (otherwise an `IllegalStateException` is thrown). So, you need to share the `MockContext` mock across tests. Thus the jndiTestSetup variable is static so that jndiTestSetup.getMockContext always returns the same MockContext instance. The end consequence is that you need to reset the MockContext mock before each test so that it doesn't carry the expectations set on it from one test to another.

The rest of the code is straightforward; it looks the same as the code you saw in the previous section.

Extending the code for other tests

Here's an observation about the time it takes to write this test and to initialize all these mocks. Looking at the code's length, it seems this last strategy may be more costly than the previous ones. This isn't completely true. Indeed, in the factory method strategy and in the factory class strategy, you unit-test *only* the PetstoreEJB.createOrder method. However, in this JNDI implementation strategy, you unit-test not only the Petstore.createOrder method but also a good part of the OrderUtil.createOrder and JMSUtil.sendToJMSQueue methods. In addition, you also test the interactions between PetstoreEJB, OrderUtil, and JMSUtil; these weren't tested with the other strategies. Thus, with the other strategies you also need to write unit tests for OrderUtil and JMSUtil—and the combined length of the test code is either greater than or equivalent to the code you had to write in the mock JNDI implementation strategy.

Said another way, you have set up all the mock objects necessary for writing all unit tests revolving around PetstoreEJB, OrderUtil, JMSUtil, and OrderEJB. It's now very easy to write unit tests for any method of these objects. Indeed, at this stage, the refactoring you should perform is to move all the setup code in the setUp method to a CommonPetstoreTestCase class that can be extended by the different TestCases (see listing 12.10).

Listing 12.10 Common mock initialization for the simple Petstore application

```
package junitbook.ejb;
[...]
public class CommonPetstoreTestCase extends TestCase
{
    protected static JNDITestSetup jndiTestSetup;

    protected Mock mockOrderLocalHome;
    protected OrderLocalHome orderLocalHome;
    protected Mock mockOrderLocal;
    protected Mock mockQueue;
    protected Queue queue;
    protected Mock mockQueueConnectionFactory;
    protected QueueConnectionFactory queueConnectionFactory;
    protected Mock mockQueueConnection;
    protected QueueConnection queueConnection;
    protected Mock mockQueueSession;
    protected Mock mockQueueSender;

    public static Test suite(Class testClass)
    {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(testClass);
        jndiTestSetup = new JNDITestSetup(suite);
        return jndiTestSetup;
    }

    protected void setUp() throws Exception
    {
        jndiTestSetup.getMockContext().reset();

        setUpOrderMocks();
        setUpJMSMocks();
        setUpJNDILookups();

        jndiTestSetup.getMockContext().matchAndReturn("close",
            null);
    }

    public void setUpOrderMocks()
    {
        // same as in listing 12.9
        [...]
    }

    public void setUpJMSMocks()
    {
        // same as in listing 12.9
        [...]
    }

    public void setUpJNDILookups()
    {
        // same as in listing 12.9
        [...]
    }

    protected void tearDown()
    {
        // same as in listing 12.9
        [...]
    }
}
```

There are only two differences from listing 12.9. You make the class variables protected instead of private, so that they can accessed by subclasses. You also add a parameter to the static suite method so that suites can be easily created by subclasses.

To demonstrate the value of CommonPetstoreTestCase, let's write unit tests for OrderUtil as shown in listing 12.11 (see listing 12.2 for the OrderUtil sources).

Listing 12.11 Unit tests for OrderUtil using CommonPetstoreTestCase

```
package junitbook.ejb.domain;

import java.util.Date;
import javax.ejb.CreateException;
import com.mockobjects.dynamic.C;
import junit.framework.Test;
import junitbook.ejb.CommonPetstoreTestCase;

public class TestOrderUtil extends CommonPetstoreTestCase
{
    public static Test suite()
    {
        return suite(TestOrderUtil.class);
    }

    public void testGetOrderHomeOk() throws Exception
    {
        OrderLocalHome olh = OrderUtil.getOrderHome();
        assertNotNull(olh);
    }

    public void testGetOrderHomeFromCache() throws Exception
    {
        // First call to ensure the home is in the cache
        OrderUtil.getOrderHome();

        // Make sure the lookup method is NOT called thus proving
        // the object is served from the cache
        jndiTestSetup.getMockContext().expectNotCalled("lookup");   ❶
        OrderUtil.getOrderHome();
    }

    public void testCreateOrderThrowsCreateException()
        throws Exception
    {
        mockOrderLocalHome.expectAndThrow("create", C.ANY_ARGS,
            new CreateException("error"));

        try
        {
            OrderUtil.createOrder(new Date(), "item 1");
            fail("Should have thrown a RuntimeException here");
        }
        catch (RuntimeException expected)
        {
            assertEquals("Failed to create "
                + "[junitbook.ejb.domain.OrderLocal]. Reason "
                + "[error]", expected.getMessage());
        }
    }
}
```

The tests are similar to those you've written so far. The only difference is in the testGetOrderHomeFromCache test, which verifies that the caching of the Order EJB home works. You test this by calling the OrderUtil.getOrderHome method twice, and you verify that the second time it's called, it isn't looking up the EJB home in JNDI (❶).

As you can see, you haven't had the burden of defining and setting up mocks again, because their definitions have been externalized in CommonPetstoreTestCase. You have focused on writing the test logic, and the tests can be read easily. However, if you run this test case, it fails in the testCreateOrderThrowsCreateException method.

Resolving the EJB home caching issue

Let's take the time to understand why the test fails, because it's a common issue in unit-testing EJB code. The reason for the testCreateOrderThrowsCreateException test failure lies in the caching of the EJB home in OrderUtil. You cache the home with the following code:

```
public class OrderUtil
{
    private static OrderLocalHome orderLocalHome;

    protected static OrderLocalHome getOrderHome()
    {
        if (orderLocalHome == null)
        {
            Object obj =
                JNDIUtil.lookup(JNDINames.ORDER_LOCALHOME);
            orderLocalHome =
                (OrderLocalHome) PortableRemoteObject.narrow(
                    obj, OrderLocalHome.class);
        }
        return orderLocalHome;
    }
[...]
```

The side effect is that the first test you run results in the OrderLocalHome mock being cached, along with the expectations set on it. Thus, the behavior of the create method defined in listing 12.11 (mockOrderLocalHome.expectAndThrow("create", C.ANY_ARGS, new CreateException("error"))) isn't used; instead, the behavior from the previous test is used (mockOrderLocalHome.matchAndReturn("create", C.ANY_ARGS, orderLocal)), leading to an error.

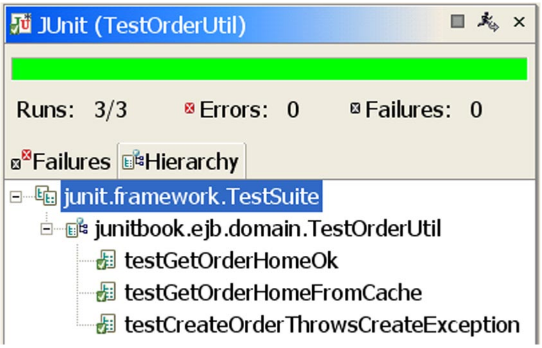
To fix this problem, you must reset the objects in their pristine states before each test. This is normally assured by the JUnit framework—except, of course, when you use static variables. One solution is to introduce an OrderUtil.clearCache method that you call in the TestOrderUtil.setUp method:

```
public class OrderUtil
{
    private static OrderLocalHome orderLocalHome;

    protected static void clearCache()
    {
        orderLocalHome = null;
    }

    protected static OrderLocalHome getOrderHome()
    {
        [...]
    }
[...]

public class TestOrderUtil extends CommonPetstoreTestCase
{
[...]
    protected void setUp() throws Exception
    {
        super.setUp();
        OrderUtil.clearCache();
    }
[...]
```

Running the tests now succeeds, as shown in figure 12.3.

12.5 Using mock objects to test message-driven beans

Unit-testing MDBs is easy when you use a mock-objects approach. The reason is that the JMS API is well designed: It uses a lot of interfaces and, in most cases, uses an IOC strategy by passing all the needed objects to method calls. Thus all the object instantiations are done in the client code, which makes it easier to control and mock. Let's see what this means on the simple Petstore application



Runs: 3/3 Errors: 0 Failures: 0

- junit.framework.TestSuite
  - junitbook.ejb.domain.TestOrderUtil
    - testGetOrderHomeOk
    - testGetOrderHomeFromCache
    - testCreateOrderThrowsCreateException

Figure 12.3 Successful test after fixing the EJB home caching issue that prevented you from running the unit tests independently of one another

by unit-testing the `OrderProcessorMDB` MDB (see figure 12.1). The method to unit test, `onMessage`, is shown in listing 12.12.

Listing 12.12 OrderProcessorMDB.java

```
package junitbook.ejb.service;

import javax.ejb.EJBException;
import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.ObjectMessage;

import junitbook.ejb.domain.OrderLocal;
import junitbook.ejb.domain.OrderUtil;

public class OrderProcessorMDB
    implements MessageDrivenBean, MessageListener
{
    public void onMessage(Message recvMsg)
    {
        ObjectMessage msg = (ObjectMessage) recvMsg;

        Integer orderId;
        try
        {
            orderId = (Integer) msg.getObject();
            OrderLocal order = OrderUtil.getOrder(orderId);
            proceedOrder(order);
        }
        catch (Exception e)
        {
            throw new EJBException("Error processing order...");
        }
    }

    private void proceedOrder(OrderLocal order) throws Exception
    {
        // Perform some business logic here and notify the customer
        // possibly by sending an email.
    }

    public void ejbCreate() {}
    public void setMessageDrivenContext(
        MessageDrivenContext context) {}
    public void ejbRemove() {}
}
```

This is similar to what you did in the previous sections when unit-testing session beans. Like session beans, MDBs can be unit-tested using several mock objects techniques: the factory method approach, the factory class strategy, or the mock JNDI implementation approach.

Let's use the mock JNDI implementation strategy and reuse the `CommonPetstoreTestCase` test case from listing 12.10. The resulting test case is shown in listing 12.13.

Listing 12.13 Unit test for OrderProcessorMDB.onMessage

```
package junitbook.ejb.service;

import javax.jms.ObjectMessage;

import com.mockobjects.dynamic.Mock;

import junit.framework.Test;
import junitbook.ejb.CommonPetstoreTestCase;
import junitbook.ejb.service.OrderProcessorMDB;

public class TestOrderProcessorMDB extends CommonPetstoreTestCase
{
    private OrderProcessorMDB orderProcessor;

    public static Test suite()
    {
        return suite(TestOrderProcessorMDB.class);       ❶ Required to
    }                                                       initialize
                                                            CommonPetstore-
                                                            TestCase
    protected void setUp() throws Exception
    {
        super.setUp();

        orderProcessor = new OrderProcessorMDB();
    }

    public void testOnMessageOk() throws Exception
    {
        Mock mockMessage = new Mock(ObjectMessage.class);
        ObjectMessage message =
            (ObjectMessage) mockMessage.proxy();

        mockMessage.expectAndReturn("getObject",
            new Integer(1234));

        orderProcessor.onMessage(message);

        mockMessage.verify();
    }
}
```

12.6 Using mock objects to test entity beans

Entity beans are the easiest to unit-test in isolation, especially if they are CMP entity beans. In that case, they are very much like standard Java beans. (For BMP entity beans, refer to chapter 11, “Unit-testing database applications,” which shows how to unit-test JDBC code.)

Listing 12.14 shows the code for the `OrderEJB` CMP EB from the Petstore application (see figure 12.1).

Listing 12.14 OrderEJB.java

```
package junitbook.ejb.domain;

import java.rmi.RemoteException;
import java.util.Date;

import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.RemoveException;

public abstract class OrderEJB implements EntityBean
{
    public abstract Integer getOrderId();
    public abstract void setOrderId(Integer orderId);
    public abstract Date getOrderDate();
    public abstract void setOrderDate(Date orderDate);
    public abstract String getOrderItem();
    public abstract void setOrderItem(String item);

    public OrderLocal ejbCreate(Date orderDate, String orderItem)
        throws CreateException
    {
        int uid = 0;

        // Note: Would need a real counter here. This is a hack!
        uid = orderDate.hashCode() + orderItem.hashCode();

        setOrderId(new Integer(uid));
        setOrderDate(orderDate);
        setOrderItem(orderItem);

        return null;
    }

    public void ejbPostCreate(Date orderDate, String orderItem)
        throws CreateException {}
    public void ejbActivate()
        throws EJBException, RemoteException {}
    public void ejbLoad()
        throws EJBException, RemoteException {}
    public void ejbPassivate()
        throws EJBException, RemoteException {}
    public void ejbRemove()
        throws RemoveException, EJBException, RemoteException {}
    public void ejbStore()
        throws EJBException, RemoteException {}
    public void setEntityContext(EntityContext context)
        throws EJBException, RemoteException {}
    public void unsetEntityContext()
        throws EJBException, RemoteException {}
}
```

Let's unit-test the `ejbCreate` method. The only issue is that when you're using the EJB 2.0 specification, a CMP EB is an abstract class and the field getters and setters are not implemented. You need to create a class that extends `OrderEJB` and implements the getters and setters. (Note that doing so is very easy with a good IDE, because every good IDE has a "Generate Getters and Setters" feature.) Let's implement it as an inner class of the `TestCase` (see listing 12.15).

Listing 12.15 Unit-testing OrderEJB.ejbCreate

```
package junitbook.ejb.domain;

import java.util.Date;

import junit.framework.TestCase;

public class TestOrderEJB extends TestCase
{
    public class TestableOrderEJB extends OrderEJB
    {
        private Integer orderId;
        private String item;
        private Date date;

        public Integer getOrderId()
            { return this.orderId; }
```

Make abstract EJB testable

```
        public void setOrderId(Integer orderId)
           { this.orderId = orderId; }
        public Date getOrderDate()
            { return this.date; }
        public void setOrderDate(Date orderDate)
            { this.date = orderDate; }
        public String getOrderItem()
            { return this.item; }
        public void setOrderItem(String item)
            { this.item = item; }
    }

    public void testEjbCreateOk() throws Exception
    {
        TestableOrderEJB order = new TestableOrderEJB();

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

        order.ejbCreate(date, item);

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

Make abstract EJB testable

NOTE If the code that generates a unique ID was more complicated (which would be the case in a real-life application), you might need to mock it.

The attentive reader will have noticed that unit-testing an `ejbCreate` method using a mock-objects approach isn't fantastically interesting. Using mock objects to unit-test EB business logic is fine; but `ejbCreate` is called by the container, which returns a proxy that implements the abstract `OrderEJB`. Thus, it usually makes more sense to perform an integration unit test for testing `ejbCreate`, as you'll see in the next section.

12.7 Choosing the right mock-objects strategy

We have discussed three strategies. Choosing the right one for a given situation isn't always easy. Table 12.1 lists the pros and cons of each strategy, to help you make the correct choice.

Table 12.1 Pros and cons of the different mock-objects strategies for unit-testing EJBs

Strategy	Pros	Cons
Factory method refactoring	Makes unit tests quick to write	Needs to refactor the class under test. Unfortunately, the refactoring doesn't usually improve the code.
Factory class refactoring	Supposed to improve code flexibility and quality	Doesn't work too well with EJBs because the EJB model doesn't lend itself to user-provided factories (only the container is supposed to provide factory implementations). In addition, requires large amounts of code refactoring.
Mock JNDI implementation	Provides a good common fixture setup for all EJB unit tests and requires minimal (if any) code refactoring	Initial tests take more time to write because all domain objects used need to be mocked. This leads to lengthy `setUp` code.

12.8 Using integration unit tests

The best feature of a J2EE container is that it provides several services for the EJBs it hosts (persistence, life cycle, remote access, transactions, security, and so on). Thus developers are freed from coding these technical details. The flip side is that there are many deployment descriptors and container configuration files. So, in order to have confidence that the J2EE application is working, it's important to write unit tests not only for the Java code but also for the descriptors/configuration files. Testing these metadata could be delegated to functional testing, but then you face the same issues as when you don't have unit tests: It's harder to find the cause of bugs, bugs are found later in the development process, and not everything that could break is tested.

As a consequence, performing integration unit tests for an EJB application is very useful. If you start doing it early, you'll also be rewarded because you'll learn as you go to configure your container properly, you'll validate your database data set, and, more generally, you'll be able to deliver a working application at each development iteration.

There are two main solutions for writing integration unit tests for EJBs: pure JUnit test cases making remote calls, and Cactus.[1] We'll discuss these solutions in sections 12.9 and 12.10.

12.9 Using JUnit and remote calls

It's possible to use JUnit directly to unit-test EJBs, without the need for any other extension. However, this technique has several limitations:

- *Requires remote interfaces*—It only works with EJBs that present remote interfaces. Any EJB that only has a local interface can't be tested using pure JUnit,
- *Doesn't run from the production execution environment*—Very often, the production code that calls the EJBs is server-side code (servlets, JSPs, taglibs, or filters). This means that if you run the tests from a standard JUnit test case, your tests will run in a different execution environment than the production one. For example, you may write `new InitialContext` in server-side code to get access to the `InitialContext` (if the servlet and EJB containers are co-located), whereas you need to use the `new InitialContext(Properties)` form when writing client-side code, because you need to specify the address of the JNDI server and the credentials to access it.
- *Requires data setup and database startup*—Before you can call your JUnit tests, you need to set up data in your database and start your container. Possibly you'll want to automate this process as well.
- *Has limitations of functional tests*—Performing remote EJB calls is more like doing functional unit testing than unit testing. How do you test exception cases, for example? Imagine that you're testing `PetstoreEJB.createOrder` (see listing 12.1) and you want to verify the behavior when `OrderUtil.createOrder` throws a `RuntimeException`. This isn't possible with a pure JUnit solution calling a remote EJB.

Let's see how you can unit-test the Petstore application. You know that you won't be able to unit-test `OrderEJB` (see figure 12.1) because it's a local interface. However, luckily, both `OrderProcessorMDB` and `PetstoreEJB` have remote interfaces you can test. Let's test them, starting with the `PetstoreEJB` SLSB.

[1] There is also JUnitEE (http://www.junitee.org/), a JUnit test runner that executes JUnit test cases in the container. However, we won't discuss it, because most (if not all) of the JUnitEE features are also available in Cactus, which we use and describe in this chapter.

12.9.1 Requirements for using JUnit directly

Because you need a J2EE container and a database for executing these tests, you'll use JBoss and Hypersonic SQL for this example. (See chapter 11 for details on how to preset database data before the test using DBUnit.)

In addition to the source code, you need some configuration files to have a working application. The mock-objects solution shown at the beginning of this chapter only unit-tests the code, not the configuration file; thus you haven't had to write these files yet.

You need at least the `application.xml` file, which is the descriptor for packaging the application as an ear, and the `ejb-jar.xml` file, which is the descriptor for the Petstore EJB-jar. You also need some other files, but you'll discover these requirements as you progress through the examples.

Finally, you need to begin thinking about the project directory structure and decide where to put these files. You also need to think about the build system (which you use to build the ear, start the server, run the tests, and so on). Which build system will you use (Maven, Ant, Centipede)? It's good to ask all these questions as early as possible in the project life cycle, because it takes time to set everything up. This is the condition to have a working application! For the purpose of this exercise, you'll use Ant as the build tool.

Defining a directory structure

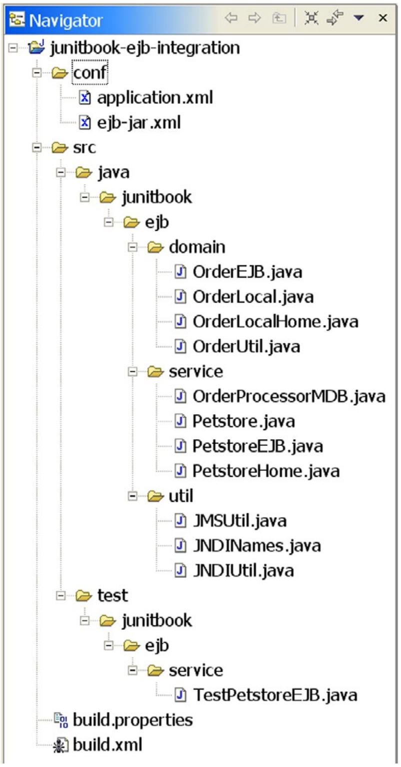
Let's use Ant to build the application and execute the JUnit tests. Figure 12.4 shows the directory structure.

Notice the `conf/` directory where you put the configuration files: `application.xml` and `ejb-jar.xml`. You'll add other configuration files as you progress with the tests.

The `src/` directory contains, as usual, the runtime sources and the test sources. The `TestPetstoreEJB.java` file contains the JUnit tests calling the `PetstoreEJB` EJB remotely. At the root are the usual Ant `build.xml` and `build.properties` files for building the project and running the tests automatically.

12.9.2 Packaging the Petstore application in an ear file

Let's start with the build file and see how to generate the application ear file. Listing 12.16 shows an Ant build script that has targets to compile the Petstore application source code, generate an EJB-jar, and wrap the whole thing in an ear file. Notice that to have a valid EJB-jar, you need to have an `ejb-jar.xml` deployment descriptor; and to have a valid ear, you need an `application.xml` descriptor.



```
junitbook-ejb-integration
  conf
    application.xml
    ejb-jar.xml
  src
    java
      junitbook
        ejb
          domain
            OrderEJB.java
            OrderLocal.java
            OrderLocalHome.java
            OrderUtil.java
          service
            OrderProcessorMDB.java
            Petstore.java
            PetstoreEJB.java
            PetstoreHome.java
          util
            JMSUtil.java
            JNDINames.java
            JNDIUtil.java
    test
      junitbook
        ejb
          service
            TestPetstoreEJB.java
  build.properties
  build.xml
```

Figure 12.4
Full directory structure, including configuration files and Ant build files

Listing 12.16 Ant build.xml (targets to generate the Petstore ear file)

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

<project name="Ejb" default="test" basedir=".">

  <property file="build.properties"/>

  <property name="conf.dir" location="conf"/>
  <property name="src.dir" location="src"/>
  <property name="src.java.dir" location="${src.dir}/java"/>

  <property name="target.dir" location="target"/>
  <property name="target.classes.java.dir"
      location="${target.dir}/classes"/>
```

Generic Ant properties pointing to location on filesystem

```
  <target name="compile">
    <mkdir dir="${target.classes.java.dir}"/>
    <javac destdir="${target.classes.java.dir}"
        srcdir="${src.java.dir}">
      <classpath>
        <pathelement location="${j2ee.jar}"/>
      </classpath>
```

Compiles runtime classes

```
    </javac>
  </target>

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

  <target name="ear" depends="ejbjar">      Generates ear file
    <ear destfile="${target.dir}/ejb.ear"
        appxml="${conf.dir}/application.xml">
      <fileset dir="${target.dir}">
        <include name="ejb.jar"/>
      </fileset>
    </ear>
  </target>

</project>
```

The build.properties file contains Ant properties whose values depend on where you have put jars and where you have installed JBoss on your machine (see listing 12.17). (Installing JBoss is easy: Download it from http://jboss.org/ and unzip the zip file anywhere on your machine.)

Listing 12.17 build.properties (initial version)

```
jboss.home.dir = C:/Apps/jboss-3.2.1
jboss.server.dir = ${jboss.home.dir}/server/default
jboss.deploy.dir = ${jboss.server.dir}/deploy

j2ee.jar = ${jboss.server.dir}/lib/jboss-j2ee.jar
```

Listing 12.18 shows the ear application.xml deployment descriptor used by the ear Ant task in listing 12.16.

Listing 12.18 application.xml

```
<?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>Sample Petstore Application</description>
  <module>
    <ejb>ejb.jar</ejb>
  </module>
</application>
```

Listing 12.19 shows the EJB deployment descriptor (`ejb-jar.xml`).[2]

Listing 12.19 ejb-jar.xml

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

<!DOCTYPE ejb-jar PUBLIC
  '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0//EN'
  'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>

<ejb-jar>
  <enterprise-beans>

   <session>
      <ejb-name>Petstore</ejb-name>
      <home>junitbook.ejb.service.PetstoreHome</home>
      <remote>junitbook.ejb.service.Petstore</remote>
      <ejb-class>junitbook.ejb.service.PetstoreEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <resource-env-ref>
        <resource-env-ref-name>
        →    jms/queue/petstore/Order</resource-env-ref-name>
        <resource-env-ref-type>
        →    javax.jms.Queue</resource-env-ref-type>
      </resource-env-ref>
    </session>

    <entity>
      <ejb-name>Order</ejb-name>
      <local-home>junitbook.ejb.domain.OrderLocalHome</local-home>
      <local>junitbook.ejb.domain.OrderLocal</local>
      <ejb-class>junitbook.ejb.domain.OrderEJB</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.Integer</prim-key-class>
      <reentrant>False</reentrant>
      <cmp-version>2.x</cmp-version>
```

[2] The deployment descriptors can also be automatically generated by XDoclet (http://xdoclet.sourceforge.net/). XDoclet is a wonderful tool that makes EJB development easier by generating most of the needed files. For more information on XDoclet, see *XDoclet in Action* by Craig Walls and Norman Richards (Greenwich, CT: Manning, 2003).

```
      <abstract-schema-name>Order</abstract-schema-name>
      <cmp-field>
        <field-name>orderId</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>orderDate</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>orderItem</field-name>
      </cmp-field>
      <primkey-field>orderId</primkey-field>
   </entity>

    <message-driven>
      <ejb-name>OrderProcessor</ejb-name>
      <ejb-class>junitbook.ejb.service.OrderProcessorMDB</ejb-class>
      <transaction-type>Container</transaction-type>
      <message-driven-destination>
        <destination-type>javax.jms.Queue</destination-type>
      </message-driven-destination>
    </message-driven>

  </enterprise-beans>
  <assembly-descriptor/>
</ejb-jar>
```

Executing `ant ear` generates the application ear, ready to be deployed in JBoss.

12.9.3 Performing automatic deployment and execution of tests

Let's now modify `build.xml` (see listing 12.20) to include a deployment target and targets for starting and stopping JBoss.

Listing 12.20 Additional targets in build.xml to deploy the ear and start and stop JBoss

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

<project name="Ejb" default="test" basedir=".">

  [...]
  <property name="src.test.dir" location="${src.dir}/test"/>

  [...]
  <property name="target.classes.test.dir"
      location="${target.dir}/classes-test"/>

  [...]
  <target name="compile.test" depends="compile">
    <mkdir dir="${target.classes.test.dir}"/>
    <javac destdir="${target.classes.test.dir}"
        srcdir="${src.test.dir}">
      <classpath>
        <pathelement location="${j2ee.jar}"/>
        <pathelement location="${junit.jar}"/>
        <pathelement location="${target.classes.java.dir}"/>
      </classpath>
    </javac>
  </target>

  [...]
  <target name="deploy" depends="ear">
    <copy todir="${jboss.deploy.dir}"
        file="${target.dir}/ejb.ear"/>
  </target>

  <target name="start" depends="deploy">
    <java classname="org.jboss.Main" fork="yes">
      <jvmarg
          value="-Dprogram.name=${jboss.home.dir}/bin/run.bat"/>
      <arg line="-c default"/>
      <classpath>
        <pathelement location="${jboss.home.dir}/bin/run.jar"/>
        <pathelement path="${java.home}/../lib/tools.jar"/>
      </classpath>
    </java>
  </target>

  <target name="stop">
    <java classname="org.jboss.Shutdown" fork="yes">
      <arg line="-s localhost"/>
      <classpath>
        <pathelement
            location="${jboss.home.dir}/bin/shutdown.jar"/>
        <pathelement path="${java.home}/../lib/tools.jar"/>
      </classpath>
    </java>
    <sleep seconds="15"/>
  </target>

</project>
```

The deployment is simple and consists of dropping the ear file into the correct JBoss `deploy` directory. The `compile.test` target compiles the test classes. You start and stop JBoss by starting a JVM and calling the correct JBoss Java class (JBoss is a Java application).

This code is fine, but you still need a `test` target that performs the orchestration of all these individual targets.

Creating a test target

In the test target, you need to deploy the ear, call the start target in a separate thread (so you can continue executing build commands), wait for the server to be started, run the tests, and stop JBoss. Fortunately, Ant 1.5+ has a nice parallel task that runs build commands on a separate thread, as shown in listing 12.21. (Note that this code can be further improved using the Ant http and socket tags from the condition task, but doing so is outside the scope of the example.)

Listing 12.21 Adding a test target to build.xml

```
<target name="test" depends="compile.test">
  <parallel>
    <antcall target="start"/>
    <sequential>
      <sleep seconds="30"/>
      <antcall target="run"/>
    </sequential>
  </parallel>
</target>

<target name="runtest">
  <junit printsummary="yes" fork="yes" errorproperty="test.error"     ❶
      failureproperty="test.failure">
    <formatter type="plain" usefile="false"/>
    <test name="junitbook.ejb.service.TestPetstoreEJB"/>
    <classpath>
      <pathelement location="${target.classes.java.dir}"/>
      <pathelement location="${target.classes.test.dir}"/>
      <fileset dir="${jboss.home.dir}/client">
        <include name="*.jar"/>
      </fileset>
    </classpath>
  </junit>
</target>

<target name="checktestfailures" if="test.failure">   ❶
  <fail>There were test failures</fail>
</target>

<target name="checktesterrors" if="test.error">   ❶
  <fail>There were test errors</fail>
</target>

<target name="run"
    depends="runtest,stop,checktesterrors,checktestfailures">   ❶
</target>
```

Note that you use the `errorproperty` and `failureproperty` attributes of the `junit` Ant task (❶). You use these attributes instead of the more familiar `haltonfailure` and `haltonerror` attributes because you want the `stop` target to be called whatever the outcome of the test, in order to cleanly stop the container.

Verifying deployment

To verify that the application deploys correctly, type **`ant test`** and relax. The output is shown in figure 12.5.

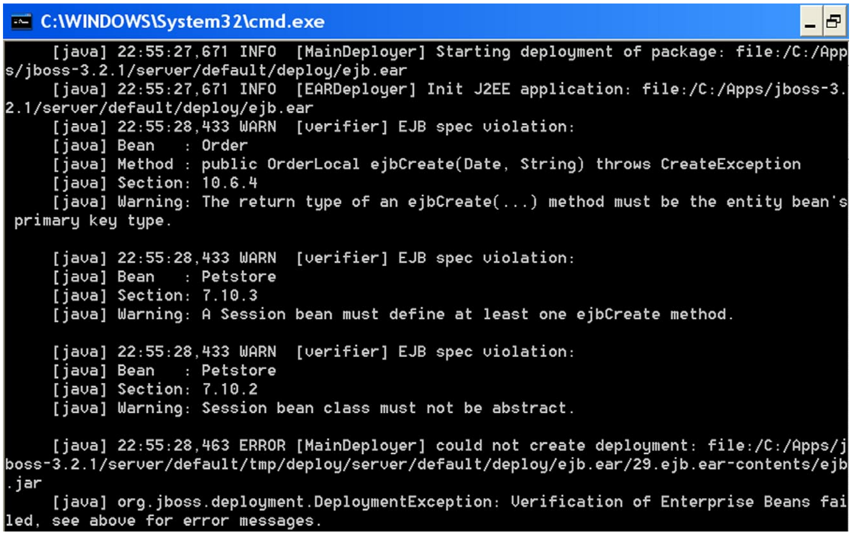
NOTE Before you run JBoss 3.2.1 for the first time, you need to edit the `server/default/conf/jboss-service.xml` file (located in the directory where you installed JBoss) and change the line

```
<attribute name="RecursiveSearch">False</attribute>
```

to

```
<attribute name="RecursiveSearch">True</attribute>
```

The reason is that JBoss 3.2.1 was shipped with an incorrect value that prevents the JMS services from being deployed correctly. If you don't make this change, you'll get a *DefaultJMSProvider not bound* error when JBoss tries to deploy the `OrderProcessorMDB` MDB.



```
    [java] 22:55:27,671 INFO  [MainDeployer] Starting deployment of package: file:/C:/App
s/jboss-3.2.1/server/default/deploy/ejb.ear
    [java] 22:55:27,671 INFO  [EARDeployer] Init J2EE application: file:/C:/Apps/jboss-3.
2.1/server/default/deploy/ejb.ear
    [java] 22:55:28,433 WARN  [verifier] EJB spec violation:
    [java] Bean    : Order
    [java] Method : public OrderLocal ejbCreate(Date, String) throws CreateException
    [java] Section: 10.6.4
    [java] Warning: The return type of an ejbCreate(...) method must be the entity bean's
 primary key type.

    [java] 22:55:28,433 WARN  [verifier] EJB spec violation:
    [java] Bean    : Petstore
    [java] Section: 7.10.3
    [java] Warning: A Session bean must define at least one ejbCreate method.

    [java] 22:55:28,433 WARN  [verifier] EJB spec violation:
    [java] Bean    : Petstore
    [java] Section: 7.10.2
    [java] Warning: Session bean class must not be abstract.

    [java] 22:55:28,463 ERROR [MainDeployer] could not create deployment: file:/C:/Apps/j
boss-3.2.1/server/default/tmp/deploy/server/default/deploy/ejb.ear/29.ejb.ear-contents/ejb
.jar
    [java] org.jboss.deployment.DeploymentException: Verification of Enterprise Beans fai
led, see above for error messages.
```

Figure 12.5 Boss deployment errors when running the Ant test target

Argh! And you thought the application was working! It doesn't even deploy! The output shows that you have at least three errors. We say "at least" because errors are like trains—one error can hide another. In practice, there are seven errors, as shown in table 12.2. Note that we purposely didn't include these errors; they are genuine errors we made when coding the EJB sample. We started by writing the mock-object tests and they ran fine. Then we moved to the integration unit tests and discovered all these errors.... This is a clear demonstration that unit testing in isolation is not enough and needs to be supplemented by either integration unit tests or functional tests.

Table 12.2 Deployment errors uncovered by the integration unit test

Error description	Fix

OrderEJB implementation of `ejbCreate` must return the primary key (`Integer`) and not the bean interface (`OrderLocal`).	Replace
	`public OrderLocal ejbCreate[...]`
	with
	`public Integer ejbCreate[...]`

OrderEJB implementation of `ejbCreate` was returning null. It must return the primary key.	Replace
	`return null;`
	with
	`return new Integer(uid);`

CMP EB fields cannot be primitive types; they must be Java objects. One method of the `OrderLocal` interface was using a primitive type.	Replace
	`void setOrderId(int orderUId);`
	with
	`void setOrderId(Integer orderUId);`

The `createOrder` method of the `Petstore` interface wasn't throwing a `RemoteException`. This is required for remote methods.	Replace
	`int createOrder([...]);`
	with
	`int createOrder([...])`
	`throws RemoteException;`

The `PetstoreEJB` session bean was wrongly declared `abstract`. Only CMP EB classes can be declared `abstract`.	Replace
	`public abstract class PetstoreEJB`
	`implements SessionBean`
	with
	`public class PetstoreEJB`
	`implements SessionBean`

continued on next page

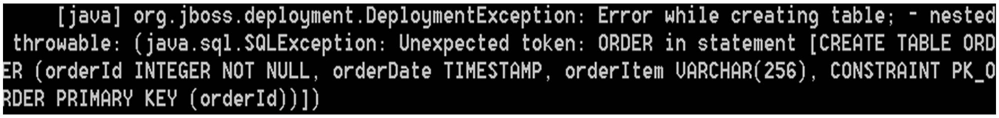
Table 12.2 Deployment errors uncovered by the integration unit test *(continued)*

Error description	Fix

The `PetstoreEJB` class was missing the implementation of `ejbCreate`. This is required for a session bean.	Add
	`public void ejbCreate()`
	`throws CreateException,`
	`RemoteException {}`

The `create` method of the `PetstoreHome` interface wasn't throwing a `RemoteException`. This is required for remote methods.	Replace
	`Petstore create() throws`
	`CreateException;`
	with
	`Petstore create() throws`
	`CreateException, RemoteException;`

Once all these errors are fixed, you will still stumble across another error, which is difficult to diagnose (see figure 12.6).



```
    [java] org.jboss.deployment.DeploymentException: Error while creating table; - nested
 throwable: (java.sql.SQLException: Unexpected token: ORDER in statement [CREATE TABLE ORD
ER (orderId INTEGER NOT NULL, orderDate TIMESTAMP, orderItem VARCHAR(256), CONSTRAINT PK_O
RDER PRIMARY KEY (orderId))])
```

Figure 12.6 This error is difficult to diagnose.

What happens is that JBoss automatically creates database tables for your CMP entity beans. However, because the CMP EB is named `Order`, JBoss generated the following SQL code:

```
[CREATE TABLE ORDER (orderId INTEGER NOT NULL, orderDate TIMESTAMP,
orderItem VARCHAR(256), CONSTRAINT PK_ORDER PRIMARY KEY (orderId))]
```

That tripped Hypersonic SQL (the default database used by JBoss) because `ORDER` is a reserved SQL keyword. We had to introduce a `jbosscmp-jdbc.xml` JBoss-specific file to solve this problem (we mapped the `Order` bean to the `Orders` table):

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

<!DOCTYPE jbosscmp-jdbc PUBLIC "-//JBoss//DTD JBOSSCMP-JDBC 3.0//EN"
  "http://www.jboss.org/j2ee/dtd/jbosscmp-jdbc_3_0.dtd">

<jbosscmp-jdbc>
  <enterprise-beans>
    <entity>
      <ejb-name>Order</ejb-name>
      <table-name>Orders</table-name>
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

These are the kind of errors you have to solve when you perform real integration unit testing.

Let's now imagine that you have fixed all the deployment errors. It's high time to write the TestPetstoreEJB test class.

12.9.4 Writing a remote JUnit test for PetstoreEJB

The principle is simple: The test is a client of the remote Petstore EJB. Listing 12.22 demonstrates how to write such a test.

Listing 12.22 TestPetstoreEJB.java

```
package junitbook.ejb.service;

import java.util.Date;

import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

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

public class TestPetstoreEJB extends TestCase
{
    public void testCreateOrderOk() throws Exception
    {
        Properties props = new Properties();                          ❶
        props.put(Context.INITIAL_CONTEXT_FACTORY,
            "org.jnp.interfaces.NamingContextFactory");
        props.put(Context.PROVIDER_URL, "localhost:1099");
        props.put(Context.URL_PKG_PREFIXES,
            "org.jboss.naming:org.jnp.interfaces");
        InitialContext context = new InitialContext(props);

        Object obj = context.lookup(JNDINames.PETSTORE_HOME);         ❷
        PetstoreHome petstoreHome =
            (PetstoreHome) PortableRemoteObject.narrow(
                obj, PetstoreHome.class);

        Petstore petstore = petstoreHome.create();
        Date date = new Date();
        String item = "item 1";

        int orderId = petstore.createOrder(date, item);
        assertEquals(date.hashCode() + item.hashCode(), orderId);
    }
}
```