CHAPTER 3

Enterprise Testing

Before we delve into the issues surrounding enterprise testing in Java, it's important to define exactly what we mean by *enterprise*.

It's hard to conceive of a word with as many meanings and connotations (and misconceptions!) as *enterprise* in Java. For many, this word is tied to the usage of the Java Enterprise Edition (J2EE, or its current incarnation, Java EE), whose APIs enable us to bless our applications with the enterprise stamp. For others, enterprise applications have specific features regardless of what APIs or even specific languages are used.

An example of using the enterprise API is an intranet application that manages a fixed set of entities, with its own backing store. It is likely that this application has a Web-based UI and that it uses some combination of servlets, JSP pages, and a persistence mechanism. In this example, the use of the ubiquitous term refers only to the API usage, and it is a relatively simple matter to ensure that this application can be tested easily, if one uses the right tools for the job.

Another example is an integration project in which a new middle tier is being added between two existing legacy systems, with the hope of slowly phasing out the old back end. This new layer has to be able to encapsulate the mapping between the two legacy systems, but more often than not, it is not allowed to modify either of the legacy systems. The mapping will likely be complex and require orchestration between a number of other external systems. In this case, we are much less likely to achieve our ideal of easy, quick-to-run unit tests and are far more likely to benefit from integration and functional tests.

That is not to say that enterprise projects cannot benefit from unit tests. It is also almost always possible to break down components into small enough pieces that meaningful unit tests can be derived, and all three types of tests go together hand in hand.

This chapter and the following one discuss testing issues with both definitions of *enterprise*. We need to be aware of a number of key concepts and issues when testing enterprise applications. These issues are not concerned with APIs but rather with the very nature of enterprise systems: complex integration issues, legacy system support, black-box testing, and so on. Generally, the assumption is that we have either a body of existing code that we need to integrate with or a system that is already in use but needs tests. Once we've established this foundation, the following chapter will discuss how to test specific J2EE or Java EE components.

Before we start, here's a brief recap of the different types of tests.

- *Unit tests*: A unit test tests an individual unit in the system in isolation. Unit tests run very quickly since they have little to no start-up costs, and almost no external dependencies.
- *Functional tests*: A functional test focuses on one piece of functionality. This usually involves interactions between different components.
- *Integration tests*: An integration test is an end-to-end test that exercises the entire stack, including any external dependencies or systems.

A Typical Enterprise Scenario

To illustrate the concepts around enterprise integration and functional testing, it's helpful to examine a real-world example. Let's say that we're consulting for a financial institution that has a legacy back-end database that houses most of its financial data. This database is one of the major bottlenecks of the system. The database is the central point for all financial trade information and is directly read by a number of front- and back-office applications.

In addition to that, some of the newer applications talk to a recently implemented abstraction layer. The abstraction layer grew organically based on the needs of specific applications and was not designed up front to be a middle tier. It has many idiosyncrasies and is so convoluted and complicated right now that it is no longer possible for new applications to easily use it.

The company decides that it is time to revamp the system. The goal is to introduce a middle tier designed from the outset to service most if not all applications that need data from the database. The database is split into a number of smaller instances and the data partitioned according to business requirements.

After the new system is implemented, it quickly proves itself profitable. Due to the phased approach of development, some applications still talk to the old legacy database, but a number have been ported over to the new system. The new system acts as a mediator between the various components and includes transformation components to ensure the correct data is still fed to legacy systems that expect the old formats and schemas.

Participants

Confused yet? You shouldn't be. Chances are that most developers have been in this situation during one project or another. Is this project bizarre or extreme in its complexity? Perhaps in the details it is, but the overall issues confronting it are fairly standard and commonplace. Let us step back a bit and see if we can identify the main participants:

- The legacy database: the source of all evil
- The shiny new API: the source of all good
- Dozens of legacy systems: the nature of the business, neither good nor bad
- Transformers: a necessary evil to allow components to talk to one another

This probably is starting to sound more familiar. Most if not all enterprise applications have to deal with legacy data at some point. This could be a migration issue, it could be a transformation issue, or it could be simply the introduction of a new layer on top of existing systems.

Testing Methodology

So what testing methodology does this successful new project employ? Judging by its success, it must consist of rigorous unit tests, countless integration and functional tests, nightly builds, email notifications of test failures—all the good developer testing habits that every successful project has.

As a matter of fact, it has none of these. The testing methodology of this project consists mainly of developers writing the odd class with a `main(String[] args)` method, running that against their data, and eyeballing the results. If it looks good, the functionality is deemed complete, the code checked in, and that's the end of that. Before a production release, there is a one- or two-week period where a QA team goes through the application and tries to find bugs. This is a manual process, but by the time it's done, the production release is in pretty good shape. The code is deployed, and everyone is happy.

The developers involved in this project range from experienced team leads to average developers. Almost all of the developers know about unit testing and have written a unit test in the past. The project did not mandate formalized test code, so there was no requirement to develop a test harness or automated tests.

Furthermore, all the developers agreed that it does not make sense to unit test the code. It is an integration project and therefore impossible to capture the important business aspects that need to be tested in a single unit test. The tests written would violate any number of popular testing recommendations; they would take a long time to run (many seconds), have complicated setup requirements (a few more seconds), and require a specific environment in that they would be highly dependent on a specific database schema, with specific data and stored procedures.

We suspect that this conclusion is far more common than many testing advocates would like us to believe. It is tempting to dismiss developers who are not obsessive about writing tests as ignorant or incompetent. Both assumptions are rather incorrect. JUnit, for example, currently makes it difficult to think in terms of integration or functional testing; there is a stigma of sorts attached to tests that have complicated environment requirements (and as a byproduct, slow running tests). Developers shy away from them. Yet for enterprise projects, such tests are far more valuable than unit tests. An integration project, unsurprisingly one would think, is exactly what integration tests excel at.

Issues with the Current Approach

So where's the problem? The project works and is a success, and everyone is happy. As the popular saying goes, if it ain't broke, why fix it? However, it turns out that the current approach has a number of inefficiencies.

QA Cycle Is Too Long

Currently, every release requires one or two weeks of full-time testing. Bugs discovered during this testing phase are added to a list of issues that should always be tested. The testing cycle often runs late if many issues are found, as many things need to be retested once the first batch of issues has been resolved.

Poor Test Capture

Developers currently write plenty of tests that are discarded as soon as the functionality being tested starts working. The main method is simply rewritten, or code is commented out and commented back in to reconfirm a test. There is no growing body of tests, nor is there a way to automate these informal tests.

Regression Testing Effort Grows Linearly

With every QA cycle, issues found are added to a growing master list of issues that need to be tested for every release. It becomes the QA team's job to perform all regression testing. This isn't such a problem with just a handful of releases, but the new system in place is expected to have a lifetime of at least five years, with many more enhancements and changes to come in future releases. Within a year or two, the mountain of regression tests is very likely to have a significant negative impact on the manual test cycle.

Lack of Unit Tests

The developers often argue that the system is too complex to be tested usefully through unit tests. This could well be true, in the general case. However, it is highly likely that a number of components or pieces of functionality do lend themselves well to unit testing. In a large, complex system, it can be a daunting task to identify these components, so the tendency is to stick to integration and functional tests.

Once we do have integration tests, unit tests more often than not will naturally emerge. Because the testing infrastructure is already in place, debugging an integration test is quite likely to result in a unit test, simply to try to narrow the scope of the bug.

A Concrete Example

So where do we start? Let's look at a typical component of this system, identify what we want to test, and then choose a strategy of how to test.

A fairly typical component in this system receives a JMS message that contains a payload of an XML document. The XML document is fairly large (400K or so) and describes a financial transaction. The component's job is to read in the message, parse the XML, populate a couple of database tables based on the message contents, and then call a stored procedure that processes the tables.

The sequence diagram in Figure 3–1 helps illustrate the message flow for this component.

Listing 3–1 shows the rough outline of the code we'd like to test.

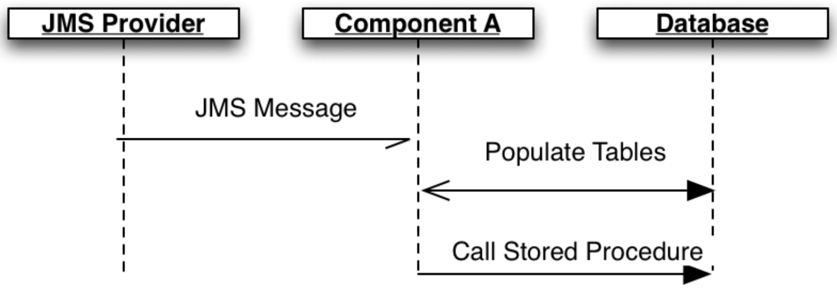

Figure 3–1 Sequence diagram for a typical component

Listing 3–1 Existing message processor for the legacy component

```
public class ComponentA implements javax.jms.MessageListener {
  private Log log = LogFactory.getLog(ComponentA.class);

  public void onMessage(Message message) {
    java.sql.Connection c = null;
    PreparedStatement ps = null;
    TextMessage tm = (TextMessage)message;
    String xml = null;
    try {
      xml = tm.getText();

      // XMLHelper is a util class that takes in an XML string
      // and parses it and returns a document
      Document doc = XMLHelper.parseDocument(xml);

      // manipulate the document to get at various elements
      // DatabaseHelper is a util class to look up and return a
      // database connection
      c = DatabaseHelper.getConnection();
      String sql = ""; // create SQL to call in db
      ps = c.prepareStatement(sql);
      // populate sql
      ps.executeUpdate();
      String spSql = "<sql to execute stored proc>";
      // call the stored procedure
      c.prepareCall(spSql).execute();
    }
```

```
      catch(Exception ex) {
        log.error("Error processing message " + message
          + " with data " + xml,ex);
      }
      finally {
        if(c != null) {
          try {
            if(ps != null) {
              try {
                ps.close();
              }
              catch(SQLException e) {
                log.error("Error closing statement", e);
              }
            }
            c.close();
          }
          catch(SQLException e) {
            log.error("Error closing connection", e);
          }
        }
      }
    }
  }
```

The focus of this exercise is to ensure that we test our component. A vital aspect of the process is also explicitly defining our test goals and non-goals up front, including the assumptions we're making.

Goals

Any functionality that we'd like to explicitly verify or check is considered one of the prime goals of the test process. For our specific case, we'd like to meet the following three goals.

1. We will create a success test. We want to ensure that if we receive a valid XML message, we process it correctly and update the correct database tables, and the stored procedure is also successfully called.
2. We will model different scenarios. We would like to be able to feed a variety of XML documents to our test to be able to easily add a growing body of sample data and use it for regression testing.

3. We will institute explicit failure tests. Failure behavior should be captured and tested so that the state of the component when it fails internally is predictable and easily captured.

Nongoals

Equally important as choosing goals is identifying nongoals. These are tasks that, if we're not careful, we might accidentally end up testing, thus having our tests focus on the wrong thing. We have three nongoals in our case.

1. We will not test the JMS provider functionality. We assume that it is a compliant implementation that has been correctly configured and will successfully deliver the intended message to us. The JMS API allows us to work with the `TextMessage` object. We can assume that this object allows us to get at the message contents without throwing any exceptions and that it will be correctly delivered. Finally, we can always have separate integration tests that verify the system's behavior end-to-end.
2. We will not perform catch-all error testing. Failure test should model explicit and reproducible failure modes. A failure test that, for example, checks what happens if a `NullPointerException` is thrown is somewhat useless.
3. We will not test APIs. The behavior of the JDBC driver is not the test subject, for example. It is also important to ensure that all our tests focus on our business functionality and to avoid tests that test Java language semantics. Therefore, we are not interested in verifying that the XML parser is able to parse XML; we assume it can.

Test Implementation

Based on our goals, we can now start to define a test for our component. The test definition involves going through each of our goals and enhancing the test so the goal is satisfied, while ensuring that we do not accidentally get distracted with any of the nongoals.

The first goal is to ensure that a valid XML document is processed correctly and the appropriate database calls made. Listing 3–2 shows the test skeleton.

Listing 3–2 Initial attempt at a functional test

```
@Test
public void componentAShouldUpdateDatabase() throws Exception {
  ComponentA component = new ComponentA();
  component.onMessage(...);
  Connection c = DatabaseHelper.getConnection();
  String verifySQL = ...;
  PreparedStatement ps = c.prepareStatement(verifySQL);
  // set parameters
  // read resultset and verify results match expectations
  String someValue = resultSet.getString(1);
  assert "foo".equals(someValue);
}
```

Testing for Success

As soon as we start to fill in our test code, we start running into problems. The first problem we have is that the component's only method is the `onMessage` method. This method takes in a `JMSMessage`. This class in the JMS API is in fact an interface, as is our expected message type, `TextMessage`. The API does not provide for an easy way to create instances of these interfaces (which, incidentally, is a good thing—an API should define contracts, not implementations). So how do we test our component?

There are two options for tackling this hurdle.

1. Use mock (or stub) objects to create our own implementation of `TextMessage`, represented by a simple POJO with setters for the message body and properties.
2. Refactor the component so the business functionality is not coupled to the JMS API.

The first approach is fairly popular but violates one of our nongoals, which is to not test external APIs. Strictly speaking, we'd be trying to use mock objects to refactor away the external API dependency. In practice, however, we'd have to model too much of it.

We would have to define a JMS message, and to ensure correctness, our implementation would have to be checked to ensure it matches the specification contract for `TextMessage`, if we hope to reuse it in any other tests that might expect different (and more compliant!) semantics of `TextMessage`. This extra code is another source of potential bugs and is yet more code to maintain. The mock object approach for external APIs should generally be used only for black-box testing, where we do not have access or rights to modify the source for the code being tested and so are forced to provide an environment that matches its expectations.

Although using mock or stub objects is the incorrect choice for our test, this is not always the case. For APIs that are complex or have very implementation-specific behavior, mocking of third-party dependencies should be avoided. However, there are times when the external API is trivial and easy to mock, in which case there is no harm in whipping up a quick stub for testing purposes.

The second approach is the correct one for our purposes. Since our goal is not to check whether we can retrieve text from a JMS message, we assume that functionality works and can be relied on. Our component should instead be modified so that the business functionality is decoupled from the incoming message. The decoupling gains us an important benefit: increased testability. We did make an implicit tradeoff in this decision, too. The modification to the code is the result not of a domain-based consideration (no business requirement is satisfied by this change) but of a testability one.

In Listing 3–3, the `onMessage` method handles all the JMS interaction and then passes the XML document string to the `processDocument` method, which then does all the work.

Listing 3–3 Refactoring component to decouple extraction from parsing

```
public void onMessage(Message message) {
  TextMessage tm = (TextMessage)message;
  processDocument(tm.getText());
}

public void processDocument(String xml) {
  // code previously in onMessage that updates DB
  // and calls stored procedure
}
```

We can now modify our functional test as shown in Listing 3–4 so that it no longer references JMS at all and instead simply passes the XML string to the `processDocument` method.

Listing 3–4 Refactored test to only test message processing

```
@Test
public void componentAUpdateDatabase() throws Exception {
  ComponentA component = new ComponentA();
  String xml = IOUtils.readFile(new File("trade.xml"));
  component.processDocument(xml);
  Connection c = DatabaseHelper.getConnection();
  String verifySQL = ...;
  PreparedStatement ps = c.prepareStatement(verifySQL);
  // set parameters
  // read resultSet and verify that results match expectations
  String someValue = resultSet.getString(1);
  assert "foo".equals(someValue);
}
```

Note how we load in the sample XML data from a file and then pass it to the component. The fact that the component happens to rely on JMS for message delivery is not relevant in terms of its business functionality, so we restructured the component to allow us to focus on testing the functionality rather than the JMS API.

An interesting side effect of this approach is that we made the `processDocument` method public. This method could well be an implementation detail that should not be exposed. To restrict its access level, we could make it protected or package protected and ensure that the test case is in the appropriate package. That way it can be invoked from the test but not from other clients.

As a side note, though we've moved the processing into another method, in practice we'd go a bit further than that and move it to another class altogether. That refactoring will result is a more reusable class that is not coupled to JMS at all.

At this point, we have a test that can verify that a sample XML file can be processed and that the database has been updated correctly.

Building Test Data

Now that we can consume a previously recorded XML file, we can easily grow the test input data and support as many files as we want. We can create a test for every file that ensures that all sorts of different input data can be verified.

Unfortunately, this approach very quickly proves itself to be rather cumbersome. The input XML files can vary significantly, and alarm bells should be going off anyway whenever we find ourselves copying and pasting, thus violating the Don't Repeat Yourself (DRY) principle.

As we have discussed previously, this is where it's useful for the testing framework to support Data-Driven Testing. We simply modify our test to use Data Providers, and parameterize the XML data as shown in Listing 3–5.

Listing 3–5 Refactored test using a Data Provider

```
@Test(dataProvider = "componentA-data-files")
public void componentAUpdateDatabase(String xml) throws Exception {
  ComponentA component = new ComponentA();
  component.processDocument(xml);
  // rest of test code
}

@DataProvider(name = "componentA-data-files")
public Iterator<Object[]> loadXML() throws Exception {
  // return data set
}
```

Note that our test now takes in a parameter and no longer has to concern itself with sourcing the sample data or determining how to load it in. All it does now is specify its Data Provider. The actual mechanism of loading the XML is now delegated to a separate loader. Our test is cleaner as a result since we have parameterized the variable data and can now invoke it multiple times for each of our sample XML files.

The type of the parameter is a `String`. Due to how the two methods are related, it is not possible to have a type-safe declaration for the method parameter. The Data Provider must return a type that matches that declared by the test method. For example, if the `loadXML` method were to return `Document` objects, we would get a runtime type mismatch exception.

The Data Provider itself can now deal with loading the XML file, as shown in Listing 3–6. Note that it does not need to have a hardcoded list. Instead it scans a specific directory and feeds all the files found to the test case. So the next time a new sample XML file needs to be added to the test suite, we just have to drop it in a specific directory and it will automatically be included, no coding or recompilation needed.

Listing 3–6 Refactored test to read in all data files from a specific directory

```
@DataProvider(name = "componentA-data-files")
  public Iterator<Object[]> loadXML() throws Exception {
    File[] f = new File("samples/ComponentA/trades").listFiles();
    final Iterator<File> files = Arrays.asList(f).iterator();
    return new Iterator<Object[]>() {

      public boolean hasNext() {
        return files.hasNext();
      }

      public Object[] next() {
        return new Object[]{IOUtils.readFile(files.next())};
      }

      public void remove() {
        throw new UnsupportedOperationException();
      }
    };
  }
```

The provider is fairly simple. It grabs a list of all the XML files in a specific directory and adds the file contents to the parameters. The file contents are added as an array of size 1 since the test method takes in just the one parameter. If we needed to parameterize other variables, that would be reflected in the array returned by the `next()` iterator method.

The provider method name does not matter at all; it can be whatever is most appropriate for a given case, as long as the `@DataProvider` annotation name matches what our test expects.

Of course, it is possible to return an array of `Object[]` from the Data Provider. However, that approach would mean that we would have to load all the file data in memory at once since the array has to be prepopulated. While this will work for a small data set, the memory requirements of the test will keep increasing over time, so the test will not scale with our data. Since this test is designed to grow over time, a little bit of upfront planning will head off this issue early on; we simply use lazy loading for the Data Provider so we only load one file's data at a time.

Test Setup Issues

Unfortunately, our test is not idempotent. An idempotent test is one where the result of running a test once is the same as running it multiple times. Describing something as idempotent is essentially saying that it does not alter state when it is invoked multiple times. So, for example, a method that reads data from a database is idempotent since calling it again will return the same data. On the other hand, a method that writes to a database may not be idempotent; invoking it again will likely result in an error since the state of the database has changed once the method has been invoked.

While we'll cover specific strategies for handling database setup and management, the concepts apply equally to any external stores we might need to interact with as part of the tests. These range from file systems to WebDAV resources to remote repositories of any format.

Not only is our test required to be idempotent, but the ordering of tests themselves shouldn't matter (assuming we haven't declared dependencies to enforce any ordering). So in addition to being idempotent, tests should not have any impact on other tests in terms of state or data.

Since the test performs a number of write operations, successive runs can easily be polluted from the results of previous runs. Any test that writes to a database will suffer from this problem, and while there is no ideal solution, a number of approaches can help us cope with this problem.

- Embedded databases
- Initialized data in the test setup
- Transaction rollbacks

Each of these approaches has its uses, and which combination of them we end up going with depends on the application and environment; some might not be options, and some might be more cumbersome than others.

Note that it might be tempting to consider using a mock library for the JDBC functionality. Resist that temptation! We discussed mocks earlier, and this is a great example of the urge and the need to resist it. A JDBC mock object would not (could not, even) cope with all the intricacies of database behavior, much less all the issues surrounding transactions or locking.

Embedded Databases

A number of Java-based database engines have been specifically designed with embedding support in mind. These databases can be created and initialized on the fly, from inside the test. They have low overhead in terms of setup costs and often perform very well.

The disadvantage of this approach, however, is that it deviates significantly from the environment the application will actually run in. There are also often significant differences between database features. While this approach is well suited to applications that use a database purely as a data store and restrict themselves to ANSI SQL database calls or use an abstraction layer (such as JPA or any similar object-relational mapping tool), it is not suitable for any applications (such as our example) that have application logic embedded in the database. Stored procedures are not portable across databases, and reimplementing them in our embedded database would be too much effort.

Initialized Data in the Test Setup

The next approach is to load our test database with a known quantity of test data. This would include all the data we'd like to manipulate, as well as any external references that our component relies on. For some tests, this might not even be sufficient, so in addition to loading data we'd have to ensure that extraneous data is also removed on start-up. While somewhat cumbersome, this approach can be combined with the embedded database engine to satisfy the needs of the component sufficiently for it to run successfully in any environment.

In practice, many tests rely on two different kinds of data. The first is *statics*. Statics are effectively constants stored in the database. For example, the list of U.S. states is considered a static, as is a list of currencies. If our test does its work against a shared remote database (a test instance, not the production one!), it can reasonably expect that the static data will be in place. After all, this information is constant across all tests, so there's no reason to load it in every run.

However, tests do also rely on data that is specific to their business functionality. For example, we might have a test that asserts that an audit trail for a financial transaction meets certain criteria, or that an invalid audit trail correctly raises the right alarms. In such cases, our test setup needs to load this test data into the database and then clear it out after the test run.

One downside of this approach is the difficulty of maintaining a robust data set that is meaningful enough to test. As the project progresses, there's a strong chance that data structures and schemas will change, and the test data can become stale. Updating the tests constantly in this situation can be quite unsatisfying as it involves duplicating the effort it has taken to implement the changes in the rest of the code base.

Thus, we have a tradeoff between capturing a meaningful data set and locking ourselves into a very specific snapshot of our model that will constantly need updating and modification to keep up to date. There is no right answer for which approach is best; the choice varies depending on the project and how much we expect the model to evolve over time.

Transaction Rollbacks

Another approach is to use Java APIs to prevent the data from being written out to the permanent data store. In both cases, the general idea is to start a transaction, perform all of our write operations, verify that everything works, and then roll back the transaction. The benefit of this approach is that we do not have to worry about cleanup; simply rolling back the transactions ensures that all the work is undone correctly, something that a manual cleanup operation might not do quite as thoroughly.

A manual rollback is also more brittle since it is more code to write and thus more code that could go wrong. Manual rollbacks becomes even trickier if we're testing multiple databases, and dealing with the hassles of ensuring that the databases are synchronized and the data is correctly cleaned up is too cumbersome for testing.

As with many of these approaches, there are disadvantages. Code that manipulates transactions or starts its own transactions cannot be tested this way without complicated nested transaction setups. For example, any code that calls `commit()` or `rollback()` should usually not be tested using this approach unless you're very clear on the semantics of what the code does and how having an external traction will impact its behavior.

Most applications will communicate with the database either through straight JDBC or through a `DataSource` implementation. The first approach involves manually working with `Driver` and `Connection` objects. Connections obtained through this mechanism are not transactional, so to prevent any writes to the database from taking place, our test would simply have to turn off autocommit, via the `Connection.setAutocommit(false)` method.

The other option is to perform database access through a `DataSource` object, which can integrate with a transaction manager and thus can be told to abort a transaction. We'll outline the specifics of this approach in Chapter 4.

Note that it is also important to ensure that the isolation level is set to `READ UNCOMMITTED`. Some databases (particularly embedded ones) have this as the default. The reason we need this is that we'd like to be able to verify some of the data we've attempted to write, and this isolation level allows us to read uncommitted data. Setting it to anything else means that we'd have to ensure that data is validated in the same transaction as it's being written, or else we'd never get to read it.

Having said that, it is important to understand the semantics of the isolation level we choose. It's very likely that in the production environment, a different isolation level is in place, and this subtle change in environments could result in some difficult-to-track bugs that do not manifest themselves in the test environment. Furthermore, this isolation level will cause issues when run in concurrent tests, as different tests might end up seeing the same data if there is just one transaction covering a particular set of tests.

Selecting the Right Strategy

For our component, we can go with disabling autocommit on the connection we obtain in the test. An embedded database is not an option since we rely on a database-specific stored procedure. So the test can expect to have a database that has been set up correctly to connect to.

Looking over the test code as we have it now, we currently obtain a database connection within the test method itself. The fact that we expect certain data to be available in the database opens us to the possibility of connecting to a database that does not have this data or, even worse, failing to connect to the database at all. In both cases, we don't get to test the business functionality, so we don't actually know if our business logic is correct or not.

In that case, our test will fail through not being tested, rather than through an explicit logic error. To distinguish between the two, another refactoring is called for.

We know that our test will be called multiple times, and we also know that it's fairly likely that we will end up with further tests that verify different aspects of our component, all of which are going to need access to the database. The database is an external dependency, so we model it accordingly in Listing 3–7, as part of the environment setup, rather than the test proper.

Listing 3–7 Extract database setup into configuration methods

```
private Connection connection;

@BeforeMethod
public void connect() throws SQLException {
  connection = DatabaseHelper.getConnection(...);
  connection.setAutoCommit(false);
}
```

```
@AfterMethod
public void rollback() throws SQLException {
  connection.rollback();
}
```

We've refactored the test to move the database connection handling into setup methods. The benefit of this approach is that if we do have an issue connecting to the database, we will get a more helpful error message that makes it clear that the failure is in setup and not in the tests themselves. We also ensure that the connection is rolled back after every test method invocation.

Of course, it might be desirable to perform a number of tests and then roll them back at the end. The rollback method can instead be marked with `@AfterClass` or `@AfterSuite`, depending on our needs.

An interesting problem we might face is that the code we're testing might explicitly call `commit`. How would we prevent the transaction from committing in this case?

To deal with this situation, we employ the Decorator pattern. We'll assume that the code has a connection provided to it. In Listing 3–8, we wrap the connection in a decorator that prevents calls to `commit` and pass that to the component instead of the real connection.

Listing 3–8 Disabling *`commit`* by using a wrapped connection

```
private WrappedConnection wrappedConnection;

@BeforeMethod
public void connect() throws SQLException {
  connection = DatabaseHelper.getConnection();
  connection.setAutoCommit(false);
  wrappedConnection = new WrappedConnection(connection);
  wrappedConnection.setSuppressCommit(true);
}
```

The `WrappedConnection` implementation is a decorator around the actual connection. It implements the `Connection` interface. The relevant parts are shown in Listing 3–9.

Listing 3–9 *`WrappedConnection`* implementation

```
public class WrappedConnection implements Connection {
  private Connection connection;
  private boolean suppressClose;
  private boolean suppressCommit;

  public WrappedConnection(Connection c) {
    this.connection = c;
  }

  public boolean isSuppressClose() {
    return suppressClose;
  }

  public void setSuppressClose(boolean suppressClose) {
    this.suppressClose = suppressClose;
  }

  public boolean isSuppressCommit() {
    return suppressCommit;
  }

  public void setSuppressCommit(boolean suppressCommit) {
    this.suppressCommit = suppressCommit;
  }

  public void commit() throws SQLException {
    if(!suppressCommit)
      connection.commit();
  }

  public void close() throws SQLException {
    if(!suppressClose)
      connection.close();
  }
  // rest of the methods all just delegate to the connection
  }
```

Using the wrapped connection now enables us to prevent any objects we use in our tests from calling `commit` or `close`, as needed.

Error Handling

At this point, we've achieved two of our stated goals, while reworking our code to ensure we don't pollute our tests with nongoals.

This test is valuable in that it successfully verifies that our component behaves the way we'd like it to, but an equally important part of testing is capturing boundary and error conditions. Invariably in the real world, things go wrong. They often go wrong in interesting and perplexing ways, and they often do so at fairly inconvenient times. What we'd like is to at least capture some of these failures and know what our code is going to do. It's fine if things blow up, as long as we know exactly what will blow up and how.

Of course, it's tempting to wrap the whole thing in a big `try/catch`, log the error, and forget about it. In fact, if we look at our component code, that's pretty much what it does. It's equally tempting to think that we can easily figure out all the failure points and account for them. Very, very few people can do this. It's important, in fact, not to get bogged down thinking of every possible thing that can go wrong and check for it. It's crucial that we remain pragmatic and practical and, at this point, handle only likely errors.

Our test will not capture everything that can go wrong. Things will go wrong over time that we did not anticipate. Some will be obvious, but others will be insidious and tricky. The crucial lesson in error handling then is to take back the feedback and results from a live run and feed them back into our tests. It's less important to have a comprehensive set of failure tests up front than it is to capture actual failures as they happen after the code has been deployed. The value of tests lies in their growth and evolution over time, not in the initial spurt, which in terms of the big picture is insignificant.

When capturing a bug that's found in production code, it's also important to label it correctly. The requirement that should always be satisfied is this: "If someone new joins the project six months after I leave, will he or she be able to look at this test case and know why it's here?" Comments in the test should include a link to the associated bug report. If that's not available, add a brief explanation of what functionality the test verifies beyond just the code.

So, what can go wrong with our component? The developer responsible for this component merrily yelled out "Nothing!" when asked. But he wasn't quite as accurate as we'd all hoped.

One interesting error that cropped up time and time again in the log files is the ubiquitous `NullPointerException`. On further investigation, it turns out that the processor extracts a currency code from the XML. It then looks up some rates associated with that currency. The problem? The currency wasn't listed in the database, hence a dazzling variety of long stack traces in the log files. No problem; the developer adds a check to verify that the currency is valid, and if not, to throw an exception.

Now that we have some tests in place, the first thing we need to do is model the failure before fixing it. Having an easy way to reproduce an error instead of clicking around a UI is a huge timesaver and is a very easy payoff for having gone to the bother of developing a testing strategy.

How do we model this failure? Thanks to our Data-Driven Testing, all we have to do is get the XML file with the invalid currency and drop it into our data directory. Running our test now will correctly show the `NullPointerException`.

We now have a reproducible error, and we know how to fix the code. The fix involves explicitly checking for invalid currencies and throwing an application-specific exception indicating invalid input data (e.g., `InvalidTradeException`). Putting that fix in shows that we correctly throw the exception, but, of course, our test will still fail since it does not expect this exception.

One option shown in Listing 3–10 is to catch the exception in the test.

Listing 3–10 Initial attempt to handle invalid data

```
@Test(dataProvider = "componentA-data-files")
public void componentAUpdateDatabase(String xml) throws Exception {
  ComponentA component = new ComponentA();
  try {
    component.processDocument(xml);
  }
  catch(InvalidTradeException e) {
    // this is OK
    return;
  }
  // rest of test code
}
```

As a side note, it's convenient to have tests declare that they throw an exception, as a reasonable catch-all mechanism for "anything that can go wrong." In production code, this is a bad idea, as it does not allow the caller to explicitly handle exceptions. Here we see yet another example of a pattern that is acceptable (and even recommended, for the sake of simplicity) in test code that should always be avoided in production code.

The problem with the approach is that it does not enable us to distinguish between the cases where that failure is expected and those where it isn't. Instead, what we should do is distinguish between expected successes and expected failures. The test as it stands can pass for two situations: It either passes when we have good data, or it passes when we have bad data. In either case, we can't make assertions about what actually happened; did we test the good data path or the bad data one? More importantly, what didn't we test?

The fact that the two paths happen to be "good data" and "bad data" in our case is a specific example. It's equally easy to accidentally write a test that has two or more conditions for passing, and we'd have the same issue with regard to what sort of assertions we can make about the success result.

The guideline to follow here is that a test shouldn't pass in more than one way. It's fine if the test verified different failure modes, but having one that can pass for both good and bad data is inviting subtle errors that are tricky and difficult to track down. We therefore define another directory and Data Provider in Listing 3–11 that handles failures, the same way as we do for valid input.

Listing 3–11 Defining a separate provider for invalid data

```
@DataProvider(name = "componentA-invalid-data-files")
  public Iterator<Object[]> loadInvalidXML() throws Exception {
    File dir = new File("samples/ComponentA/trades/invalid");
    File[] f = dir.listFiles();
    final Iterator<File> files = Arrays.asList(f).iterator();
    return new Iterator<Object[]>() {

      public boolean hasNext() {
        return files.hasNext();
      }

      public Object[] next() {
        return new Object[]{IOUtils.readFile(files.next())};
      }

      public void remove() {
        throw new UnsupportedOperationException();
      }
    };
  }
```

```
@Test(dataProvider = "componentA-invalid-data-files",
      expectedExceptions = InvalidTradeException.class)
public void componentAInvalidInput(String xml) throws Exception {
  ComponentA component = new ComponentA();
  component.processDocument(xml);
  // rest of test code
}
```

Here we defined another set of inputs with an associated test that always expects an exception to be thrown. If we do end up putting a valid XML file in the invalid `trades` directory, we will also correctly get a test failure since our `processDocument` method will not throw `InvalidTradeException` (since the input is valid).

Emerging Unit Tests

As we receive more bug reports from the QA team, our test sample grows steadily. Each sample will test a certain code branch. It won't be long, however, before we find that it's actually rather cumbersome to run the entire functional test to narrow an issue with our XML parsing.

The example of invalid data that we highlighted earlier demonstrates this perfectly. In this case, our first step in the functional test fails. We never even get to the database. Given that our problem is restricted to a small part of our functional test, it would be tremendously useful if we could isolate that one part and split it off into a unit test.

This highlights one of the important paths through which we can grow our unit tests; let them evolve naturally through functional tests. As we start to debug failures, unit tests will become more apparent as a quick and easy way to reproduce fast failures. This top-down approach is very useful in isolating bugs and in testing an existing code base. Once we're in the habit of testing, using the bottom-up approach of unit tests followed by functional tests is well suited when developing new functionality.

We have an important principle here. Unit tests do not necessarily have to be written before any other kind of test—they can be derived from functional tests. Particularly in large projects or an existing code base, writing useful unit tests at first can be tricky because, without an understanding of the bigger picture, they're likely to be too trivial or unimportant. They can instead be derived from meaningful functional tests intelligently, as the process of debugging and developing functional and integration tests will reveal their unit test components. Since functional tests are (hopefully!) derived from specifications and requirements, we know that they satisfy a core piece of functionality, whereas in a large project it might be difficult to immediately spot meaningful units that are relevant and in need of testing.

So, for our example, we will go through another round of refactoring. We need to split up the XML validation and database processing into separate methods so that we can invoke and test them separately.

Our component code now becomes something like Listing 3–12.

Listing 3–12 Refactored component to separate processing from validation

```
public void processDocument(String xml)
  throws InvalidDocumentException {
  Document doc = XMLHelper.parseDocument(xml);
  validateDocument(doc);
  // do DB work
}

public void validateDocument(Document doc)
  throws InvalidDocumentException {
  // perform constraint checks that can't be captured by XML
}
```

We separated the document validation from document processing so that we can test them separately. The upshot of this refactoring is that we now have a simple unit test that has very few (and more importantly, light and inexpensive) external dependencies that can be used to validate all our documents. This test does not require a database or much of an environment since all it does is look through our set of XML documents to ensure that any constraints that cannot be expressed via the document's DTD or schema are not violated.

Since we already have a good set of input XML, why not reuse it for our unit test, too? By the very nature of how we derived our unit test, we know that if it fails, the functional test will also fail. This is an important aspect of functional testing; a good functional test can be decomposed into a number of unit tests. And no matter what anyone tells you, the order in which you write them is not important at all. For new code, it's likely easier to start with unit tests and then develop functional tests that likely build on the rest of the unit tests. For existing code, the reverse is true. The principle remains the same in both cases.

Having said that, it is important to note that regardless of what order they're written in, functional and unit tests are complementary. A functional test is a more horizontal test that touches on many different components and exercises many portions of the code. A unit test, on the other hand, is more vertical in that it focuses on a narrow subject and tests far more exhaustively than a functional test would.

How do we express this relationship between functional tests and unit tests? We place them into logical groupings and explicitly specify the dependency. Putting these concepts together gives us the tests shown in Listing 3–13.

Listing 3–13 Dependency between unit and functional tests

```
@Test(dataProvider = "componentA-data-files", groups="unit-tests")
public void componentAValidateInput(String xml) throws Exception {
  ComponentA component = new ComponentA();
  component.validateDocument(XMLHelper.parseDocument(xml));
  // rest of test code
}

@Test(dataProvider = "componentA.xml", groups = "func-tests",
        dependsOnGroups = "unit-tests")
public void componentAUpdateDatabase(String xml) throws Exception {
  ComponentA component = new ComponentA();
  component.processDocument(xml);
  // rest of test code
}
```

Here we have two tests, one unit and one functional, both belonging to their respective groups, and the dependency between them is explicitly specified. Our test engine will ensure that they are run in the correct order. We can also use the same approach we did for our functional test to add a unit test that verifies that invalid inputs fail correctly as well.

Coping with In-Container Components

One thing we assumed in our test is that the component to be tested can be instantiated easily. Unfortunately, most code out in the real world isn't quite blessed with that convenience. In many cases, the component we dissected earlier would be a Message-Driven Bean (MDB), which runs in an application server. It could also be a servlet or any other managed component that expects a specific environment and cannot be easily instantiated.

We use the term *in-container* to denote that the code needs to be run and deployed into a container and so requires an expensive and heavy environment.

Obviously, this makes testing much trickier and more difficult, and a recurring theme of frameworks like Spring is to always try to abstract away the container dependency, to promote better reuse and code testability.

So, how do we test components in such situations? The answer lies in the same way we managed to get rid of the JMS dependency in our component test. The trick is to refactor the component so that its business functionality is isolated from its environment. The environment is either handled by an external class or injected into the component. For example, if our component were an MDB, we would have gone through the same approach as we did earlier to get rid of JMS. If it were a servlet, we would have used a delegate.

This is not to say that we should always avoid tests that have external dependencies or need to run inside of a server. Such tests are complimentary to our other unit and functional tests. For example, we do need a test at some point to verify that we don't have a typo in the code that reads a JMS message property, and such a test cannot be done without JMS in place.

The Delegate pattern means that the functionality of the component would have been moved away from the servlet class itself into a POJO that can be easily instantiated and tested. The servlet would act as a delegate, and all it would do is ensure that the actual component receives the correct environment and settings based on the request the servlet receives.

Having said that, components are sometimes more intimately tied to their environments and APIs. While it is possible to modify code that, for example, relies on JTA and JNDI, it might be more convenient to simulate that environment in our tests to minimize the testing impact on the code being tested. Chapter 4 will go through a number of Java EE APIs and outline approaches for simulating the correct test environment for them.

Another option that is worth considering is using an in-container test. We will explore this further Chapter 4. The main concept is that the test engine is embedded in the application server, so we can invoke tests that run in the actual deployment environment and interact with the results remotely.

Putting It All Together

We started out with a transformation component that was written without taking testing into consideration at all. It consisted of a monolithic method where a number of concerns and concepts were intertwined, without any clear separation of responsibility. More informally, it was what's often called messy code.

When we tried to test this code, we immediately ran into hurdles. The test encouraged us to shuffle things about a bit in the component itself to