CHAPTER 10

Testing

During the course of this book, you have seen how Spring helps you build robust Java applications. This chapter covers testing your applications. We start by discussing two important testing concepts, unit testing and integration testing (also called functional testing), explaining the difference between them. We also talk about test-driven development as a way to develop your software.

Then we introduce JUnit and EasyMock, two frameworks that are the de facto standard when it comes to testing Java applications. We will show you how to use the power of both frameworks to extensively test your application.

Of course, this chapter would not fit in this book if it did not expand on some of the testing features provided by Spring. You will learn how to test your Spring configuration files and the configuration of the objects created by the Spring container. Using the information in this chapter, you will be able to test the configuration of your web applications.

Introducing Testing Approaches

Testing is often considered to be the responsibility of a separate department within a software development organization. This quality assurance (QA) department will take all written code and test it against certain requirements set forth by the client. This form of testing, often referred to as *system testing*, is an important step in developing high-quality software applications. However, software testing should start with the developers. They are the front line when it comes to ensuring the quality of their software prior to handing it off to QA. This chapter focuses on testing performed by developers.

Generally speaking, testing should be used to ensure the following aspects of an application:

Correctness: You want to ensure the correctness of your application. For example, suppose that you have written a `calculate()` method on a `Calculator` class. You want to make sure that certain input for this method results in a correct calculation result.

Completeness: Testing can be used to ensure that your application is complete by verifying that all required operations have been executed. Suppose you have a signup process that includes creating an invoice for newly signed-up members. You want to test whether a member is actually added to the database, and also if an invoice has been created for that user.

Quality: Testing can ensure the quality of your application, and this goes beyond software-quality metrics. A well-tested piece of software creates confidence with developers. When existing code needs to be changed, it's less likely that developers will be afraid of unintentionally breaking the software or reintroducing bugs.

Another major advantage of testing is that it provides you with a harness for your code. This harness enables you to refactor your code at a later stage with more confidence.

As you have probably experienced, most code is not perfect when it is originally written. In many cases, you will revisit the code a couple of times during the development and rewrite parts of it, either to fit new requirements or just to improve its quality.

For example, suppose that when the previously mentioned `Calculator` class is implemented, we have written tests to assert that certain results are returned on certain input. Now let's assume we revisit the code after a couple of weeks and decide our initial implementation needs to be improved. We decide to rewrite our calculation algorithm to calculate the result. Because we have written a set of tests, we will know whether our new implementation works as well as the previous implementation. We can assert this by running the tests we wrote.

Testing efforts can be split into two separate ways of testing applications. Specific parts of an application, called *units*. can be tested in isolation. Those units can also be tested together to assert that they work together as expected. These types are referred to as unit testing and integration testing, respectively.

Unit Testing

A unit in the context of unit testing depicts *a specific piece of functionality* in your code, usually residing in a specific method or constructor in a specific class. Unit testing is the microscale of testing and is typically done by developers who know how such a unit should function.

As you have learned by reading this book, the key to flexible applications is abstractions. As such, it's important to abstract Java code from the environment in which it will operate. We say that this code is unaware of its environment, unless it chooses to become aware of it in a minority of cases.

This so-called plain old Java object (POJO) approach to Java coding, combined with defining interfaces for important parts of your application (see Chapter 8), provides the basis for thorough testing. The Spring Framework promotes exactly this approach (we could also say the Spring Framework makes this approach possible). By separating your code into well-defined interfaces and objects, you have already defined the units that are eligible for unit testing.

One goal of unit testing is to ensure that each unit of an application functions correctly in isolation. Another goal is to define a *framework*, *harness*, or *contract* (all referring to a strict set of rules that must be respected) that must be satisfied by the unit test. As long as the tests can be run successfully, the unit is considered to work properly. (If there are bugs in the test code, the unit will function properly according to this buggy test code.)

Unit testing offers a number of benefits for developers:

Facilitate change: As previously mentioned, having a set of unit tests for a specific piece of code provides confidence in refactoring existing code. The unit tests will ensure the module continues to function correctly according to the available tests as long as the tests succeed. Given there are enough tests for all the code in the application, this promotes and facilitates changing implementation details of units in the application. An important aspect of facilitating change is preventing solved problems or bugs from reentering the code.

Simplify integration: Unit testing provides a bottom-up testing approach, which ensures that low-level units function properly according to their tests. This makes it easier to write integration tests at a later stage. There is no need to have two tests for units. In integration testing, described in the next section, the *interaction* between units can be tested, rather than individual units. This makes integration tests much easier to write because their scope is limited.

Promote well-defined separation: In order for you to be able to completely and efficiently write unit tests, your code needs to be separated into well-defined units. Each unit needs to be tested in isolation and should therefore allow the replacement of dependencies with test-specific ones. Thus, writing unit tests promotes the separation of your application into well-defined units.

Integration Testing

As opposed to unit testing, integration testing is the testing of a number of units and how they work together. Integration testing obviously spans the boundaries of a unit, but can also span across the boundaries of your code. Often, integration testing also includes testing of integration with external systems, such as databases or legacy systems.

Integration testing is just as important as unit testing. While unit testing ensures that all units work properly according to their tests in isolation, integration testing ensures they collaborate as expected.

For example, suppose you have a repository adapter implementation using JDBC to modify the database. Such a class is very hard to unit test, since it's difficult to isolate the data-access code from the database. With integration tests, you can assert the data-access configuration of the application and whether the data-access code and SQL statements are valid for the target database. This is where integration testing comes in. This chapter will show you how Spring helps you to create sophisticated integration tests.

Test-Driven Development

Test-driven development (TDD) is a way of implementing code by writing a unit test before implementing the actual class. This will ensure that you first think about what the implementation should do—what contract it should fufill. This manner of implementing code is highly emphasized in Extreme Programming. TDD defines the following development cycle:

Write the test: You should always begin with writing a test. In order to be able to write a test, you should have a clear understanding of the specification and the requirements. In order to get those specifications and requirements, you first need to ask questions and get answers. (In Extreme Programming, this question-and-answer interaction with the user translates into use cases and stories.)

Write the code: Next, you should write the code that will make the test pass. The implementation is finished only when the test passes. At this stage, you should look back at your test code to ensure it tests what you are actually trying to implement. It's not uncommon to get better insights in the implementation you are trying to create, which results in correcting or improving your test.

Run the automated tests: The next step is to run the automated test cases and observe if they pass or fail. If they pass, you can be confident that the code meets the test cases as written. If there are failures, the code did not meet the test cases and you should continue to the next step.

■ **Note** It's important to make sure the test fails before the implementation has been written. Otherwise, from a psychological perspective, there would be no difference between writing the test and writing a correct implementation. In other words, a test that only succeeds if the implementation is correct is a reward for the developer.

Refactor: The final step is the refactoring step. This step involves improving the code that actually passed the test. Because you know the code already fulfills the contract, you can refactor here with confidence. The tests are then rerun and observed. Refactoring can happen at any time. It does not need to happen immediately after the first implementation is written, although developers may be more focused at that time.

Repeat: After completing the refactor step, the cycle will then repeat, starting with either adding functionality or fixing any errors.

Using TDD has a number of benefits. First, the tests you start off creating act as the first user of your code. Secondly, it can help you write modular software. Next, it also provides some sort of documentation for your software. Assuming you have written a test for a specific unit, another developer can check the test in order to see how the unit is supposed to be used and works. And lastly, TDD promotes refactoring, on the one hand because it is part of the development cycle, and also because it provides developers with the confidence that is required to refactor code in the first place.

For more information about TDD, see *Test-Driven Development: A J2EE Example* (Apress, 2004).

UNIT TESTS AS DOCUMENTATION?

It's debatable whether unit tests can ever provide all the documentation required for an application. Moreover, writing tests to test application code *and* to serve as documentation unavoidably overloads unit tests with responsibilities. This overload may not always be apparent. However, we've seen cases where unit tests could not be altered to become more efficient in the way they tested the software because developers feared they would not document the code anymore.

We believe that "unit tests that serve as documentation" is a misleading approach to unit testing and that many developers take it too seriously. We can understand that developers welcome any approach that relieves them from having to write documentation. We also believe that this argument has helped to sell TDD books in the past (and probably still does today). But as far as we are concerned, this argument has no strong foundation. Good documentation is much more beneficial to the consumer of an application. And it requires developers to profoundly think about the approaches they are taking.

Writing Unit Tests Using JUnit

JUnit is the de facto standard for unit testing of your Java applications. It is an automated testing framework that allows you to easily create tests for your Java classes. JUnit provides the most commonly used functionality for building robust unit tests. Another advantage of using JUnit is that most IDEs provide support for running JUnit tests. Also, JUnit is supported out of the box by build tools like Ant (`http://ant.apache.org`) and Maven (`http://maven.apache.org`).

In this section, we will use JUnit to build a test for a unit of our tennis club application. First, we will need to develop the functionality to meet the requirements.

Establishing the Requirements

We need to calculate the membership fee per member based on a number of factors. These factors can change over time, but currently we use these rules to determine the rate:

- Members younger than 14 years old pay $25 for three months or $90 for a full year membership.
- Members between 14 years and less than 18 years old and members over 50 years old pay $35 for three months or $126 for a full year membership.
- Members over 18 years and less than 50 years old pay $50 for three months or $180 for a full year membership.
- A member's age is his or her age at the date of the invoice.
- All members get a 25% reduction on all the membership rates if they have a membership with the national tennis federation.

We could create an interface that is responsible for calculating the membership fee, like this:

```
package  com.apress.springbook.chapter10;

public interface MembershipFeeCalculator {
  double calculateMembershipFee(Member member);
}
```

The `MembershipFeeCalculator` interface looks adequate for other classes in the sample application to use whenever they need to calculate the membership fee for a member. However, some issues make this interface less than acceptable.

First of all, for now we only need to know the age of a member, the member's payment preference, and whether the member has a membership with the national tennis federation. For this data, there are getter methods on the `Member` class. However, it seems like the `Member` class is too generic for this calculation since it has many other getter methods. Hence, the `MembershipFeeCalculator` doesn't clearly communicate how membership fees are calculated. This may sound like a good abstraction, since callers should not care about how the membership fee is calculated, but if we take a closer look, we'll find that other classes can calculate a membership fee only for the current age of a member. The `MembershipFeeCalculator` can't calculate future membership fees. It also cannot calculate fees for prospective members, since they do not have a `Member` object.

■ **Note** We could create a dummy `Member` object, but that goes against the nature of the `Member` class. `Member` represents an active or past member of the tennis club. Using a concrete class for multiple purposes will unavoidably lead to problems and bugs, since developers will need to take all the roles of a class into account when they change it. For the `Member` class, these roles would be real members and prospective members. As you can imagine, a real member has many more properties than a prospective member.

We clearly need a better abstraction mechanism to calculate membership fees. Here is a reworked `MembershipFeeCalculator`:

```
package  com.apress.springbook.chapter10;

public interface MembershipFeeCalculator {
  double calculateMembershipFee(
    int age,
    boolean perTrimesterNotPerAnnum,
    boolean tennisFederationMembership);
}
```

This `MembershipFeeCalculator` is much more specific than the previous version, but unfortunately, it is too specific. Callers now need to know about all the gory details to calculate a membership fee. They must get a member's age, whether the member wants to pay per trimester or per annum, and whether the member has a valid national tennis federation membership at the time of the calculation. This may be exactly what's needed for some callers, but is likely to be painful, complicated, and not cost-effective for other callers. This is especially true if the requirements for calculating the membership fee change. All callers would suddenly have to get other data, and this change would badly affect the application. So this second version of the `MembershipFeeCalculator` is not a good abstraction for our purposes.

Here is yet another reworked version of the `MembershipFeeCalculator`:

```
package com.apress.springbook.chapter10;

public interface MembershipFeeCalculator {
  double calculateMembershipFee(PayingMember payingMember);
}
```

This MembershipFeeCalculator version uses the PayingMember interface, which is shown in Listing 10-1.

Listing 10-1. *The PayingMember Interface*

```
package com.apress.springbook.chapter10;

public interface PayingMember {
  int getAge();
  boolean isPaymentPerTrimester();
  boolean isMemberOfNationalTennisFederation();
}
```

One possible next step could be to let the Member class implement the PayingMember interface. Another option is to create an adapter class for Member objects, as shown in Listing 10-2.

Listing 10-2. *PayingMember Adapter Class for Member Objects*

```
package com.apress.springbook.chapter10;

import java.util.Date;

public class PayingMemberAdapterForMember {
  private Member member;
  private Date calculationDate;

  public PayingMemberAdapterForMember(Member member, Date calculationDate) {
    this.member = member;
    this.calculationDate = calculationDate;
  }
  public int getAge() {
    // calculate age at calculation date based on member
    // birth date.
  }
  public boolean isPaymentPerTrimester() {
    // get payment option for member and return
  }
  boolean isMemberOfNationalTennisFederation() {
    // get membership details of member at the calculation date
  }
}
```

We can now create as many implementation classes of the PayingMember interface as are required. One interesting advantage of this adapter approach is that callers of the MembershipFee Calculator interface no longer need to know exactly which details are required to perform the calculation. They just need to pass an instance of the PayingMember interface as they see fit.

If the fee calculation changes, and with it the required data changes, we just need to refactor the PayingMember interface and its implementation classes. This may affect callers of the MembershipFeeCalculator interface, but it will always be possible to keep this impact limited yet maintain flexibility. Membership fee calculation will never change in such a way that our approach would become inflexible.

In order to be able to run the tests at a later stage, we also want to create a default implementation of the MembershipFeeCalculator interface. It works best if we start off by creating a default implementation that just throws an UnsupportedOperationException for all method invocations. This allows us to run the tests and get an indication of which methods have not yet been implemented. Listing 10-3 shows the initial implementation of the MembershipFeeCalculator interface.

Listing 10-3. *Initial Default Implementation of the MembershipFeeCalculator Interface*

```
package com.apress.springbook.chapter10;

public class RegularMembershipFeeCalculator implements MembershipFeeCalculator {
  public double calculateMembershipFee (PayingMember member) {
    throw new UnsupportedOperationException("not implemented yet!");
  }
}
```

SHOULD WE ACT DUMB?

Extreme Programming and some related methodologies do not encourage you to deliberate on your interfaces during the first iteration. You are encouraged to go through this kind of deliberation only when the need to find a better abstraction becomes imminent. We can't agree with such a dogma.

We believe that encouraging developers to act "dumb" when it comes to creating flexible applications is the wrong approach. We do agree that very often requirements can't be known in advance, but not all requirements are equal. Some requirements, like calculating a membership fee, are sufficiently limited in scope. It's fairly straightforward to come up with working abstractions, and it's easy to prove how they can adapt to potential changes. As such, it's also easy to prove these abstractions can save time and money, and many small profits can make a big whole.

If you encounter comparable cases where some brainstorming can lead to good abstractions, we advise you to embrace the flexibility of your application and go for it. There's a risk in acting dumb, in that you may not be able to improve your applications if there's no urgent need.

Writing the Test

Taking the TDD approach, we now want to create our test to define the contract for our `MembershipFeeCalculator` implementation. We're ready to write a unit test. Listing 10-4 shows the `RegularMembershipFeeCalculatorTests` class.

Listing 10-4. *RegularMembershipFeeCalculatorTests Class*

```
package com.apress.springbook.chapter10;

import junit.framework.TestCase;

public class RegularMembershipFeeCalculatorTests
     extends TestCase {

  public void testLessThan14YearsOldPerTrimesterNoNTFMember() {
    MembershipFeeCalculator mfc = new RegularMembershipFeeCalculator();

    PayingMember payingMember = new TestPayingMember(13, true, false);

    double result = mfc.calculateMembershipFee(payingMember);

    assertEquals((double)25, result);
  }
```

```
  public void testLessThan14YearsOldPerAnnumNoNTFMember() {
    MembershipFeeCalculator mfc = new RegularMembershipFeeCalculator();

    PayingMember payingMember = new TestPayingMember(13, false, false);

    double result = mfc.calculateMembershipFee(payingMember);

    assertEquals((double)90, result);
  }

  public void testLessThan14YearsOldPerTrimesterNTFMember() {
    MembershipFeeCalculator mfc = new RegularMembershipFeeCalculator();

    PayingMember payingMember = new TestPayingMember(13, true, true);

    double result = mfc.calculateMembershipFee(payingMember);

    assertEquals((double)25 * 0.75, result);
  }

  public void testLessThan14YearsOldPerAnnumNTFMember() {
    MembershipFeeCalculator mfc = new RegularMembershipFeeCalculator();

    PayingMember payingMember = new TestPayingMember(13, false, true);

    double result = mfc.calculateMembershipFee(payingMember);

    assertEquals((double)90 * 0.75, result);
  }

  private class TestPayingMember implements PayingMember {
    private int age;
    private boolean paymentPerTrimester;
    private boolean memberOfNationalTennisFederation;

    private TestPayingMember(
            int age,
            boolean paymentPerTrimester,
            boolean memberOfNationalTennisFederation) {
      this.age = age;
      this.paymentPerTrimester = paymentPerTrimester;
      this.memberOfNationalTennisFederation = memberOfNationalTennisFederation;
    }

    public int getAge() {
      return age;
    }

    public boolean isPaymentPerTrimester() {
      return paymentPerTrimester;
    }

    public boolean isMemberOfNationalTennisFederation() {
      return memberOfNationalTennisFederation;
    }
  }
}
```

Notice that we used the name of the class we're testing and appended `Tests` to it. This is not a requirement for JUnit, but is considered a best practice, and the `Tests` extension is often used to determine which classes should be executed as tests. Furthermore, our test extends the JUnit `TestCase` base class. This class provides the base functionality for building tests.

If you take a closer look at the `test*()` methods, you should notice the use of the `assertEquals()` method, which is made available by the `TestCase` superclass. This method does exactly what its name suggests: it asserts whether the first and second argument are equal.

`TestCase` provides many methods for you to assert results and influence the outcome of the test. Table 10-1 lists the most commonly used methods of the JUnit `TestCase` base class. Each of those methods also has a version that allows you to specify a message for when the test fails, which might help when you're debugging failing test methods.

Table 10-1. *Commonly Used Methods of the JUnit TestCase Base Class*

Method	Description
`assertEquals`	Asserts that two objects are equal. Along with checking two `Object` instances for equality, a number of convenience versions exist. These take as input primitives or regularly used objects, such as `long`, `int`, `double`, `String`, and so on.
`assertTrue`	Asserts that a Boolean condition evaluates to `true`.
`assertFalse`	Asserts that a Boolean condition evaluates to `false`.
`assertNull`	Asserts that an object is `null`.
`assertNotNull`	Asserts that an object is not `null`.
`assertSame`	Asserts that two objects refer to the same instance.
`assertNotSame`	Asserts that two objects do not refer to the same instance.
`fail`	Explicitly makes a test fail.

Definining a Test Suite

JUnit normally executes all methods that contain `test` as part of the name. You can override this behavior, as well as temporarily disable it or exclude certain test methods, by defining a *test suite*. Using JUnit, this is fairly simple—you just need to add a static method to your test class called `suite`, which returns a `TestSuite` instance.

Listing 10-5 shows a sample test suite, which makes sure only the first three test methods are executed.

Listing 10-5. *A Sample Test Suite Definition for the RegularMembershipFeeCalculatorTests Class*

```
public static TestSuite suite() {
  TestSuite suite = new TestSuite();
  suite.addTest(new RegularMembershipFeeCalculatorTests(
    "testLessThan14YearsOldPerTrimesterNoNTFMember"
  ));
  suite.addTest(new RegularMembershipFeeCalculatorTests(
    "testLessThan14YearsOldPerAnnumNoNTFMember"
  ));
  suite.addTest(new RegularMembershipFeeCalculatorTests(
    "testLessThan14YearsOldPerTrimesterNTFMember"
  ));
  return suite;
}
```

Now that we have created fully functional tests, we can run them (for instance, in an IDE). Of course, the tests will all fail because they are not yet implemented.

Next, we need to finish the `RegularMembershipFeeCalculator` class, as shown in Listing 10-6.

Listing 10-6. *The Implementation of the RegularMembershipFeeCalculator Class*

```
package com.apress.springbook.chapter10;

public class RegularMembershipFeeCalculator implements MembershipFeeCalculator {
  public double calculateMembershipFee (PayingMember member) {
    double fee = 0;

    int age = member.getAge();

    // Member is under 14
    if (age < 14) {
      if (member.isPaymentPerTrimester()) {
        fee = 25;
      } else {
        fee = 90;
      }
    }

    // Member is a between 14 and 18, or is over 50
    if ((age >= 14 && age < 18) || age > 50) {
      if (member.isPaymentPerTrimester()) {
        fee = 35;
      } else {
        fee = 126;
      }
    }

    // Member is between 18 and 50
    if (age >= 18 && age <= 50) {
      if (member.isPaymentPerTrimester()) {
        fee = 50;
      } else {
        fee = 180;
      }
    }

    // Reduce the fee if they are a member of the national federation
    if (member.isMemberOfNationalTennisFederation()) {
      fee = fee * 0.75;
    }

    return fee;
  }
}
```

You can find more information about JUnit at the JUnit website (`http://www.junit.org`).

Creating Mock Implementations with EasyMock

The previous section demonstrated how to write unit tests for your classes using JUnit. For the membership calculator example, this works fine, but now let's consider testing a class with a number of dependencies. Suppose we have a currency converter object that converts amounts in one currency to the appropriate amounts in another currency. In this case, the currency converter class has a dependency: an exchange rate service that can be queried for the current exchange rates. We'll use this example to demonstrate the use of EasyMock.

EasyMock (`http://www.easymock.org/`) is an open source library that allows you to easily create mock implementations of interfaces (or even classes for that matter). You can use EasyMock to dynamically create mock objects for interfaces of dependencies (also called *collaborators*). EasyMock is not the only mock object framework for Java, but it is considered one of the best.

Remember that unit testing is defined as testing classes or methods in isolation. Using the approach of unit testing described in the previous section does not work when testing classes that depend on collaborators. In order to test a class that depends on collaborators, you need to be able to eliminate the testing of the collaborators (they are tested by their own unit tests). This is where mock objects come in. A *mock object* is a dummy interface or class where you define the dummy output for a certain method call. In the case of the currency converter, we need to define a mock version of the exchange rate service to be able to truly unit test the converter. You could create mock objects by hand for each collaborator and have them return certain values. However, using a library for this makes your life much easier and gives you some added functionality, as you will see in this example.

Defining and Implementing the Interface

For this example, we will use the interface for the currency converter shown in Listing 10-7 and the default implementation for it shown in Listing 10-8.

Listing 10-7. *The CurrencyConverter Interface*

```
package com.apress.springbook.chapter10;

public interface CurrencyConverter {
  double convert(double amount, String fromCurrency, String toCurrency)
    throws UnknownCurrencyException;
}
```

Listing 10-8. *The Default Implementation of the CurrencyConverter Interface*

```
package com.apress.springbook.chapter10;

public class DefaultCurrencyConverter implements CurrencyConverter {
  private ExchangeRateService exchangeRateService;

  public void setExchangeRateService(ExchangeRateService exchangeRateService) {
    this.exchangeRateService = exchangeRateService;
  }

  public double convert(double amount, String fromCurrency, String toCurrency)
   throws UnknownCurrencyException {
    // get the current exchange rate for the specified currencies
```

```
    double exchangeRate =
      exchangeRateService.getExchangeRate(fromCurrency, toCurrency);
    // return the amount multiplied with the exchange rate
    return amount * exchangeRate;
  }
}
```

The default implementation of the `CurrencyConverter` has a collaborator defined by an interface, `ExchangeRateService`, shown in Listing 10-9.

Listing 10-9. *The ExchangeRateService Interface*

```
package com.apress.springbook.chapter10;

public interface ExchangeRateService {
  double getExchangeRate(String fromCurrency, String toCurrency)
   throws UnknownCurrencyException;
}
```

Notice the `getExchangeRate()` method, which takes the two currencies as arguments and returns the exchange rate as a `double`. This is the method that is used by our default implementation of the `CurrencyConverter` interface.

Creating a Mock Object

Now we want to unit test our currency converter. We would probably start out the same as we did in the previous section, by defining a test class and implementing a number of test methods on it. Listing 10-10 shows the test skeleton using EasyMock to create a mock object for the collaborators of the converter.

Listing 10-10. *The Test Skeleton for the DefaultCurrencyConverter Class*

```
package com.apress.springbook.chapter10;

import junit.framework.TestCase;

import org.easymock.EasyMock;

public class DefaultCurrencyConverterTests extends TestCase {
   private DefaultCurrencyConverter converter;

   private ExchangeRateService exchangeRateService;

   public DefaultCurrencyConverterTests(String name) {
      super(name);
   }

   protected void setUp() throws Exception {
      converter = new DefaultCurrencyConverter();

      exchangeRateService = EasyMock.createMock(ExchangeRateService.class);
      converter.setExchangeRateService(exchangeRateService);
   }

   // tests go here ...
}
```

Note that the `setUp()` method is overridden in order to create and assign the class under testing. However, the converter relies on an implementation of the `ExchangeRateService` interface. We could also instantiate and assign an actual implementation for this interface, but that would mean that our test would also test the inner workings of that class. This would result in this test no longer being a real unit test, as unit tests are supposed to test units in isolation.

Instead of using an actual implementation of the collaborator interface, Listing 10-10 uses EasyMock to dynamically create a mock object for the interface. As you can see, it takes only one line of code to create a mock object for an interface. We use the static `createMock()` method on the `EasyMock` class and provide it with the interface for which we want it to create a mock object. The return value is a mock implementation of the provided interface, which we can directly set as a collaborator on the converter instance.

■ **Note** EasyMock also supports the creation of mock objects for existing classes instead of just interfaces by using the EasyMock class extensions.

Testing with EasyMock

As mentioned earlier, using EasyMock offers much more functionality than just creating dynamic mock objects. It also allows you to test which methods are called on the collaborators by the class under testing. You can specify which methods you expect to be called by your implementation. For methods that have a return value, you can also specify which value the method call should return. It is also possible to have the method throw an exception in order to verify the exception handling of the class under testing.

Listing 10-11 shows a simple test method of the `DefaultCurrencyConverterTests`, which tests the `convert()` method with valid input and asserts the result.

Listing 10-11. *A Simple Test Method of the DefaultCurrencyConverterTests Class*

```
public void testConvertWithValidInput() throws UnknownCurrencyException {
  EasyMock.expect(exchangeRateService.getExchangeRate("EUR", "USD"))
          .andReturn(1.2).times(2);

  EasyMock.replay(exchangeRateService);

  assertEquals(12.0, converter.convert(10.0, "EUR", "USD"));
  assertEquals(24.0, converter.convert(20.0, "EUR", "USD"));

  EasyMock.verify(exchangeRateService);
}
```

The first line of the test method tells EasyMock to expect a method call to the `getExchangeRate()` method on the mock `ExchangeRateService` with `"EUR"` and `"USD"` as arguments. By calling the `andReturn()` method, you can specify the value the mock object should return. The last part of the first line of code specifies the number of times we expect this method to be called. In this case, it expects the method to be called twice because the `convert()` method is called twice with different amounts, so the `getExchangeRate()` should also be called twice: once for each `convert()` invocation.

EasyMock mock objects have state: they are either in record or replay mode. When they are first created, they are in record mode. This allows you to specify the expected behavior of the mock object as just demonstrated. However, in order to use the mock object for testing, you need to inform EasyMock that the specified behavior should be replayed. You do this by calling the

static replay() method on the EasyMock class with the mock object to replay as an argument. If you do not put the mock object in replay mode, EasyMock will not replay the behavior and will not inform you of unexpected method calls.

After making sure the mock object is in replay mode, you test the class. In this case, the convert() method is called twice, and the result is asserted to be correct. Note that the correct result is based on the return value that was specified for the exchange rate service. If the return value specified in the first line of code were changed to, for instance, 1.3, the correct return values for assertion would be 13.0 and 26.0, respectively.

Finally, you should inform EasyMock that testing has finished. This is done by calling the static verify() method on the EasyMock class, again with the mock object as the argument. Informing EasyMock that testing has finished is not required, but it gives you the added value that EasyMock will verify that all specified expected method calls have actually been called. So if the test in Listing 10-11 calls the convert() method just once, the verify() method will throw an exception, because the expected method call is done only once.

You also will want to have your tests check whether exception handling is correctly implemented. In this case, we want to make sure that the UnknownCurrencyException thrown by the exchange rate service is correctly handled (in this example, just thrown and not handled by the implementation). To have the mock object throw an exception, you can just specify which exception to throw instead of specifying a return value, as demonstrated in Listing 10-12.

Listing 10-12. *Using EasyMock for Testing Exception Handling*

```
public void testConvertWithUnknownCurrency() throws UnknownCurrencyException {
  EasyMock.expect(exchangeRateService.getExchangeRate(
         (String)EasyMock.isA(String.class), (String)EasyMock.isA(String.class)))
         .andThrow(new UnknownCurrencyException()).times(2);

  EasyMock.replay(exchangeRateService);

  try {
    converter.convert(10.0, "EUR", "-UNKNOWN-");
    fail("an unknown currency exception was expected");
  } catch (UnknownCurrencyException e) {
    // do nothing, was expected
  }

  try {
    converter.convert(10.0, "-UNKNOWN-", "EUR");
    fail("an unknown currency exception was expected");
  } catch (UnknownCurrencyException e) {
    // do nothing, was expected
  }

  EasyMock.verify(exchangeRateService);
}
```

First, notice the replacement of the andReturn() invocation with the andThrow() invocation, which has an instance of the UnknownCurrencyException as an argument. The rest of the test is much of the same, except that this test method tests for the exceptions actually being thrown.

Also notice that the currency arguments are switched for the second convert() invocation in order to fully test the method. In the previous test method, the currency arguments were fixed. This allowed us to specify expected arguments to the expected method call. In this case, however, we can't specify arguments, because the value is different for both invocations. To help us, EasyMock allows for an argument matcher to be specified for each argument. You can either implement your

own argument matcher or use one of the default argument matchers provided by EasyMock, which are listed in Table 10-2. Note that when using argument matchers, you need to use an argument matcher for all arguments.

Table 10-2. *Argument Matchers Provided by EasyMock*

Argument Matcher	Description
`eq(X value)`	Matches if the actual value is equal to the expected value. Available for all primitive types and for objects.
`anyBoolean()`, `anyByte()`, `anyChar()`, `anyDouble()`, `anyFloat()`, `anyInt()`, `anyLong()`, `anyObject()`, `anyShort()`	Match any value of a specific type. Available for all primitive types and for objects.
`eq(X value, X delta)`	Matches if the actual value is equal to the given value allowing the given delta. Available for `float` and `double`.
`aryEq(X value)`	Matches if the actual value is equal to the given value according to `Arrays.equals()`. Available for primitive and object arrays.
`isNull()`	Matches if the actual value is `null`. Available for objects.
`notNull()`	Matches if the actual value is not `null`. Available for objects.
`same(X value)`	Matches if the actual value is the same as the given value. Available for objects.
`isA(Class clazz)`	Matches if the actual value is an instance of the given class, or if it is an instance of a class that extends or implements the given class. Available for objects.
`lt(X value)`, `leq(X value)`, `geq(X value)`, `gt(X value)`	Match if the actual value is less than/less or equal/greater or equal/greater than the given value. Available for all numeric primitive types.

To review, the most important aspect of EasyMock is that it dynamically generates mock objects for your interfaces. In addition, EasyMock allows you to verify that only expected methods are called, that they are actually being called, the number of times they are called, and the arguments that are provided to the methods. Also, EasyMock makes it easy to have the methods return values or throw exceptions without having to code all that yourself. For more information, about EasyMock see the EasyMock project page (`http://www.easymock.org/`).

Using Spring Support for Integration Testing

When working with Spring for building your applications, you will typically use one or more XML configuration files for defining your application context. These configuration files are not Java files and will therefore not be compiled. Of course, if you include the Spring DTD or use Spring's namespace support, some aspects of your configuration files will be validated. But issues such as defining a nonexisting class as the class for a bean in your application context or setting a nonexisting property on a bean definition are discovered only when you actually load the application context at runtime. This is where integration testing comes into the picture.

The goal of integration testing is to test how the individually tested units of your application collaborate with each other. When working with Spring, you wire those dependencies together using Spring's configuration files. In order to test part of your whole application, you typically want to load the Spring application context and test one or more beans configured in that application

context. Fortunately, Spring provides convenient support for this. In order to use these features, you need to add `spring-mock.jar` to your classpath.

Spring provides three convenient test base classes for integration testing of your Spring applications: `AbstractDependencyInjectionSpringContextTests`, `AbstractTransactionalSpringContextTests`, and `AbstractTransactionalDataSourceSpringContextTests`. These classes are discussed in the following sections.

Testing Without Transactions

The `org.springframework.test.AbstractDependencyInjectionSpringContextTests` base class is the test class you will typically use when testing parts of your application that do not require access to a database or any other transactional support. You should extend this class by first implementing the `getConfigLocations()` method, which should return an array of application context locations to be loaded by the test. When the test is executed, the specified configurations will be loaded as an application context.

The major advantage of using this base class is that the application context will be loaded only once for each test method. If you were to load the application yourself in the `setUp()` method of a test, the application context would be reloaded for every test method. This is especially useful when loading configurations that require a lot of initialization, such as a Hibernate session factory. Another advantage of using this base class is that you can define fields for this test that are populated automatically by Spring based on your application context.

Assume we have a Spring configuration file that defines the default implementation of the currency converter we created earlier. The configuration file, named `applicationContext.xml`, also contains all required dependencies—in this case, only the exchange rate service. We can just add a field of type `CurrencyConverter`, and it will get injected into our test on creation. The property is then available to our test methods for testing the converter. Listing 10-13 shows this configuration file.

Listing 10-13. *A Sample Application Context for Integration Testing*

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC
      "-//SPRING//DTD BEAN//EN"
      "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

  <bean id="currencyConverter"
        class="com.apress.springbook.chapter10.DefaultCurrencyConverter">
    <property name="exchangeRateService" ref="exchangeRateService"/>
  </bean>

  <bean id="exchangeRateService"
        class="com.apress.springbook.chapter10.DefaultExchangeRateService"/>

</beans>
```

The next step is to create an integration test class for testing the currency converter using Spring's test support. Listing 10-14 shows this integration test, which extends the `AbstractDependencyInjectionSpringContextTests` base class.

Listing 10-14. *An Integration Test for the Currency Converter Using Spring's Test Support*

```
package com.apress.springbook.chapter10;

import org.springframework.test.AbstractDependencyInjectionSpringContextTests;

public class CurrencyConverterIntegrationTests
  extends AbstractDependencyInjectionSpringContextTests {

  private CurrencyConverter currencyConverter;

  public void setCurrencyConverter(CurrencyConverter currencyConverter) {
    this.currencyConverter = currencyConverter;
  }

  protected String[] getConfigLocations() {
    return new String[] { "classpath:/applicationContext.xml" };
  }

  public void testWithValidInput() {
    try {
      assertEquals(12.234, currencyConverter.convert(10.0, "EUR", "USD"));
    } catch (UnknownCurrencyException e) {
      fail("something went wrong in testWithValidInput()");
    }
  }

  public void testConvertWithUnknownCurrency() {
    try {
      currencyConverter.convert(10.0, "EUR", "-UNKNOWN-");
      fail("an unknown currency exception was expected");
    } catch (UnknownCurrencyException e) {
      // do nothing, was expected
    }
  }
}
```

As you can see, the `getConfigLocations()` method is implemented to return the previously created application context. This application context contains the currency converter, which is automatically injected by Spring. This injection is done in one of two ways:

- Through setter injection—creating setter methods for the dependencies you want to have injected. They will be satisfied by autowiring by type.
- Through field injection—declaring protected variables of the required type that match named beans in the context. This is autowiring by name, rather than type.

By default, all dependencies are injected using setter injection. You need to set the `populateProtectedVariables` property to `true` in the constructor of the test to switch on field injection.

Note that the test methods in the test class in Listing 10-14 perform an integration test, because they also include the exchange rate service dependency of the currency converter.

Testing with Transactions

Another convenient test base class is `org.springframework.test.AbstractTransactionalSpringContextTests`, which builds on top of the functionality offered by the `AbstractDependencyInjectionSpringContextTests` test base class. Each test method that is executed by a subclass of this base class will automatically participate in a transaction. Because the default is to roll back the transaction after each test method execution, no actual modifications will be made to any transactional resources. This makes it the perfect choice for performing integration tests using a transactional data source.

Using this base class allows you to integration test functionality that will make modifications to the database without having to worry about the changes affecting any other test methods. It also allows you test code that requires a transactional context. And you can write data to the database without worrying about cleaning it up afterwards.

As mentioned, all modifications to the database are rolled back at the end of each test method execution. In order to override this behavior, you have two alternative approaches:

- Set the `defaultRollback` property to `false`, to make the transaction not roll back by default after each test method execution.
- Call `setComplete()` in a test method in order to inform the test not to roll back the transaction after the test methods complete.

Testing with a DataSource

A third convenient test base class is `org.springframework.test.AbstractTransactionalDataSourceSpringContextTests`, which builds on top of the functionality provided by `AbstractTransactionalSpringContextTests`. In order to use this base class, you need to include a `DataSource` definition in the application context loaded by this test. The data source is automatically injected, as explained earlier.

The main feature offered by this base class is that it provides you with a `JdbcTemplate` as a protected field, which you can use to modify the data source, within the transactional context. You could, for instance, insert some data that the test needs in order to succeed. Because the statements to the `JdbcTemplate` are also executed within the transactional context, you do not need to worry about cleaning up the database or modifying the existing data.

Using Spring Mock Classes

So far, you have seen how Spring helps you integration test the beans defined in your application context and provides support for testing code that actually modifies the database. Spring also provides support for testing your J2EE-specific application code. Because much of your web application code is very much tied to J2EE classes, it is hard to test. For instance, testing a servlet or a Spring controller implementation requires you to somehow mock the `HttpServletRequest` and `HttpServletResponse` classes. You could use EasyMock to do this, but there is a much easier alternative: Spring's web mock classes.

Table 10-3 lists the web mock classes provided by Spring to help you test your web application code.

Table 10-3. *Web Mock Classes Provided by Spring*

Class	Description
`MockHttpServletRequest`	Mock implementation of the `HttpServletRequest` interface. Can be used to test servlets and controllers.
`MockHttpServletResponse`	Mock implementation of the `HttpServletResponse` interface. Can be used to test servlets and controllers.
`MockHttpSession`	Mock implementation of the `HttpSession` interface. Can be used to test code that requires a session in order to function.
`MockFilterConfig`	Mock implementation of the `FilterConfig` interface. Can be used to test filter implementations.
`MockServletConfig`	Mock implementation of the `ServletConfig` interface. Can be used to test servlets.
`MockServletContext`	Mock implementation of the `ServletContext` interface. Can be used to test servlets.
`MockRequestDispatcher`	Mock implementation of the `RequestDispatcher` interface. Can be used to test servlets and controllers.
`MockPageContext`	Mock implementation of the `PageContext` abstract class. Can be used to test JSP tag implementations.
`MockExpressionEvaluator`	Mock implementation of the `ExpressionEvaluator` abstract class. Can be used to test JSP tag implementations.

Along with these web-specific mock objects, Spring also provides mock objects for a number of other hard-to-mock J2EE interfaces and classes. Currently, it provides mock objects mainly for working with JNDI, such as the `SimpleNamingContext`, which provides a mock object for the `Context` interface.

Summary

This chapter started with an overview of testing, including unit testing, integration testing, and TDD. We then introduced JUnit as a framework for writing tests, and EasyMock to mock the dependencies your classes may have. Finally, you saw how Spring provides a number of convenient base test classes and mock objects to perform integration testing and test your web application code.