9 Spring Testing

Testing is an indispensable part of developing high quality, reliable enterprise software. Testing can typically be divided into two broad categories: functional and non-functional tests. Functional tests help verify whether the product under test executes correctly with a pre-determined set of use cases, whereas non-functional tests help verify whether the product under test can meet the performance, reliability, scalability and/or any other requirements that are operational-driven.

In this chapter, we are more concerned with functional tests. The first level of functional testing is *unit testing*, which verifies that the methods of a class work as expected with one or more given states as represented by the attributes of the class. The next level of functional testing is *integration testing*, which verifies that a class or component works as expected when other components get involved. The third level of functional testing is *system testing*, which verifies that the product still work as expected when it is deployed to an internal test environment and driven with realistic use cases that mimic how real users would use it. The scope becomes broader and broader as we go higher from one level to the next. For example, with the SOBA sample included in this book, I have to verify that it does not only work in a base environment of MySQL/Tomcat/Windows, but also works with other commonly used databases, application servers and operating systems such as Oracle, SQL Server, Jboss, GlassFish, WebLogic, Linux and Mac OS X. However, if you are working with a real enterprise software product, you should convince your company to minimize the variety of DB/AppServer/OS that your product has to support. That will help your company save time and money in the end.

In this chapter, we start with how to use JUnit and TestNG to perform functional tests in general. Then we cover how Spring Testing supports testing Spring components with features such as TestContext, automatically injecting test fixtures, transactional support, and so on. If you are already familiar with JUnit and TestNG, feel free to jump to Spring Testing directly.

9.1 UNIT TESTING WITH JUNIT AND TESTNG

JUnit is a testing framework that has been in use widely for testing Java applications. At the time of this writing, the latest formal release is 4.11. Spring 2.5 and prior releases supported JUnit 3.8,

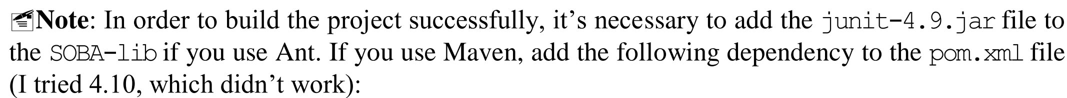
which was referred to as JUnit 3.8 legacy support, while Spring 3.x started supporting JUnit 4. JUnit 3.x depends on the following two features:

- The testing class must extend the JUnit framework class `TestCase`.
- Each test case must be a public method and the method name must be prefixed with `test`.

In contrast, JUnit 4 takes advantages of the Java annotation feature introduced in Java 1.5. It provides a framework `@Test` annotation so that any public methods can be run as a test case. Next, we use an example to illustrate how JUnit 4 works in general.

9.1.1 Unit Testing with JUnit 4

Let us use our SOBA sample as the context for demonstrating how JUnit 4 works. Let's say we have a requirement that dividend should be paid with an banking account on a monthly basis. This requirement can be implemented with a `DividentCalculator.java` interface and a `SimpleDividentCalculator.java` class that implements the interface. Listings 9.1 and 9.2 illustrates the Java source code for this interface and class.



Note: In order to build the project successfully, it's necessary to add the `junit-4.9.jar` file to the `SOBA-lib` if you use Ant. If you use Maven, add the following dependency to the `pom.xml` file (I tried 4.10, which didn't work):

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.9</version>
</dependency>
```

As we see from the interface and the implementation class shown in Listings 9.1 and 9.2, the method `calculate (double balance, int numOfDays)` returns the amount of dividend to be credited to the customer's account, which can be initiated on a monthly basis.

Listing 9.1 DividentCalculator.java

```
package com.perfmath.spring.soba.testing;

public interface DividentCalculator {

   public void setDailyRate(double dailyRate);
   public double calculate(double balance, int numOfDays);
}
```

Listing 9.2 SimpleDividentCalculator.java

```
package com.perfmath.spring.soba.testing;

public class SimpleDividentCalculator implements DividentCalculator {

   private double dailyRate;

   public void setDailyRate(double dailyRate) {
      this.dailyRate = dailyRate;
   }

   public double calculate(double balance, int numOfDays) {
      if (numOfDays <0) {
         throw new IllegalArgumentException("number of days must be positive");
      } else if (balance < 0) {
          return 0;
      } else {
         return balance * numOfDays * dailyRate;
      }
   }
}
```

Listing 9.3 shows how to test the `SimpleDividentCalculator.java` class as shown in Listing 9.2. Unlike JUnit 3.x, this JUnit test class does not extend the JUnit framework `TestCase` base class. Besides, it does not have to prefix each public method with `test`. Instead, it uses the `@Test` annotation to indicate that the annotated method is a test case. In addition, you can use the `assertEquals` to ensure that the dividend would be $15.0 with a daily rate of 0.1%, an average balance of $500 over a 30-day period. This method has the signature of

```
Public static void assertEquals (double expected, double actual, double delta)
```

which asserts that the `expected` and `actual` doubles will be considered `equal` within a positive `delta` as specified. Finally, you can use the `expected` attribute of the `@Test` annotation to specify the exception to be thrown when an invalid parameter is input.

Listing 9.3 SimpleDividentCalculatorJUnitTest.java

```
package com.perfmath.spring.soba.testing.junit;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

import com.perfmath.spring.soba.testing.DividentCalculator;
import com.perfmath.spring.soba.testing.SimpleDividentCalculator;
```

```
public class SimpleDividentCalculatorJUnitTest {

   private DividentCalculator dividentCalculator;

   @Before
   public void init() {
      dividentCalculator = new SimpleDividentCalculator();
      dividentCalculator.setDailyRate(0.001);
   }

   @Test
   public void calculate() {
      double divident = dividentCalculator.calculate(500.0, 30);
      assertEquals(divident, 15.0, 0);
   }

   @Test(expected = IllegalArgumentException.class)
   public void illegalCalculate() {
      dividentCalculator.calculate(100.0, -10);
   }
}
```

To run the above JUnit test class `SimpleDividentCalculatorJUnitTest`, first, make sure you build the project with either Maven or Ant without encounter any errors. Then, you can right-click on the class name of `SimpleDividentCalculatorJUnitTest.java` in the *src/main/java/testing/junit* package, and select *Run As … 2 JUnit Test*, as shown in Figure 9.1. If you built the project with Maven, you should see an output as shown in Figure 9.2, indicating zero errors and failures. However, if you built the project with Ant, you might get the notorious exception of `ClassDefNotFound`, complaining that the `SimpleDividentCalculatorJUnitTest` class cannot be found. Next, we describe how to resolve this error.

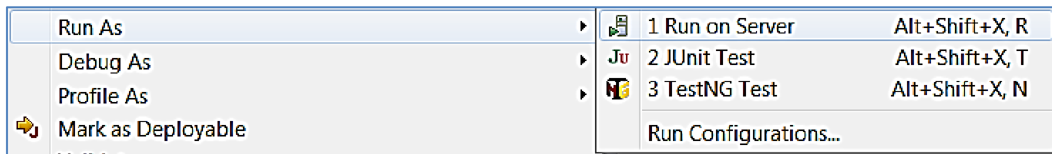


Figure 9.1 Running JUnit test on Eclipse IDE

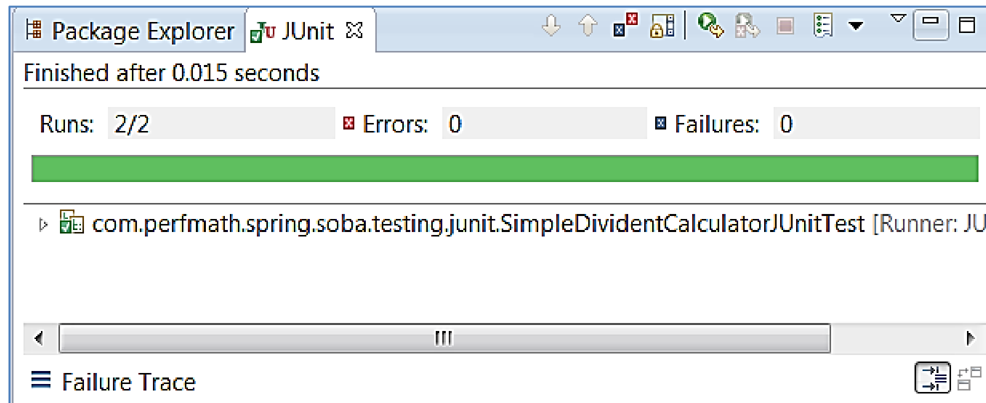


Figure 9.2 Output from running the JUnit test

The error of `ClassDefNotFound` typically is a CLASSPATH issue. When Maven is enabled on Eclipse, the *target/classes* folder is automatically added to the project's classpath. However, this is not the case when the project is built with Ant. In this case, you have to add the *build/classes* folder according to the instructions given in the *SOBA_project_ReadMeFirst.PDF* file included in the project download zip file.

Next, we discuss how to use TestNG to perform unit testing.

9.1.2 Unit Testing with TestNG

TestNG is marketed as the *Next Generation Java Testing Framework* (http://testng.org). As it is stated at its website, it is inspired from JUnit and NUnit and introduces the following new features:

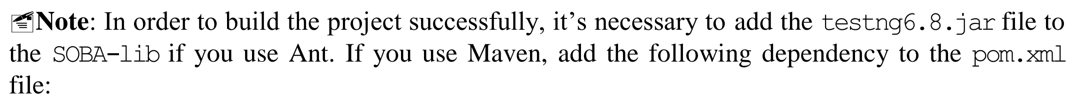
- Annotations
- Run your tests in arbitrarily big thread pools (all methods in their own thread and one thread per test case, etc.)
- Test your code is multithread safe
- Supper for data-driven testing with `@DataProvider`
- Support for parameters
- Powerful execution model with no more need to extend `TestCase`
- Supported by a variety of tools and plug-ins such as Eclipse, Maven, IDEA, etc.
- Dependent methods for application server testing

TestNG is designed to cover all categories of testing such as unit testing, functional testing, end-to-end testing, integration testing, and so on. However, a detailed coverage of TestNG is beyond the scope of this text. We can only present a few examples to help you get a glimpse of how it works.

Listing 9.4 shows how to use TestNG to test the `SimpleDividentCalculator` class as an alternative to JUnit. First, it applies the `@BeforeMethod` annotation to the `init` () method to

create a new `DividentCalculator` instance and sets the `dailyRate` to 0.1%. Then, note the two methods of `createLegalDividentParameters()` and `createIllegalDividentParameters()`, annotated with `@DataProvider` named with "`legal`" and "`illegal`", respectively. Note how the `Object[]` and `Object[][]` are used with the `@DataProvider` annotation. Finally, note how these two DataProviders are used with the `calculate` and `illegalCalculate` methods, both of which are annotated with the `@Test` annotation.

You can build and execute this test class using the Eclipse *Run As …* feature as shown in Figure 9.1. Listing 9.5 shows the output obtained in my environment.



Note: In order to build the project successfully, it's necessary to add the `testng6.8.jar` file to the `SOBA-lib` if you use Ant. If you use Maven, add the following dependency to the `pom.xml` file:

```
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>6.8</version>
</dependency>
```

Listing 9.4 SimpleDividentCalculatorTestNGTest.java

```
package com.perfmath.spring.soba.testing.testng;

import static org.testng.Assert.*;

import org.testng.annotations.BeforeMethod;
import org.testng.annotations.DataProvider;
import org.testng.annotations.Test;

import com.perfmath.spring.soba.testing.DividentCalculator;
import com.perfmath.spring.soba.testing.SimpleDividentCalculator;

public class SimpleDividentCalculatorTestNGTest {

  private DividentCalculator dividentCalculator;

  @BeforeMethod
  public void init() {
    dividentCalculator = new SimpleDividentCalculator();
    dividentCalculator.setDailyRate(0.001);
  }
```

```
  @DataProvider(name = "legal")
  public Object[][] createLegalDividentParameters() {
    return new Object[][] { new Object[] { 500, 30, 15.0 } };
  }

  @DataProvider(name = "illegal")
  public Object[][] createIllegalDividentParameters() {
    return new Object[][] {new Object[] { 500, -30 }, new Object[] { -500, -30 }};
  }

  @Test(dataProvider = "legal")
  public void calculate(double balance, int numOfDays, double result) {
    double divident= dividentCalculator.calculate(balance, numOfDays);
    assertEquals(divident, result);
  }

  @Test(dataProvider = "illegal", expectedExceptions = IllegalArgumentException.class)
  public void illegalCalculate(double balance, int numOfDays) {
   dividentCalculator.calculate(balance, numOfDays);
  }
}
```

Listing 9.5 TestNG output from running SimpleDividentCalculatorTestNGTest

```
[TestNG] Running:
 C:\Users\henry\AppData\Local\Temp\testng-eclipse--972939339\testng-customsuite.xml

PASSED: calculate(500, 30, 15.0)
PASSED: illegalCalculate(500, -30)
PASSED: illegalCalculate(-500, -30)

===============================================
  Default test
  Tests run: 3, Failures: 0, Skips: 0
===============================================
```

9.2 INTEGRATION TESTING WITH TESTNG

The scope of a unit test is limited to a single programming unit, which is typically a class with its methods in object-oriented languages. As we described earlier, the next level of testing is integration testing, the scope of which becomes wider that it has to deal with the inter-dependencies among multiple units or objects. In this section, let us explore how to conduct integration testing with TestNG. We start with testing DAO units with InMemoryDaos.

9.2.1 Integration Testing DAOs with InMemoryDaos

In principle, a DAO unit or class should be tested against a real database. However, during the early stage of a product development process, it's very common to test DAOs with InMemoryDaos. In this section, we present an example of integration-testing DAOs with InMemoryDaos.

The example we use in this section is the Account domain object and the AccountDao interface from the SOBA project. Listings 9.6 and 9.7 show the Account domain object and the AccountDao interface, while Listing 9.8 shows the InMemoryAccountDao class that implements the AccountDao interface. You can take a quick look at these listings, but our main subject here is how to test the InMemoryDAO with TestNG. In addition, we focus on the insert, update, delete and findByAccountID methods only, as highlighted in Listing 9.8. Note that this InMemoryDao is implemented with a synchronized HashMap.

Listing 9.6 Account.java

```
package com.perfmath.spring.soba.model.domain;

import java.io.Serializable;
import java.sql.Timestamp;

public class Account implements Serializable {

  private String accountId;
  private String name;
  private String type;
  private String description;
  private String status;
  private double balance;
  private Timestamp openDate;
  private Timestamp closeDate;
  private String customerId;

  // getters and setters are omitted to save space
  public boolean equals(Object obj) {
    if (!(obj instanceof Account)) {
      return false;
    }
    Account account = (Account) obj;
    return account.accountId.equals(accountId) && account.balance == balance;
  }
}
```

Listing 9.7 AccountDao.java

```
package com.perfmath.spring.soba.model.dao;

import java.util.List;
import java.util.Map;

import com.perfmath.spring.soba.model.domain.Account;

public interface AccountDao {
   public List<Account> getAccountList();
   public void insert(Account account);
   public void update(Account account);
   public double updateAccountBalance (double amount, String accountID);
   public void delete(Account account);
   public Account findByAccountID(String accountID);
   public void insertBatch(List<Account> accounts);
   public List<Map<String, Object>> findAll();
   public String getCustomerId(String accountID);
   public int countAll();
   public String getAccountIdByCustomerId (String customerId, String accountType);
}
```

Listing 9.8 InMemoryAccountDao.java

```
package com.perfmath.spring.soba.testing;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;

import java.util.Set;

import com.perfmath.spring.soba.model.domain.Account;
import com.perfmath.spring.soba.model.dao.AccountDao;

public class InMemoryAccountDao implements AccountDao {

   private Map<String, Account> accounts;

   public InMemoryAccountDao() {
```

```
    accounts = Collections.synchronizedMap(new HashMap<String, Account>());
  }

  public boolean accountExists(String accountNo) {
    return accounts.containsKey(accountNo);
  }

  public void insert(Account account) {
    if (accountExists(account.getAccountId())) {
      throw new DuplicateAccountException();
    }
    accounts.put(account.getAccountId(), account);
  }

  public void update(Account account) {
    if (!accountExists(account.getAccountId())) {
      throw new AccountNotFoundException();
    }
    accounts.put(account.getAccountId(), account);
  }

  public void delete(Account account) {
    if (!accountExists(account.getAccountId())) {
      throw new AccountNotFoundException();
    }
    accounts.remove(account.getAccountId());
  }

  public Account findByAccountID(String accountId) {
    Account account = accounts.get(accountId);
    if (account == null) {
      throw new AccountNotFoundException();
    }
    return account;
  }

   public List<Account> getAccountList() {
        List<Account> list = (List<Account>) accounts.values();
        return list;
   }
  public double updateAccountBalance (double amount, String accountID) {
   Account account = findByAccountID(accountId);
   double balance = amount + account.getBalance();
   account.setBalance(balance);
```

```
    update (account);
    return balance;
  }
  public void insertBatch(List<Account> accounts) {
    ;
  }
  public List<Map<String, Object>> findAll() {
    List<Map<String, Object>> list = new ArrayList ();
    Map <String, Object> objectMap = new HashMap <String, Object>();
    Set set = (Set) accounts.entrySet();
    Iterator it = set.iterator ();
    while (it.hasNext()) {
        Map.Entry<String, Account> map = (Map.Entry)it.next();
        objectMap.put (map.getKey().toString(), (Object) map.getValue());
        list.add(objectMap);
    }
    return list;
  }
  public String getCustomerId(String accountID) {
    return "";
  }
  public int countAll() {
    return 0;
  }
  public String getAccountIdByCustomerId (String customerId, String accountType) {
    return "";
  }
}
```

Listing 9.9 shows the `InMemoryAccountDaoTest.java` class implemented with TestNG. If you have gone through the TestNG unit testing example with the `SimpleDividentCalculator.java` class presented earlier, the annotations used in this `InMemoryAccountDaoTest.java` class should look familiar to you. You can build this project with either Ant or Maven and run the test as you did with the previous TestNG example. For your reference, Listing 9.10 shows the console output from running this test in my Eclipse environment. You can also view the results of running class `InMemoryAccountDaoTest` by clicking on the tab next to the Console tab as shown in Figure 9.3.

Listing 9.9 InMemoryAccountDaoTest.java

```
package com.perfmath.spring.soba.testing.testng;

import static org.testng.Assert.*;

import org.testng.annotations.BeforeMethod;
```

```
import org.testng.annotations.Test;

import com.perfmath.spring.soba.model.domain.Account;
import com.perfmath.spring.soba.testing.InMemoryAccountDao;
import com.perfmath.spring.soba.testing.AccountNotFoundException;
import com.perfmath.spring.soba.testing.DuplicateAccountException;

public class InMemoryAccountDaoTest {

  private static final String EXISTING_ACCOUNT_ID = "88888888";
  private static final String NEW_ACCOUNT_ID = "99999999";

  private Account existingAccount;
  private Account newAccount;
  private InMemoryAccountDao accountDao;

  @BeforeMethod
  public void init() {
    existingAccount = new Account();
    existingAccount.setAccountId(EXISTING_ACCOUNT_ID);
    existingAccount.setBalance(800.0);

    newAccount = new Account();
    newAccount.setAccountId(NEW_ACCOUNT_ID);
    newAccount.setBalance(900.0);

    accountDao = new InMemoryAccountDao();
    accountDao.insert(existingAccount);
  }

  @Test
  public void createNewAccount() {
    accountDao.insert(newAccount);
    assertEquals(accountDao.findByAccountID(NEW_ACCOUNT_ID), newAccount);
  }

  @Test
  public void accountExists() {
    assertTrue(accountDao.accountExists(EXISTING_ACCOUNT_ID));
    assertFalse(accountDao.accountExists(NEW_ACCOUNT_ID));
  }

  @Test(expectedExceptions = DuplicateAccountException.class)
  public void createDuplicateAccount() {
```

```
        accountDao.insert(existingAccount);
    }

    @Test
    public void updateExistingAccount() {
        existingAccount.setBalance(20);
        accountDao.update(existingAccount);
        assertEquals(accountDao.findByAccountID(EXISTING_ACCOUNT_ID), existingAccount);
    }
    @Test
    public void updateAccountBalance() {
      accountDao.updateAccountBalance(UPDATE_AMOUNT, EXISTING_ACCOUNT_ID);
        assertEquals(accountDao.findByAccountID(EXISTING_ACCOUNT_ID).getBalance(),
                (EXISTING_ACCOUNT_INITIAL_BALANCE + UPDATE_AMOUNT), 0);
    }

    @Test(expectedExceptions = AccountNotFoundException.class)
    public void updateNonExistingAccount() {
        accountDao.update(newAccount);
    }

    @Test
    public void removeExistingAccount() {
        accountDao.delete(existingAccount);
        assertFalse(accountDao.accountExists(EXISTING_ACCOUNT_ID));
    }

    @Test(expectedExceptions = AccountNotFoundException.class)
    public void removeNonExistingAccount() {
        accountDao.delete(newAccount);
    }

    @Test
    public void findExistingAccount() {
        Account account = accountDao.findByAccountID(EXISTING_ACCOUNT_ID);
        assertEquals(account, existingAccount);
    }

    @Test(expectedExceptions = AccountNotFoundException.class)
    public void findNonExistingAccount() {
        accountDao.findByAccountID(NEW_ACCOUNT_ID);
    }
}
```

Listing 9.10 Console output from running the InMemoryAccountDao TestNG testing

```
[TestNG] Running:
  C:\Users\henry\AppData\Local\Temp\testng-eclipse-1229054272\testng-customsuite.xml

PASSED: accountExists
PASSED: createDuplicateAccount
PASSED: createNewAccount
PASSED: findExistingAccount
PASSED: findNonExistingAccount
PASSED: removeExistingAccount
PASSED: removeNonExistingAccount
PASSED: updateAccountBalance
PASSED: updateExistingAccount
PASSED: updateNonExistingAccount

===============================================
    Default test
    Tests run: 10, Failures: 0, Skips: 0
===============================================
```

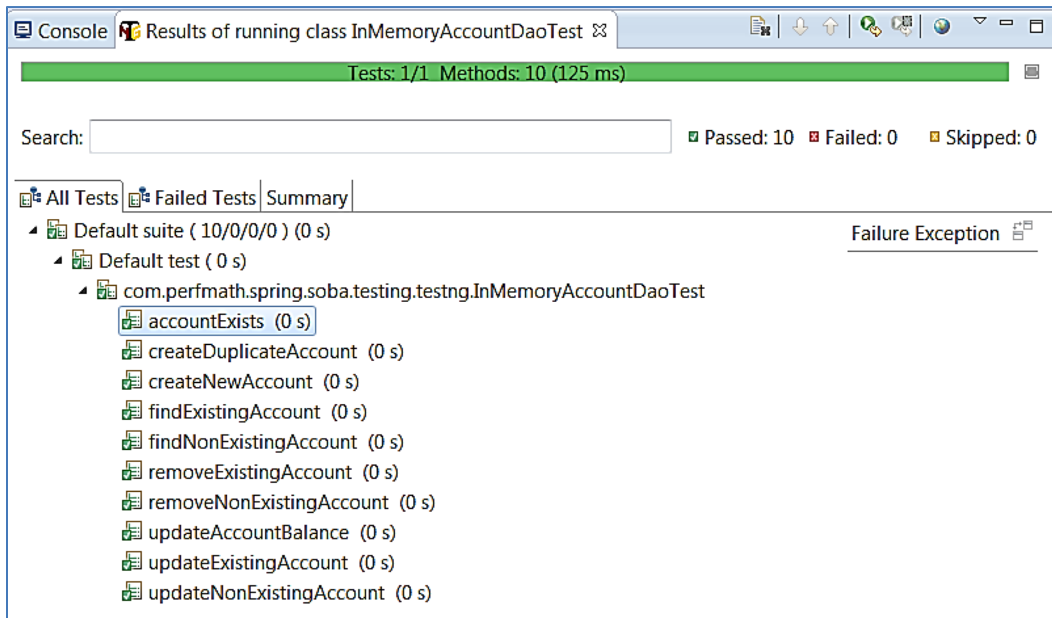


Figure 9.3 Tab for the Results of running class InMmemoryAccountDaoTest

9.2.2 Integration Testing Service Layer Classes with TestNG

Since the service layer of an enterprise application typically depends on the DAO layer, the scope of testing a service layer classes belongs in the category of integration testing. In this section, we use the example of AccountManager in the package of com.perfmath.spring.soba.service of the SOBA project to demonstrate how to conduct integration testing with TestNG. We continue using the InMemoryAccountDao class used in the previous section for this example. In the next section, we replace this class with a mock DAO object to demonstrate how to use mock objects to assist integration testing.

For this example, we continue using Account.java as the domain object class, AccountDao.java as the DAO class, and InMemoryAccountDao.java as the DAO implementation class. In addition, we introduce the AccountManager.java interface and the SimpleAccountManager.java class as the service layer class to be tested. Listings 9.11 and 9.12 show how this interface and the implementation class are coded.

Note from Listing 9.12 that the SimpleAccountManager class depends on the AccountDao class. Therefore, testing the SimpleAccountManager class cannot be conducted without getting the class AccountDao involved. Listing 9.13 shows the SimpleAccountManagerIntegrationTest.java class, implemented with TestNG. In addition to the TestNG specific annotations such as @BeforeMethod, @Test, and @AfterMethod, note the account instance created in the init() method and how an InMemoryAccountDao object is hooked up with the accountManager object, which characterizes this testing being integration testing.

Note in Listing 9.13 that, for demonstrating how to conduct integration testing with TestNG, only two test cases are included: createAccount and updateAccountBalance. The createAccount test case creates the account set up in the init method, and then asserts that the ACCOUNT_ID and BALANCE of the account created match the account id and the initial balance as set in the init method. The two asserts are accomplished with the help of the findByAccountID method.

In addition, keep in mind that the init method only creates an account instance, which is not persisted into the InMemoryAccountDao object. Therefore, in the updateAccountBalance test case, the account instance created in the init method must be persisted into the HashMap of the InMemoryAccountDao object; otherwise, an AccountNotFound exception will be thrown when the test case is executed. We cannot simply assume that the createAccount test case would always be executed first so that the account instance would be available in the updateAccountBalance test case without creating it again as shown in the updateAccountBalance method.

You can build the project with either Ant or Maven and run the test as you did with the previous TestNG example. For your reference, Listing 9.13 shows the console output from running this test in my Eclipse environment, indicating that the test was executed successfully.

Listing 9.11 AccountManager.java

```
package com.perfmath.spring.soba.service;

import java.io.Serializable;
import java.util.List;
```

```
import com.perfmath.spring.soba.model.domain.Account;

public interface AccountManager extends Serializable{

   public void createAccount(Account account);
   public List<Account> getAccounts();
   public String getAccountIdByCustomerId (String customerId, String accountType);
   public double updateAccountBalance (double amount, String accountID);
   public void deleteAccount(String accountID);
}
```

Listing 9.12 SimpleAccountManager.java

```
package com.perfmath.spring.soba.service;
import java.util.List;
import com.perfmath.spring.soba.model.dao.AccountDao;
import com.perfmath.spring.soba.model.domain.Account;

public class SimpleAccountManager implements AccountManager {

   private AccountDao accountDao;

   public List<Account> getAccounts() {
      return accountDao.getAccountList();
   }

   public void createAccount(Account account) {
     accountDao.insert(account);
   }
   public void updateAccount(Account account) {
     accountDao.update(account);
   }
   public void setAccountDao(AccountDao accountDao) {
      this.accountDao = accountDao;
   }

   public String getAccountIdByCustomerId (String customerId, String accountType) {
         return accountDao.getAccountIdByCustomerId(customerId, accountType);
   }

   public double updateAccountBalance (double amount, String accountID) {
     Account account = accountDao.findByAccountID(accountID);
     double balance = amount + account.getBalance();
     account.setBalance(balance);
```

```
    accountDao.update(account);
    return balance;
  }
  public void deleteAccount(String accountID) {
    Account account = accountDao.findByAccountID(accountID);
    accountDao.delete(account);
  }
}
```

Listing 9.13 SimpleAccountManagerIntegrationTest.java

```
package com.perfmath.spring.soba.testing.testng;

import static org.testng.Assert.*;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;
import com.perfmath.spring.soba.model.domain.Account;
import com.perfmath.spring.soba.model.dao.AccountDao;
import com.perfmath.spring.soba.service.SimpleAccountManager;
import com.perfmath.spring.soba.testing.InMemoryAccountDao;

public class SimpleAccountManagerIntegrationTest {

private static final String TEST_ACCOUNT_ID = "88889999";
  private static final double INITIAL_BALANCE = 200.0;
  private static final double UPDATE_AMOUNT = 50.0;
  private AccountDao accountDao;
  private SimpleAccountManager accountManager;
  private Account account;

  @BeforeMethod
  public void init() {
    account = new Account ();
    account.setAccountId (TEST_ACCOUNT_ID);
    account.setBalance (200.0);

    accountManager = new SimpleAccountManager ();
    accountDao = new InMemoryAccountDao();
    accountManager.setAccountDao (accountDao);
  }

  @Test
  public void createAccount () {
```

```
    accountManager.createAccount(account);
    assertEquals((accountDao.findByAccountID(TEST_ACCOUNT_ID)).getAccountId(),
      TEST_ACCOUNT_ID);
    assertEquals((accountDao.findByAccountID(TEST_ACCOUNT_ID)).getBalance(),
      INITIAL_BALANCE, 0);
  }

  @Test
  public void updateAccountBalance () {
   accountManager.createAccount(account);
   accountManager.updateAccountBalance(UPDATE_AMOUNT, TEST_ACCOUNT_ID);
   assertEquals((accountDao.findByAccountID(TEST_ACCOUNT_ID)).getBalance(),
        (UPDATE_AMOUNT + INITIAL_BALANCE), 0);
  }

  @AfterMethod
  public void cleanup () {
   accountManager.deleteAccount (TEST_ACCOUNT_ID);
  }
}
```

Listing 9.14 Console output of running SimpleAccountManagerIntegrationTest

```
[TestNG] Running:
 C:\Users\henry\AppData\Local\Temp\testng-eclipse--289187755\testng-customsuite.xml

PASSED: createAccount
PASSED: updateAccountBalance

===============================================
  Default test
  Tests run: 2, Failures: 0, Skips: 0
===============================================
```

9.2.3 Integration Testing with Mock Objects

As we pointed earlier, using mock objects is a helpful approach to conducting integration testing that has to resolve dependencies for the class under test. To demonstrate how to use mock objects to facilitate integration testing, we re-use the `SimpleAccountManager` example presented in the preceding section, with one change that instead of using an `InMemoryAccountDao` object, we use mock objects.

For this example, we use EasyMock (http://www.easymock.org), which is a Java testing framework for creating mock objects on the fly using Java's proxy mechanism. To demonstrate

how mock objects generated with EasyMock can help conduct integration testing on the SimpleAccountManager class that was tested with an InMemoryAccountDao object in the preceding section, Listing 9.15 shows the integration testing implemented with TestNG and EasyMock. Let's summarize how EasyMock is used for this integration testing:

- **Creating the mock object**: This is accomplished in the init method with the statement of mock = createMock(AccountDao.class). This is a mock object that mocks an AccountDao object.
- **Hooking up the mock object with the class under test**: This is accomplished in the init method with the statement of accountManager.setAccountDao (mock). As is seen, the mocking AccountDao object named mock is assigned to the class under test as its AccountDao object. This mock object will play the role of a DAO object like a more real DAO object such as the InMemoryAccountDao object we used in the preceding section or a real JDBC or Hibernate DAO object that actually persists domain objects into a real database.
- **A mock object in replay**: Let's use the createAccount test case to demonstrate how this works. First, the statement of mock.insert(account) is executed. Since mock is an accountDao mocking object, this statement can be considered *inserting* an account object into a "mocking" store just as if this mock object were a real DAO object. Once a mock object is created, it will be put into the *record* state, meaning that any operations performed upon it will be recorded for future referencing and verification. The next statement of replay (mock) will put the mock object in the *replay* state, meaning that the mock object is active so that you can perform operations like accountManager.createAccount(account) (note that without this mock object, the account instance would have no place to persist). Finally, the verify (mock) statement verifies that the accountManager.createAccount(account) statement has achieved the same result as the mock.insert(account) statement recorded earlier.
- **Returning a domain object from the mock object**: Note this interesting feature with the expect (mock.findByAccountID(TEST_ACCOUNT_ID)).andReturn (account) statement in the updateAccountBalance test case. This expect statement finds the account object stored in the mock object and returns it to the executing thread. This account object can then be used as if it were retrieved from an InMemoryAccountDao object or a real relational store. The rest of the updateAccountBalance test case is similar to what we explained with the createAccount test case, so we would not repeat it again.

You can build the project with either Ant or Maven and run the test as you did with the previous TestNG example. For your reference, Listing 9.16 shows the console output from running this test in my Eclipse environment, indicating that the test was executed successfully.

In the next section, we explore Spring testing support for testing various Spring components. You will see that the concept of a mock object is equally applicable to testing Spring components. Of course, Spring has introduced its own more advanced concepts such as MockHttpSession, MockHttpServletRequest, MockHttpServletResponse, TestContext, and so on.

Note: In order to build the project successfully, it's necessary to add the easymock-3.1.jar

file to the SOBA-lib if you use Ant. If you use Maven, add the following dependency to the pom.xml file:

```
<dependency>
    <groupId>org.easymock</groupId>
    <artifactId>easymock</artifactId>
    <version>3.1</version>
</dependency>
```

Listing 9.15 SimpleAccountManagerMockTest.java

```
package com.perfmath.spring.soba.testing.testng;

import static org.easymock.EasyMock.*;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;
import static org.testng.Assert.*;

import com.perfmath.spring.soba.model.domain.Account;
import com.perfmath.spring.soba.model.dao.AccountDao;
import com.perfmath.spring.soba.service.SimpleAccountManager;

public class SimpleAccountManagerMockTest {

  private static final String TEST_ACCOUNT_ID = "88889999";
  private static final double AMOUNT = 50.0d;
  private AccountDao mock;
  private SimpleAccountManager accountManager;
  private Account account;

  @BeforeMethod
  public void init() {
    account = new Account ();
    account.setAccountId (TEST_ACCOUNT_ID);
    account.setBalance (200.0);

    mock = createMock(AccountDao.class);
    accountManager = new SimpleAccountManager ();
    accountManager.setAccountDao (mock);
  }

  @Test
  public void createAccount () {
```

```
    mock.insert(account);
    replay (mock);

      accountManager.createAccount(account);
      verify (mock);
   }

   @Test
   public void updateAccountBalance() {
      expect (mock.findByAccountID(TEST_ACCOUNT_ID)).andReturn (account);
      account.setBalance(AMOUNT + account.getBalance());
      mock.update(account);
      replay(mock);

      accountManager.updateAccountBalance(AMOUNT, TEST_ACCOUNT_ID);
      verify(mock);
   }
}
```

Listing 9.16 Console output from running SimpleAccountManagerMockTest

```
[TestNG] Running:
  C:\Users\henry\AppData\Local\Temp\testng-eclipse-1570924868\testng-customsuite.xml

PASSED: createAccount
PASSED: updateAccountBalance

===============================================
   Default test
   Tests run: 2, Failures: 0, Skips: 0
===============================================
```

9.3 SPRING TESTING SUPPORT

As one of the most popular enterprise Java development framework, Spring takes testing seriously and supports testing various Spring components on top of the generic Java testing frameworks such as TestNG and JUnit. In this section, we explore Spring testing support using the SOBA project as an example. Let’s start with testing Spring MVC controllers next.

9.3.1 Testing Spring MVC Controllers

Note: In order to build the project with all test cases presented in this section successfully, it’s

necessary to add the spring-mock-2.0.8.jar file and spring-test-3.2.0.RELEASE.jar file to the SOBA-lib if you use Ant. If you use Maven, add the following dependencies to the pom.xml file (note that I used spring-mock-version 2.0.8 and spring-test version 3.2.0- RELEASE at the time of this writing):

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-mock</artifactId>
    <version>${spring-mock.version}</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${org.springframework.version}</version>
</dependency>
```

To demonstrate how to test Spring MVC controllers, we use the AccountController class in the SOBA project. As you have learnt from the previous chapters of this text, a Spring MVC controller can be either annotation-based or configuration-based. We present an example for each case. Let's start with testing an annotation-based Spring MVC controller next.

#1 Testing Annotation-based Spring MVC Controllers

Listing 9.17 shows the annotated AccountController class to be tested. Note that we name it AccountControllerAnnotated to distinguish it from the configuration-based version to be discussed next. For your Spring MVC controllers, you should pick either annotation-based or configuration-based without having *Annotated* or *Configured* to be part of the name of your controller).

First, note the annotations of @Controller, @Autowired, @RequestMapping, and @RequestParam that we covered in the earlier chapters. These annotations should look familiar to you and we would not repeat explaining them here. Next, note that this controller depends on an accountManager object, which is used to update the account balance by calling its updateAccountBalance method. This method can be multiple purposes, for example, customer initiated transactions such as *deposit* and *withdraw*, or transactions such as *debit* and *credit* conducted by the bank or third parties, depending on whether the amount parameter value is positive or negative (refer to Listing 9.12 for more details about this method). Finally, note the accountId and balance model attributes and the success *viewName* returned at the exit of this method call.

Listing 9.17 AccountControllerAnnotated.java

```
package com.perfmath.spring.soba.web;

import org.springframework.beans.factory.annotation.Autowired;
```

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

import com.perfmath.spring.soba.service.AccountManager;

@Controller
public class AccountControllerAnnotated {

  private AccountManager accountManager;
  @Autowired
  public AccountControllerAnnotated(AccountManager accountManager) {
    this.accountManager = accountManager;
  }
  @RequestMapping("/updateAccountBalance.htm")
   public String updateAccountBalance (
       @RequestParam("accountId") String accountId,
       @RequestParam("amount") double amount,
       ModelMap model) {
    double balance = accountManager.updateAccountBalance(amount, accountId);
    model.addAttribute("accountId", accountId);
    model.addAttribute("balance", balance);
    return "success";
  }
}
```

Listing 9.18 shows the `AccountControllerAnnotatedTest.java` class. Given what we have covered so far about TestNG and EasyMock, it should be easy to comprehend this test class. However, I have to mention that a common exception of "`missing behavior definition for the preceding method call`" as shown in Listing 9.19 would occur if the first two statements in the `updateAccountBalance` method were implemented as follows:

```
mock.updateAccountBalance (TEST_AMOUNT, TEST_ACCOUNT_ID);
expect (mock.getBalance(TEST_ACCOUNT_ID)).andReturn(NEW_BALANCE);
replay (mock);
```

That is because the `mock.updatAccountBalance` call has a `double` return type instead of `void`. It would work properly if the mock call prior to the `expect` statement had a `void` return type. Other than that, this class is straightforward, and you can just build and run it as you did with all previous TestNG-based examples. For your reference, Listing 9.20 shows the console output obtained from running this class in my Eclipse environment.

Listing 9.18 AccountControllerAnnotatedTest.java

```
import static org.junit.Assert.*;

import static org.easymock.EasyMock.*;

import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;
import org.springframework.ui.ModelMap;

import com.perfmath.spring.soba.service.AccountManager;
import com.perfmath.spring.soba.web.AccountControllerAnnotated;

public class AccountControllerAnnotatedTest {

   private static final String TEST_ACCOUNT_ID = "88889999";
   private static final double TEST_AMOUNT = 100;
   private static final double NEW_BALANCE = 200;
   private AccountManager mock;
   private AccountControllerAnnotated accountController;

   @BeforeMethod
   public void init() {
     mock = createMock (AccountManager.class);
      accountController = new AccountControllerAnnotated(mock);
   }

   @Test
   public void updateAccountBalance() {
      expect (mock.updateAccountBalance(TEST_AMOUNT,
         TEST_ACCOUNT_ID)).andReturn(NEW_BALANCE);
      replay(mock);

      ModelMap model = new ModelMap();
      String viewName =
         accountController.updateAccountBalance (TEST_ACCOUNT_ID, TEST_AMOUNT, model);
      verify(mock);

      assertEquals(viewName, "success");
      assertEquals(model.get("accountId"), TEST_ACCOUNT_ID);
      assertEquals(model.get("balance"), NEW_BALANCE);
   }
}
```

Listing 9.19 The exception caused by the improper use of mock objects

```
[TestNG] Running:
  C:\Users\henry\AppData\Local\Temp\testng-eclipse--701496993\testng-customsuite.xml

FAILED: updateAccountBalance
java.lang.IllegalStateException: missing behavior definition for the preceding method call:
AccountManager.updateAccountBalance(100.0, "88889999")
Usage is: expect(a.foo()).andXXX()
    at org.easymock.internal.MockInvocationHandler.invoke(MockInvocationHandler.java:42)
    at org.easymock.internal.ObjectMethodsFilter.invoke(ObjectMethodsFilter.java:85)
    at $Proxy6.getBalance(Unknown Source)
    at com.perfmath.spring.soba.testing.spring.AccountControllerAnnotatedTest.
        updateAccountBalance(AccountControllerAnnotatedTest.java:30)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)
    at sun.reflect.DelegatingMethodAccessorImpl.
        invoke(DelegatingMethodAccessorImpl.java:25)
    at java.lang.reflect.Method.invoke(Method.java:597)
    at org.testng.internal.MethodInvocationHelper.
        invokeMethod(MethodInvocationHelper.java:80)
    at org.testng.internal.Invoker.invokeMethod(Invoker.java:714)
    at org.testng.internal.Invoker.invokeTestMethod(Invoker.java:901)
    at org.testng.internal.Invoker.invokeTestMethods(Invoker.java:1231)
    at org.testng.internal.TestMethodWorker.invokeTestMethods(TestMethodWorker.java:128)
    at org.testng.internal.TestMethodWorker.run(TestMethodWorker.java:111)
    at org.testng.TestRunner.privateRun(TestRunner.java:767)
    at org.testng.TestRunner.run(TestRunner.java:617)
    at org.testng.SuiteRunner.runTest(SuiteRunner.java:334)
    at org.testng.SuiteRunner.runSequentially(SuiteRunner.java:329)
    at org.testng.SuiteRunner.privateRun(SuiteRunner.java:291)
    at org.testng.SuiteRunner.run(SuiteRunner.java:240)
    at org.testng.SuiteRunnerWorker.runSuite(SuiteRunnerWorker.java:52)
    at org.testng.SuiteRunnerWorker.run(SuiteRunnerWorker.java:86)
    at org.testng.TestNG.runSuitesSequentially(TestNG.java:1203)
    at org.testng.TestNG.runSuitesLocally(TestNG.java:1128)
    at org.testng.TestNG.run(TestNG.java:1036)
    at org.testng.remote.RemoteTestNG.run(RemoteTestNG.java:111)
    at org.testng.remote.RemoteTestNG.initAndRun(RemoteTestNG.java:204)
    at org.testng.remote.RemoteTestNG.main(RemoteTestNG.java:175)

===============================================
   Default test
   Tests run: 1, Failures: 1, Skips: 0
===============================================
```

Listing 9.20 Console output of running AccountControllerAnnotatedTest.java class

```
[TestNG] Running:
  C:\Users\henry\AppData\Local\Temp\testng-eclipse-480456488\testng-customsuite.xml

PASSED: updateAccountBalance

===============================================
    Default test
    Tests run: 1, Failures: 0, Skips: 0
===============================================
```

#2 Testing Configuration-based Spring MVC Controllers

Listing 9.21 shows the configuration-based `AccountController` to be tested. It has the same dependency on an `accountManager` object and is *HttpServlet*-based with the `accountId` and `amount` parameters extracted using the `ServletRequestUtils` class. It then calls the `accountManager's updateAccountBalance` with the `amount` and `accoundId` parameters extracted from the `HttpServletRequest` argument of the `handleRequestInternal` method. Finally, it returns a `ModelAndView` object with the *viewName*, *accountId* and *balance* attributes as specified.

Listing 9.21 AccountControllerConfigured.java

```
package com.perfmath.spring.soba.web;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.bind.ServletRequestUtils;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

import com.perfmath.spring.soba.service.AccountManager;

public class AccountControllerConfigured extends AbstractController{

    private AccountManager accountManager;

    public AccountControllerConfigured(AccountManager accountManager) {
        this.accountManager = accountManager;
    }

      public ModelAndView handleRequestInternal (HttpServletRequest request,
              HttpServletResponse response) throws Exception {
```

```
    String accountId =
          ServletRequestUtils.getRequiredStringParameter (request, "accountId");
    double amount =
          ServletRequestUtils.getRequiredDoubleParameter (request, "amount");
    double balance = accountManager.updateAccountBalance (amount, accountId);
    return new ModelAndView ("success", "accountId", accountId).addObject("balance",
       balance);
  }
}
```

Listing 9.22 shows the `AccountControllerConfiguredTest.java` class. Note that this test class uses Spring's `MockHttpServletRequest` and `MockHttpServletResponse` to facilitate testing this configuration-based `accountController`. The mock object is used similarly in terms of *expect*, *replay* and *verify*. However, instead of using `assertEquals` as is the case with the preceding class, it uses Spring-specific `assertViewName` and `assertModelAttributeValue` for asserting the `ModelAndView` object. Figure 9.4 shows the TestNG console output of running this test class, indicating that the test was executed successfully.

Listing 9.22 AccountControllerConfiguredTest.java

```
package com.perfmath.spring.soba.testing.spring;

import static org.springframework.test.web.ModelAndViewAssert.*;
import static org.easymock.EasyMock.*;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;
import org.springframework.mock.web.MockHttpServletRequest;
import org.springframework.mock.web.MockHttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import com.perfmath.spring.soba.service.AccountManager;
import com.perfmath.spring.soba.web.AccountControllerConfigured;

public class AccountControllerConfiguredTest {

  private static final String TEST_ACCOUNT_ID = "88889999";
  private static final double TEST_AMOUNT = 100.0;
  private static final double NEW_BALANCE = 200.0;

  private AccountManager mock;
  private AccountControllerConfigured accountController;

  @BeforeMethod
  public void init() {
    mock = createMock (AccountManager.class);
```

```
    accountController = new AccountControllerConfigured(mock);
  }

  @Test
  public void updateAccountBalance() throws Exception {
   MockHttpServletRequest request = new MockHttpServletRequest ();
   request.setMethod ("POST");
   request.addParameter ("accountId", TEST_ACCOUNT_ID);
   request.addParameter ("amount", String.valueOf(TEST_AMOUNT));
   MockHttpServletResponse response = new MockHttpServletResponse ();

    expect (mock.updateAccountBalance(TEST_AMOUNT,
      TEST_ACCOUNT_ID)).andReturn(NEW_BALANCE);
    replay(mock);

    ModelAndView modelAndView =
         accountController.handleRequest (request, response);
    verify (mock);

    assertViewName (modelAndView, "success");
    assertModelAttributeValue (modelAndView, "accountId", TEST_ACCOUNT_ID);
    assertModelAttributeValue (modelAndView, "balance", NEW_BALANCE);
  }
}
```

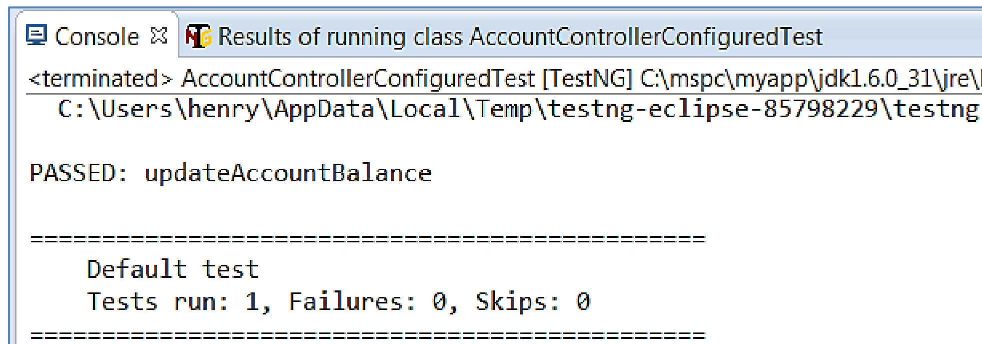


Figure 9.4 Console output of running class AccountControllerConfiguredTest

9.3.2 Spring's TestContext Support for Integration Testing

Recall that in §9.2.2, we tested a service layer class named `SimpleAccountManager.java` as shown in Listing 9.13. We used the Java `new` operator to create the dependency object `accountDao` in the TestNG's `init` method as follows:

accountDao = new InMemoryAccountDao();

By doing that, we have completed by-passed the application context configuration file. It's okay to test a pure POJO that way, but for most Spring beans, it's preferable to test them against their application contexts. Spring provides TestContext support to manage application context for this kind of integration testing. Specifically, the Spring TestContext framework provides the following two test execution listeners related to context management:

- **DependencyInjectionTestExecutionListener**: This listener helps inject dependencies into the tests.
- **DirtiesContextTestExecutionListener**: This listener handles the `@DirtiesContext` annotation, which may trigger the re-loading of the application context when necessary.

The good news is that these listeners are registered by default if we don't specify our own explicitly so that we don't have to worry about them. However, the test classes we create must implement the following interface or extend the following base class:

- The `ApplicationContextAware` for both TestNG and JUnit4**.**
- The `AbstractTestNGSpringContextTests` base class for TestNG or the `AbstractJUnit4SpringContextTests` base class for JUnit4.

Next, let's explore this Spring `TestContext` feature with the same `SimpleAccountManager.java` class we mentioned at the beginning of this section. Since the TestNG and JUnit test classes share the same application context configuration file, Listing 9.23 shows the `beans-app-context.xml` file to be used for the examples presented in this section. We start with the TestNG-based example for demonstrating Spring `TestContext` feature following Listing 9.23.

Listing 9.23 beans-app-context.xml file

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

  <bean id="accountDao"
    class="com.perfmath.spring.soba.testing.InMemoryAccountDao">
  </bean>

  <bean id="accountManager"
    class="com.perfmath.spring.soba.service.SimpleAccountManager">
    <property name="accountDao" ref="accountDao" />
  </bean>
</beans>
```

#1 TestNG-based example for demonstrating the Spring TestContext feature

Listing 9.24 shows the `AccountManagerTestNGContextAbstractTest.java` class, which demonstrates how the `AbstractTestNGSpringContextTests` class is extended with this test

class. First, note the `@ContextConfiguration (locations = "/beans-app-context.xml")` annotation applied to this class. This annotation instructs the Spring testing framework about where to find the application context configuration file, which is specified with the `locations` property as is shown. The locations are classpath locations relative to the test class by default. Keep in mind that the application context will be cached and reused for each test case if the test case under test is not annotated with the `@DirtiesContext` annotation to reload the application context. The `createAccount` and `updateAccountBalance` test cases are similar to what we discussed before, so we would not explain them here again. For the console output of running this class with TestNG on Eclipse, see Listing 9.25.

Listing 9.24 AccountManagerTestNGContextAbstractTest.java

```
package com.perfmath.spring.soba.testing.testng;

import static org.junit.Assert.assertEquals;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.AbstractTestNGSpringContextTests;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;
import com.perfmath.spring.soba.model.domain.Account;
import com.perfmath.spring.soba.service.AccountManager;

@ContextConfiguration(locations = "/beans-test-app-context.xml")
public class AccountManagerTestNGContextAbstractTest extends
    AbstractTestNGSpringContextTests {

  private static final String TEST_ACCOUNT_ID = "88889999";
  private static final double INITIAL_BALANCE = 500.0;
  private static final double TEST_AMOUNT = 300.0;
  private AccountManager accountManager;

  @BeforeMethod
  public void init() {
    accountManager = (AccountManager) applicationContext.getBean("accountManager");
  }

  @Test
  public void createAccount () {
    Account account = new Account ();
    account.setAccountId(TEST_ACCOUNT_ID);
    account.setBalance(INITIAL_BALANCE);
    accountManager.createAccount(account);
    assertEquals(accountManager.getBalance(TEST_ACCOUNT_ID), INITIAL_BALANCE, 0);
  }
```

```
  @Test
  public void updateAccountBalance () {
    accountManager.updateAccountBalance (TEST_AMOUNT, TEST_ACCOUNT_ID);
    assertEquals(accountManager.getBalance(TEST_ACCOUNT_ID),
          (INITIAL_BALANCE + TEST_AMOUNT), 0);
  }
}
```

Listing 9.25 Console output of running the AccountManagerTestNGAbstractTest class

```
[TestNG] Running:
  C:\Users\henry\AppData\Local\Temp\testng-eclipse--1205327052\testng-customsuite.xml

PASSED: createAccount
PASSED: updateAccountBalance

===============================================
   Default test
   Tests run: 2, Failures: 0, Skips: 0
===============================================
```

#2 JUnit4-based example for demonstrating the Spring TestContext feature

We have an `AccountManagerTJUnit4ContextAbstractTest.java` class in SOBA's `testing/junit` package, but it's almost identical with the TestNG-based example we have just presented above except that it extends the `AbstractJUni4SpringContextTests` interface rather than the `AbstractTestNGSpringContextTests` base class. Therefore, we present a JUnit 4 example that implements the `ApplicationContextAware` interface next.

Listing 9.26 demonstrates how the `ApplicationContextAware` interface is implemented with this test class. First, in addition to the `@ContextConfiguration (locations = "/beans-app-context.xml")` annotation as we saw in the previous example, note the `@RunWith` annotation, which instructs the Spring testing framework explicitly to use this test runner of `SpringJUnit4ClassRunner` to run this test. Next, note the `applicationContext` attribute and its setter method. These features are unique with the test classes that implement the `ApplicationContextAware` interface. Similarly, the `locations` for the application context configuration file are classpath locations relative to the test class by default. The `createAccount` and `updateAccountBalance` test cases are similar to what we discussed before, so we would not explain them here again. For the console output of running this class with JUnit4 on Eclipse, see Figure 9.5.

Listing 9.26 AccountManagerJUnit4ContextAwareTest.java

```
package com.perfmath.spring.soba.testing.junit;
```

```
import static org.junit.Assert.assertEquals;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import com.perfmath.spring.soba.service.AccountManager;
import com.perfmath.spring.soba.model.domain.Account;

@RunWith (SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "/beans-test-app-context.xml")
public class AccountManagerJUnit4ContextAwareTest implements
    ApplicationContextAware {

  private static final String TEST_ACCOUNT_ID = "88889999";
  private static final double INITIAL_BALANCE = 500.0;
  private static final double TEST_AMOUNT = 300.0;
  private ApplicationContext applicationContext;
  private AccountManager accountManager;

  public void setApplicationContext (ApplicationContext applicationContext ) {
   this.applicationContext = applicationContext;
  }

  @Before
  public void init() {
    accountManager = (AccountManager) applicationContext.getBean("accountManager");
  }

  @Test
  public void createAccount () {
    Account account = new Account ();
    account.setAccountId(TEST_ACCOUNT_ID);
    account.setBalance(INITIAL_BALANCE);
    accountManager.createAccount(account);
    assertEquals(accountManager.getBalance(TEST_ACCOUNT_ID), INITIAL_BALANCE, 0);
  }

  @Test
  public void updateAccountBalance () {
    accountManager.updateAccountBalance (TEST_AMOUNT, TEST_ACCOUNT_ID);
    assertEquals(accountManager.getBalance(TEST_ACCOUNT_ID),
```

```
            (INITIAL_BALANCE + TEST_AMOUNT), 0);
    }
}
```

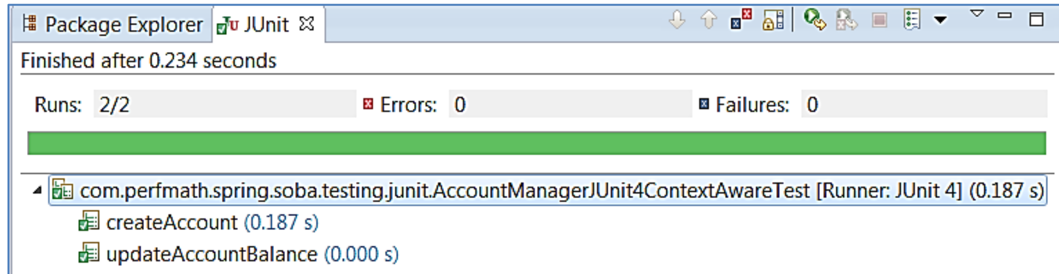


Figure 9.5 Console output of running AccountManagerJUnit4ContextAwareTest class

9.3.3 Automatically Injecting Spring Test Fixtures into Integration Tests

Note in the test cases presented in the preceding section, such as Listing 9.26 `AccountManagerJUnit4ContextAwareTest.java`, that the `accountManager` dependency bean needs to be injected into the test class explicitly in its `init` method with the following statement:

```
@Before
    public void init() {
        accountManager = (AccountManager) applicationContext.getBean("accountManager");
    }
```

Such dependency beans are called *test fixtures* and in fact can be injected into the test classes automatically. The fixture can be injected by type with the `@Autowired` annotation or by name with the `@Resource` annotation. Listing 9.27 shows how this feature works exactly with the `AccountManagerTestNGContextFixtureAbstractTest.java` class, which is retrofitted from Listing 9.24 `AccountManagerTestNGContextAbstractTest.java` presented in §9.3.2. Note that the `@Autowired` annotation, applied to the `accountManager` dependency bean, has eliminated the need to create the `accountManager` bean in the `init` method of the test class. Refer to Listing 9.28 for the console output of running this class. Note that this feature applies to JUnit as well, but we would not repeat it here.

Listing 9.27 AccountManagerTestNGContextFixtureAbstractTest.java

```
package com.perfmath.spring.soba.testing.testng;

import static org.junit.Assert.assertEquals;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.AbstractTestNGSpringContextTests;
import org.testng.annotations.BeforeMethod;
```

```
import org.testng.annotations.Test;
import com.perfmath.spring.soba.model.domain.Account;
import com.perfmath.spring.soba.service.AccountManager;

@ContextConfiguration(locations = "/beans-test-app-context.xml")
public class AccountManagerTestNGContextFixtureAbstractTest extends
AbstractTestNGSpringContextTests {

   private static final String TEST_ACCOUNT_ID = "88889999";
   private static final double INITIAL_BALANCE = 500.0;
   private static final double TEST_AMOUNT = 300.0;
   @Autowired
   private AccountManager accountManager;

   @BeforeMethod
   public void init() {
   }

   @Test
   public void createAccount () {
      Account account = new Account ();
      account.setAccountId(TEST_ACCOUNT_ID);
      account.setBalance(INITIAL_BALANCE);
      accountManager.createAccount(account);
      assertEquals(accountManager.getBalance(TEST_ACCOUNT_ID), INITIAL_BALANCE, 0);
   }

   @Test
   public void updateAccountBalance () {
      accountManager.updateAccountBalance (TEST_AMOUNT, TEST_ACCOUNT_ID);
      assertEquals(accountManager.getBalance(TEST_ACCOUNT_ID),
            (INITIAL_BALANCE + TEST_AMOUNT), 0);
   }
}
```

Listing 9.28 The console output of running AccountManagerTestNGContextFixtureAbstractTest class

```
[TestNG] Running:
  C:\Users\henry\AppData\Local\Temp\testng-eclipse-583467160\testng-customsuite.xml

PASSED: createAccount
PASSED: updateAccountBalance
```

```
===============================================
   Default test
   Tests run: 2, Failures: 0, Skips: 0
===============================================
```

9.3.4 Spring's Transactional Support for Integration Testing

So far, we have been using either mock objects or an in-memory database to facilitate integration testing. It might be more meaningful to use real databases to test Spring-based enterprise applications at the integration testing level to close the gap between artificiality and reality. In this section, we explore Spring's transactional support for integration testing with real data persistence, while in the next section, we explore Spring's support for conducting integration testing using real databases to facilitate assertions.

Spring 3.x provides two abstract base classes to support transactionality for conducting JUni4-based and TestNG-based integration testing. For JUnit4, the abstract base class is named `AbstractTransactionalJUnit4SpringContextTests`, while for TestNG, the counterpart abstract base class is named `AbstractTransactionalTestNGSpringContextTests`. These classes are registered with a test context manager by default if it's not specified explicitly. In addition, a Spring `TransactionalTestExecutionListener` makes it possible to use `@Transactional` annotation to mark a class or method to be transactional. By default, those two abstract base classes for the JUnit4 and TestNG frameworks enable a test class with @Transactional at the class level, meaning that all the test cases will be transactional. However, keep in mind that when we say a test class or method is transactional in Spring's test context, what that really means is that all persistence related operations will be rolled back at the end of a test so that the state of the database remains the same before and after the test. This helps save us time restoring database to its state before testing if we need to make multiple tests repeatable.

To proceed, we need to create an application context configuration file and have a JDBC Dao class in place. Listing 9.29 shows the application context configuration file for the examples presented in this section, while Listing 9.30 shows the `JdbcAccountDao.java` class we will use for the examples in this section and in the next section. This application context configuration file and the `JdbcAccountDao.java` class should look familiar to you, so we would not spend time here explaining how they work. However, note that we have to specify a `transactionManager` in the `beans-test-tx.xml` file to support the transactionality under test.

Listing 9.29 beans-test-tx.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd">

   <bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
```

```
    <property name="driverClassName"
      value="com.mysql.jdbc.Driver" />
    <property name="url"
      value="jdbc:mysql://localhost:3306/soba32" />
    <property name="username" value="soba32admin" />
    <property name="password" value="soba32admin" />
  </bean>

  <bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
  </bean>

  <bean id="accountDao"
    class="com.perfmath.spring.soba.model.dao.JdbcAccountDao">
    <property name="dataSource" ref="dataSource" />
  </bean>

  <bean id="accountManager"
    class="com.perfmath.spring.soba.service.SimpleAccountManager">
    <property name="accountDao" ref="accountDao" />
  </bean>
</beans>
```

Listing 9.30 JdbcAccountDao.java

```
package com.perfmath.spring.soba.model.dao;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;
import java.util.Map;
import javax.sql.DataSource;
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSourceUtils;
import org.springframework.jdbc.core.simple.ParameterizedRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;
import com.perfmath.spring.soba.model.domain.Account;

public class JdbcAccountDao  implements AccountDao {
    private JdbcTemplate jdbcTemplate;
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;
```

```
public List<Account> getAccountList() {
    List<Account> accounts = this.jdbcTemplate.query("SELECT ACCOUNT_ID, NAME, TYPE,
        DESCRIPTION, STATUS, "
                + " BALANCE, OPEN_DATE, CLOSE_DATE, CUSTOMER_ID FROM ACCOUNT",
                new AccountMapper());
    return accounts;
}

public String getAccountIdByCustomerId(String customerId, String accountType) {
    String sql = "SELECT ACCOUNT_ID FROM ACCOUNT WHERE CUSTOMER_ID = ?
        AND TYPE = ?";

    String accountId = this.jdbcTemplate.queryForObject(sql,
            String.class, customerId, accountType);
    return accountId;
}

public void insert(Account account) {
    String sql = "INSERT INTO ACCOUNT (ACCOUNT_ID, NAME, TYPE, DESCRIPTION, STATUS, "
            + " BALANCE, OPEN_DATE, CLOSE_DATE, CUSTOMER_ID) "
            + "VALUES (:accountId, :name, :type,:description, "
            + ":status, :balance, :openDate, :closeDate, :customerId)";

    SqlParameterSource parameterSource = new BeanPropertySqlParameterSource(
            account);

    int count = this.namedParameterJdbcTemplate.update(sql, parameterSource);
}

public void insertBatch(List<Account> accounts) {
    String sql = "INSERT INTO ACCOUNT (ACCOUNT_ID, NAME, TYPE, DESCRIPTION, STATUS, "
            + " BALANCE, OPEN_DATE, CLOSE_DATE, CUSTOMER_ID) "
            + "VALUES (:accountId, :name, :type,:description, "
            + ":status, :balance, :openDate, :closeDate, :customerId)";
    SqlParameterSource[]  batch = SqlParameterSourceUtils.createBatch(accounts.toArray());
    namedParameterJdbcTemplate.batchUpdate (sql, batch);
}

public Account findByAccountID(String accountID) {
    String sql = "SELECT ACCOUNT_ID, NAME, TYPE, DESCRIPTION, STATUS, BALANCE, " +
            "  OPEN_DATE, CLOSE_DATE, CUSTOMER_ID FROM ACCOUNT WHERE
                ACCOUNT_ID = ?";
    Account account = this.jdbcTemplate.queryForObject(sql, new AccountMapper(),
            accountID);
```

```
        return account;
    }

    public void update(Account account) {
        String sql = "UPDATE ACCOUNT SET NAME = ?, SET TYPE = ?," +
                " SET DESCRIPTION = ?, SET STATUS = ?, SET BALANCE = ?," +
                " SET OPEN_DATE = ?, SET CLOSE_DATE = ?, SET CUSTOMER_ID = ? " +
                " WHERE ACCOUNT_ID = ?";
        this.jdbcTemplate.update(sql, account.getName(), account.getType(),
                account.getDescription(), account.getStatus(), account.getBalance(),
                account.getOpenDate(), account.getCloseDate(), account.getCustomerId(),
                account.getAccountId());
    }
    public double updateAccountBalance (double amount, String accountID) {
        String sql0 = "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = ?";

        Double Balance = this.jdbcTemplate.queryForObject(sql0,
                Double.class, accountID);
        double currBalance = Balance.doubleValue();

        String sql1 = "UPDATE ACCOUNT SET BALANCE = ? WHERE ACCOUNT_ID = ?";
        int count = this.jdbcTemplate.update(sql1, (amount + currBalance), accountID);
        return (amount + currBalance);
    }

    public void delete(Account account) {
        String sql = "DELETE FROM ACCOUNT WHERE ACCOUNT_ID = ?";
        this.jdbcTemplate.update(sql, account.getAccountId());
    }

    public List<Map<String, Object>> findAll() {
        String sql = "SELECT * FROM ACCOUNT";

        List<Map<String, Object>> accounts = this.jdbcTemplate.queryForList(sql, new
            AccountMapper());
        return accounts;
    }

    public String getCustomerId(String accountID) {
        String sql = "SELECT CUSTOMER_ID FROM ACCOUNT WHERE ACCOUNT_ID = ?";

        String customerId = this.jdbcTemplate.queryForObject(sql,
                String.class, accountID);
        return customerId;
```

```
    }
    public double getBalance (String accountID) {
        String sql = "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = ?";

        double balance = this.jdbcTemplate.queryForObject(sql, Double.class, accountID);
        return balance;
    }
    public int countAll() {
        String sql = "SELECT COUNT(*) FROM ACCOUNT";

        int count = this.jdbcTemplate.queryForInt(sql);
        return count;
    }
    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
        this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate (dataSource);
    }
    private static class AccountMapper implements
            ParameterizedRowMapper<Account> {
        public Account mapRow(ResultSet rs, int rowNum) throws SQLException {
            Account account = new Account();
            account.setAccountId(rs.getString("ACCOUNT_ID"));
            account.setName(rs.getString("NAME"));
            account.setType(rs.getString("TYPE"));
            account.setDescription(rs.getString("DESCRIPTION"));
            account.setStatus(rs.getString("STATUS"));
            account.setBalance(rs.getInt("BALANCE"));
            account.setOpenDate(rs.getTimestamp("OPEN_DATE"));
            account.setCloseDate(rs.getTimestamp("CLOSE_DATE"));
            account.setCustomerId(rs.getString("CUSTOMER_ID"));
            return account;
        }
    }
}
```

#1 Transactional integration testing with JUnit4

Our first example in this section is the `AccountManagerJUnit4ContextTxTest.java` as shown in Listing 9.31. First, note the `@RunWith` annotation, which specifies that the `SpringJUnit4ClassRunner` class should be used as the test runner. Next, note the `@Transactional` annotation as we explained at the beginning of this section. Another annotation of `@NotTransactional` used to be available for marking the test cases that are expected to be non-transactional, but has been deprecated as of Spring 3.0. Finally, note that this test class does not need to extend an abstract base class. I have run this test class in my environment successfully

as indicated in Figure 9.6. I have also double checked my MySQL database and verified that no traces left in the database by this test run.

Listing 9.31 AccountManagerJUnit4ContextTxTest.java

```
package com.perfmath.spring.soba.testing.junit;

import static org.junit.Assert.assertEquals;
import java.sql.Timestamp;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.transaction.annotation.Transactional;
import com.perfmath.spring.soba.model.domain.Account;
import com.perfmath.spring.soba.service.AccountManager;

@RunWith (SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "/beans-test-tx.xml")
@Transactional
public class AccountManagerJUnit4ContextTxTest {

  private static final String TEST_ACCOUNT_ID = "88889999";
  private static final double INITIAL_BALANCE = 500.0;
  private static final double TEST_AMOUNT = 300.0;
  @Autowired
  private AccountManager accountManager;

  @Before
  public void init() {
  }

  @Test
  public void createAccount () {
    Account account = new Account ();
    account.setAccountId(TEST_ACCOUNT_ID);
    account.setCustomerId("585855478");
    account.setName("Testing");
    account.setType("Testing");
    account.setDescription("Spring integration testing");
    account.setBalance(INITIAL_BALANCE);
    account.setStatus("0");
```

```
    account.setOpenDate(new Timestamp(System.currentTimeMillis()));
    accountManager.createAccount(account);
    assertEquals(accountManager.getBalance(TEST_ACCOUNT_ID), INITIAL_BALANCE, 0);
  }
  @Test
  public void updateAccountBalance () {
   createAccount ();
    double balance = accountManager.updateAccountBalance (TEST_AMOUNT,
       TEST_ACCOUNT_ID);
    assertEquals(accountManager.getBalance(TEST_ACCOUNT_ID),
       (INITIAL_BALANCE + TEST_AMOUNT), 0);
  }
}
```

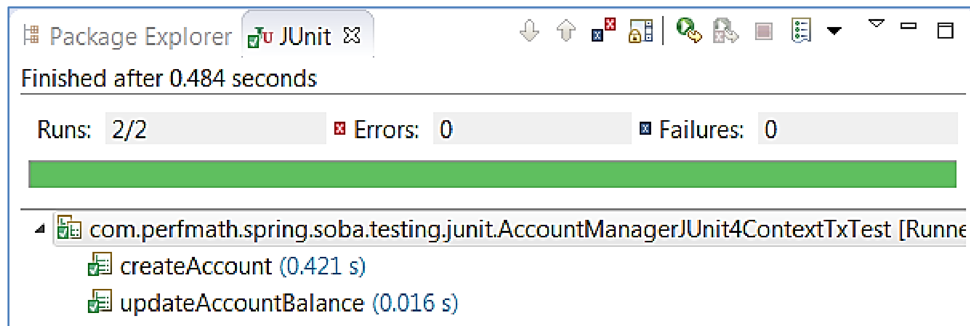


Figure 9.6 Successful execution of a transactional test case with JUnit 4

Note that the `@RunWith` and `@Transactional` annotations can be avoided if the test class as shown in 9.31 extends the `AbstractTransactionalJUnit4SpringContextTests` base class. You can find the `AccountManagerJUnit4ContextAbstractTxTest.java` class in the same package as the `AccountManagerJUnit4ContextTxTest.java` class and try it out yourself if you are interested in it. This is a trivial change and we would not repeat it here.

#2 Transactional integration testing with TestNG

Our second example is the `AccountManagerTestNGAbstractContextTxTest.java` as shown in Listing 9.32. Since it extends the `AbstractTransactionalTestNGSpringContextTests` base class, this test class does not need the `@RunWith` annotation and the `@Transactional` annotation. I have run this test class in my environment and got successful results for the two test cases as indicated in Listing 9.33. I have also double checked my MySQL database and verified that no traces left in the database by this test run.

Listing 9.32 AccountManagerTestNGAbstractContextTxTest.java

```
package com.perfmath.spring.soba.testing.testng;
```

```
import java.sql.Timestamp;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import
org.springframework.test.context.testng.AbstractTransactionalTestNGSpringContextTests;
import static org.junit.Assert.assertEquals;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;
import com.perfmath.spring.soba.model.domain.Account;
import com.perfmath.spring.soba.service.AccountManager;

@ContextConfiguration(locations = "/beans-test-tx.xml")
public class AccountManagerTestNGContextAbstractTxTest extends
    AbstractTransactionalTestNGSpringContextTests {

  private static final String TEST_ACCOUNT_ID = "88889999";
  private static final double INITIAL_BALANCE = 500.0;
  private static final double TEST_AMOUNT = 300.0;
  @Autowired
  private AccountManager accountManager;

  @BeforeMethod
  public void init() {
  }

  @Test
  public void createAccount () {
    Account account = new Account ();
    account.setAccountId(TEST_ACCOUNT_ID);
    account.setCustomerId("585855478");
    account.setName("Testing");
    account.setType("Testing");
    account.setDescription("Spring integration testing");
    account.setBalance(INITIAL_BALANCE);
    account.setStatus("0");
    account.setOpenDate(new Timestamp(System.currentTimeMillis()));
    accountManager.createAccount(account);
    assertEquals(accountManager.getBalance(TEST_ACCOUNT_ID), INITIAL_BALANCE, 0);
  }

  @Test
  public void updateAccountBalance () {
   createAccount ();
```

```
    accountManager.updateAccountBalance (TEST_AMOUNT, TEST_ACCOUNT_ID);
    assertEquals(accountManager.getBalance(TEST_ACCOUNT_ID),
          (INITIAL_BALANCE + TEST_AMOUNT), 0);
  }
}
```

Listing 9.33 Console output of running AccountManagerTestNGContextAbstractTxTest

```
[TestNG] Running:
  C:\Users\henry\AppData\Local\Temp\testng-eclipse--1910367704\testng-
customsuite.xml

PASSED: createAccount
PASSED: updateAccountBalance

===============================================
    Default test
    Tests run: 2, Failures: 0, Skips: 0
===============================================
```

9.3.5 Spring Integration Testing against Real Databases

In the previous section, we discussed how to make a test case and its methods *transactional* by having the test case extend either `AbstractTransactionalJUnit4SpringContextTests` base class for JUnit4 or the `AbstractTransactionalTestNGSpringContextTests` base class for TestNG. However, note from both Listing 9.31 and 9.32 that `assertEquals` are done by calling the proper method of the dependent object. In fact, one can issue SQLs directly to prepare data or retrieve results from the database for verification purposes if such needs arise. Listing 9.34 shows an example of extending the `AbstractTransactionalTestNGSpringContextTests` base class for TestNG (Junit4 case would be similar). Note those Spring `jdbcTemplate` API calls in the `init` method for creating the account and add an initial balance to it. Then, the Spring `jdbcTemplate.queryForObject` calls are made in the `deposit` and `withdraw` methods to verify the balance in the account.

This test class as shown in Listing 9.34 was executed successfully in my environment with the console output as shown in Listing 9.35. There is a similar JUnit4-based test case named `AccountManagerJUnit4ContextDBTest.java` in the `testing.junit` package, and you can try it out if you are interested.

Listing 9.34 AccountMangerTestNGContextDBTest.java

```
package com.perfmath.spring.soba.testing.testng;

import static org.testng.Assert.assertEquals;
import java.sql.Timestamp;
```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import
org.springframework.test.context.testng.AbstractTransactionalTestNGSpringContextTests;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

import com.perfmath.spring.soba.service.AccountManager;

@ContextConfiguration(locations = "/beans-test-tx.xml")
public class AccountManagerTestNGContextDBTest extends
    AbstractTransactionalTestNGSpringContextTests  {

  private static final String TEST_ACCOUNT_ID = "88889999";
  private static final double INITIAL_BALANCE = 500.0;
  private static final double TEST_AMOUNT = 300.0;

  @Autowired
  private AccountManager accountManager;

  @BeforeMethod
  public void init() {
    jdbcTemplate.update ( "INSERT INTO ACCOUNT (ACCOUNT_ID, NAME, TYPE,
       DESCRIPTION, STATUS, "
              + " BALANCE, OPEN_DATE, CLOSE_DATE, CUSTOMER_ID) "
              + "VALUES (?, ?, ?,?, ?, ?, ?, ?, ?)", TEST_ACCOUNT_ID,
              "Testing", "Testing", "Spring integration testing",  "0", INITIAL_BALANCE,
              new Timestamp(System.currentTimeMillis()), null, "585855478");
   jdbcTemplate.update ("UPDATE ACCOUNT SET BALANCE = ? WHERE ACCOUNT_ID = ?",
              (INITIAL_BALANCE + TEST_AMOUNT), TEST_ACCOUNT_ID);
  }

  @Test
  public void deposit() {
    accountManager.updateAccountBalance (TEST_AMOUNT, TEST_ACCOUNT_ID);
    double balance = jdbcTemplate.queryForObject (
         "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = ?",
         Double.class, TEST_ACCOUNT_ID);
    assertEquals(balance, (INITIAL_BALANCE + 2*TEST_AMOUNT), 0);
  }

  @Test
  public void withdraw() {
    accountManager.updateAccountBalance (-TEST_AMOUNT, TEST_ACCOUNT_ID);
```

```
        double balance = jdbcTemplate.queryForObject (
            "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = ?",
            Double.class, TEST_ACCOUNT_ID);
        assertEquals(balance, INITIAL_BALANCE, 0);
    }
}
```

Listing 9.35 Console output of running AccountManagerTestNGContextDBTest

```
[TestNG] Running:
  C:\Users\henry\AppData\Local\Temp\testng-eclipse--598883244\testng-customsuite.xml

PASSED: deposit
PASSED: withdraw

===============================================
    Default test
    Tests run: 2, Failures: 0, Skips: 0
===============================================
```

9.4 SPRING JUNIT TESTING ANNOTATIONS

Spring supports annotations such as @Timed, @Repeat (n), @IfProfileValue, and so on. However, according to Spring Reference Documentation 3.2, these annotations are supported on JUnit only; or in other words, they are not supported on TestNG. In this section, we present a simple example to demonstrate how some of these annotations work.

Listing 9.36 shows an example, which uses: (1) the @Timed annotation to limit that the deposit method should complete its execution within 900 milliseconds, and (2) the @Repeat (5) annotation to specify that the withDraw method should repeat five times. Because how these annotations work is so obvious, I'd like to show the result of running this test class in my environment in Figure 9.7, which verifies that the deposit method completed within 359 milliseconds. This wraps up our coverage on Spring testing support.

Listing 9.36 AccountManagerJUnit4ContextCommonTest .java

```
package com.perfmath.spring.soba.testing.junit;
import static org.junit.Assert.assertEquals;
import java.sql.Timestamp;
import org.junit.Before;
import org.junit.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.annotation.Repeat;
import org.springframework.test.annotation.Timed;
```

```
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.AbstractTransactionalJUnit4SpringContextTests;
import com.perfmath.spring.soba.service.AccountManager;

@ContextConfiguration(locations = "/beans-test-tx.xml")
public class AccountManagerJUnit4ContextCommonTest extends
    AbstractTransactionalJUnit4SpringContextTests {

  private static final String TEST_ACCOUNT_ID = "88889999";
  private static final double INITIAL_BALANCE = 1500.0;
  private static final double TEST_AMOUNT = 300.0;

  @Autowired
  private AccountManager accountManager;

  @Before
  public void init () {
    jdbcTemplate.update( "INSERT INTO ACCOUNT (ACCOUNT_ID, NAME, TYPE,
             DESCRIPTION, STATUS, "
             + " BALANCE, OPEN_DATE, CLOSE_DATE, CUSTOMER_ID) "
             + "VALUES (?, ?, ?,?, ?, ?, ?, ?, ?)", TEST_ACCOUNT_ID,
             "Testing", "Testing", "Spring integration testing",  "0", INITIAL_BALANCE,
             new Timestamp(System.currentTimeMillis()), null, "585855478");
      jdbcTemplate.update("UPDATE ACCOUNT SET BALANCE = ? WHERE ACCOUNT_ID = ?",
             (INITIAL_BALANCE + TEST_AMOUNT), TEST_ACCOUNT_ID);
  }

  @Test
  @Timed(millis = 900)
  public void deposit() {
    accountManager.updateAccountBalance (TEST_AMOUNT, TEST_ACCOUNT_ID);
    double balance = jdbcTemplate.queryForObject(
        "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = ?",
        Double.class, TEST_ACCOUNT_ID);
    assertEquals(balance, (INITIAL_BALANCE + 2*TEST_AMOUNT), 0);
  }

  @Test
  @Repeat(3)
  public void withDraw() {
    accountManager.updateAccountBalance (-TEST_AMOUNT, TEST_ACCOUNT_ID);
    double balance = jdbcTemplate.queryForObject(
        "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_ID = ?",
        Double.class, TEST_ACCOUNT_ID);
```

```
        assertEquals(balance, INITIAL_BALANCE, 0);
    }
}
```

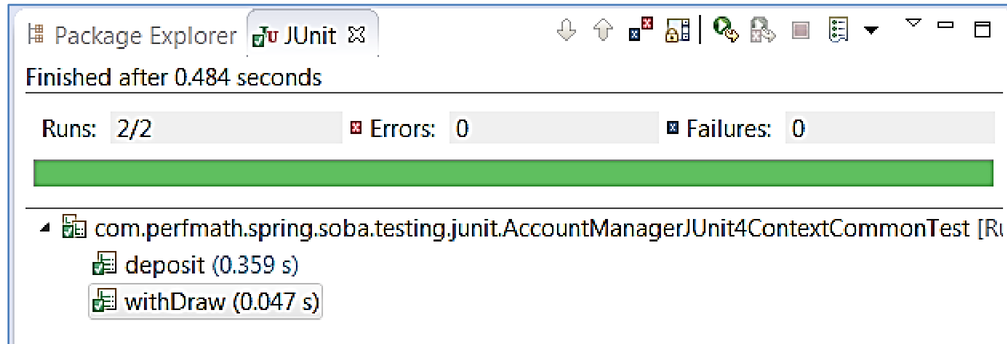


Figure 9.7 Test result of running AccountManagerJUnit4ContextCommonTest

9.5 SUMMARY

This chapter covered Spring testing support for testing Spring MVC controllers, service and DAO layer Spring beans. We presented test cases in association with both JUnit4 and TestNG Java testing frameworks, as Spring testing framework really is not a standalone testing framework. We demonstrated three most important Spring testing framework features: TestContext, automatically injecting Spring test fixtures, and transaction-related annotations. These features should meet most of your testing needs for your Spring-based project. Another equally important subject is test automation, which is much needed for every large-scale enterprise application. However, a detailed coverage of such a broad topic is beyond the context and scope of this book.

9.6 RECOMMENDED READING

Chapter 11 of the Spring Framework Reference Documentation 3.2 is dedicated to Spring testing. It includes the following sections that you can reference if you need more information on certain specific features:

- **11.1 Introduction to Spring Testing**
- **11.2 Unit Testing**
- **11.3 Integration Testing**
- 11.4 Further Resources

9.7 EXERCISES

9.1 If you have experience with both JUnit and TestNG, what are the pros and cons of each test framework from your perception, and which one is in use with your project (and what was the main reason for choosing one over the other)?

9.2 What are the advantages and disadvantages of using some artifacts such as mock objects and in-memory databases in place of real backend layers?

9.3 What is test context and how is it managed in Spring's testing framework? How does it facilitate testing Spring beans?

9.4 What issue does the Spring test fixture injection attempt to resolve? How do you inject test fixtures automatically into Spring-specific test classes?

9.5 Which abstract base classes do the test classes need to extend to enable transactionality to work with the JUnit4 and TestNG testing frameworks?