Chapter 3. Developing the Business Logic

This is Part 3 of a step-by-step tutorial on how to develop a Spring application. In this section, we will adopt a pragmatic Test-Driven Development (TDD) approach for creating the domain objects and implementing the business logic for our inventory management system. This means we'll "code a little, test a little, code some more then test some more". In Part 1 we configured the environment and set up a basic application. In Part 2 we refined the application by decoupling the view from the controller.

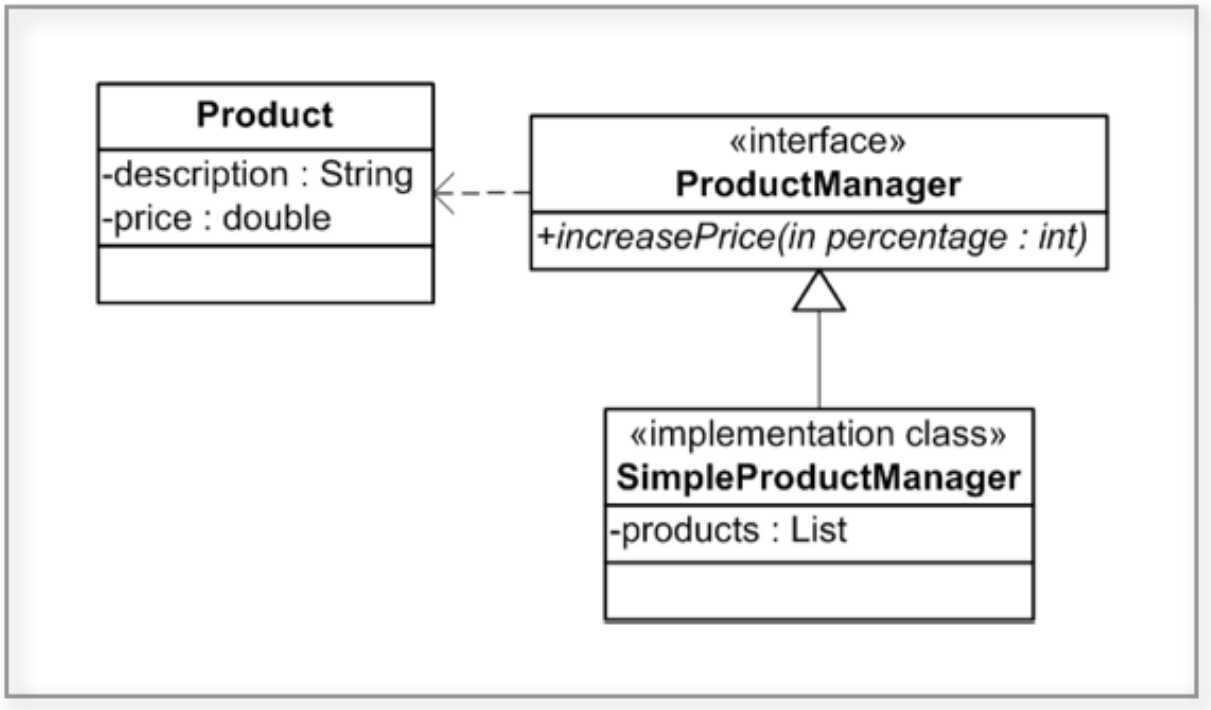
Spring is about making simple things easy and the hard things possible. The fundamental construct that makes this possible is Spring's use of Plain Old Java Objects (POJOs). POJOs are essentially plain old Java classes free from any contract usually enforced by a framework or component architecture through subclassing or the implementation of interfaces. POJOs are plain old objects that are free from such constraints, making object-oriented programming possible once again. When you are working with Spring, the domain objects and services you implement will be POJOs. In fact, almost everything you implement should be a POJO. If it's not, you should be sure to ask yourself why that is. In this section, we will begin to see the simplicity and power of Spring.

3.1. Review the business case of the Inventory Management System

In our inventory management system, we have the concept of a product and a service for handling them. In particular, the business has requested the ability to increase prices across all products. Any decrease will be done on an individual product basis, but this feature is outside the scope of our application. The validation rules for price increase are:

- The maximum increase is limited to 50%.
- The minimum increase must be greater than 0%.

Find below a class diagram of our inventory management system.



The class diagram for the inventory management system

3.2. Add some classes for business logic

Let's now add some business logic in the form of a `Product` class and a service called `ProductManager` service that will manage all the products. In order to separate the web dependent logic from the business logic, we will place classes related to the web tier in the `'web'` package and create two new packages: one for service objects called `'service'` and another for domain objects called `'domain'`.

First we implement the `Product` class as a POJO with a default constructor (automatically provided if we don't specify any constructors) and getters and setters for its properties `'description'` and `'price'`. Let's also make it `Serializable`, not necessary for our application, but could come in handy later on when we persist and store its state. The class is a domain object, so it belongs in the `'domain'` package.

`'springapp/src/springapp/domain/Product.java'`:

```
package springapp.domain;

import java.io.Serializable;

public class Product implements Serializable {

    private String description;
    private Double price;

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
```

```
    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    public String toString() {
        StringBuffer buffer = new StringBuffer();
        buffer.append("Description: " + description + ";");
        buffer.append("Price: " + price);
        return buffer.toString();
    }
}
```

Now we write the unit tests for our `Product` class. Some developers don't bother writing tests for getters and setters or so-called 'auto-generated' code. It usually takes much longer to engage in the debate (as this paragraph demonstrates) on whether or not getters and setters need to be unit tested as they're so 'trivial'. We write them because: a) they are trivial to write; b) having the tests pays dividends in terms of the time saved for the one time out of a hundred you may be caught out by a dodgy getter or setter; and c) they improve test coverage. We create a `Product` stub and test each getter and setter as a pair in a single test. Usually, you will write one or more test methods per class method, with each test method testing a particular condition in a class method such as checking for a `null` value of an argument passed into the method.

'springapp/test/springapp/domain/ProductTests.java':

```
package springapp.domain;

import junit.framework.TestCase;

public class ProductTests extends TestCase {

    private Product product;

    protected void setUp() throws Exception {
        product = new Product();
    }

    public void testSetAndGetDescription() {
        String testDescription = "aDescription";
        assertNull(product.getDescription());
        product.setDescription(testDescription);
        assertEquals(testDescription, product.getDescription());
    }

    public void testSetAndGetPrice() {
        double testPrice = 100.00;
        assertEquals(0, 0, 0);
        product.setPrice(testPrice);
        assertEquals(testPrice, product.getPrice(), 0);
    }

}
```

Next we create the `ProductManager`. This is the service responsible for handling products. It contains two methods: a business method `increasePrice()` that increases prices for all products and a getter method `getProducts()` for retrieving all products. We have chosen to make it an interface instead of a concrete class for an number of reasons. First of all, it makes writing unit tests for `Controllers` easier (as we'll see in the next chapter). Secondly, the use of interfaces means JDK proxying (a Java language feature) can be used to make the service transactional instead of CGLIB (a code generation library).

'springapp/src/springapp/service/ProductManager.java':

```
package springapp.service;

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

import springapp.domain.Product;

public interface ProductManager extends Serializable{

    public void increasePrice(int percentage);

    public List<Product> getProducts();

}
```

Let's create the `SimpleProductManager` class that implements the `ProductManager` interface.

'springapp/src/springapp/service/SimpleProductManager.java':

```
package springapp.service;

import java.util.List;

import springapp.domain.Product;

public class SimpleProductManager implements ProductManager {

    public List<Product> getProducts() {
        throw new UnsupportedOperationException();
    }

    public void increasePrice(int percentage) {
        throw new UnsupportedOperationException();
    }

    public void setProducts(List<Product> products) {
        throw new UnsupportedOperationException();
    }

}
```

Before we implement the methods in `SimpleProductManager`, we're going to define some tests first. The strictest definition of `Test Driven Development` (TDD) is to always write the tests first, then the code. A looser interpretation of it is more akin to `Test Oriented Development` (TOD), where we alternate between writing code and tests as part of the development process. The most important thing is for a codebase

to have as complete a set of unit tests as possible, so how you achieve it becomes somewhat academic. Most TDD developers, however, do agree that the quality of tests is always higher when they are written at around the same time as the code that is being developed, so that's the approach we're going to take.

To write effective tests, you have to consider all the possible pre- and post-conditions of a method being tested as well as what happens within the method. Let's start by testing a call to `getProducts()` returns `null`.

'springapp/test/springapp/service/SimpleProductManagerTests.java':

```
package springapp.service;

import junit.framework.TestCase;

public class SimpleProductManagerTests extends TestCase {

    private SimpleProductManager productManager;

    protected void setUp() throws Exception {
        productManager = new SimpleProductManager();
    }

    public void testGetProductsWithNoProducts() {
        productManager = new SimpleProductManager();
        assertNull(productManager.getProducts());
    }

}
```

Rerun all the Ant `tests` target and the test should fail as `getProducts()` has yet to be implemented. It's usually a good idea to mark unimplemented methods by getting them to throw an `UnsupportedOperationException`.

Next we implement a test for retrieving a list of stub products populated with test data. We know that we'll need to populate the products list in the majority of our test methods in `SimpleProductManagerTests`, so we define the stub list in JUnit's `setUp()`, a method that is invoked before each test method is called.

'springapp/test/springapp/service/SimpleProductManagerTests.java':

```
package springapp.service;

import java.util.ArrayList;
import java.util.List;

import springapp.domain.Product;

import junit.framework.TestCase;

public class SimpleProductManagerTests extends TestCase {

    private SimpleProductManager productManager;
    private List<Product> products;

    private static int PRODUCT_COUNT = 2;

    private static Double CHAIR_PRICE = new Double(20.50);
    private static String CHAIR_DESCRIPTION = "Chair";
```

```
    private static String TABLE_DESCRIPTION = "Table";
    private static Double TABLE_PRICE = new Double(150.10);

    protected void setUp() throws Exception {
        productManager = new SimpleProductManager();
        products = new ArrayList<Product>();

        // stub up a list of products
        Product product = new Product();
        product.setDescription("Chair");
        product.setPrice(CHAIR_PRICE);
        products.add(product);

        product = new Product();
        product.setDescription("Table");
        product.setPrice(TABLE_PRICE);
        products.add(product);

        productManager.setProducts(products);
    }

    public void testGetProductsWithNoProducts() {
        productManager = new SimpleProductManager();
        assertNull(productManager.getProducts());
    }

    public void testGetProducts() {
        List<Product> products = productManager.getProducts();
        assertNotNull(products);
        assertEquals(PRODUCT_COUNT, productManager.getProducts().size());

        Product product = products.get(0);
        assertEquals(CHAIR_DESCRIPTION, product.getDescription());
        assertEquals(CHAIR_PRICE, product.getPrice());

        product = products.get(1);
        assertEquals(TABLE_DESCRIPTION, product.getDescription());
        assertEquals(TABLE_PRICE, product.getPrice());
    }
}
```

Rerun all the Ant `tests` target and our two tests should fail.

We go back to the `SimpleProductManager` and implement the getter and setter methods for the `products` property.

'springapp/src/springapp/service/SimpleProductManager.java':

```
package springapp.service;

import java.util.ArrayList;
import java.util.List;

import springapp.domain.Product;

public class SimpleProductManager implements ProductManager {

    private List<Product> products;
```

```
    public List<Product> getProducts() {
        return products;
    }

    public void increasePrice(int percentage) {
        // TODO Auto-generated method stub
    }

    public void setProducts(List<Product> products) {
        this.products = products;
    }

}
```

Rerun the Ant `tests` target and all our tests should pass.

We proceed by implementing the following tests for the `increasePrice()` method:

- The list of products is null and the method executes gracefully.
- The list of products is empty and the method executes gracefully.
- Set a price increase of 10% and check the increase is reflected in the prices of all the products in the list.

'springapp/test/springapp/service/SimpleProductManagerTests.java':

```
package springapp.service;

import java.util.ArrayList;
import java.util.List;

import springapp.domain.Product;

import junit.framework.TestCase;

public class SimpleProductManagerTests extends TestCase {

    private SimpleProductManager productManager;

    private List<Product> products;

    private static int PRODUCT_COUNT = 2;

    private static Double CHAIR_PRICE = new Double(20.50);
    private static String CHAIR_DESCRIPTION = "Chair";

    private static String TABLE_DESCRIPTION = "Table";
    private static Double TABLE_PRICE = new Double(150.10);

    private static int POSITIVE_PRICE_INCREASE = 10;

    protected void setUp() throws Exception {
        productManager = new SimpleProductManager();
        products = new ArrayList<Product>();

        // stub up a list of products
        Product product = new Product();
```

```
        product.setDescription("Chair");
        product.setPrice(CHAIR_PRICE);
        products.add(product);

        product = new Product();
        product.setDescription("Table");
        product.setPrice(TABLE_PRICE);
        products.add(product);

        productManager.setProducts(products);
    }

    public void testGetProductsWithNoProducts() {
        productManager = new SimpleProductManager();
        assertNull(productManager.getProducts());
    }

    public void testGetProducts() {
        List<Product> products = productManager.getProducts();
        assertNotNull(products);
        assertEquals(PRODUCT_COUNT, productManager.getProducts().size());

        Product product = products.get(0);
        assertEquals(CHAIR_DESCRIPTION, product.getDescription());
        assertEquals(CHAIR_PRICE, product.getPrice());

        product = products.get(1);
        assertEquals(TABLE_DESCRIPTION, product.getDescription());
        assertEquals(TABLE_PRICE, product.getPrice());
    }

    public void testIncreasePriceWithNullListOfProducts() {
        try {
            productManager = new SimpleProductManager();
            productManager.increasePrice(POSITIVE_PRICE_INCREASE);
        }
        catch(NullPointerException ex) {
            fail("Products list is null.");
        }
    }

    public void testIncreasePriceWithEmptyListOfProducts() {
        try {
            productManager = new SimpleProductManager();
            productManager.setProducts(new ArrayList<Product>());
            productManager.increasePrice(POSITIVE_PRICE_INCREASE);
        }
        catch(Exception ex) {
            fail("Products list is empty.");
        }
    }

    public void testIncreasePriceWithPositivePercentage() {
        productManager.increasePrice(POSITIVE_PRICE_INCREASE);
        double expectedChairPriceWithIncrease = 22.55;
        double expectedTablePriceWithIncrease = 165.11;

        List<Product> products = productManager.getProducts();
        Product product = products.get(0);
```

```
        assertEquals(expectedChairPriceWithIncrease, product.getPrice());

        product = products.get(1);
        assertEquals(expectedTablePriceWithIncrease, product.getPrice());
    }

}
```

We return to `SimpleProductManager` to implement `increasePrice()`.

'springapp/src/springapp/service/SimpleProductManager.java':

```
package springapp.service;

import java.util.List;

import springapp.domain.Product;

public class SimpleProductManager implements ProductManager {

    private List<Product> products;

    public List<Product> getProducts() {
        return products;
    }

    public void increasePrice(int percentage) {
        if (products != null) {
            for (Product product : products) {
                double newPrice = product.getPrice().doubleValue() *
                                  (100 + percentage)/100;
                product.setPrice(newPrice);
            }
        }
    }

    public void setProducts(List<Product> products) {
        this.products = products;
    }

}
```

Rerun the Ant `tests` target and all our tests should pass. *HURRAH* JUnit has a saying: “keep the bar green to keep the code clean.” For those of you running the tests in an IDE and are new to unit testing, we hope you're feeling imbued with a sense of greater sense of confidence and certainty that the code is truly working as specified in the business rules specification and as you intend. We certainly do.

We're now ready to move back into the web layer to put a list of products into our `Controller` model.

3.3. Summary

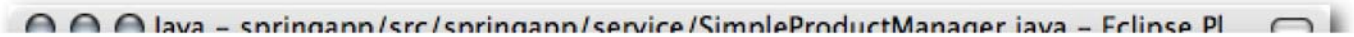
Let's take quick look at what we did in Part 3.

- We implemented a domain object `Product` and a service interface `ProductManager` and concrete class `SimpleProductManager` all as POJOs.

- We wrote unit tests for all the classes we implemented.

- We didn't write a line of code to do with Spring. This is an example of how non-invasive the Spring Framework really is. One of its core aims is to enable developers to focus on tackling the most important task of all: to deliver value by modelling and implementing business requirements. Another of its aims is to make following best practices easy, such as implementing services using interfaces and unit testing as much as is pragmatic given project constraints. Over the course of this tutorial, you'll see the benefits of designing to interfaces come to life.

Find below a screen shot of what your project directory structure must look like after following the above instructions.



The project directory structure at the end of part 3

Prev

Chapter 2. Developing and Configuring the Views and the Controller

Home

Sponsored by SpringSource

Next

Chapter 4. Developing the Web Interface