Capítulo 3. Desarrollando la Lógica de Negocio

Ésta es la Parte 3 del tutorial paso a paso para desarrollar una aplicación Spring MVC. En esta sección, adoptaremos un acercamiento pragmático al Test-Driven Development (TDD o Desarrollo Conducido por Tests) para crear los objetos de dominio e implementar la lógica de negocio para nuestro sistema de mantenimiento de inventario. Esto significa que "escribiremos un poco de código, lo testearemos, escribiremos un poco más de código, lo volveremos a testear..." En la Parte 1 hemos configurado el entorno y montado la aplicación básica. En la Parte 2 hemos refinado la aplicación desacoplando la vista del controlador.

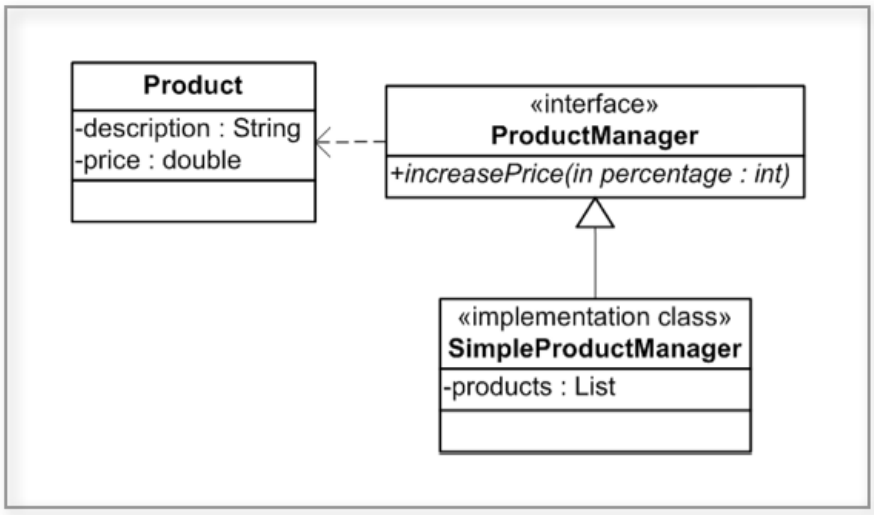
Spring permite hacer las cosas simples fáciles y las difíciles posibles. La estructura fundamental que hace esto posible es el uso de `Plain Old Java Objects (POJOs u Objetos Normales Java)` por Spring. Los POJOs son esencialmente clases nomales Java libres de cualquier contrato (normalmente impuesto por un framework o arquitectura a traves de subclases o de la implementación de interfaces). Los POJOs son objetos normales Java que están libres de dichas obligaciones, haciendo la programación orientada a objetos posible de nuevo. Cuando trabajes con Spring, los objetos de dominio y los servicios que implementes seran POJOs. De hecho, casi todo lo que implementes debería ser un POJO. Si no es así, deberías preguntarte a ti mismo porqué no ocurre esto. En esta sección, comenzaremos a ver la simplicidad y potencia de Spring.

3.1. Revisando la regla de negocio del Sistema de Mantenimiento de Inventario

En nuestro sistema de mantenimiento de inventario tenemos dos conceptos: el de producto, y el de servicio para manejarlo. Supongamos en este tutorial que el negocio solicita la capacidad de incrementar precios sobre todos los productos. Cualquier decremento será hecho sobre cada producto en concreto, pero esta característica está fuera de la funcionalidad de nuestra aplicación. Las reglas de validación para incrementar precios son:

- El incremento máximo esta limitado al 50%.
- El incremento mínimo debe ser mayor del 0%.

A continuación puedes ver el diagrama de clases de nuestro sistema de mantenimiento de inventario.



El diagrama de clases para el sistema de mantenimiento de inventario

3.2. Añadir algunas clases a la lógica de negocio

Añadamos ahora más lógica de negocio en la forma de una clase `Product` y un servicio al que llamaremos `ProductManager` que gestionará todos los productos. Para separar la lógica de la web de la lógica de negocio, colocaremos las clases relacionadas con la capa web en el paquete `'web'` y crearemos dos nuevos paquetes: uno para los objetos de servicio, al que llamaremos `'service'`, y otro para los objetos de dominio al que llamaremos `'domain'`.

Primero implementamos la clase `Product` como un POJO con un constructor por defecto (que es provisto si no especificamos ningun constructor explícitamente), así como métodos getters y setters para las propiedades `'description'` y `'price'`. Además haremos que la clase implemente la interfaz `Serializable`, aspecto no necesario en este momento para nuestra aplicación, pero que será necesario más tarde cuando persistamos y almacenemos su estado. Esta clase es un objeto de dominio, por lo tanto pertenece al paquete `'domain'`.

`'springapp/src/main/java/com/companyname/springapp/domain/Product.java'`:

```
package com.companyname.springapp.domain;

import java.io.Serializable;
```

```
public class Product implements Serializable {

    private static final long serialVersionUID = 1L;

    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();
    }
}
```

Escribamos ahora una unidad de test para nuestra clase `Product`. Algunos programadores no se molestan en escribir tests para los getters y setters, también llamado código 'auto-generado'. Normalmente, supone demasiado tiempo enfrascarse en el debate (como este párrafo demuestra) sobre si los getters y setters necesitan ser testeados, ya que son métodos demasiado 'triviales'. Nosotros escribiremos los tests debido a que: a) son triviales de escribir; b) tendremos siempre los tests y preferimos pagar el precio de perder un poco de tiempo por la sola ocasión entre cien en que nos salvemos de un error producido por un getter o setter; y c) porque mejoran la cobertura de los tests. Creamos un stub de `Product` y testeamos cada método getter y setter como una pareja en un test simple. Normalmente, escribirás uno o más métodos de test por cada método de la clase, donde cada uno de estos métodos compruebe una condicion particular en el método de la clase (como por ejemplo, verificar un valor `null` pasado al método, etc.).

'springapp/src/test/java/com/companyname/springapp/domain/ProductTests.java':

```
package com.companyname.springapp.domain;

import static org.junit.Assert.*;

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

public class ProductTests {

    private Product product;

    @Before
    public void setUp() throws Exception {
        product = new Product();
    }

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

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

}
```

A continuación, creamos el servicio `ProductManager`. Éste es el servicio responsable de gestionar los productos. Contiene dos métodos: un método de negocio, `increasePrice()`, que incrementa el precio de todos los productos, y un método getter,

`getProducts()`, para recuperar todos los productos. Hemos decidido diseñarlo como una interface en lugar de como una clase concreta por algunas razones. Primero, es más fácil escribir tests unitarios para los `Controllers` (como veremos en el próximo capitulo). Segundo, el uso de interfaces implica que JDK Proxying (una característica del lenguaje Java) puede ser usada para hacer el servicio transaccional, en lugar de usar CGLIB (una librería de generación de código).

'springapp/src/main/java/com/companyname/springapp/service/ProductManager.java':

```
package com.companyname.springapp.service;

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

import com.companyname.springapp.domain.Product;

public interface ProductManager extends Serializable {

    public void increasePrice(int percentage);

    public List<Product> getProducts();

}
```

Vamos a crear ahora la clase `SimpleProductManager` que implementa la interface `ProductManager`.

'springapp/src/main/java/com/companyname/springapp/service/SimpleProductManager.java':

```
package com.companyname.springapp.service;

import java.util.List;

import com.companyname.springapp.domain.Product;

public class SimpleProductManager implements ProductManager {

    private static final long serialVersionUID = 1L;

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

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

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

Antes de implementar los métodos en `SimpleProductManager`, vamos a definir algunos tests. La definición más estricta de `Test Driven Development` (TDD) implica escribir siempre los tests primero, y a continuación el código. Una interpretación aproximada se conoce como `Test Oriented Development` (TOD - Desarrollo Orientado a Tests), donde se alternan las tareas de escribir el código y los tests como parte del proceso de desarrollo. En cualquier caso, lo más importante es tener para el código base el conjunto más completo de tests que sea posible. La forma de alcanzar este objetivo es más teoría que práctica. Muchos programadores TDD, sin embargo, están de acuerdo en que la calidad de los tests es siempre mayor cuando son escritos al mismo tiempo que el código, por lo que ésta es la aproximación que vamos a tomar.

Para escribir test efectivos, tienes que considerar todas las pre- y post-condiciones del método que va a ser testeado, así como lo que ocurre dentro del método. Comencemos testeando una llamada a `getProducts()` que devuelve `null`.

'springapp/src/test/java/com/companyname/springapp/service/SimpleProductManagerTests.java':

```
package com.companyname.springapp.service;

import static org.junit.Assert.*;

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

public class SimpleProductManagerTests {

    private SimpleProductManager productManager;

    @Before
    public void setUp() throws Exception {
        productManager = new SimpleProductManager();
    }

    @Test
    public void testGetProductsWithNoProducts() {
        productManager = new SimpleProductManager();
```

```
        assertNull(productManager.getProducts());
    }
}
```

Si ejecutamos ahora los tests de `SimpleProductManagerTests` fallarán; ya que, por ejemplo, `getProducts()` todavía no ha sido implementado. Normalmente, es una buena idea marcar los métodos aún no implementados haciendo que lancen una excepción de tipo `UnsupportedOperationException`.

A continuación, vamos a implementar un test para recuperar una lista de objectos de respaldo en los que han sido almacenados datos de prueba. Sabemos que tenemos que almacenar la lista de productos en la mayoria de nuestros tests de `SimpleProductManagerTests`, por lo que definimos la lista de objetos de respaldo en el metodo `setUp()` de JUnit. Este método, anotado como `@Before`, será invocado previamente a cada llamada a un método de test.

'springapp/src/test/java/com/companyname/springapp/service/SimpleProductManagerTests.java':

```
package com.companyname.springapp.service;

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

import static org.junit.Assert.*;

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

import com.companyname.springapp.domain.Product;

public class SimpleProductManagerTests {

    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);

    @Before
    public 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);

    }

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

    @Test
    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());
    }
```

```
}
```

Si volvemos a lanzar los test de `SimpleProductManagerTests` seguirán fallando.Para solucionarlo, volvemos a `SimpleProductManager` e implementamos los métodos getter y setter de la propiedad `products`.

'springapp/src/main/java/com/companyname/springapp/service/SimpleProductManager.java':

```
package com.companyname.springapp.service;

import java.util.List;

import com.companyname.springapp.domain.Product;

public class SimpleProductManager implements ProductManager {

    private static final long serialVersionUID = 1L;

    private List<Product> products;

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

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

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

Relanza los test de nuevo y ahora todos ellos deben pasar.

Ahora procedemos a implementar los siguientes test para el método `increasePrice()`:

- La lista de productos es null y el método se ejecuta correctamente.
- La lista de productos esta vacía y el método se ejecuta correctamente.
- Fija un incremento de precio del 10% y comprueba que dicho incremento se ve reflejado en los precios de todos los productos de la lista.

'springapp/src/test/java/com/companyname/springapp/service/SimpleProductManagerTests.java':

```
package com.companyname.springapp.service;

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

import static org.junit.Assert.*;

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

import com.companyname.springapp.domain.Product;

public class SimpleProductManagerTests {

    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;

    @Before
    public 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);

    }

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

    @Test
    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());
    }

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

    @Test
    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.");
        }
    }

    @Test
    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(), 0);

        product = products.get(1);
        assertEquals(expectedTablePriceWithIncrease, product.getPrice(), 0);
    }
}
```

Por último, volvemos a `SimpleProductManager` para implementar el método `increasePrice()`.

'springapp/src/main/java/com/companyname/springapp/service/SimpleProductManager.java':

```
package com.companyname.springapp.service;

import java.util.List;

import com.companyname.springapp.domain.Product;
```

```
public class SimpleProductManager implements ProductManager {

    private static final long serialVersionUID = 1L;

    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;
    }
}
```

Si volvemos a lanzar los tests de `SimpleProductManagerTests` ahora deberán pasar todos. ¡HURRA!. JUnit tiene un dicho: "keep the bar green to keep the code clean (mantén la barra verde para mantener el código limpio)".

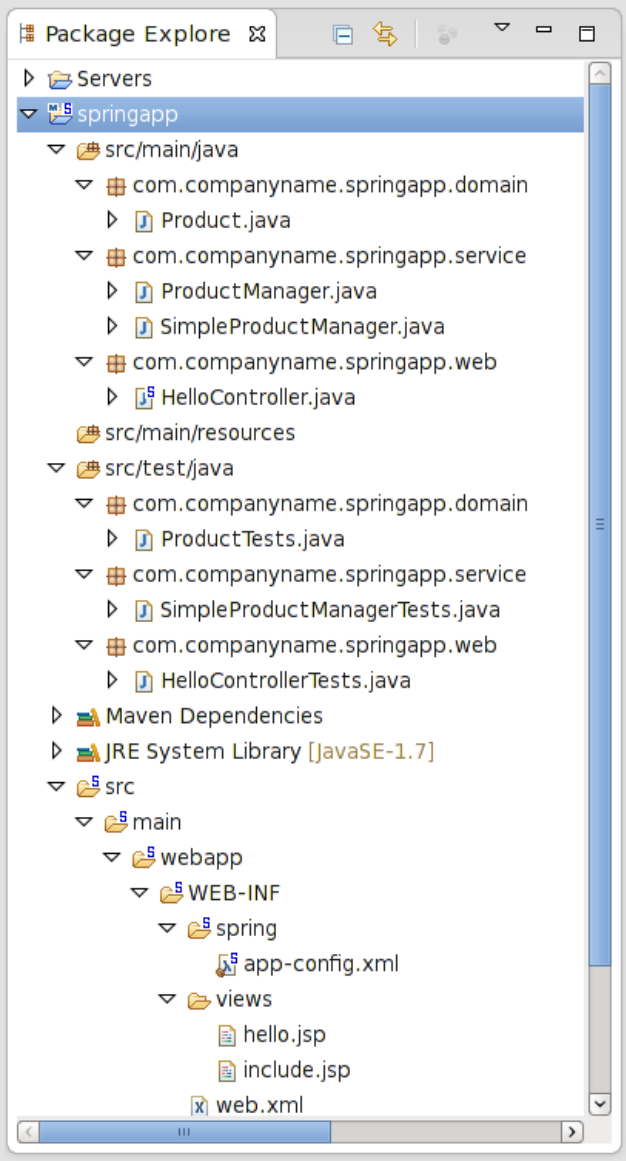
Ahora estamos listos para movernos a la capa web y para poner una lista de productos en nuestro modelo `Controller`.

3.3. Resumen

Echemos un rápido vistazo a lo que hemos hecho en la Parte 3.

- Hemos implementado el objeto de dominio `Product`, la interface de servicio `ProductManager` y la clase concreta `SimpleProductManager`, todos como POJOs.
- Hemos escrito tests unitarios para todas las clases que hemos implementado.
- No hemos escrito ni una sola linea de código de Spring. Éste es un ejemplo de lo no-intrusivo que es realmente Spring Framework. Uno de sus propósitos principales es permitir a los programadores centrarse en la parte más importante de todas: modelar e implementar requerimientos de negocio. Otro de sus propósitos es hacer seguir las mejores prácticas de programación de una manera sencilla, como por ejemplo implementar servicios usando interfaces y usar tests unitarios más allá de las obligaciones prágmaticas de un proyecto dado. A lo largo de este tutorial, verás como los beneficios de diseñar interfaces cobran vida.

A continuación puedes ver una captura de pantalla que muestra el aspecto que debería tener la estructura de directorios del proyecto después de seguir todas las instrucciones anteriores.



La estructura de directorios del proyecto al final de la parte 3