Capitulo 3. Desarrollando la Logica de Negocio

Esta es la Parte 3 del tutorial paso a paso para desarrollar una aplicacion Spring MVC. En esta seccion, adoptaremos un acercamiento pragmatico a Test-Driven Development (TDD o Desarrollo Conducido por Tests) para crear los objetos de dominio e implementar la logica de negocio para nuestro sistema de mantenimiento de inventario. Esto significa que "escribiremos un poco de codigo, lo testearemos, escribiremos un poco mas de codigo, lo volveremos a testear..." En la Parte 1 hemos configurado el entorno y montado la aplicacion basica. En la Parte 2 hemos refinado la aplicacion desacoplando la vista del controlador.

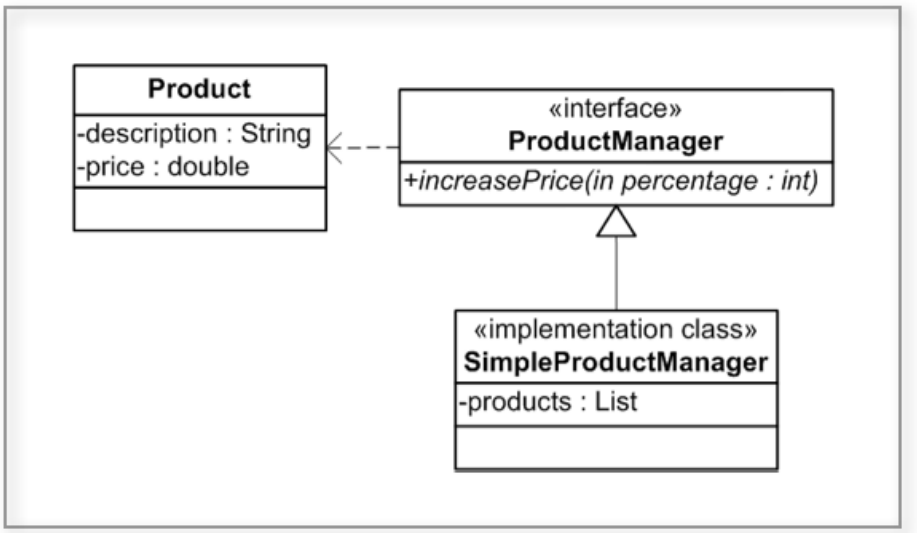
Spring permite hacer las cosas simples faciles y las dificiles posibles. La estructura fundamental que hace esto posible es el uso de `Plain Old Java Objects (POJOs u Objetos Normales Java)` por Spring. Los POJOs son esencialmente clases nomales Java libres de cualquier contrato (normalmente impuesto por un framework o arquitectura a traves de subclases o de la implementacion de interfaces). Los POJOs son objetos normales Java que estan libres de dichas obligaciones, haciendo la programacion 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 deberia ser un POJO. Si no es asi, deberias preguntarte a ti mismo porque no ocurre esto. En esta seccion, 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. Ahora el negocio solicita la capacidad de incrementar precios sobre todos los productos. Cualquier decremento sera hecho sobre productos en concreto, pero esta caracteristica esta fuera de la funcionalidad de nuestra aplicacion. Las reglas de validacion para incrementar precios son:

- El incremento maximo esta limitado al 50%.
- El incremento minimo debe ser mayor del 0%.

Debajo puedes ver un diagrama de clase para nuestro sistema de mantenimiento de inventario.



El diagrama de clase para el sistema de mantenimiento de inventario

3.2. Añadir algunas clases a la logica de negocio

Añadamos ahora mas logica de negocio en la forma de una clase `Product` y un sercicio al que llamaremos `ProductManager` que gestionara todos los productos. Para separar la logica de la web de la logica 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 explicitamente), asi como metodos getters y setters para las propiedades `'description'` y `'price'`. Ademas haremos que la clase implemente la interfaz `Serializable`, no necesariamente para nuestra aplicacion, pero que sera necesario mas tarde cuando persistamos y almacenemos su estado. Esta clase es un objeto de dominio, por lo tanto pertenece al paquete `'domain'`.

`"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();
    }
}
```

Escribamos ahora una unidad de test para nuestra clase `Product`. Algunos programadores no se molestan en escribir tests para los getters y setters, tambien llamado codigo 'auto-generado'. Normalmente conlleva mucho tiempo retirarse del debate (como este parrafo demuestra) sobre si los getters and setters necesitan ser testeados, ya que son metodos demasiado 'triviales'. Nosotros escribiremos los tests debido a: a) son triviales de escribir; b) tenemos siempre los tests pagando dividendos en terminos de tiempo salvado si en solo una ocasion de cien nos vemos salvados 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 metodo getter y setter como un pareja en un test simple. Normalmente, escribiras uno o mas metodos de test por cada metodo de la clase, con cada metodo de test comprobando una condicion particular en el metodo de la clase (como verificar un valor `null` pasado al metodo).

"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);
    }

}
```

A continuacion creamos `ProductManager`. Este es el servicio responsable de manejar productos. Contiene dos metodos: un metodo de negocio, `increasePrice()`, que incrementa el precio de todos los productos, y un metodo 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 mas facil escribir tests de unidad para `Controllers` (como veremos en el proximo capitulo). Segundo, el uso de interfaces implica que JDK Proxying (una caracteristica del lenguaje Java) puede ser usada para hacer el servicio transaccional, en lugar de usar CGLIB (una libreria de generacion de codigo).

"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();

}
```

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

"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();
    }

}
```

Antes de implementar los metodos en SimpleProductManager, vamos a definir algunos tests. La definicion mas estricta de Test Driven Development (TDD) implica escribir siempre los tests primero, y a continuacion el codigo. Una interpretacion aproximada seria mas parecido a Test Oriented Development (TOD - Desarrollo Orientado a Tests), donde alternariamos entre escribir el codigo y los tests como parte del proceso de desarrollo. Lo mas importante es tener para el codigo base el conjunto mas completo de tests que sea posible, de manera que la forma en que alcances este objetivo es mas teoria que practica. Muchos programadores TDD, sin embargo, estan de acuerdo en que la calidad de los tests es siempre mayor cuando son escritos al mismo tiempo que el codigo, por lo que esta es la aproximacion que vamos a tomar.

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

}
```

Relanza Ant con la opcion tests y el test debe fallar, ya que getProducts() todavia no ha sido implementado. Normalmente es una buena idea marcar los metodos aun no implementados haciendo que lancen una excepcion de tipo UnsupportedOperationException.

A continuacion 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, el cual es invocado previamente a cada llamada a un metodo de test.

"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());
    }
}
```

Relanza Ant con la opcion `tests` y nuestros dos tests deben fallar.

Volvemos a `SimpleProductManager` e implementamos ambos metodos getter and setter para la propiedad `products`.

"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;
    }

}
```

Relanza Ant con la opcion `tests` y ahora todos los tests deben pasar.

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

- La lista de productos es null y el metodo se ejecuta correctamente.
- La lista de productos esta vacia y el metodo 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/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());
    }

}
```

Volvemos a SimpleProductManager para implementar 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;
    }

}
```

Relanzamos Ant con la opcion tests y todos nuestros tests deben pasar. ¡HURRA!. JUnit tiene un dicho: “keep the bar green to keep the code clean (manten la barra verde para

mantener el codigo limpio)". Esto es debido a que los IDE con soporte para JUnit utilizan una barra de color verde para indicar que todos los tests han pasado, y una de color morado para indicar que han habido fallos. Para aquellos que estais ejecutando los tests en un IDE y sois nuevos en tests de unidad, esperamos que os sintais invadidos por una gran sensacion de seguridad y confianza al saber que el codigo es verdaderamente operativo como esta especificado en las reglas de negocio que habeis definido previamente. Nosotros realmente lo estamos.

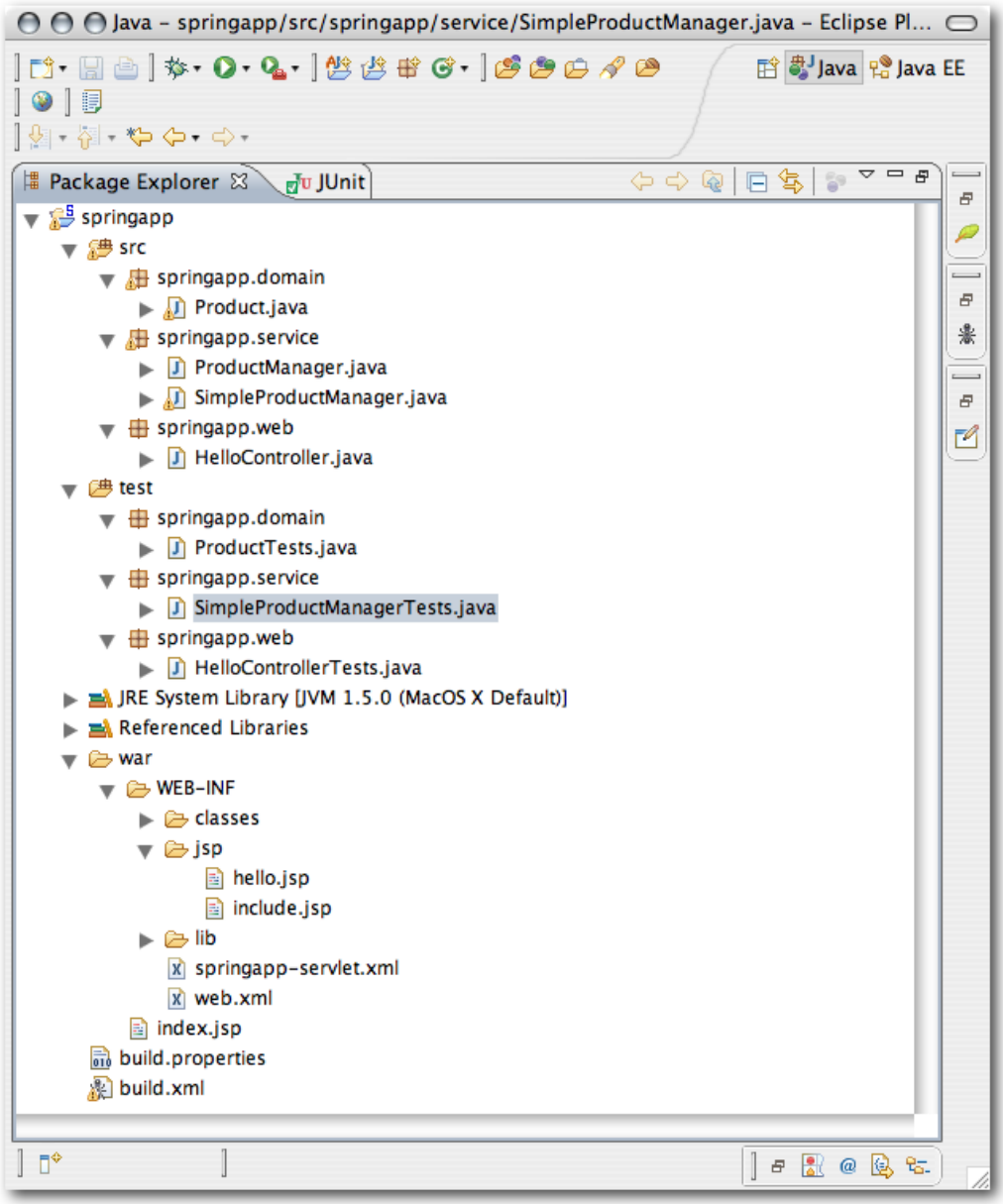
Ahora estamos listos para movernos a la capa web para poner una lista de productos en nuestro modelo `Controller`.

3.3. Resumen

Echemos un rapido 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 de unidad para todas las clases que hemos implementado.
- No hemos escrito ni una sola linea de codigo de Spring. Este es el ejemplo de lo no-intrusivo que es realmente Spring Framework. Uno de sus propositos principales es permitir a los programadores centrarse en la parte mas importante de todas: crear valor modelando e implementando requerimientos de negocio. Otro de sus propositos es hacer seguir las mejores practicas de programacion mas faciles, como implementar servicios usando interfaces y usando tests de unidad, mas alla de las obligaciones pragmaticas de un proyecto dado. A lo largo de este tutorial, veras como los beneficios de diseñar interfaces cobran vida.

Debajo puedes ver una captura de pantalla mostrando como debe aparecer tu estructura de directorios despues de seguir todas las instrucciones anteriores.



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

Anterior	Inicio	Siguiente
Capitulo 2. Desarrollando y Configurando la Vista y el Controlador	Traducido por David Marco	Capitulo 4. Desarrollando la Interface Web