Auditoría, Calidad y Gestión de Sistemas
(ACG)

Práctica
Testing Alquiler de Películas

Con Eclipse y Tests JUnit

Curso 22/23

1. Alquiler de películas

A través de un sencillo proyecto Java vamos a seguir trabajando con el framework de pruebas JUnit y a realizar algunas refactorizaciones. El objetivo de la práctica es *aprender a diseñar tests y automatizarlos con JUnit* a la vez que mejoramos el código. Para ello, utilizaremos el ejemplo de gestión de alquiler de películas que se muestra en el primer capítulo del libro de Martin Fowler[1]. Utilizaremos el entorno eclipse.

Como algunos alumnos pueden no estar familiarizados con el testing y JUnit, esta práctica la hemos dividido en dos sesiones:

- Sesión 1: crearemos el proyecto, analizaremos el código y nos plantearemos las pruebas que realizaríamos de forma manual. Secciones 1 y 2 de este documento. Si no tienes experiencia con JUnit es recomendable leer el documento "Introducción a JUnit".
- Sesión 2: realizaremos la pruebas unitarias con el framework JUnit. Secciones 3 y 4 de este documento. Las soluciones de la práctica están disponibles en Poliformat, comprueba si tu solución es correcta.

2. Planteamiento del ejercicio

En primer lugar, debéis crearos un proyecto Java en Eclipse e importar las clases Java de la carpeta PoliformaT- ACG- Recursos- Laboratorio - Práctica 1- Código.

La Figura 1 muestra el diseño de estas clases, usando la notación UML.

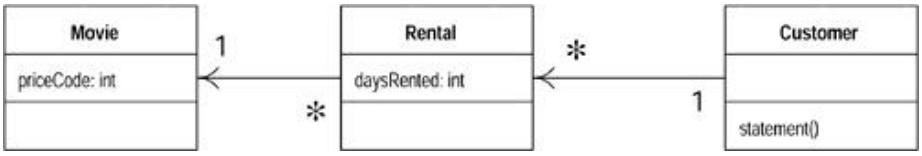

Figura 1: Diagrama de clases inicial

Os proponemos que analicéis el código de las clases; fijaros especialmente en el código del método `statement`().

Como habréis comprobado, el diseño de estas clases podría mejorarse. Esta implementación, de manera aislada, podría considerarse un mal diseño, pero no un problema grave. Sin embargo, si este diseño forma parte de un sistema más complejo, puede entorpecer mucho el diseño y la implementación de la solución. Este diseño dificultará, entre otras cosas, la modificación de código. Lo que afecta al factor de calidad de software visto en clase: *mantenibilidad*. Por ejemplo, suponed que en lugar de querer que el resultado

de `statement`() sea texto en lenguaje natural, quisiéramos que fuera código html para ser mostrado a través de un navegador. ¿Qué impacto tendría este cambio? Como observaréis, no se puede aprovechar el método `statement`(). Lo que haríamos sería copiar el método, y modificarlo, obteniendo como resultado un nuevo método `htmlstatement`(). Copiar y pegar supone un problema cuando el software va a sufrir cambios (que es lo que sucede en prácticamente todo desarrollo). ¿Qué ocurriría si ahora cambiaran las reglas de negocio para clasificar películas; si cambiaran las reglas que calculan el precio de los alquileres y los puntos de regalo? Tendríamos que cambiar tanto el método `statement`() como `htmlstatement`() y asegurarnos de que los cambios son consistentes.

Para evitar este diseño pobre, vamos a refactorizar este código y realizar las pruebas unitarias correspondientes.

3. Construir una batería de pruebas para nuestro código y probar el código.

Lo primero que debemos implementar antes de modificar el código es una batería de pruebas que permita asegurar que el método `statement`() devuelve el string correcto. Las pruebas las haremos utilizando el framework JUnit.

Un ejemplo de prueba para la clase Rental podría ser:

```
public class RentalTest {

// Declaramos variables Rental, Customer y Movie

@Before
public void setUp() throws Exception {
// Creamos películas de diferentes tipo
  // Creamos alquileres
// Creamos clientes
}

@Test
public void testMovies(){
// Comprobamos que el nombre de las películas creadas es correcto
    // Comprobamos que el código del precio de las películas es correcto
}
@Test
public void testCustomers(){
// Comprobamos que los clientes se han creado con el nombre correcto
```

```
}

@Test
public void testRentasl() {
// Comprobamos que la película asociada al alquiler es correcta
// Comprobamos que los días asociados a cada alquiler son correctos
}

@Test
public void testStatement(){

// Añadimos los alquileres a uno de los clientes
// Obtenemos el string que devuelve statement()
      // Construimos un string que el resultado que creemos debe devolver
      // statement()
      // Comprobamos que statement() ha devuelto lo mismo que esperábamos

}
```

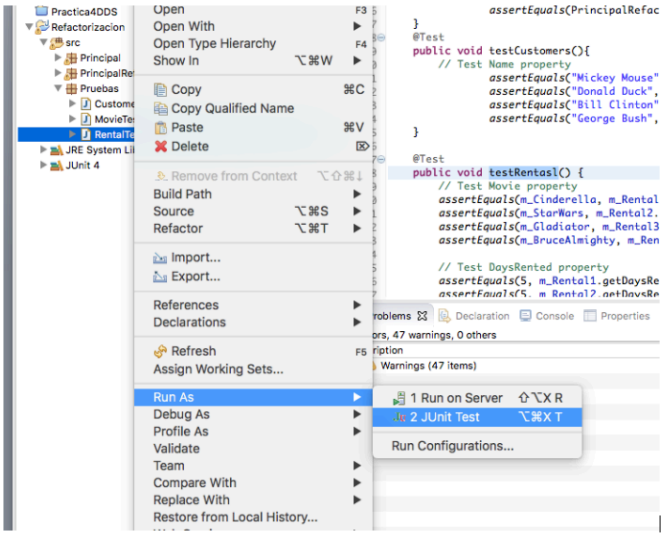

Figura 2: Opción para ejecutar un test JUnit en el entorno Eclipse

Una vez creado el test lo ejecutamos (ver Figura 2). Si el test detecta algún error, lo solucionamos, sino continuamos con el siguiente paso.

4. Refactorizar el código y probar

Una vez realizadas las pruebas, pasaremos a llevar a cabo la primera refactorización. Empezaremos por el método statement(). El método es muy largo, lo que nos hace pensar que se puede descomponer en pequeños métodos. Vamos primero a analizar el siguiente código:

```
switch(each.getMovie().getPriceCode()) {
    case Regular:
     thisAmount += 2;
     if (each.getDaysRented() > 2) {
     thisAmount += (each.getDaysRented() - 2) * 1.5;
     }
     break;
    case NewRelease:
     thisAmount += each.getDaysRented() *3;
     break;
    case Childrens:
     thisAmount += 1.5;
     if (each.getDaysRented() > 3) {
     thisAmount = (each.getDaysRented() - 3) * 1.5;
     }
     break;
}
```

En este trozo de código se incrementa la variable thisAmount en una determinada cantidad según el tipo de la película en alquiler. Podemos extraer este código del método statament() y ponerlo en un nuevo método. Llamaremos a este nuevo método amountFor() y devolverá un valor con la cantidad a incrementar. Una vez creado el nuevo método lo modificaremos renombrando las variables para hacer el código más legible. El nombre de las variables debería ser autodescriptivo.

El resultado de la refactorización podría ser un método amountFor() como el siguiente:

```
public double amountFor(Rental aRental){
     double result = 0;
     switch(aRental.getMovie().getPriceCode()) {
     case Regular:
     result += 2;
     if (aRental.getDaysRented() > 2) {
```

```
    result += (aRental.getDaysRented() - 2) * 1.5;
    }
    break;
    case NewRelease:
    result += aRental.getDaysRented() *3;
    break;
    case Childrens:
    result += 1.5;
    if (aRental.getDaysRented() > 3) {
    result = (aRental.getDaysRented() - 3) * 1.5;
    }
    break;
    }
    return result;
   }
```

Debemos cambiar el método `statement()` para que haga uso de este nuevo método.

Tras implementar esta refactorización, se debe probar el código con la batería de pruebas que hemos diseñado en el paso 1. De esta forma nos aseguramos de que la modificación de código que hemos realizado no cambia su funcionalidad.

Volvemos sobre el método aumountFor() que hemos creado. Este método ¿utiliza información de la clase Customer? O ¿más bien está utilizando datos de otra clase? Si nos fijamos podemos observar como, en realidad, el método utiliza datos de la clase Rental. Esto resulta en una clase Customer con baja cohesión y alto acoplamiento con la clase Rental. Lo que lo correcto sería definir el método aumountFor() en la clase Rental. Así que debemos mover el método amountFor() y realizar los cambios necesarios para utilizar dicho método.

Esta refactorización, que incluye cambiar el nombre al método (ahora lo llamamos getCharge()), podría dar como resultado el siguiente código:

```
public class Rental{
...
public double getCharge(){
    double result = 0;
    switch(movie.getPriceCode ()) {
    case Regular:
    result += 2;
    if (daysRented > 2) {
    result += (daysRented - 2) * 1.5;
    }
```

```
     break;
     case NewRelease:
     result += daysRented *3;
     break;
     case Childrens:
     result += 1.5;
     if (daysRented > 3) {
     result = (daysRented - 3) * 1.5;
     }
     break;
     }
     return result;
    }
}
```

En la clase Customer el método amountFor() ahora únicamente hará uso del método getCharge() de la clase Rental. Y lo hará a través del objeto Rental que recibe como parámetro. El siguiente código muestra el método amountFor() de la clase Customer:

```
public class Customer{
...
public double amountFor(Rental rental){
     return rental.getCharge();
}
}
```

Sin embargo, puesto que amountFor() únicamente hace una llamada a getCharge(), podemos eliminar el método amountFor() de la clase Customer, y utilizar directamente getCharge() en el método statement().

Tras estas modificaciones, se debe volver a probar el código. Será necesario realizar cambios en los tests para probar el nuevo método getCharge(). Esto implica que en la clase RentalTest debemos crear un nuevo método @test. A continuación se muestra un ejemplo:

```
@Test
public void testGetCharge(){
 // Comprobamos que el precio que devuelve getCharge de los
 //alquileres que hemos creado es el que hemos calculado
 //a mano que debe ser
}
```

Ahora vamos a observar la variable thisAmount del método statement(). Esta variable toma el valor del resultado de each.getCharge() y no cambia

después de esto. Es una variable redundante. Por ello, podemos eliminar la variable temporal thisAmount sustituyéndola por una llamada al método getCharge(). El código resultante de aplicar la refactorización sería:

```
public String statement() {
     double totalAmount = 0;
     int frequentRenterPoints = 0;
     Iterator<Rental> iteradorRentals = rentals.iterator();

     String result = "Rental record for " + name + "\n";

     while ( iteradorRentals.hasNext() ) {

     Rental each = iteradorRentals.next();

     frequentRenterPoints++;
     if ((each.getMovie().getPriceCode() ==
         PrincipalRefact.Movie.PriceCodes.NewRelease)
     && (each.getDaysRented() > 1)) {
     frequentRenterPoints ++;
     }

     result += "\t" + each.getMovie().getTitle()  + "\t" +
         Double.toString(each.getCharge()) +"\n";

     totalAmount += each.getCharge();
     }

     result += "Amount owed is " +
        Double.toString(totalAmount) + "\n";
     result += "You earned " + Integer.toString(frequentRenterPoints) +
        " frequent renter points.";
     return result;
    }
```

Volvemos a probar el código.

Retomamos el método statement() y revisamos el siguiente código:

```
frequentRenterPoints++;
if ((each.getMovie().getPriceCode() == PrincipalRefact.Movie.PriceCodes.NewRelease)
    && (each.getDaysRented() > 1)) {
     frequentRenterPoints ++;
}
```

En la primera línea de código se añade un punto de regalo (FrequentRenterPoint) a los puntos actuales; y en las siguientes líneas se añaden puntos de regalo en función de si el alquiler (each) es de más de 1 día y el tipo de película en alquiler es NewRelease. Estos datos son propios de la clase Rental, por lo que este código estaría mejor situado si lo definimos en un método de la clase Rental. Así pues, movemos las líneas de código del método statement() que calculan los puntos de regalo obtenidos en el alquiler, a un nuevo método, getFrequentRenterPoint(), de la clase Rental. El siguiente código muestra esta refactorización:

```
public class Rental{
...
public int getFrequentRenterPoint(){
     if ((movie.getPriceCode() == Movie.PriceCodes.NewRelease) &&
         (daysRented > 1)) {
     return 2;
     }
     else return 1;
}
}
```

Al extraer estas líneas de código del método statement() al nuevo método getFrequentRenterPoint(), el código del método statement() de la clase Customer queda de la siguiente manera:

```
public String statement() {
     double totalAmount = 0;
     int frequentRenterPoints = 0;
     Iterator<Rental> iteradorRentals = rentals.iterator();

     String result = "Rental record for " + name + "\n";

     while ( iteradorRentals.hasNext() ) {

     Rental each = iteradorRentals.next();

     frequentRenterPoints+=each.getFrequentRenterPoint();

     result += "\t" + each.getMovie().getTitle() + "\t" +
         Double.toString(each.getCharge())  + "\n";

     totalAmount += each.getCharge();
     }
```

```
 result += "Amount owed is " + Double.toString(totalAmount) + "\n";
 result += "You earned " + Integer.toString(frequentRenterPoints) +
     " frequent renter points.";
 return result;
}
```

Una vez realizada esta refactorización, realizaremos los cambios oportunos en los test (se debe crear un nuevo método @test para probar getFrequentRenterPoints()) y volver a probar el código.

El código se podría seguir refactorizando para mejorar muchos aspectos del mismo. Por ejemplo, el método statement() se puede simplificar aún más extrayendo métodos que calculen los totales de precio y puntos de regalo asociados a un cliente; o la sentencia condicional del método getCharge() que calcula el precio del alquiler también se puede simplificar utilizando poliformismo (lo que supondría crear nuevas clases hijas de la clase Movie). Podéis seguir refactorizando el ejemplo para mejorar la calidad del código.