
Auditoría, Calidad y Gestión de Sistemas (ACG)

Una **solución** a la práctica 1

Testing Alquiler de Películas

Curso 22/23

Solución Sección 2. Planteamiento del ejercicio

En este apartado se pedía que analizarais el código de una sencilla aplicación java compuesta por tres clases, especialmente se pedía que os fijarais en el método `statement()`, y que os plantearais las siguientes cuestiones: >Qué tipo de pruebas realizaríais si estuvierais ante un proyecto real? >Qué pruebas concretas realizarías? >Echas en falta alguna herramienta que os ayude a realizar estas pruebas?

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. 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.

Respecto a las pruebas, necesitaremos empezar por pruebas unitarias. Este tipo de pruebas verifica trozos de código de forma independiente. Decidir qué pruebas realizar no es un trabajo trivial. En muchas situaciones es difícil o imposible probar ciertas partes del código (<https://www.toptal.com/qa/how-to-write-testable-code-and-why-it-matters>). Lo primero que debíais hacer es analizar las clases y sus métodos:

1. La clase **Movie** es una clase sencilla, con dos métodos *setter* y otros dos *getter*. No posee una lógica compleja, por lo que las pruebas a implementar se limitarían a la creación de algunas películas y a comprobar si los métodos `set` y `get` funcionan correctamente.
2. La clase **MovieRental**, de nuevo, es una clase sencilla con dos métodos *getter* que consultan los atributos (datos) creados en el constructor. De nuevo, en estas clases, no es estrictamente necesario (en este caso) crear casos de prueba, aunque se pueden crear casos sencillos como los de la clase **Movie**.

3. La clase **Customer** es un poco más compleja, tiene un método **statement()** que calcula el dinero que debe el cliente por alquilar las películas y genera un informe sobre las películas alquiladas y los puntos conseguidos. En este caso debemos centrarnos en probar bien este método que va a ser candidato a refactorizar.



Realizar las pruebas sin ninguna herramienta de ayuda es un trabajo costoso, que implica hacer un uso intensivo del debugger de los entornos de programación o utilizar mensajes por consola que nos van indicando el resultado de los trozos de código a probar. Utilizar herramientas de pruebas unitarias, agilizará este tipo de pruebas.

Solución Sección 3. Construir una batería de pruebas

En esta sección se pedía que implementarais antes de modificar el código una batería de pruebas que permitiera asegurar que el método **statement()** devuelve el string correcto. La solución que os proponemos en esta sección utiliza Eclipse más el framework JUnit. Las últimas versiones de Eclipse incluyen las librerías JUnit 5 (podéis comprobarlo en Eclipse -> Acerca de Eclipse -> Installation Details -> Plugins), por lo que no se requiere de ninguna instalación adicional.

Las librerías JUnit permiten al diseñador crear automáticamente esquemas de clases de prueba para las clases Java. Para ello se debe utilizar la opción de menú **New => JUnit Test Case** del menú contextual que aparece al posicionar el ratón sobre clases Java. Al seleccionar esta opción el asistente nos da la opción de elegir los métodos de prueba que queremos crear.

Para ejecutar las pruebas en Eclipse se debe seleccionar la opción **Run As->JUnit Test** en el menú contextual de las clases de Prueba. JUnit para Eclipse ofrece una vista que muestra de forma visual si las pruebas han fallado o no. La vista JUnit se divide en dos zonas:

- La zona superior muestra los resultados de las pruebas. El icono  indica una ejecución de pruebas exitosa. Mientras que el icono  indica una ejecución de pruebas con fallos.
- Al seleccionar un fallo, en la zona inferior se mostrará la traza de ejecución del fallo en forma de árbol. Haciendo doble clic sobre la traza del fallo se accede al código correspondiente que ha provocado el fallo.

¿Cómo creo los tests para las clases **Movie**, **Rental** y **Customer**?

En el explorador de paquetes de Eclipse accedemos al proyecto y seleccionamos una clase, por ejemplo **Movie**, con botón derecho seleccionamos **New => JUnit Test Case** (ver figura 1). Esta opción nos abrirá un formulario

(ver figura 2) donde seleccionaremos la versión última de JUnit (Jupyter, versión 5 del framework). Si pulsamos **Next** nos pasará a la pantalla de la figura 3 donde seleccionaremos los métodos para los cuales queremos que nos genere un método de prueba. Tal y como lo hemos hecho para Movie lo realizaremos para las demás clases (Rental y Customer).

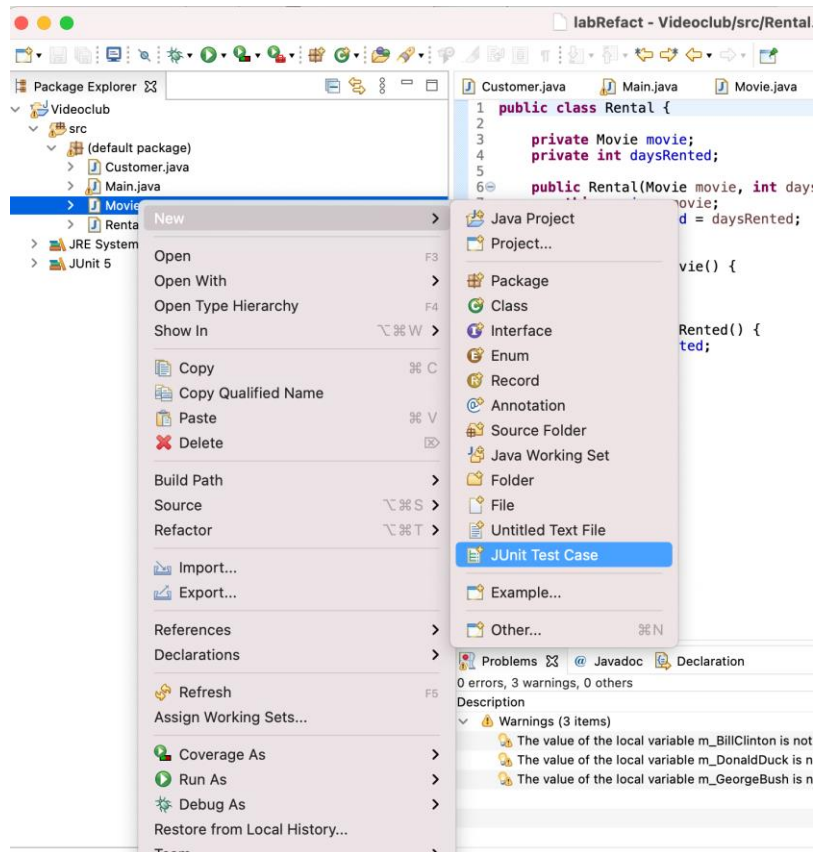


Figura 1. Creación de un caso de prueba JUnit para la clase Movie

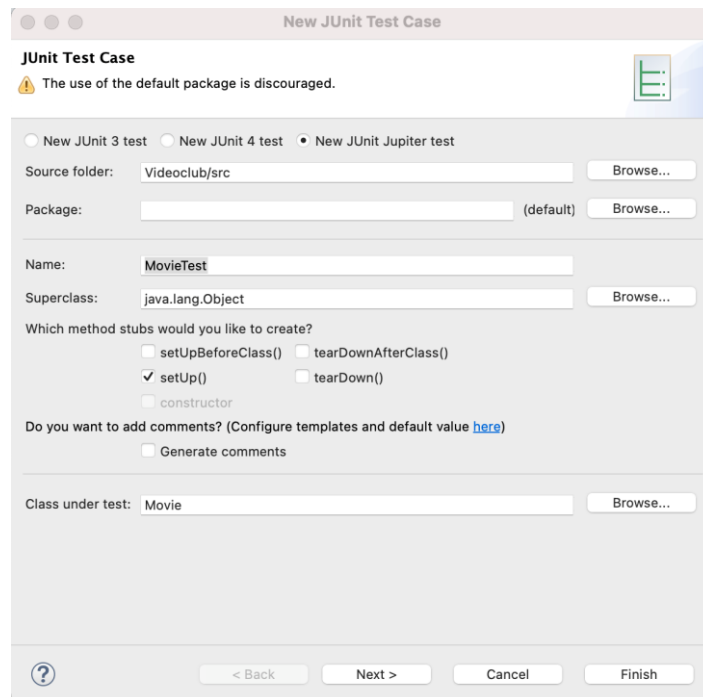


Figura 2. Selección de la versión de Junit

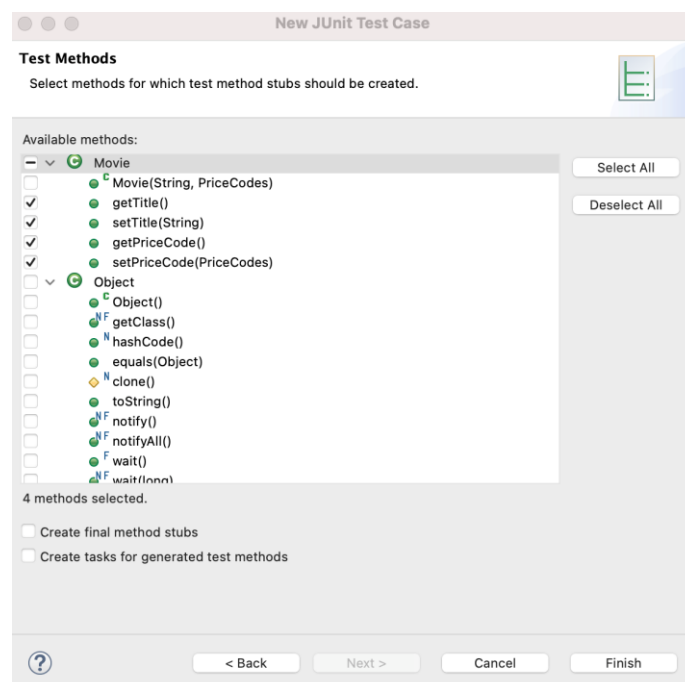


Figura 3. Diagrama de clases inicial

En este caso, por ser un ejemplo relativamente sencillo, no hemos creado un proyecto o paquete aparte para las pruebas. Si lo hubiésemos creado, Eclipse nos proporciona un mecanismo sencillo y potente para referenciar a proyectos existentes es el **Java Build Path**, seleccionaríamos esta opción y elegiríamos el proyecto donde está el código (las clases) a probar (ver figura 4).

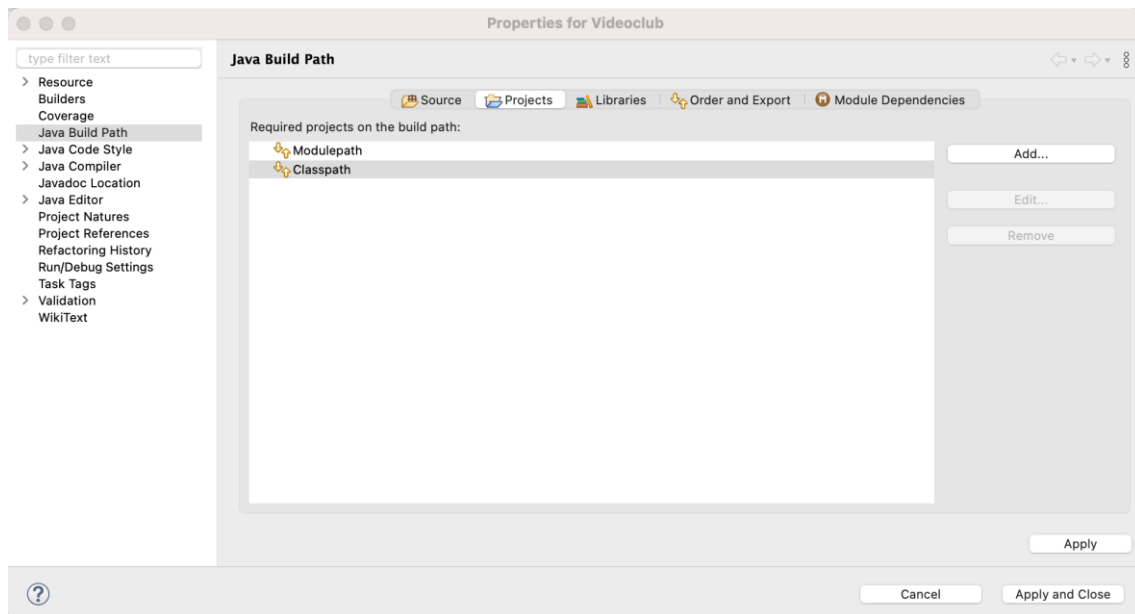


Figura 4. Java Build Path

Probando el método statement()

Cuando un cliente alquila una película, especifica por cuántos días la va a alquilar. Dependiendo del tipo de película (REGULAR, CHILDRENS o NEW_RELEASE) y de los días de alquiler, pagará un precio diferente y obtendrá una cantidad diferente de puntos por el bonus de alquileres frecuentes. La tabla 1 detalla como se definen estos cálculos.

| | Price | Frequent Renter Points |
|-------------|--|--|
| REGULAR | \$2.00 for the first 2 days, \$1.50/day thereafter | 1 pt/rental |
| CHILDRENS | \$1.50 for the first 3 days, \$1.25/day thereafter | 1 pt/rental |
| NEW_RELEASE | \$3.00/day | 1 frequent-renter point 1 day; 2 points for 2 or more days |

Tabla 1. Calculo del Precio y los Puntos para cada tipo de Película

Tenemos que centrarnos en probar el método statement() identificando todas las posibles combinaciones de casos que pueden darse. Crearemos tantos métodos de prueba como combinaciones posibles de datos nos salgan. Por ejemplo:

1. Para las películas REGULAR: Crearemos **dos métodos** de prueba: 1) que calcule el precio para 1 o 2 días ($0 < \text{days} \leq 2$, cálculo normal sin descuento), 2) que calcule el precio para 3 o más días ($\text{days} \geq 3$, cálculo con descuento). Genera 1 punto por su alquiler.
2. Para las películas CHILDRENS: Crearemos **dos métodos** de prueba: 1) que calcule el precio para 1, 2 o 3 días ($0 < \text{days} \leq 3$, cálculo normal sin descuento), 2) que calcule el precio para 4 o más días ($\text{days} \geq 4$, cálculo con descuento). Genera un punto por su alquiler.

3. Para las películas NEW_RELEASE: Crearemos **dos métodos** de prueba:
 - 1) que calcule el precio para 1 día de alquiler, genera un punto, 2) que calcule el precio para days>1, generando dos puntos por su alquiler.

Por último, todos los métodos de prueba están pensados para probar el alquiler de 1 película, nos falta cubrir el alquiler de N películas (N>1). En esta situación deberemos implementar 7 métodos que prueben distintas combinaciones de datos a procesar por el método.

En este ámbito se puede dar dos tipos de estrategias para implementar los métodos de pruebas:

- 1) Un método de prueba por cada combinación o situación. Esta estrategia facilita encontrar el problema o el fallo a primera vista cuando se ejecutan los tests, pero implica implementar más métodos. Nosotros optaremos por esta opción.
- 2) Un método de prueba que incluya (los asserts) de todas las combinaciones (aquí se pueden agrupar por tipo de película). Aunque JUnit nos facilita encontrar el assert que ha fallado, no es tan fácil identificar/visualizar el fallo a primera vista.

A continuación os mostramos un ejemplo de implementación de los métodos de prueba de las películas de niños (CHILDRENS) y un método de ayuda (**expectedMessageFor**¹) para facilitar la implementación de los asserts:

```
@Test
public void precioPrimeros3DiasSinDescuentoChildrenRental() {
    customer.addRental(new Rental(NEW_RELEASE, 2));
    assertEquals(customer.statement(), expectedMessageFor("The Hulk", 1.5, 1.5, 1));
}

@Test
public void precioMasDe3DiasConDescuentoChildrenRental() {
    customer.addRental(new Rental(NEW_RELEASE, 4));
    assertEquals(customer.statement(), expectedMessageFor("The Hulk", 2.75, 2.75, 1));
}

private static String expectedMessageFor(String rental, double price, double total, int renterPointsEarned) {
    return "Rental record for Vicente\n" + rental + "\n" + price + "\nAmount owed is " + total + "\nYou earned " + renterPointsEarned + " frequent renter points.";
}
```

Cuando finalicemos la implementación de los tests, los ejecutaremos y comprobaremos si la implementación pasa todos los tests.

Solución Sección 4. Refactorizar el código y probar

En esta sección se pedía refactorizar el código del método statement() y probar. Eclipse ofrece una serie de excelentes herramientas de refactorización automática que eliminan gran parte del tedio de abordar la refactorización

¹ ExpectedFor se ha creado pensando que hay solo un rental en el array, para el caso de que existan más rentals en el array el método ExpectedFor debe recibir un array de Strings y un array de ints.

“manual”. Por supuesto, ninguna de estas herramientas es perfecta y aún necesitaremos hacer parte de la refactorización a mano, pero puede reducir la cantidad total de tiempo invertido en refactorizar.

Para evitar este diseño pobre, vamos a refactorizar este código. La refactorización la vamos a llevar a cabo en varios pasos.

Paso 1. Aplicar **Extract Method**² al método `statement()`.

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) * 1.25;
        }
        break;
}
```

Fijémonos en que cada película en el vector llamará (entrará) en el switch. 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 `statement()` y ponerlo en un nuevo método (mediante la refactorización **Extract Method**). Para que Eclipse extraiga el método por nosotros, seleccionad el bloque de Código, empezando por la línea donde tenemos la siguiente sentencia `double thisAmount = 0` y abarcamos toda la sentencia switch. Con el botón derecho del ratón seleccionamos **Refactor => Extract Method...** (ver figura 5) y llamaremos a este nuevo método `amountFor()` (ver figura 6). El resultado final es que Eclipse ha generado automáticamente este método (ver figura 7) al final del fichero de código. Este método devolverá un valor con la cantidad a incrementar.

² Hemos utilizado la nomenclatura de las refactorizaciones del libro de Fowler [1].

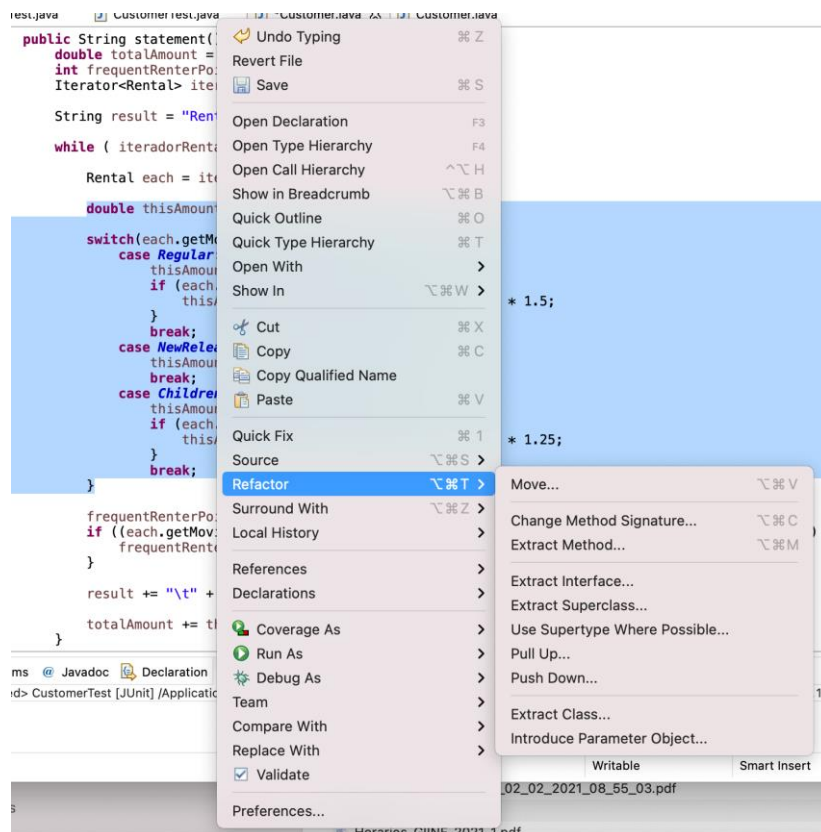


Figura 5. Refactor Extract Method (del código seleccionado)

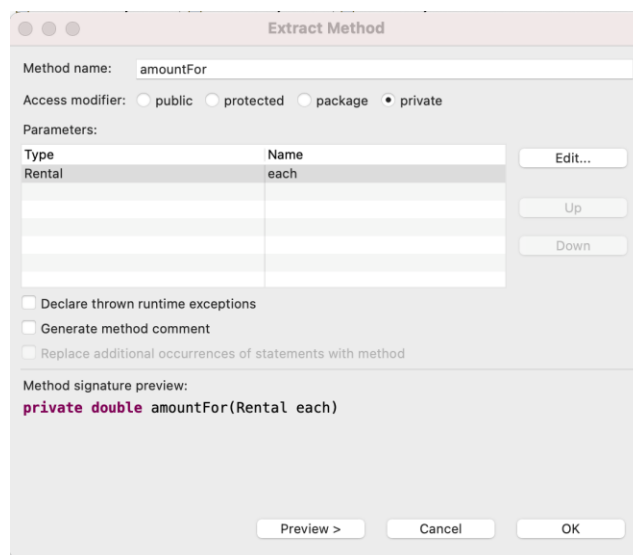


Figura 6. Renombramos el método refactorizado (amountFor)

```

private double amountFor(Rental each) {
    double thisAmount = 0;

    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) * 1.25;
            }
            break;
    }
    return thisAmount;
}

```

Figura 7. Método (amountFor) generado al final del fichero

Paso 2. Aplicar Rename a las variables del método amountFor().

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. Para llevar a cabo esto debemos seleccionar la variable `each` en el método `amountFor` y con el botón derecho pulsar **Refactor => Rename...** y asignarle el nuevo nombre `aRental`. Hagamos lo mismo para `thisAmount` y renombrémosla como `result`. Eclipse renombrará todas las instancias de ambas variables.

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;
}

```

Paso 3. Aplicar Move Method al método amountFor()

Volvemos sobre el método *amountFor()* 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*, por lo que lo correcto sería definir el método en dicha clase. Así que debemos mover el método *amountFor()* a la clase *Rental* (con el refactoring *Move Method*) y aplicar, en la clase *Customer*, los cambios necesario para utilizar dicho método.

Eclipse puede hacer este trabajo por nosotros. Seleccionar el método *amountFor()* y pulsar con el botón derecho para acceder a la opción **Refactor => Move...** Eclipse es suficientemente inteligente para detectar automáticamente que este método debería trasladarse a la clase *Rental*, así que no tenemos que hacer nada más que cambiar el nombre del método por algo más legible. Vamos a llamarle *getCharge* y pulsamos ok.

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;
}
}
```

Podemos utilizar directamente *getCharge()* en el método *statement()*, por lo que cambiaremos el código de esta forma: `double thisAmount = each.getCharge();`

Paso 4. Aplicar Inline Method a la variable *thisAmount*

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. Esa variable se usa solo en 2 lugares del código, y podríamos ahorrarnos declararla. Por ello, podemos eliminar la variable temporal *thisAmount* sustituyéndola por una llamada al método *getCharge()* con el refactoring *Inline*

Method. Con Eclipse convertir fácilmente una variable en un método inline. Eclipse facilitará que todas las referencias a esa variable se reemplacen por llamadas al método `each.getCharge()`. Seleccionemos la variable `any thisAmount` y con el botón derecho seleccionamos y pulsamos **Refactor => Inline...** y pulsamos ok.

El código resultante de aplicar la refactorización:

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

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

    while ( iteratorRentals.hasNext() ) {

        Rental each = iteratorRentals.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;
}
```

Paso 5. Aplicar Extract Method al método `statement()` y Move Method a `getCharge()`

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` (aplicaremos los refactorings `Extract Method` y `Move Method`). Así pues, extraeremos las líneas de código del método `statement()` que calculan los puntos de regalo obtenidos en el alquiler, a un nuevo método, `getFrequentRenterPoints()`. Para hacer esto seleccionaremos el Código desde `frequentRenterPoints++` hasta que finalice la sentencia con el `if` que hemos comentado. A continuación pulsaremos botón derecho

y seleccionaremos **Refactor => Extract Method...** Esta vez llamaremos al método `getFrequentRenterPoints`. Una vez extraído el método debe moverlo a la clase `Rental`, seleccionando el método y pulsando con el botón derecho del ratón para acceder a la opción **Refactor => Move** que nos facilitará mover el método a la clase `Rental`.

El siguiente código muestra esta refactorización:

```
public class Rental{

    public int getFrequentRenterPoints(int frequentRenterPoints) {
        frequentRenterPoints++;
        if ((getMovie().getPriceCode() == Movie.PriceCodes.NewRelease) && (getDaysRented()
> 1)) {
            frequentRenterPoints ++;
        }
        return frequentRenterPoints;
    }
    ...
}
```

Limpiemos un poco este método, ya que no es necesario que pasemos `frequentRenterPoints`. Podemos cambiar el método para devolver el valor de los puntos ganados solo en esta película y, agregar ese valor devuelto al valor actual de la variable `frequentRenterPoints` del método `statement()`. Ahora eliminemos el parámetro `frequentRenterPoints` del método `getFrequentRenterPoints`. Para hacer esto, seleccionemos el método `getFrequentRenterPoints` y vayamos a **Refactor => Change Method Signature**. Seleccionemos el parámetro `frequentRenterPoints` y hagamos clic en Eliminar. Cuando aparezcan advertencias sobre errores, solo digamos ok. Todos esos errores en rojo se pueden arreglar de manera muy simple, devolviendo 2 si es `NewRelease` y 1 en cualquiera de los otros casos:

```
public int getFrequentRenterPoints(){
    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 `getFrequentRenterPoints()`, el código del método `statement()` de la clase `Customer` quedará de la siguiente manera:

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

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

    while ( iteratorRentals.hasNext() ) {

        Rental each = iteratorRentals.next();
```

```

        frequentRenterPoints+=each.getFrequentRenterPoints();

        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;
}

```

Las refactorizaciones que hemos realizado nos llevan al diseño de clases que muestra la Figura 8.

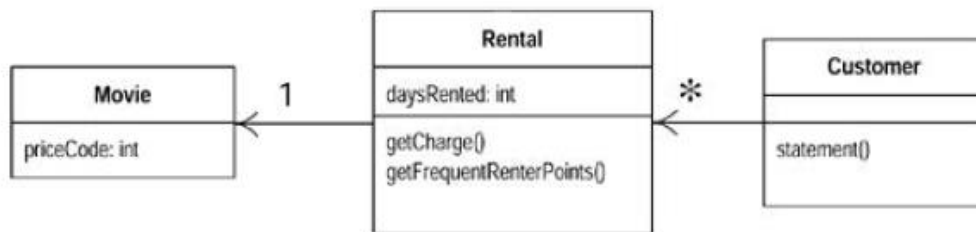


Figura 8. Diseño de clases refactorizado

Ya hemos aplicado muchos refactorings, ejecutemos esas pruebas unitarias para ver qué está pasando. Ejecuta tus pruebas unitarias seleccionando el archivo **CustomerTest.java** en el navegador de paquetes y selecciona **Run => Run As => JUnit test**.

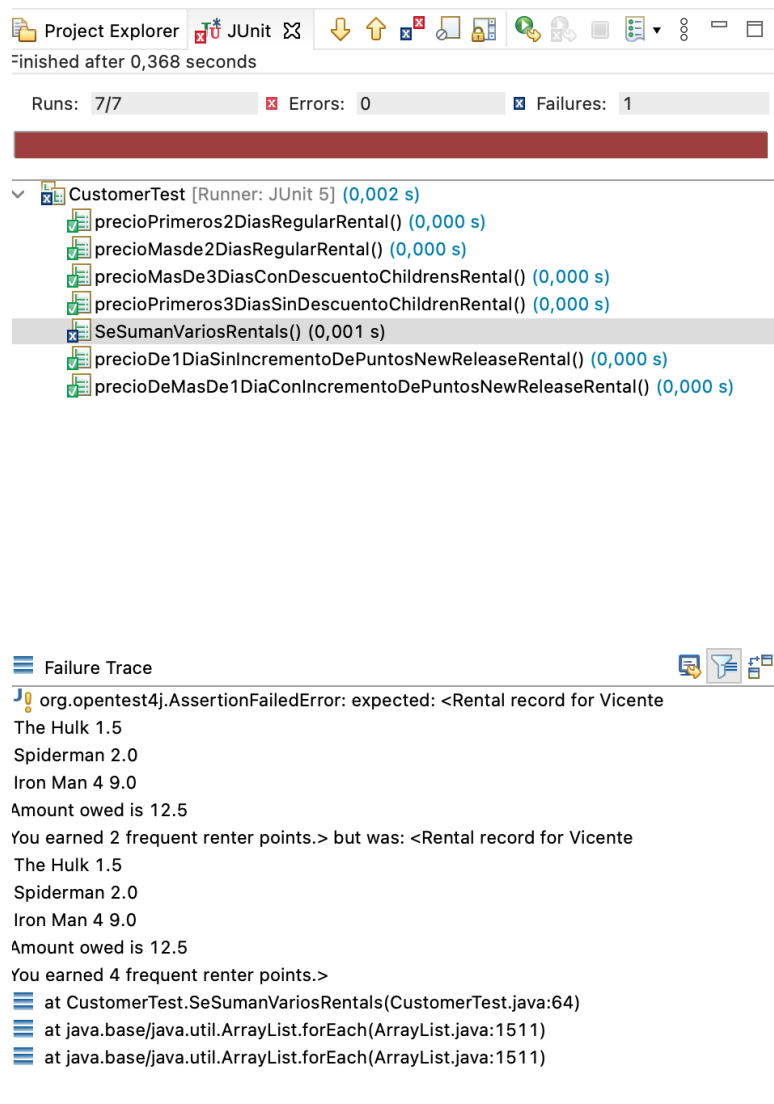


Figura 9. Vista del resultado de las pruebas con un fallo

Vaya!!! En rojo, un fallo!! (ver figura 9) ¿Qué ocurre? Viendo la traza del fallo podemos observar que el cálculo de la suma de *frequent renter points* no funciona bien... o algo parecido. Es necesario cambiar un carácter en una línea del método `statement()` (`frequentRenterPoints += each.getFrequentRenterPoints();`). Cambiémoslo. Como podéis observar, por esto son necesarios los tests unitarios (entre otras razones), pequeños errores pueden estropear o romper la funcionalidad de nuestro programa.

El código se podría seguir refactorizando para mejorar muchos aspectos de este. 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 (Regular, Children, NewRelease) de la clase Movie). Podéis seguir refactorizando el ejemplo para mejorar la calidad del código.

Bibliografía

- [1] Martin Fowler and Kent Beck. *Refactoring. Improving the design of Existing Code*. Addison-Wesley.