PRÁCTICA

1

(2ª parte)

Refactoring

(Videoclub)

Diseño del Software

Grado en Ingeniería Informática del Software

Curso 2022-2023

Escuela de Ingeniería Informática
Universidad de Oviedo

César Acebal
Raúl Izquierdo

Principios

¿En qué consiste factorizar»?

Factorizar es hacer cambios en la estructura interna del software, sin añadir ninguna funcionalidad nueva (sin alterar su comportamiento), para que éste sea más fácil de entender y de modificar.

¿Por qué factorizar?

- Mejora el diseño del software
- Hace que sea más fácil de entender
- Ayuda a encontrar fallos
- Nos hace programar mucho más rápido

¿Cuándo factorizar?

- **Al añadir nuevas funcionalidades**
- **Cuando necesitemos corregir un fallo**
- **Al hacer revisión de código**
  - En diseños muy grandes, también con diagramas UML, escenarios y fichas CRC
  - En XP, con la programación por parejas

La primera vez que haces algo, simplemente lo haces. La segunda vez que haces algo parecido, te molesta la duplicación de código, pero lo duplicas de todas formas. La tercera vez, factorizas.

Refactoring y diseño

¿Ya no hay diseño?

- No, claro que todo el mundo (hasta en XP) hace algún diseño previo
- La diferencia está en dónde ponemos el énfasis
  - Ya no se trata de encontrar «la» solución, sino una solución razonable que iremos mejorando
  - Es decir, no debemos caer en error del «sobrediseño»
  - Pero tampoco en la ausencia de él
    - No todo se puede arreglar simplemente factorizando

Malas sensaciones en el código

Malas sensaciones

- Código duplicado
- Métodos largos
  - Más de diez líneas, malo
  - Lo ideal, sobre cinco o seis líneas la mayoría de los métodos
  - Métodos pequeños con nombres bien escogidos contribuyen a tener código autodocumentado
- Lógica condicional
- Tipos primitivos

Malas sensaciones

- **Falta de ocultación de la información**
  - No todas las clases tienen por qué ser públicas (es decir, tener un constructor público)
    - *Encapsulate Classes with Factory*
- **Responsabilidad repartida en varias clases**
- **Clases grandes**
- **Switch**
- **Explosión de combinaciones**
- **Comentarios**

Ejercicio de factorización

videoclub

¿Qué problemas presenta el diseño actual?

Una pista

- **Además de las «malas sensaciones» anteriores recordemos los principios de acoplamiento, cohesión…**
- **¿Hay alguna clase, método, etcétera, que esté haciendo demasiadas cosas?**
  - (Y demasiadas es más de una)
- **Efectivamente, el el enorme (y muy poco OO) método** statement

Problema

- **¿Qué pasa si cambia la política de precios?**
  - ¿Es evidente que hay que cambiar el método que imprime la factura?
  - ¿Es Customer la clase adecuada para tratar los precios?
- **Más aún: ¿qué ocurriría si ahora queremos generar también la factura en HTML?**
  - ¡Habría que volver a escribir el código que calcula el importe y los puntos frecuentes!
  - No sólo eso, sino que... ¡habría que acordarse de cambiarlo en dos sitios distintos!

Un diseño es bueno cuando ante un cambio en los requisitos sólo hay que tocar un sitio (o incluso ninguno, si se trata de añadir nueva funcionalidad)... y ese sitio es evidente.

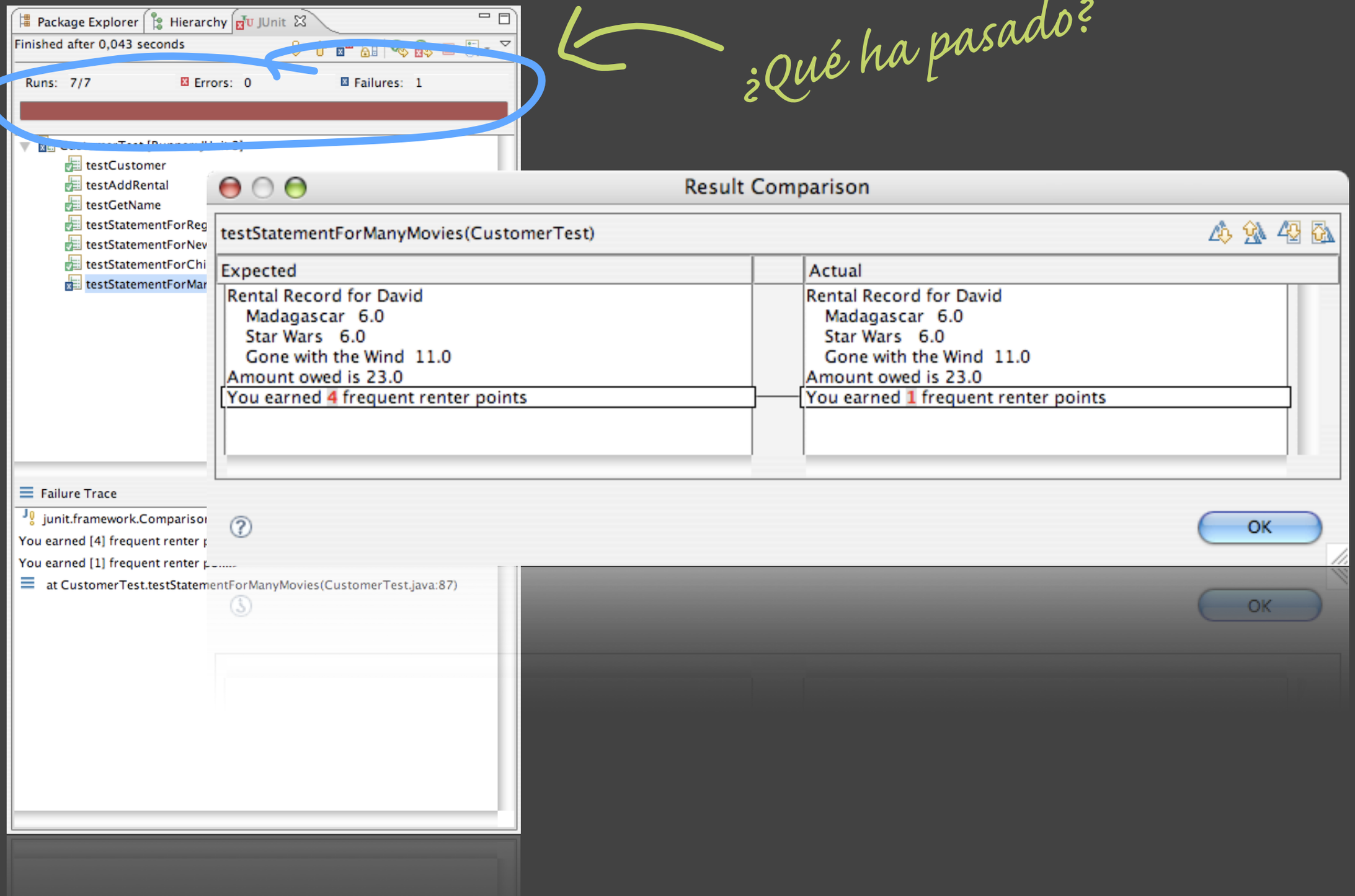
Primer cambio

Sacar el cálculo del importe y de los puntos frecuentes fuera de ese método (por ahora seguirá en «Customer»)

Extract Method

- Aplicamos *Extract Method* para el cálculo del importe por película y de los puntos frecuentes

Ejecutamos las pruebas



¿Qué ha pasado?

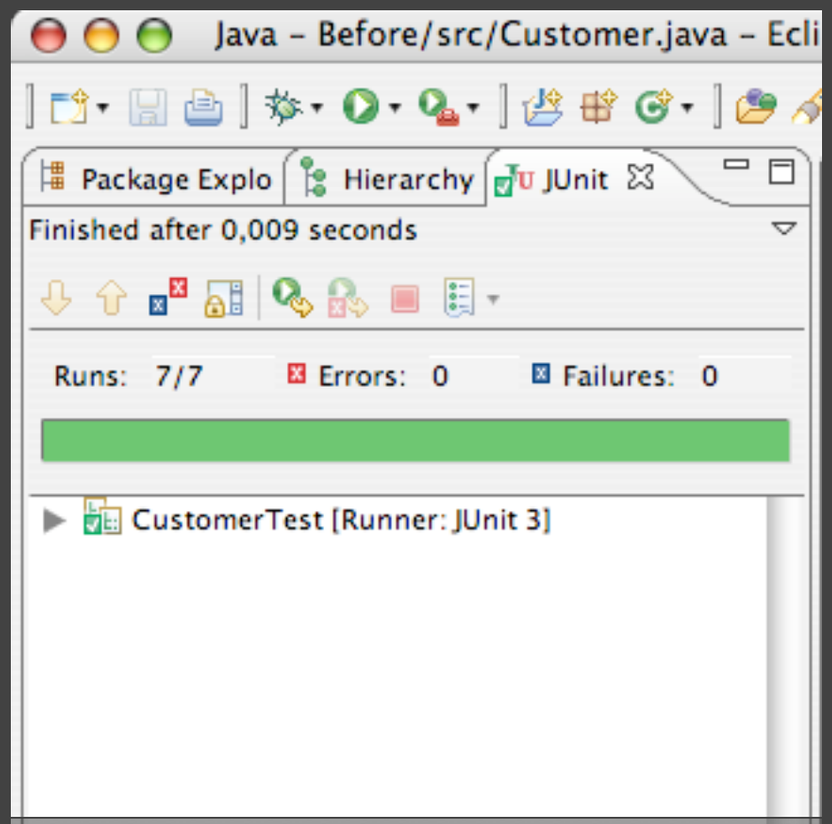
- El problema está en la llamada al método que acabamos de extraer

```
frequentRenterPoints = getFrequentRenterPoints(each);
```

- Hemos quitado el acumulador
- Debería ser:

```
frequentRenterPoints += getFrequentRenterPoints(each);
```

Probamos de nuevo y...



¡Ahora sí!

¿Alguna conclusión?

getFrequentRenterPoints

- ¿Cómo queda el código de este método?

```
private int getFrequentRenterPoints(Rental rental)
{
    int frequentRenterPoints = 1;
    // add bonus for a two day new release rental
    if ((rental.getMovie().getPriceCode() == Movie.NEW_RELEASE &&
            rental.getDaysRented() > 1)
        frequentRenterPoints++;
    return frequentRenterPoints;
}
```

- Es corto, sí, pero... ¿sabríais decir qué hace de un solo vistazo?

getFrequentRenterPoints

- ¿Qué podríamos mejorar?
- Por un lado, simplificar la expresión condicional
  - *Introduce Explaining Variable*, *Extract Method*...
- Por otro, eliminar la variable temporal
- (Éstas ya serían mejoras a nivel de código, de implementación, más que de diseño)

getFrequentRenterPoints

```
private int getFrequentRenterPoints(Rental rental)
{
    int frequentRenterPoints = 1;
    // add bonus for a two day new release rental
    if ((rental.getMovie().getPriceCode() == Movie.NEW_RELEASE &&
            rental.getDaysRented() > 1)
        frequentRenterPoints++;
    return frequentRenterPoints;
}
```

```
private int getFrequentRenterPoints(Rental rental)
{
    if ((rental.isNewRelease() && rental.getDaysRented() > 1)
        return 2;
    return 1;
}
```

getFrequentRenterPoints

- **Antes de hacer eso, no obstante... ¿vemos algo «sospechoso» en este método?**
- **¿Utiliza alguna propiedad o llama a algún método de la clase** Customer**?**
  - No
- **¿De dónde saca los datos que utiliza?**
  - De Rental
- **¿No será que esta responsabilidad no pertenece entonces a esta clase?**

Segundo cambio

Sacar el cálculo del importe y de los puntos frecuentes de «Customer» (asignar correctamente las responsabilidades)

Aplicamos *Move Method*

- **Movemos el método a la clase** Rental
- **Lo mismo para el método** amountFor
- **Que renombraremos como** getCharge

(De nuevo, podríamos seguir aplicando algunas factorizaciones de bajo nivel, aunque no es el objetivo fundamental de esta práctica)

Replace Temp with Query

- **Ahora** thisAmount **es redundante**
- **Se le asigna el valor de** each.getCharge() **pero luego no se cambia nunca**

```
while (rentals.hasMoreElements()) {
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement();

    thisAmount = each.getCharge();

    frequentRenterPoints += each.getFrequentRenterPoints();

    // show figures for this rental
    result += "\t" + each.getMovie().getTitle() + "\t" +
    String.valueOf(each.getCharge()) + "\n";
    totalAmount += each.getCharge();
}
```

Replace Temp with Query

- **Las variables temporales pueden ser un problema**
  - Promueven los métodos largos
- **¿Qué pasa si queremos sacar la factura en HTML?**
  - Que tanto la versión ASCII como la HTML necesitarán el importe y los puntos de alquiler totales
- **Susituimos** totalAmount **y** frequentRenterPoints **por llamadas a métodos**

Replace Temp with Query

- **Hmm... ¿qué ha ocurrido aquí?**
- **Cuando normalmente al factorizar reducimos código, aquí lo hemos aumentado**
  - Pero eso es culpa del modismo («idiom») de Java para recorrer colecciones
- **Y ahora hay tres bucles donde antes sólo había uno**
- **¿Qué opináis de esto? ¿Puede haber un problema de rendimiento? ¿Merece la pena esta factorización?**

¡Sí, sin duda!

- **Los nuevos métodos están disponibles para cualquier otra parte del sistema que los pueda necesitar**
- **O, dentro de la propia clase, si se añade un método para sacar la factura en HTML**
- **Y, lo que es más importante, ahora** statement **hace una sola cosa, no tres**

Tercer cambio

¿Nos damos ya por satisfechos con los dos métodos anteriores en la clase «Rental»? ¿O podemos aplicar el mismo razonamiento de antes?

El problema

Rental

```
public double getCharge() {
    double result = 0;
    switch (getMovie().getPriceCode()) {
    case Movie.REGULAR:
        result += 2;
        if (getDaysRented() > 2)
            result += (getDaysRented() - 2) * 1.5;
        break;
    case Movie.NEW_RELEASE:
        result += getDaysRented() * 3;
        break;
    case Movie.CHILDRENS:
        result += 1.5;
        if (getDaysRented() > 3)
            result += (getDaysRented() - 3) * 1.5;
        break;
    }
    return result;
}
```

¿Qué hay de malo en este código?

Dos cosas

- **La lógica condicional**
  - Hay que procurar huir de los switch e if anidados (siempre que sea posible)* como de la peste
- **Que si se añaden nuevos tipos de película, hay que cambiar también el código de la clase** Rental

(*) Es decir, siempre que se pueda sustituir por el polimorfismo.

Como norma general

- Hay que procurar que los cambios afecten al menor número de clases posible

Solución

- **En primer lugar, mover dicho método a la clase** Movie
  - Que es de quien depende mayormente
- **Y lo mismo para los puntos frecuentes**
- **A continuación, eliminar la lógica condicional aplicando polimorfismo**

Aplicamos *Move Method*

- **Pero ahora con un matiz:**
  - Ahora las responsabilidades están bien donde están (un alquiler debe saber cómo calcular su importe y los puntos de fidelidad)
    - Es la implementación la que depende de otra clase
- **Marcaremos por tanto la casilla para mantener ambos métodos en el alquiler**
  - (Pero sin marcarlos como obsoletos)
- **Delegando así la implementación en la película**

Cuarto cambio

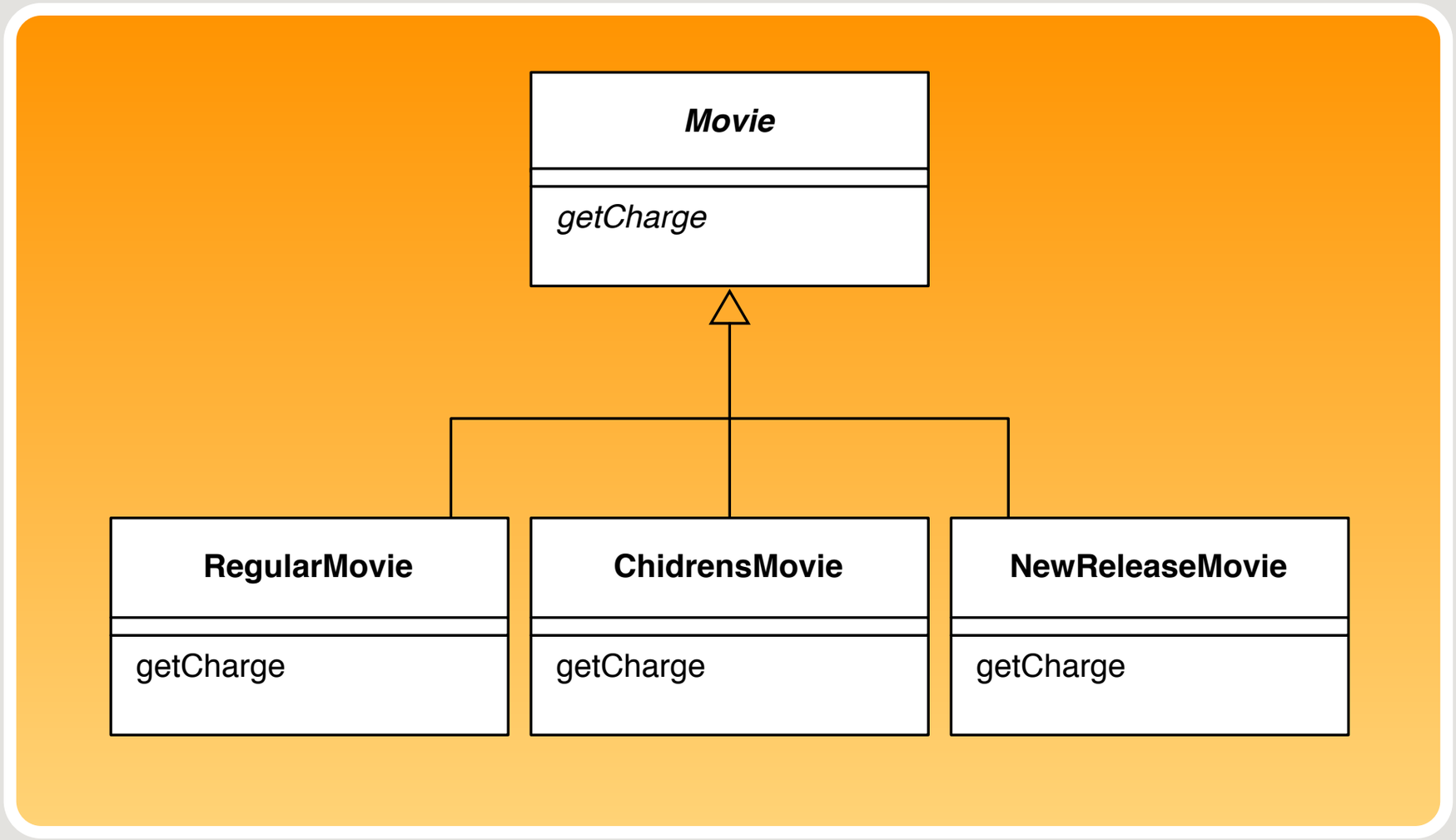
Eliminar la lógica condicional

¿Se puede mejorar más?

- ¿Qué ocurre si se añaden nuevos tipos de películas?
- ¿O si cambia la política de precios o de puntos de fidelidad?

¿Valdría una cosa así?

Con herencia «normal»



¿Resuelve esto nuestro problema?

No

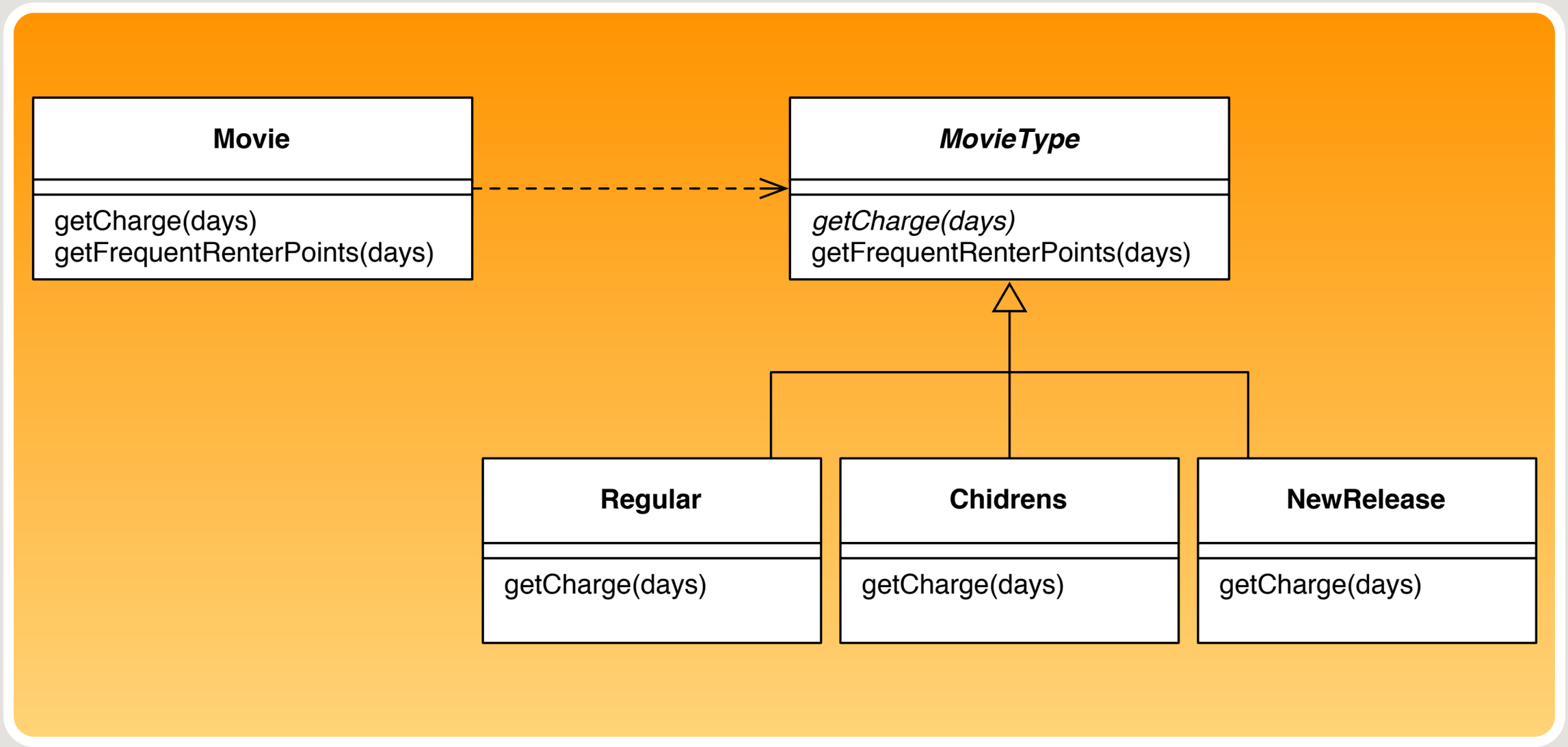
- **No se puede cambiar el tipo de la película**
- **Solución**
  - *Replace Type Code with State/Strategy*

Las películas no tienen siempre el mismo estado

El estado de las películas

- Una película, cuando se estrena, es novedad, pero pasado un tiempo deja de serlo
- El cliente quiere poder introducir nuevos tipos de película y poder cambiarlos dinámicamente

Solución



Que, por cierto, resulta ser un patrón de diseño (que ya estudiaremos)

Conclusión

Conclusión

- **¿Qué pasa ahora si aparecen nuevos tipos de películas o cambia la forma de calcular el importe o los puntos frecuentes de alguna de ellas?**
  - En el primer caso, no hay que cambiar nada del código existente (sólo añadir una nueva clase)
  - En el segundo, se toca sólo lo mínimo imprescindible en el sitio más obvio posible: el tipo de película que ha cambiado
- **Lo hemos conseguido asignando correctamente las responsabilidades**

¡Diseñar para el cambio!

- **Todo programa sufrirá cambios**
- **El objetivo del diseño es facilitar dichos cambios (inevitables)**
- **Un cambio será fácil de realizar si:**
  - Sólo hay que hacerlo en un único sitio
  - Resulta fácil determinar dónde es dicho lugar

Sólo habrá que cambiar en un lugar si...

- **Las clases y los métodos son cohesivos**
  - Sólo tienen una responsabilidad
  - Si hacen varias cosas habrá que modificarlas por distintos tipos de cambios
- **Se puede cambiar su implementación sin afectar al resto**
  - Como vimos en el ejemplo de las temperaturas

Será fácil determinar el lugar si...

- **Cada clase tiene una única responsabilidad y lo indica claramente**
- **Se siguen estructuras reconocibles**
  - Patrones de diseño

En resumen

- Clases con pocas responsabilidades, métodos con nombres que las indiquen claramente y el uso de estructuras habituales (patrones de diseño) hacen fácil saber cuál es la clase a modificar cuando aparezca algún cambio en los requisitos