Desarrollo guiado por pruebas

Adolfo Sanz De Diego

Junio de 2011

# Refactorización

**Refactorizar** un software es modificar su estructura interna con el objeto de que sea más fácil de entender y de modificar a futuro, de tal forma que el comportamiento observable del software al ejecutarse no se vea afectado.

**¿Por qué se refactoriza un software?**

* Para mejorar su diseño
* Para hacerlo más fácil de entender
* Para hacerlo más fácil de modificar
* Para encontrar errores

# ¿Cuándo refactorizar?

Código duplicado.

Método demasiado largo.

Clase demasiado grande.

Diferentes funcionalidades en una misma clase.

Mismas funcionalidades en distintas clases.

Muchos parámetros en una función.

Sentencias switch largas.

Números mágicos

Cadenas mágicas.

Excesos de comentarios.

# Ejemplo de refactorización

Original:

if (date.before(SUMMER\_START) || date.after(SUMMER\_END)) {  
 charge = quantity \* \_winterRate + winterServiceCharge;  
} else {  
 charge = quantity \* \_summerRate;  
}

Refactorizado:

if (summer(date)) {  
 charge = summerCharge(quantity);  
} else {  
 charge = winterCharge(quanity);  
}

# Refactoricación en Eclipse

**Renombrar o mover**: Eclipse nos revisa el resto de las clases del proyecto y nos cambia automáticamente lo necesario para que siga compilando.

**Convertir una variable en un atributo**: Seleccionando una variable, Eclipse nos crea un nuevo atributo en la clase y reemplaza todas las apariciones en el código de esa clase.

**Convertir un valor fijo en una constante**: Seleccionando un valor fijo (numérico o cadena de texto), Eclipse nos crea una constante en la clase y reemplaza todas las apariciones en el código de esa clase.

**Extraer método**: Seleccionamos varias líneas de código, y Eclipse nos crea el método y nos cambia el código para que haga una llamada a ese método.

**Cambiar método**: Seleccionamos el método, y Eclipse nos pone o quita parámetros y revisa todo el código para arreglar las llamadas.

**Extraer Interfaz**: Eclipse extrae las llamadas de los métodos públicos de una clase y crea una Interfaz con ellos, y hace que la clase implemente dicha interfaz.

**Push Down**: Eclipse mueve método o atributo de la clase padre a las hijas.

**Push Up**: Eclipse mueve método o atributo de las clases hijas a la padre.

# Pruebas de software

**Pruebas Unitarias**: Testean el correcto funcionamiento de un módulo.

**Pruebas de Integración**: Testean el correcto funcionamiento del sistema.

**Pruebas de Regresión**: Pila de test que se hacen a cada nuevo cambio introducido en un módulo comprobando que estos no hayan afectado al resto del sistema.

**Pruebas de Carga**: Testean la carga máxima de trabajo que el sistema puede soportar.

**Pruebas de Aceptación**: Pruebas desarrolladas por el cliente, comprobando que el sistema realiza correctamente el trabajo para el que fue diseñado.

# Pruebas Unitarias

**Principio FIRST**:

* **Fast**: Los test deben ejecutarse rápido. Si no al final el equipo terminará por no pasarlos.
* **Independant**: Un test nunca debe depender de otros para pasar. Los puedes ejecutar en cualquier orden y deben funcionar igual.
* **Repeteable**: El test se debe ejecutar en cualquier entorno. En el equipo de desarrollo, en el servidor de test, etc.
* **Self-validating**: Un test debe devolver "correcto" o "fallo". No debe requerir ningún tipo de intervención manual posterior que determine si el test pasó o no.
* **Timely**: Los test deben ser escritos en el momento adecuado, es decir, antes que el código que prueban.

**Frameworks**:

* **JUnit**: el más conocido y utilizado, viene incluido en Eclipse.
* **DBUnit**: para cargar datos en una base de datos antes de hacer los test.
* hay muchos más...

**Mocks Objects**:

* Se usan para simular el comportamiento de objetos complejos cuando es imposible o impracticable usar al objeto real en la prueba.
* Se crean con herramientas como **EasyMock** y **Mockito**.

# Concepto de TDD

**TDD**, Test Driven Development o Desarrollo guiado por pruebas: 1. **Escribir la prueba**. El desarrollador debe entender claramente las especificaciones y los requisitos. Deberá cubrir todos los escenarios de prueba y todas las condiciones de error. 2. **Escribir el código haciendo que pase la prueba**. Escribir el código más sencillo que haga que la prueba funcione. Se usa la metáfora KISS (”Keep It Simple, Stupid”). 3. **Ejecutar las pruebas de regresión**. Por cada cambio realizado en un módulo, por pequeño que sea, obliga a ejecutar todas las pruebas del sistema y comprobar que todo sigue funcionando correctamente. 4. **Refactorización y limpieza en el código**. Antes de añadir nuevas funcionalidades y nuevas pruebas hay que refactorizar y limpiar el código. Después se vuelven a efectuar todas las pruebas del sistema y se comprueba que todo sigue funcionando correctamente. 5. **Repetición**. Después se repetirá el ciclo y se comenzará a agregar las funcionalidades adicionales o a arreglar cualquier error.

# Ventajas TDD

Cómo desarrollador:

* **Mejora la estabilidad del sistema**, pues con cada cambio, las pruebas de regresión comprueban que todos los componentes sigues funcionando.
* **Mejora la documentación**, pues los propios test unitarios actúan como tal.
* **Mejora el diseño**, pues el desarrollador no puede crear código sin entender realmente cuales deberían ser los resultados deseados y como probarlos.

Cómo docente:

* Una vez escritos los test, estos se pueden pasar a los alumnos para que verifiquen que su trabajo está correctamente realizado.
* Los test nos podrían valer para **corregir prácticas y exámenes automáticamente**.