# UT5 – OPTIMIZACIÓN Y DOCUMENTACIÓN

# REFACTORIZACIÓN

Entornos de Desarrollo

# Introducción



- Debido a la complejidad del software se está generando muchas aplicaciones mal diseñadas que son ineficientes y difíciles de mantener.
- El 80% del desarrollo de software es el mantenimiento (evolución) a sistemas existentes
- En el pasado…
  - Las prioridades eran tener un código rápido, pequeño (ocupa poca memoria), optimizado, utilizando los algoritmos mas eficaces, etc.
- Hoy en día se busca…
  - Software simple en su diseño y en su codificación.
  - Software funcional
- Para realizar esto hay que **reestructurar** el software (no sólo se da en código, sino en todo el proceso de desarrollo de software: datos, diseño externo, documentación)

# Refactorización o Refactoring



- Según Martin Fowler:
  - Es mejorar el código existente.
- Para Roger Pressman:
  - Es modificar el **comportamiento interno** (generalmente código fuente) **sin modificar su comportamiento externo** (apariencia, funcionalidad).
- Si se llega a modificar su comportamiento externo formalmente no se le considera "refactorización" sino más bien una modificación.
- Por tanto, no se trata de encontrar y/o corregir errores.
- En definitiva, es hacer un **cambio a la estructura interna** del software para hacer más fácil su comprensión, y más barato al modificar, todo esto sin cambiar su comportamiento externo [Fowler, 1999] .

# Refactorización o Refactoring



- Es una forma sistemática de introducir mejoras en el código que minimiza la posibilidad de introducir errores (bugs) en él.
- Consta básicamente de dos pasos:
  - Introducir un **cambio** simple (refactorización)
  - **Probar** el sistema tras el cambio introducido
- Consiste en realizar modificaciones como:
  - Añadir un argumento a un método
  - Mover un atributo de una clase a otra
  - Mover código hacia arriba o hacia abajo en una jerarquía de herencia
- Ventajas:
  - El código funciona bien
  - El código comunica claramente lo que está haciendo
  - El código no está duplicado

# Refactorización o Refactoring



- Dos 'reglas de oro' de la refactorización:
  1. Cuando necesitamos **añadir una nueva funcionalidad** a una aplicación, si la estructura de la aplicación no es adecuada para introducir los cambios necesarios, **primero refactoriza** el código para que el cambio sea fácil de introducir y luego añade la nueva funcionalidad.
  2. Antes de aplicar técnicas de refactorización, debemos asegurarnos de que disponemos de una batería de **pruebas** robusta y completa. Estas pruebas deben ser auto-comprobantes (como las pruebas unitarias)
- **Refactorizar** durante todo el ciclo de vida de una aplicación **ahorra tiempo** e **incrementa la calidad** del proyecto.
- Objetivo último: Mantener un **código conciso y claro, fácil** de comprender, modificar y extender (incluso por terceros). Un aspecto clave para ello es la total **ausencia de código duplicado**.

# Refactorización o Refactoring



**Motivos para refactorizar**

- Mejorar el diseño del software
- Hacer que el código se más fácil de entender
- Hacer que sea más sencillo encontrar fallos
- Permite programar más rápidamente

**¿Cuándo refactorizar?**

- Metáfora de los dos sombreros: *un programador tiene **dos sombreros**: uno para modificar código (refactorizar) y otro para añadir nuevas funcionalidades*
- *Cuando trabaja lleva puesto uno (y solo uno) de los dos sombreros.*
- *Cuando añade código nuevo, NO modifica el existente. Si está arreglando el existente, NO añade funcionalidades nuevas.*

# ¿Cuándo refactorizar?



- No hay que refactorizar según un plan establecido, sino cuando pienses que es necesario.
- Los momentos más adecuados pueden ser los siguientes:
  - Al añadir un método/función
  - Cuando necesites arreglar un fallo
  - Al revisar código
  - Cuando 'algo huele mal‘

# ¿Cuándo refactorizar?



- Reestructurar después de añadir nueva funcionalidad, especialmente cuando la funcionalidad es difícil de integrar en el código base existente.
- Durante la depuración (debugging)
  - Si es difícil seguir la pista a un error, reestructura para hacer el código más comprensible.
- Durante la inspección formal del código (revisiones de código)
- "Malos olores" o "Bad Smells": si algo huele mal… probablemente esté mal. Para cada "mal olor" existen una serie de refactorings propuestos
  - Código duplicado
  - Métodos largos
  - Clases grandes
  - Lista de parámetros larga
  - Otros problemas más complejos

# ¿Cuándo refactorizar?



| BAD SMELL | REFACTORING PROPUESTO |
|---|---|
| CÓDIGO DUPLICADO | EXTRAER EL MÉTODO<br>SUBIR VARIABLES<br>SUSTITUIR EL ALGORITMO |
| MÉTODOS LARGOS | EXTRAER EL MÉTODO<br>INTRODUCIR OBJETOS COMO PARÁMETROS<br>REEMPLAZAR EL MÉTODO CON UN OBJETO MÉTODO |
| CLASES GRANDES | EXTRAER CLASES<br>EXTRAER SUBCLASES |
| ….. | |

# Cuándo refactorizar: código sospechoso



- A menudo encontramos código sospechoso: algo nos dice que ese código podría ser mejor. A menudo a esto se le llama código con mal olor (*bad code smells*, en inglés).
- Algunos ejemplos:
  - **Código duplicado**: líneas de código exactamente iguales o muy parecidas en varios sitios. Se debe unificar en un sólo sitio. Es el 'mal olor' más común y se debe evitar a toda costa.
  - **Métodos muy largos**: cuanto más largo es el código, más difícil de entender y mantener. Un método muy largo normalmente está realizando tareas que deberían ser responsabilidad de otros. Se deben identificar éstas y descomponer el método en otros más pequeños.
  - **Clases muy grandes**: clases con demasiados métodos, demasiados atributos o incluso demasiadas instancias. La clase está asumiendo, por lo general, demasiadas responsabilidades. Se debe identificar si realmente todas esas cosas tienen algo que ver entre sí y si no es así, hacer clases más pequeñas, de forma que cada una trate con un conjunto pequeño de responsabilidades bien delimitadas (por ejemplo, ocuparse de la conexión con una base de datos, o manejar cierto tipo de información específica, como fechas, DNI, etc.)
  - **Métodos con demasiados parámetros**: los métodos de una clase suelen disponer de la mayor parte de la información que necesitan en su propia clase o clases base. Tener demasiados parámetros puede estar indicando un problema de encapsulación de datos o la necesidad de crear una clase a partir de varios de esos parámetros y pasar ese objeto como argumento en vez de todos los parámetros. Especialmente si esos parámetros suelen tener que ver unos con otros y suelen ir juntos siempre.

# Problemas en la refactorización



- La reestructuración de códigos no es una etapa sencilla, requiere de mucha creatividad.
- En muchas ocasiones, malas reestructuraciones traen consigo más errores y quizás un rendimiento más pobre.

**Cuándo no se debe refactorizar**

- Es más fácil hacerlo de nuevo
- Una señal importante: El código no funciona
- Demasiado cerca de la fecha de entrega comprometida

# Pruebas



- Recordemos que al hacer refactoring no se debe modificar la funcionalidad, sólo la estructura interna.
- Para ello se parte de un código que funciona, se modifica y el código debe seguir funcionando igual que antes.
- Si hacemos refactoring con frecuencia, es importante tener algún medio de probar que el código sigue funcionando después de "arreglarlo".
- Una opción es, después de arreglar el código, compilarlo, ejecutarlo y probar que sigue funcionado, especialmente en las partes de código que hemos tocado.
- *Opción recomendable: hacer pruebas con **JUnit** antes y después. Si pasa el test, podemos tener cierta garantía de que todo sigue funcionando igual que antes.*

# Técnicas de refactorización simples



**Añadir un parámetro**

- Motivo: Un método necesita más información al ser invocado
- Solución: Añadir como parámetro un objeto que proporcione dicha información
- Ejemplo

  *cliente.getContacto()*

  *cliente.getContacto(Fecha fecha)*

- Observaciones:
  - Evitar listas de argumentos demasiado largas.

**Refactoring_1_Parametro (Pasos 01 y 02)**

# Técnicas de refactorización simples



**Quitar un parámetro**

- Motivo: Un parámetro ya no es usado en el cuerpo de un método
- Solución: Eliminarlo
- Ejemplo:

  *cliente.getContacto(Fecha fecha)*

  *cliente.getContacto()*

- Observaciones:
  - Si el método está sobrescrito, puede que otras implementaciones del método sí lo usen. En ese caso no quitarlo. Considerar sobrecargar el método sin ese parámetro.

**Refactoring_1_Parametro (Paso 03)**

# Técnicas de refactorización simples



**Cambiar el nombre de un método**

- Motivo: el nombre de un método no indica su propósito
- Solución: cambiarlo
- Ejemplo:

  *cliente.getLimCrdFact()*

  *cliente.getLimiteCreditoFactura()*
- Observaciones:
  - Comprobar si el método está implementado en clases base o derivadas.
  - Si es parte de una interface publicada, crear un nuevo método con el nuevo nombre y el cuerpo del método. Anotar la versión anterior como @deprecated y hacer que invoque a la nueva.
  - Modificar todas las llamadas a este método con el nuevo nombre.

**Refactoring_2_NombreMetodo**

# Técnicas de refactorización comunes



**Mover un atributo**

- Motivo: Un atributo es (o debe ser) usado por otra clase más que en la clase donde se define.
- Solución: Crear el atributo en la clase destino
- Observaciones:
  - Si el atributo es público, encapsularlo primero.
  - Reemplazar las referencias directas al atributo por llamadas al getter/setter correspondiente.
- Ejemplo:

```
class Cuenta
{
private TipoCuenta tipo;
private double tipoInteres;
double calculaInteres(double saldo, int dias)
{
        return tipoInteres * saldo * dias /365;
}
…
```

```
class TipoCuenta
{
private double tipoInteres;
void setInteres(double d) {…}
double getInteres() {…}
…
```

```
double calculaInteres(double saldo, int dias) {
return tipo.getInteres() * saldo * dias /365;
}
…
```

# Técnicas de refactorización comunes



**Mover un método**

- Motivo: Un método es usado más en otro lugar que en la clase actual
- Solución: Crear un nuevo método allí donde más se use.
- Ejemplo:

  *cliente.getLimiteCreditoFactura()*

  *cuenta.getLimiteCreditoFactura(Cliente c) //parámetro opcional, sólo en caso de necesitar información de Cliente*
- Observaciones:
  - Es una de las refactorizaciones más comunes
  - Hacer que el antiguo método invoque al nuevo o bien eliminarlo.
  - Comprobar si el método está definido en la jerarquía de clases actual. En ese caso puede no ser posible moverlo.

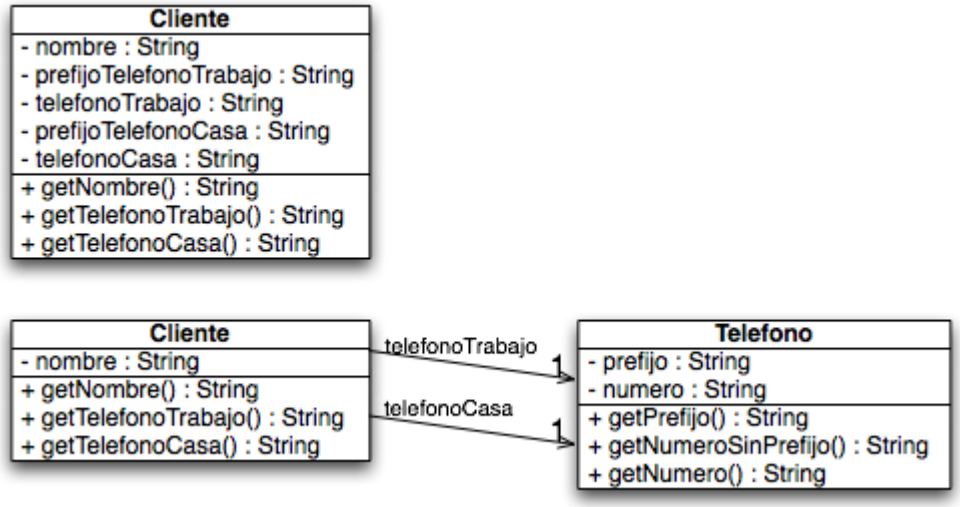
**ejemplorefactoring._03_mover**

# Técnicas de refactorización comunes



**Extraer una clase**

- Motivo: Una clase hace el trabajo que en realidad deberían hacer entre dos.
- Solución: Crear una nueva clase y mover los métodos y atributos relevantes de la clase original a la nueva.
- Observaciones:
  - La nueva clase debe asumir un conjunto de responsabilidades bien definido.
  - Implica crear una relación entre la nueva clase y la antigua.
  - Implica 'Mover atributo' y 'Mover método'
  - Se debe considerar si la nueva clase ha de ser pública o no.




**ejemplorefactoring._04_extraer (paso 01)**

# Técnicas de refactorización comunes

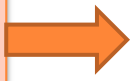


**Extraer un método**

- Motivo: Existe un fragmento de código (quizás duplicado) que se puede agrupar en una unidad lógica.
- Solución: Convertir el fragmento en un método cuyo nombre indique su propósito e invocarlo desde donde estaba el fragmento.
- Observaciones:
  - Se realiza a menudo cuando existen métodos muy largos.
  - Las variables locales que sólo se leen en ese trozo de código se convertirán en parámetros del nuevo método
  - Si algunas variables locales son modificadas en el código, son candidatas a ser valor de retorno del método
  - Considerar si debe ser público el nuevo método.

```
public void imprimir(double importe)
{
   imprimirCabecera();

   //Imprimir detalles
   System.out.println("nombre: "+ nombre);
   System.out.println("importe: "+ importe);
}
```



```
public void imprimir(double importe) {
   imprimirCabecera();
   imprimirDetalles(importe);
}

private void imprimirDetalles(double importe)
{
   System.out.println("nombre: "+ nombre);
   System.out.println("importe: "+ importe);
}
```

ejemplorefactoring._04_extraer (paso 02)