# REFACTORING Y DUPLICACIÓN DE CÓDIGO

# Refactoring

Hacer *refactoring* es hacer pequeñas transformaciones en el código que mantienen el sistema funcional, sin añadir nuevas funcionalidades.

> Refactoring is a **disciplined** technique for restructuring an existing body of code, altering its internal structure without changing its **external** behavior
>
> — M. Fowler, www.refactoring.com
>
> A change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing its observable behavior
>
> — M. Fowler. Refactoring. Improving the Design of Existing Code. Addison-Wesley, 2nd Edition, 2008.

## Motivos para hacer refactoring

- Duplicación de código
- Diseño no ortogonal
- Cambios (de requisitos, más conocimiento del problema)
- Uso del sistema (se descubre la imporancia de las cosas)
- Rendimiento
- Pasan todos los tests (es la oportunidad)

**Lecturas recomendadas**

- A. Hunt & D. Thomas. The Pragmatic Programmer. Addison-Wesley, 1999. Capítulo 40: *Refactoring*
- Steve McConnell. Code Complete: A practical handbook of software construction, 2nd edition, 2004.

## Conceptos relacionados con el refactoring

- Deuda técnica
- *Clean code* vs *dirty code*
- Tests unitarios y *Test-Driven Development* (TDD)
- Tufos o *code smells*

> **Lecturas recomendadas**
>
> - Refactoring Guru: What is Refactoring?
> - Refactoring Guru: Code Smells
> - Refactoring Guru: Refactoring techniques

¿Cuál es la primera razón para hacer refactoring?

## Ejemplos de refactoring

- **Código duplicado**
- Rutinas demasiado largas
- Bucles demasiado largos o demasiado anidados
- Clases poco cohesionadas
- Interfaz de una clase con un nivel de abstracción poco consistente
- Demasiados parámetros en una función
- Jerarquías de herencia en paralelo
- Muchas sentencias *case* en paralelo
- Hay muchos cambios que requieren modificaciones en paralelo a varias clases
- Etc.

# CASO PRÁCTICO: Cálculo de nóminas

## Implementación de nóminas v0.1

```
public class Empleado {
  Comparable id;
  String name;
  public Empleado(String id, String name) {
      this.id = id;
      this.name = name;
  }
  public void print() {
      System.out.println(id+" "+name);
  }
}
```

```java
public class Autonomo extends Empleado {
  String vatCode;
  public Autonomo(String id, String name, String vat) {
      this.id = id;
      this.name = name;
      this.vatCode = vat;
  }
  public void print() {
      System.out.println(id+" "+name+" "+vatCode);
  }
}
public class Prueba {
  public static void main(String[] args) {
    Empleado e = new Empleado("0001","Enrique");
    Empleado a = new Autonomo("0002","Ana","12345-A");
    e.print();
    a.print();
  }
}
```

En la implementación anterior, ¿dónde hay código duplicado?

- Código duplicado en los constructores de las clases y subclases
- Refactorizar delegando hacia la superclase

## Implementación de nóminas v0.2

- Requisito: los trabajadores autónomos cobran por horas (no tienen un salario fijo bruto)
- Incluimos el método `computeMonthlySalary` para el cálculo de la nómina mensual

```
public class Empleado {
  Comparable id;
  String name;
  float yearlyGrossSalary;
  public Empleado(String id, String name) {
      this.id = id;
      this.name = name;
  }
  void setSalary( float s ) { yearlyGrossSalary=s; }
  public void print() {
      System.out.print(id+" "+name);
  }
  public float computeMonthlySalary() {
      return yearlyGrossSalary/12;
  }
}
```

```java
public class Autonomo extends Empleado {
  String vatCode;
  float workingHours;
  public Autonomo(String id, String name, String vat) {
      super(id,name);
      this.vatCode = vat;
      this.workingHours = 0.0;
  }
  public float computeMonthlySalary() {
      return workingHours*Company.getHourlyRate()*(1.0+Company.getVatRate());
  }
  @Override
  public void print() {
      super.print();
      System.out.print(" "+vatCode);
  }
}
```

```
public class Prueba {
  public static void main(String[] args) {
    Empleado e = new Empleado("0001", "Enrique");
    Empleado a = new Autonomo("0002", "Ana", "12345-A");
    e.print();  System.out.println();
    a.print();  System.out.println();
  }
}
```

¿Están descohesionadas las clases?

- ¿Todos los empleados deben tener un salario anual `yearlyGrossSalary` bruto? Los autónomos no...
- El método de cálculo del salario está descohesionado

## Implementación de nóminas v0.3

```
public class Prueba {
  public static void main(String[] args) {
    Empleado e = new Plantilla("0001", "Pepe");
    e.setSalary(25000.0);
    Empleado a = new Autonomo("0002", "Ana", "12345-A");
    a.addWorkingHours(30.0);
    e.print(); System.out.println(" Salario: "+e.computeMonthlySalary()+" EUR");
    a.print(); System.out.println(" Salario: "+a.computeMonthlySalary()+" EUR");
  }
}
```

```
public abstract class Empleado {
  /* ... */
  public abstract float computeMonthlySalary();
}

public class Plantilla extends Empleado {
  float yearlyGrossSalary;

  /* ... */
  float setSalary( float s ) { yearlyGrossSalary=s; }
  public float computeMonthlySalary() {
      return yearlyGrossSalary/12;
  }
}
```

```java
public class Autonomo extends Empleado {
  String vatCode;
  float workingHours;

  public Autonomo(String id, String name, String vat) {
      super(id,name);
      this.vatCode = vat;
      this.workingHours = 0.0;
  }

  public void addWorkingHours(float workingHours){
    this.workingHours += workingHours;
  }

  public float computeMonthlySalary() {
      return workingHours*Company.getHourlyRate()*(1.0+Company.getVatRate());
  }

  @Override
  public void print() {
      super.print();
      System.out.print(" "+vatCode);
  }
}
```

# Código duplicado

## ¿Por qué no duplicar?

- Mantenimiento
- Cambios (no sólo a nivel de código)
- Trazabilidad

## Causas de la duplicación

1. **Impuesta**: No hay elección
2. **Inadvertida**: No me he dado cuenta
3. **Impaciencia**: No puedo esperar
4. **Simultaneidad**: Ha sido otro

> **Lectura recomendada**
>
> A. Hunt & D. Thomas. The Pragmatic Programmer. Addison-Wesley, 2019.
>
> Capítulo *DRY—The Evils of Duplication*

## Principio DRY – *Don't Repeat Yourself!*

- DRY no tiene que ver con el código, sino con el **conocimiento**. No se trata de no repetir código, sino de no repetir la lógica.

> Every piece of **knowledge** must have a single, unambiguous, authoritative representation within a system.
> -- Andrew Hunt & David Thomas. The Pragmatic Programmer. Addison-Wesley, 1999.

- Evitar abstracciones prematuras
- Preguntarse por el motivo de la duplicación

> Duplication is far cheaper than the wrong abstraction.
> -- Sandi Metz, RaisConf 2014.

# 1. Duplicación impuesta

- Representaciones múltiples de la información:
  - Varias implementaciones de un TAD que necesita guardar elementos de distintos tipos, cuando el lenguaje no permite genericidad
  - Esquema de BD configurado en la BD y en código fuente a través de un ORM
- Documentación del código:
  - Código incrustado en javadocs
- Casos de prueba:
  - Pruebas unitarias con jUnit (Cuidado!)
- Características del lenguaje:
  - C/C++ header files
  - IDL specs

## Cómo evitaba Java la duplicación en sus *containers*

Cuando el lenguaje no tenía capacidad de usar tipos genéricos (hasta el JDK 1.4), podría aparecer la necesidad de duplicar código a la hora de implementar un TAD contenedor, pues habría que repetir todo el código de manejo del TAD para cada tipo de elemento contenido.

Para evitarlo, Java usó un *workaround*: todas las clases en Java heredan de `Object`. Así una clase que implementara un TAD contenedor de elementos de otra clase, tan solo tenía que declarar los elementos contenidos de tipo `Object`.

A partir del JDK 1.5, se introdujeron los tipos genéricos y ya no era necesario usar dicho *workaround*, que se mantuvo por compatibilidad con versiones anteriores.

## Técnicas de solución

- **Generadores de código**: para evitar duplicar representaciones múltiples de la información
- Herramientas de **ingeniería inversa**: para generar código a partir de un esquema de BD – v.g. jeddict para crear clases JPA, visualizar y modificar BDs y automatizar la generación de código Java EE.
- **Plantillas**: Tipos genéricos del lenguaje (Java, C++, TypeScript, etc.) o mediante un motor de plantillas – v.g. Apache Velocity template language (VTL)
- **Metadatos**: Anotaciones @ en Java, decoradores en TypeScript, etc.
- Herramientas de **documentación** (v.g. asciidoctor: inclusión de ficheros).
- Herramientas de **programación literaria**
- Ayuda del **IDE**

¿Cómo reducir la duplicación de código al programar pruebas unitarias?

## Property-based testing

- Herramientas de *property-based testing*, como Hypothesis (python), RapidCheck (C++), jqwik (Java) o QuickCheck (originalmente para Haskell).
- Leer el Consejo nº 71 del libro de D. Thomas & A. Hunt. The Pragmatic Programmer: your journey to mastery, 20th Anniversary Edition, 2nd Edition, Addison-Wesley Professional, 2020.

## Ejemplo de Hypothesis en Python

Ejemplo de property-based testing con Hypothesis en Python:

```python
from hypothesis import given
import hypothesis.strategies as some

@given(some.lists(some.integers()))
def test_list_size_is_invariant_across_sorting(a_list):
  original_length = len(a_list)
  a_list.sort()
  assert len(a_list) == original_length

@given(some.lists(some.text()))
def test_sorted_result_is_ordered(a_list):
  a_list.sort()
  for i in range(len(a_list) - 1):
    assert a_list[i] <= a_list[i + 1]
```

# 2. Duplicación inadvertida

- Normalmente tiene origen en un diseño inapropiado.
- Fuente de numerosos problemas de integración.

## Ejemplo: código duplicado – versión 1

```
public class Line {
  public Point start;
  public Point end;
  public double length;
}
```

¿Dónde está la duplicación?

Realmente `length` ya está definido con `start` y `end`.

¿Mejor así...?

```
public class Line {
  public Point start;
  public Point end;
  public double length() {
     return start.distanceTo(end);
  }
}
```

¿Es conveniente aplicar siempre DRY?

- A veces se puede optar por violar DRY por razones de rendimiento...
  - *Memoization*: cachear los resultados de cómputos costosos
  - La técnica de memoization es menos problemática si queda dentro de los límites de la clase/módulo.
  - Otras razones de rendimiento: las cachés y los optimizadores de código también hacen su labor

## Ejemplo: aplicando memoization – versión 2

```
public class Line {
  private boolean changed;
  private double length;
  private Point start;
  private Point end;

  public void setStart(Point p) { start = p; changed = true; }
  public void setEnd(Point p)   { end   = p; changed = true; }
  public Point getStart() { return start; }
  public Point getEnd() { return end; }
  public double getLength() {
    if (changed) {
      length = start.distanceTo(end);
      changed = false;
    }
    return length;
  }
}
```

¿Es tan importante DRY en tiempos de la IA?

- Si se usa IA, ¿cuál es el coste de mantener código duplicado vs el coste de mantener sistemas muy acoplados?
  - El coste de la duplicación de código es principalmente el mantenimiento (esfuerzo humano). La IA puede ayudar con esto.
  - Pero el coste de elegir una abstracción incorrecta no disminuye. La IA todavía tiene dificultades con los sistemas sobreacoplados.

## Principio de acceso uniforme

> All services offered by a module should be available through a uniform notation, which does not betray whether they are implemented through storage or through computation
>
> – B. Meyer. Object-Oriented Software Construction. Prentice-Hall, 2nd edition, 1997.

Conviene aplicar el principio de acceso uniforme para que sea más fácil añadir mejoras de rendimiento (por ejemplo, caching)

## Ejemplo: acceso uniforme en C# – versión 3

```
public class Line {
  private Point Start;
  private Point End;
  private double Length;

  public Point Start {
    get { return Start; }
    set { Start = value; }
  }

  public Point End {
    get { return End; }
    set { Start = value; }
  }

  public double Length {
    get { return Start.distanceTo(End); }
  }
}
```

## Ejemplo: acceso uniforme en Scala

Llamadas a métodos con paréntesis:

```
class Complejo(real: Double, imaginaria: Double) {
  def re() = real
  def im() = imaginaria
  override def toString() =
    "" + re() + (if (im() < 0) "" else "+") + im() + "i"
}

object NumerosComplejos {
  def main(): Unit = {
    val c = new Complejo(1.2, 3.4)
    println("Número complejo: " + c.toString())
    println("Parte imaginaria: " + c.im())
  }
}
```

Llamadas a métodos sin paréntesis, igual que si fueran atributos:

```
class Complejo(real: Double, imaginaria: Double) {
  def re = real
  def im = imaginaria
  override def toString() =
    "" + re + (if (im < 0) "" else "+") + im + "i"
}

object NumerosComplejos {
  def main(): Unit = {
    val c = new Complejo(1.2, 3.4)
    println("Número complejo: " + c)
    println("Parte imaginaria: " + c.im)
  }
}
```

# 3. Duplicación por impaciencia

## El peligro del copy&paste

> Copy and paste is a design error
>
> -- Steve McConnell. Code Complete: A practical handbook of software construction, 2nd edition, 2004.

## Las prisas y los ahorros

"Vísteme despacio que tengo prisa" (*shortcuts make for long delays*).

Ejemplo: Fiasco del año 2000

# 4. Duplicación por simultaneidad

- No resoluble a nivel de técnicas de construcción
- Hace falta metodologías de integración, gestión de equipos y herramientas de comunicación
  - CI/CD (*Continuous Integration* / *Continuous Delivery*)
  - Prácticas DevOps

# Reglas para hacer refactoring

Según Fowler:

1. No hacer refactoring y añadir funcionalidad al mismo tiempo
2. Disponer de buenos tests antes de empezar. Pasarlos a menudo.
3. Dar pasos cortos:
    - mover un campo de una clase a otra
    - dividir un método
    - renombrar una variable

Añadimos...

- Reflejar cada cambio en un *commit* separado