

# UA4. Optimización y documentación

Entornos de Desarrollo – 1ºDAM  
Luis del Moral Martínez

versión 21.03  
Bajo licencia CC BY-NC-SA 4.0



# Contenidos del tema

## 1. Introducción

- 1.1 Mapa conceptual del tema
- 1.2 Refactorización, GIT y documentación
- 1.3 Algunos conceptos importantes

## 2. Refactorización

- 2.1 ¿En qué consiste?
- 2.2 Refactorizaciones más usuales
- 2.3 Code “smells”

## 3. Patrones de diseño

- 3.1 ¿En qué consisten?
- 3.2 Patrones de diseño más usuales

## 4. Control de versiones

- 4.1 ¿En qué consiste?
- 4.2 Almacenamiento de versiones
- 4.3 Flujos de trabajo

## 5. Documentación

- 5.1 Importancia de la documentación
- 5.2 Documentación de calidad
- 5.3 Tipos de documentación
- 5.4 Generación automática de documentación

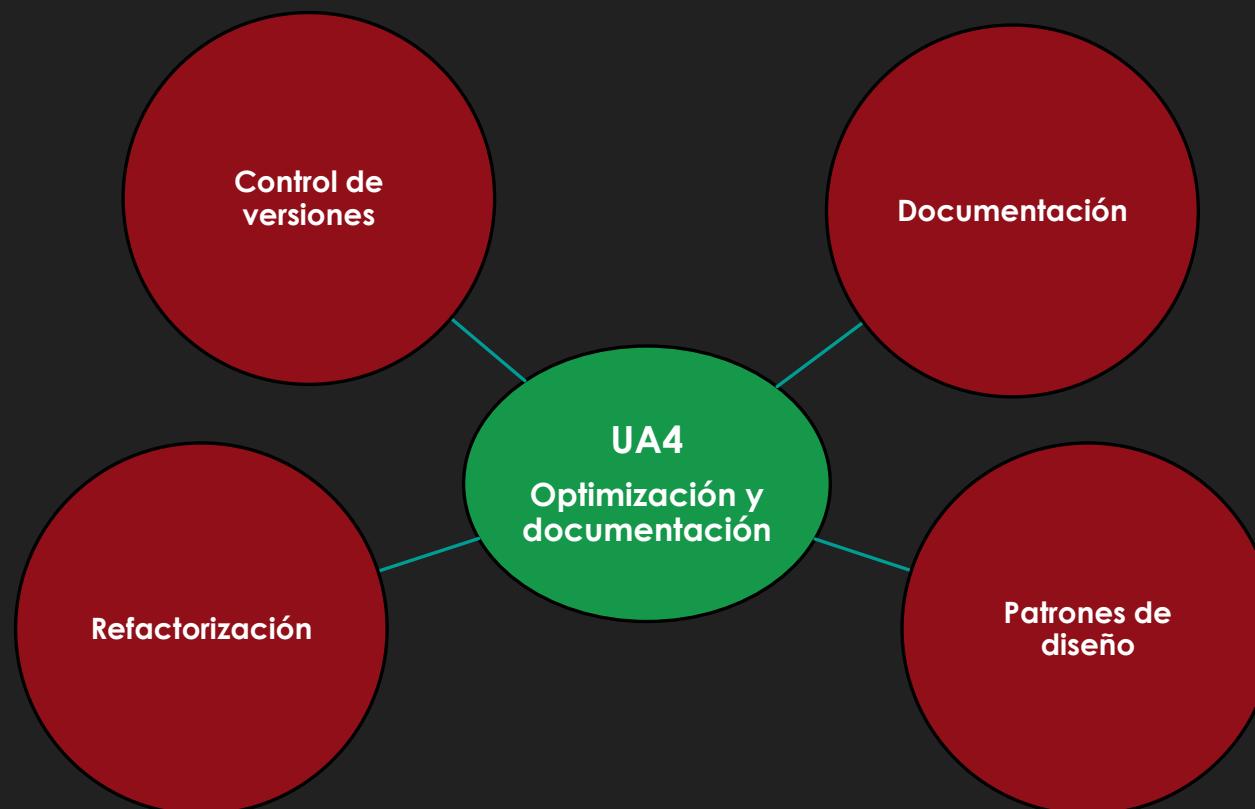
# Contenidos de la sección

## 1. Introducción

- 1.1 Mapa conceptual del tema
- 1.2 Refactorización, GIT y documentación
- 1.3 Algunos conceptos importantes

# 1. Introducción

## 1.1 Mapa conceptual del tema



# 1. Introducción

## 1.2 Refactorización, GIT y documentación

- Para convertirse en un buen programador hay que saber **refactorizar** código
- Al principio se programa “como se puede”, pero hay que buscar **arquitecturas eficientes**
- También es raro participar en un proyecto y no usar un **sistema de control de versiones**
- Finalmente, ningún desarrollador que se precie prescinde de la **documentación**
- Un programa poco o mal documentado es un mal producto final
- En este tema abordaremos estas y otras cuestiones que mejorarán la labor de desarrollo

# 1. Introducción

## 1.3 Algunos conceptos importantes

- **Factorizar**: dividir parte de la funcionalidad de un código en funciones (+ fácil de mantener)
- **Getter y Setter**: métodos de una clase que permiten acceder a un atributo privado
- **Inline**: agrupar en uno sólo el código que estaba dividido para que sea más legible
- **Refactorizar**: consiste en hacer que un código sea más fácil de entender y mantener
- **SCV**: siglas de sistemas de control de versiones (VCS)
- **Workflow (flujo de trabajo)**: forma en la que se trabaja con un sistema o software

# Contenidos de la sección

## 2. Refactorización

- 2.1 ¿En qué consiste?
- 2.2 Refactorizaciones más usuales
- 2.3 Code “smells”

# 2. Refactorización

## 2.1 ¿En qué consiste?

- Consiste en realizar **cambios internos** y **reestructurar** componentes de una aplicación
- El comportamiento de la aplicación **debe seguir siendo el mismo**
- **Ventajas de la refactorización**
  - Ayuda a encontrar errores
  - Ayuda a programar más rápido
  - Los programas se interpretan de forma más fácil
  - Los diseños son más robustos
  - Los programas tienen más calidad
  - Se evita duplicar la lógica de los programas

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- A continuación analizaremos las **refactorizaciones** más usuales
- Puedes aprender más sobre refactorizaciones en este [enlace](#)
- En este tema aplicaremos los conceptos aprendidos mediante ejercicios, usando Eclipse
- Al realizar refactorizaciones, se recomienda crear **casos de prueba** con **Junit**
- Los casos de prueba garantizarán que la aplicación sigue funcionando correctamente

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Extract method (reducción lógica)**
  - Es una de las refactorizaciones más sencillas
  - Se deben crear métodos que hagan lo que verdaderamente dicen
  - Los métodos cortos realizan tareas más específicas y crean métodos más complejos

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Extract method (reducción lógica)**
  - Código antes de refactorizar

```
void printCuenta (String nombre, double cantidad) {  
    printLogo();  
    System.out.println("nombre: " + nombre);  
    System.out.println("cantidad: " + cantidad);  
}
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Extract method (reducción lógica)**

- Código después de refactorizar

```
void printCuenta2 (String nombre, double cantidad) {  
    printLogo();  
    printDetalles(nombre, cantidad);  
}  
  
void printDetalles (String nombre, double cantidad) {  
    System.out.println("nombre: " + nombre);  
    System.out.println("cantidad: " + cantidad);  
}
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Métodos Inline (código embebido)**
  - En ocasiones interesa no factorizar el código
  - A veces, ocultar la lógica del programa factorizando el código puede ser una mala idea

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Métodos Inline (código embebido)**
  - **Ejemplo 1:** código antes de refactorizar

```
int entradaGratis (int edad) {  
    return (jubilado(edad)) ? 1 : 0;  
}  
  
boolean jubilado (int edad) {  
    return edad >= 65;  
}
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Métodos Inline (código embebido)**
  - **Ejemplo 1:** después de refactorizar

```
int entradaGratis (int edad) {  
    return (edad >= 65) ? 1 : 0;  
}
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Métodos Inline (código embebido)**
  - **Ejemplo 2:** código antes de refactorizar

```
int precioBase = pedido.getPrecioBase();
return (precioBase > 100);
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Métodos Inline (código embebido)**
  - **Ejemplo 2:** código después de refactorizar

```
return (pedido.getPrecioBase() > 100);
```

- Debemos evitar el uso de **variables temporales** para almacenar resultados intermedios
- Se recomienda el uso de **final** si la variable no va a ser instanciada de nuevo ni va a tomar otros valores

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Uso de variables autoaplicativas**
  - A veces nos obcecamos por escribir líneas de código muy complejo
  - Usando **variables autoaplicativas** el código resulta más **legible** y **sencillo**

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Uso de variables autoaplicativas**
  - **Ejemplo:** código antes de refactorizar

```
if ((idioma.toUpperCase().indexOf("RUS") > -1) &&
    (idioma.toUpperCase().indexOf("ALE") > -1) &&
    (nivelIngles > 0)) {
    // Mostrar mensajes en inglés
}
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Uso de variables autoaplicativas**
  - **Ejemplo:** código después de refactorizar

```
final boolean esRuso = idioma.toUpperCase().indexOf("RUS") > -1;
final boolean esAleman = idioma.toUpperCase().indexOf("ALE") > -1;
final boolean ingles = nivelIngles > 0;

if (esRuso && esAleman && ingles) {
    // Mostrar mensajes en inglés
}
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Mal uso de variables temporales**
  - Las **variables temporales** sólo deberían usarse o instanciarse una vez
  - No es buena idea que una variable temporal cambie su valor en el mismo bloque de código
  - En este caso, sería mejor usar otra variable auxiliar

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Mal uso de variables temporales**
  - **Ejemplo 1:** código antes de refactorizar

```
double tmp = pi * radio * radio;  
System.out.println (tmp);  
tmp = 2* pi * radio;  
System.out.println(tmp);
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Mal uso de variables temporales**
  - **Ejemplo 1:** código después de refactorizar

```
final double area = pi * radio * radio;  
System.out.println (area);  
final double perimetro = 2* pi * radio;  
System.out.println(perimetro);
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Mal uso de variables temporales**
  - **Ejemplo 2:** código antes de refactorizar

```
double salario (int horas, int horasExtra, int salarioBase) {  
    horas = horas + horasExtra * 1.5;  
    ...  
}
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Mal uso de variables temporales**
  - **Ejemplo 2:** código después de refactorizar

```
double salario (int horas, int horasExtra, int salarioBase) {  
    final int horasTrabajadas = horas + horasExtra * 1.5;  
    ...  
}
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Cambiar algoritmos**

- Si crees que existe una forma más fácil de hacer algo, sustituye el código
- La factorización persigue simplificar cosas más complejas en otras más sencillas
- Si crees que puedes usar librerías externas, ¡úsalas!
- Al sustituir algoritmos, puedes usar **pruebas de regresión** para comparar resultados

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- Cambiar algoritmos

- Ejemplo: código antes de refactorizar

```
String buscarAnimal (String[] animales) {  
    for (int i = 0; i < animales.length; i++) {  
        if (animales[i].equals("perro")) { return "perro"; }  
        if (animales[i].equals("loro")) { return "loro"; }  
        if (animales[i].equals("tortuga")) { return "tortuga"; }  
    }  
    return "no encontrado";  
}
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- Cambiar algoritmos
  - Ejemplo: código después de refactorizar

```
String buscarAnimal (String[] animales) {  
    for (String animal : animales){  
        if (animal.equals("perro")) { return "perro"; }  
        if (animal.equals("loro")) { return "loro"; }  
        if (animal.equals("tortuga")) { return "tortuga"; }  
    }  
    return "no encontrado";  
}
```

# 2. Refactorización

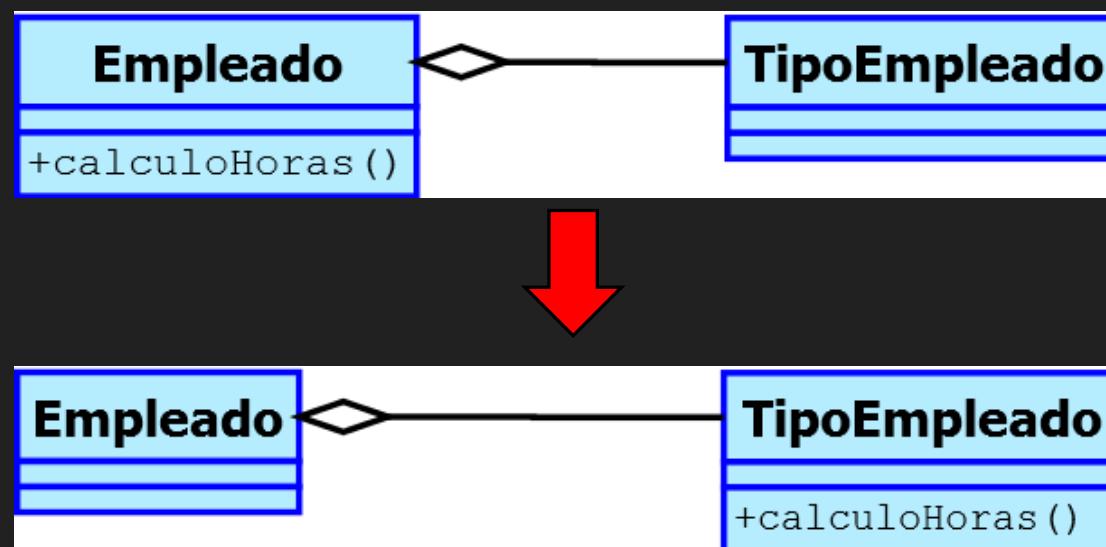
## 2.2 Refactorizaciones más usuales

- **Mover métodos entre clases**
  - En ocasiones se crean clases complejas y otras que realizan pocas tareas
  - Resulta interesante **balancear** y **repartir la carga de trabajo**
  - Aunque aún no hemos visto los diagramas de clases, realizaremos un breve planteamiento

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Mover métodos entre clases**
  - **Ejemplo:** mover métodos entre clases



# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Mover métodos entre clases**

- **Ejemplo:** Clases **Empleado** y **TipoEmpleado** (antes de la refactorización)

```
public class TipoEmpleado {  
    private String tipo;  
    private double horaBase;  
  
    public tipoEmpleado (String t, double h)  
    { ... }  
  
    public String getTipo()  
    { ... }  
  
    public String getHorasBase()  
    { ... }  
}
```

```
public class Empleado {  
    private int horas;  
    private int horasExtra;  
    private TipoEmpleado tipo;  
  
    public double calculoHoras() {  
        if (tipo.getTipo().equals("supervisor")) {  
            return horas + horasExtra * 1.75;  
        } else if (tipo.getTipo.equals("cajero")) {  
            return horas + horasExtra * 1.40;  
        }  
    }  
    public double getSueldo() {  
        return tipo.getHorasBase() * calculoHoras();  
    }  
}
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Mover métodos entre clases**

- **Ejemplo:** Clases **Empleado** y **TipoEmpleado** (después de la refactorización)

```
public class TipoEmpleado {  
    private String tipo;  
    private double horaBase;  
  
    public tipoEmpleado (String t, double h)  
    { ... }  
  
    public String getTipo()  
    { ... }  
  
    public String getHorasBase()  
    { ... }  
  
    public double calculoHoras()  
    { ... }  
}
```

```
public class Empleado {  
    private int horas;  
    private int horasExtra;  
    private TipoEmpleado tipo;  
  
    public double getSueldo() {  
        return tipo.getHorasBase() * calculoHoras();  
    }  
}
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Mover variables miembro entre clases**

- En ocasiones puede resultar conveniente mover atributos (variables miembro) entre clases
- Resulta útil ubicarla en la clase donde va a tener más uso
- **Del ejemplo anterior:**

```
String class Empleado {  
    private int horas;  
    private int horasExtra;  
    private double horaseBase;  
    private TipoEmpleado tipo;  
    ...  
}
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Autoencapsular campos (*getters* y *setters*)**
  - Cuando una variable va a ser utilizada con asiduidad, se recomienda **encapsularla**
  - El encapsulamiento se realiza usando métodos ***getters*** y ***setters***
  - **Ejemplo rápido**

```
private String tipo;  
public String getTipo() { return tipo; }  
public void setTipo(String tipo) { this.tipo = tipo; }
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Extraer clases**
  - Imagina que en el ejemplo anterior no existiera la clase **TipoEmpleado**
  - Si continúan añadiéndose métodos y atributos a la clase **Empleado**, ésta puede **crecer infinitamente**
  - La mejor alternativa es extraer su funcionalidad a otra clase (**TipoEmpleado**)

# 2. Refactorización

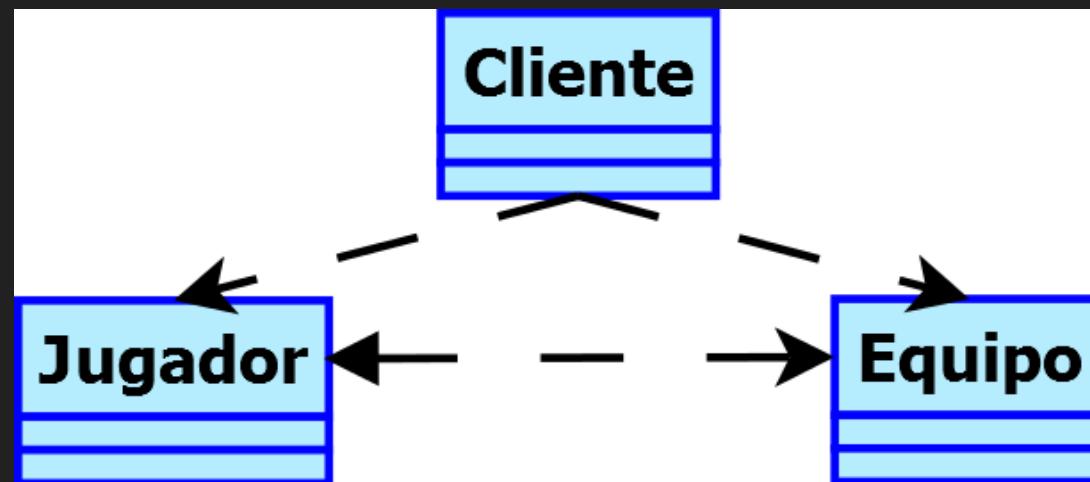
## 2.2 Refactorizaciones más usuales

- **Clases delegadas**
  - A veces interesa que ciertas clases no sean accesibles a cualquier parte de una clase concreta

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Clases delegadas**
  - **Ejemplo:** equipos de fútbol)

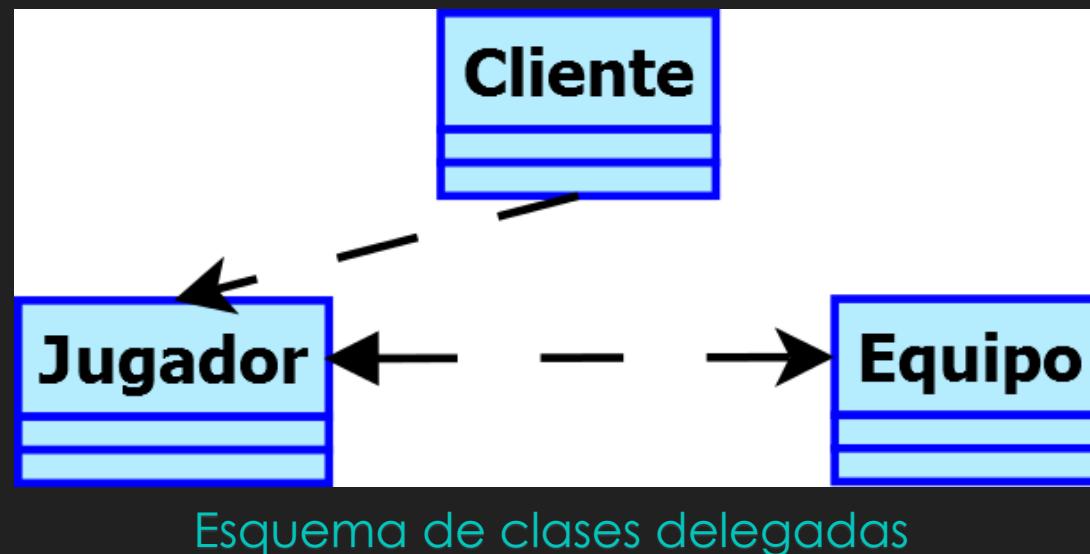


Interrelación entre una clase cliente y dos clases que están relacionadas entre sí

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Clases delegadas**
  - **Ejemplo:** equipos de fútbol)



# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Usar constantes**

- Hay que usar **constantes** siempre que sea posible
- **Ejemplo rápido**

```
static final double DESCUENTO_DIRECTO = 0.95;
double descuento(double unidades, double precioUnitario)
{
    return unidades * DESCUENTO-DIRECTO * precioUnitario;;
}
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Encapsular Arrays**

- Los **arrays** son usados en infinidad de aplicaciones
- Se recomienda usar **colecciones** (que son más eficientes)
- Si el uso de un array es irremediable, se recomienda **encapsularlo**

# 2. Refactorización

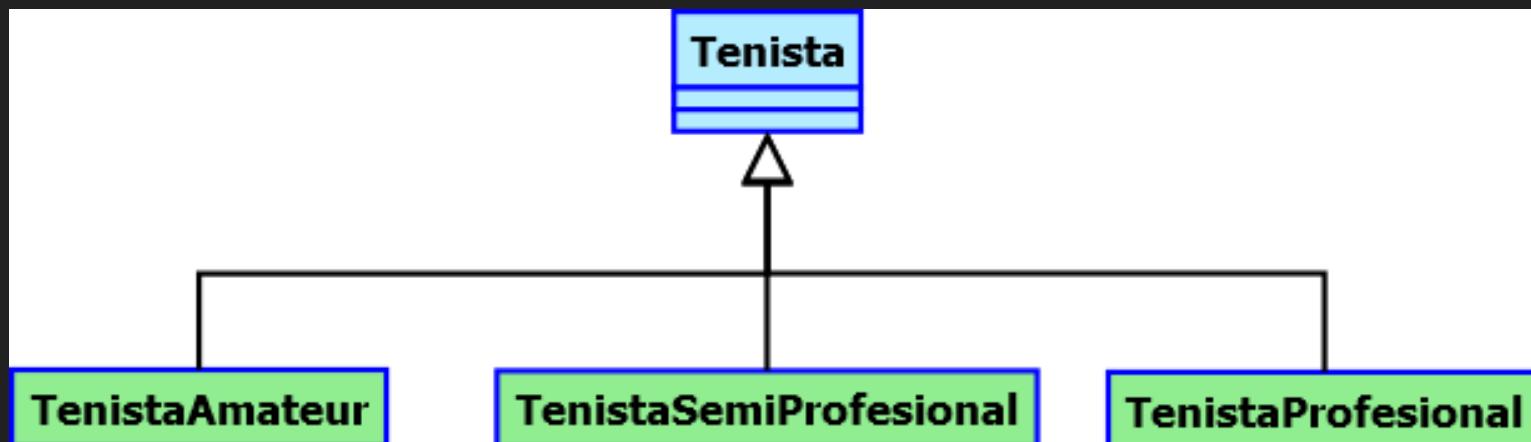
## 2.2 Refactorizaciones más usuales

- **Reemplazar tipos de objeto con subclases**
  - En ocasiones se usan variables internas para clasificar objetos
  - Se recomienda cambiar este código y usar **polimorfismo**, creando **clases derivadas**

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Reemplazar tipos de objeto con subclases**
  - **Ejemplo:** diagrama de clases con herencia



# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Reemplazar tipos de objeto con subclases**

- **Ejemplo:** clase Tenista (antes de la refactorización)

```
String class Tenista {  
    private final int AMATEUR = 1;  
    private final int SEMIPROFESIONAL = 2;  
    private final int PROFESIONAL = 3;  
    private final int tipo;  
  
    public Tenista(int tipo)  
    {  
        this.tipo = tipo;  
    }  
  
    public int getTipo()  
    {  
        return tipo;  
    }  
}
```

# 2. Refactorización

## 2.2 Refactorizaciones más usuales

- **Reemplazar tipos de objeto con subclases**
  - **Ejemplo:** clase Tenista (después de la refactorización)

```
String class Tenista {  
    private final int AMATEUR = 1;  
    private final int SEMIPROFESIONAL = 2;  
    private final int PROFESIONAL = 3;  
    private final int tipo;  
    public int getTipo() { return tipo; }  
    public Tenista create(int Tipo)  
    {  
        switch (tipo) {  
            case AMATEUR: return new TenistaAmateur(); break;  
            case SEMIPROFESIONAL: return new TenistaSemiProfesional(); break;  
            case PROFESIONAL: return new TenistaProfesional(); break;  
            default: throw new IllegalArgumentException("tipo incorrecto"); break;  
        }  
    }  
}
```

# 2. Refactorización

## 2.3 Code “smells”

- Es el código que “huele”, y debe ser **mejorado (refactorización)**
- A continuación, analizaremos este [\*\*enlace\*\*](#) para conocer los **code “smells”** más comunes
- Realizaremos ejemplos y en **Programación** podrás aplicar los conceptos estudiados

# Contenidos de la sección

## 3. Patrones de diseño

- 3.1 ¿En qué consisten?
- 3.2 Patrones de diseño más usuales

# 3. Patrones de diseño

## 3.1 ¿En qué consisten?

- Los patrones de diseño están íntimamente ligados a la programación orientada a objetos
- Un patrón es una solución a un **problema típico de diseño**
  - **Principios SOLID**
    - **Single responsibility (S)**: los objetos deben tener una única responsabilidad
    - **Open-Closed (O)**: cualquier entidad debe estar abierta para su extensión, pero cerrada a su modificación
    - **Liskov substitution (L)**: si se sustituye un objeto por una subclase suya, el funcionamiento debería ser idéntico
    - **Interface segregation (I)**: es mejor crear muchas interfaces específicas separadas antes de una interfaz genérica
    - **Dependency inversion (D)**: se debe depender de abstracciones, no de implementaciones concretas

# 3. Patrones de diseño

## 3.1 ¿En qué consisten?

- **Características de los patrones de diseño**
  - Pueden usarse bajo distintas circunstancias (reusabilidad)
  - Han resuelto problemas parecidos en otras ocasiones (efectividad)
  - Permiten comunicación fluida y común entre ingenieros y desarrolladores
  - Estandarización
  - Facilitan la comprensión de proyectos o estructuras más complejas
  - No impiden que se use otra estrategia de diseño

# 3. Patrones de diseño

## 3.1 ¿En qué consisten?

- **Ley de Demeter**
  - Estas reglas establecen el **principio del menor conocimiento**
  - Persiguen que un objeto tenga una **mínima interdependencia** con otros objetos
  - Se desarrollaron en Boston y se basan en los siguientes principios:
    - Cada unidad de software sólo debe interactuar con sus “amigos” y no con “extraños”
    - Cada unidad de software sólo puede hablar con sus “amigos inmediatos”
    - Cada unidad de software sólo debería conocer las unidades con las que está íntimamente relacionada

# 3. Patrones de diseño

## 3.2 Patrones de diseño más usuales

- A continuación, analizaremos los patrones más útiles en este [enlace](#)
- Realizaremos ejemplos y en **Programación** podrás aplicar los conceptos estudiados
- Los patrones fueron propuestos por la “panda de los cuatro” (*Gang of Four*)
- Aparecen por primera vez en el libro [Design Patterns](#) (Gamma E. et al)

# Contenidos de la sección

## 4. Control de versiones

- 4.1 ¿En qué consiste?
- 4.2 Almacenamiento de versiones
- 4.3 Flujos de trabajo

# 4. Control de versiones

## 4.1 ¿En qué consiste?

- Un producto software se modifica en infinidad de ocasiones durante su ciclo de vida
- Es necesario tener una herramienta que mantenga un control de los cambios realizados
- En este ciclo usaremos **GIT**, un sistema de control de versiones desarrollado por **Linus Torvalds**

# 4. Control de versiones

## 4.1 ¿En qué consiste?

- **Características de GIT**
  - Es un **sistema de control de versiones** gratuito, de código abierto y distribuido
  - Puede emplearse tanto para proyectos **pequeños** como proyectos de **mayor envergadura**
  - Es fácil de aprender y proporciona un gran rendimiento

# 4. Control de versiones

## 4.2 Almacenamiento de las versiones

- **Clasificación de los sistemas de control de versiones**
  - Los sistemas de control de versiones se clasifican en función de **cómo se almacena el código**:
    - **Sistemas distribuidos**
    - **Sistemas centralizados**

# 4. Control de versiones

## 4.2 Almacenamiento de las versiones

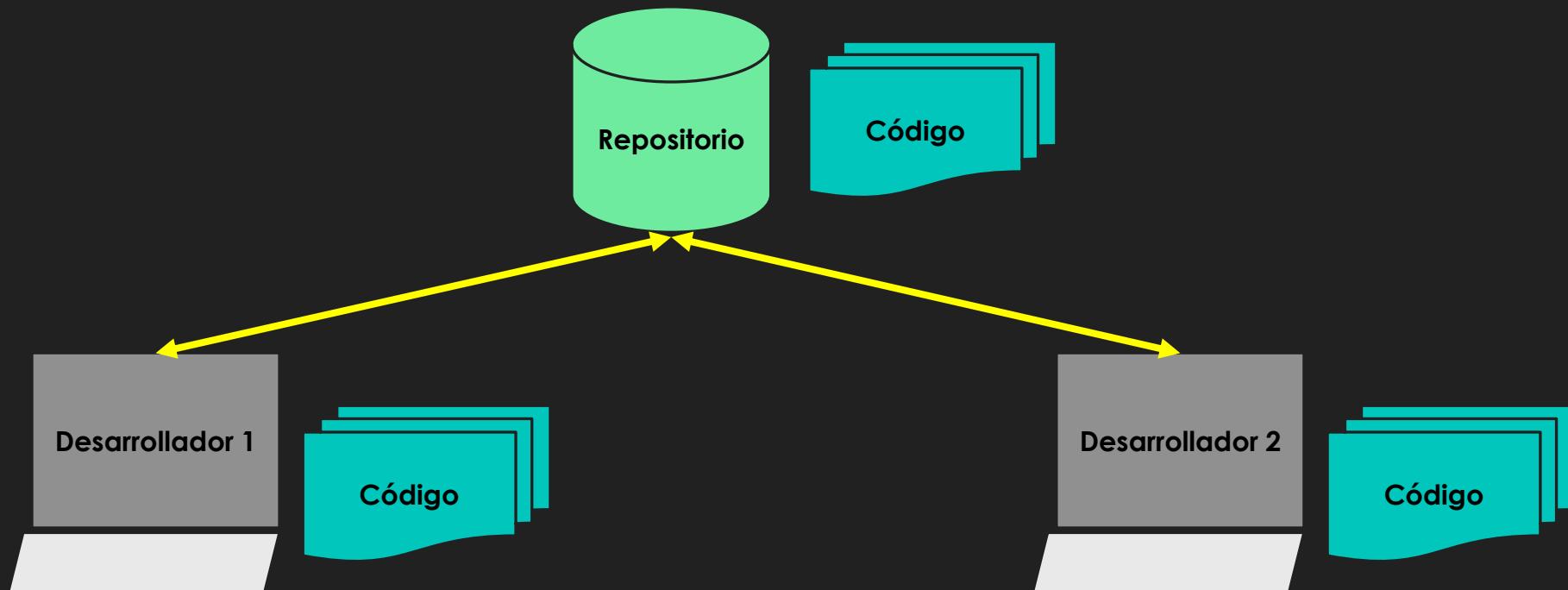
- **Sistemas centralizados**

- Tienen una gestión más sencilla
- El repositorio es **único** y en él se almacena todo el código del software
- Tienen un mayor control y existe un número único de versión
- Las ramas o versiones alternativas se crean únicamente en el servidor
- Por contrapartida, sólo existe una **única ubicación de la información** (cuidado con los backups)

# 4. Control de versiones

## 4.2 Almacenamiento de las versiones

- Sistemas centralizados



# 4. Control de versiones

## 4.2 Almacenamiento de las versiones

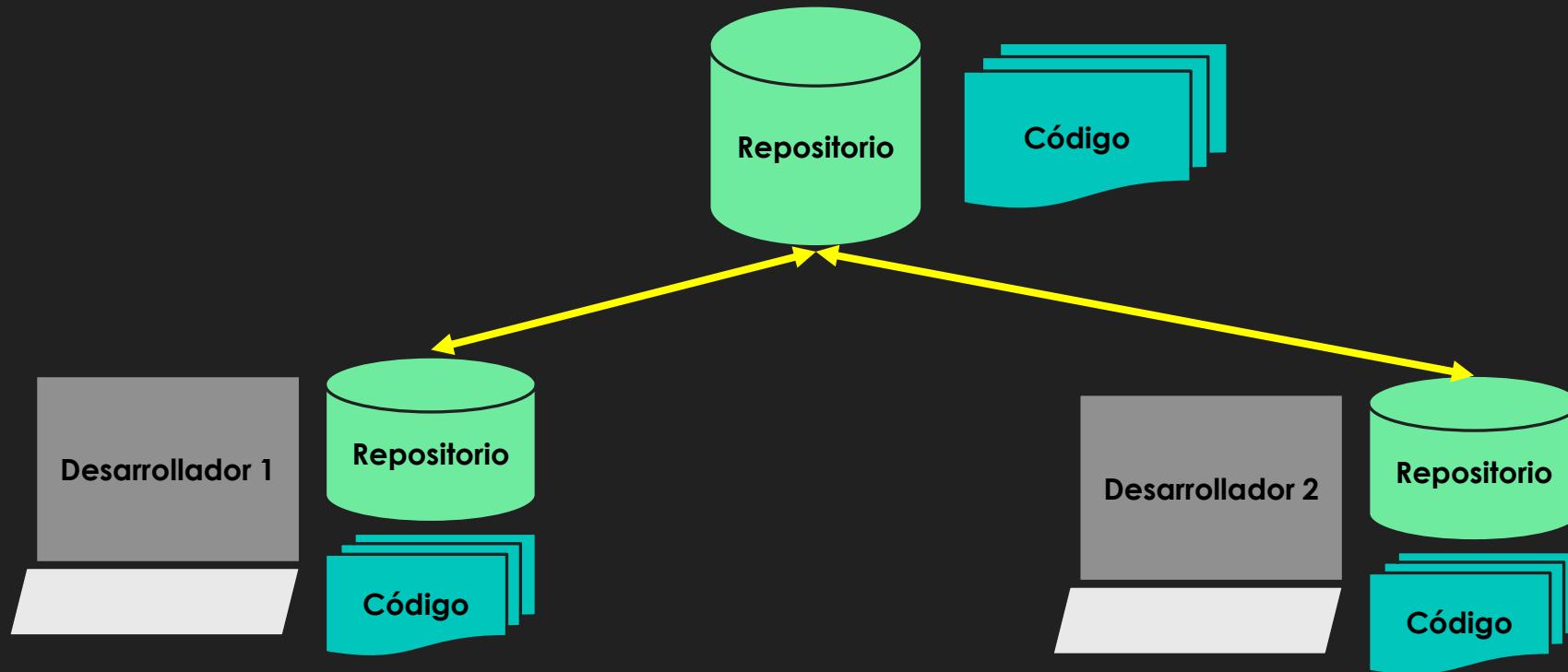
### ■ Sistemas distribuidos

- Cada programador tiene una **copia del repositorio** (cada desarrollador tiene una copia completa)
- En caso de fallo hay **múltiples copias** para poder restaurar
- En el servidor principal reside la **copia principal** (repositorio principal)
- Los desarrolladores pueden crear ramas locales para hacer ciertas pruebas o modificaciones
- El trabajo es mucho más **ágil**
- Se reduce la **carga de trabajo** del servidor
- **GIT** es un sistema de control de versiones **distribuido**

# 4. Control de versiones

## 4.2 Almacenamiento de las versiones

- Sistemas distribuidos



# 4. Control de versiones

## 4.3 Flujos de trabajo

- **Formas de trabajar en un sistema de control de versiones**
  - Generalmente existen dos formas de trabajar con un sistema de control de versiones (flujo de trabajo):
    - **En exclusividad:**
      - Un desarrollador trabaja con un módulo concreto y es responsable del mismo
      - El repositorio se bloquea para que no sea modificada
      - Cuando termina el trabajo, lo notifica al repositorio y ya queda disponible al resto de usuarios

# 4. Control de versiones

## 4.3 Flujos de trabajo

- **Formas de trabajar en un sistema de control de versiones**
  - Generalmente existen dos formas de trabajar con un sistema de control de versiones (flujo de trabajo):
    - **Colaborativamente:**
      - Cada usuario tiene su propia copia local
      - Cuando termina los cambios, los sube al repositorio común
      - Pueden aparecer conflictos en las diferentes modificaciones (es crucial coordinar los cambios)

# 4. Control de versiones

## 4.3 Flujos de trabajo

- **Flujo de trabajo en un sistema centralizado**
  - Cada desarrollador publica sus actualizaciones en el servidor centralizado
  - Funcionan de forma **exclusiva**
  - Si alguien está modificando un elemento ningún otro desarrollador puede realizar modificaciones

# 4. Control de versiones

## 4.3 Flujos de trabajo

- **Flujo de trabajo en un sistema distribuido**
  - Proporciona un mayor control sobre las versiones del proyecto
  - Los desarrolladores actualizan sus repositorios públicos
  - El gestor de integración actualiza el repositorio común con los cambios realizados
  - Como se ha comentado, pueden surgir **conflictos** a la hora de actualizar los cambios

# Contenidos de la sección

## 5. Documentación

- 5.1 Importancia de la documentación
- 5.2 Documentación de calidad
- 5.3 Tipos de documentación
- 5.4 Generación automática de documentación

# 5. Documentación

## 5.1 Importancia de la documentación

- Durante el proceso de desarrollo del software se genera mucha **documentación**
- Es mejor no generar documentación a generar **documentación inservible**
- La documentación va dirigida tanto a programadores como a usuarios, clientes, testers...
- La documentación debe ser de **calidad** para que pueda servir como referencia
- Lamentablemente, se suele dedicar poco tiempo a la documentación (**fase transversal**)
- **Anécdota:** “la ayuda no ayuda”

# 5. Documentación

## 5.2 Documentación de calidad

- Algunas pautas para escribir **documentación de calidad**:
  1. Hacer **esquemas** antes de escribir e incluirlos a la documentación
  2. Clasificar la información según su **importancia** (organizar y dividir en secciones, anexos...)
  3. Realizar resúmenes, plantillas, síntesis...
  4. Usar **estándares** y **herramientas** de documentación (Javadoc...)
  5. Anticiparse y responder las **preguntas** que pueda tener el público de dicha documentación

# 5. Documentación

## 5.2 Documentación de calidad

- Algunas pautas para escribir **documentación de calidad**:
  6. Escribir con **claridad** y de forma **concisa**
  7. Elegir la **herramienta** de documentación adecuada
  8. Crear **manuales de referencia** (más técnicos y específicos) y **manuales de usuario** (para el usuario final)
  9. Considerar el **nivel técnico** del usuario que leerá la documentación
  10. Especificar lo que el usuario **puede** y lo que **no debe** hacer

# 5. Documentación

## 5.3 Tipos de documentación

- La documentación de un software debe cubrir **aspectos técnicos** (qué) y **funcionales** (cómo)
- La tarea o fase de documentación de un proyecto es **transversal** (inicia y acaba con él)
- Podría decirse, en realidad, que **la documentación nunca termina**
- Aunque ya no se use el **modelo en cascada**, en **SCRUM** también precisa de documentación

# 5. Documentación

## 5.3 Tipos de documentación

- **Documentación en las diferentes fases del proyecto**
  - **Inicio**
    - Planificación del proyecto
    - Estimaciones y rentabilidad
    - Debe realizarla personal cualificado y con experiencia
    - **Documentos:** planificación, estimaciones, soluciones, análisis de costes

# 5. Documentación

## 5.3 Tipos de documentación

- **Documentación en las diferentes fases del proyecto**
  - **Análisis**
    - Recopilación y análisis de requisitos del cliente
    - La especificación de requisitos se puede ver como un **acuerdo contractual** cliente-desarrollador
    - **Documentos:** especificación de requisitos

# 5. Documentación

## 5.3 Tipos de documentación

- **Documentación en las diferentes fases del proyecto**
  - **Diseño**
    - Determina requisitos generales y la arquitectura de la aplicación
    - Se define cada subconjunto de la aplicación
    - Los documentos son más técnicos
    - **Documentos**: diagramas de clases, diagramas de casos de uso, historias de usuario...

# 5. Documentación

## 5.3 Tipos de documentación

- **Documentación en las diferentes fases del proyecto**
  - **Implementación**
    - Gran parte del código se documenta en el propio código fuente
    - También se generan documentos auxiliares que documentan los métodos, entradas, salidas...
    - Hay que tener en cuenta que el código será **mejorado** o **mantenido** en el futuro
    - **Un programador o desarrollador que no documenta no está haciendo bien su trabajo**

# 5. Documentación

## 5.3 Tipos de documentación

- **Documentación en las diferentes fases del proyecto**
  - **Pruebas**
    - Documentación de las pruebas funcionales y las pruebas técnicas
    - **El documento de las pruebas recibirá el visto bueno el cliente**

# 5. Documentación

## 5.3 Tipos de documentación

- **Documentación en las diferentes fases del proyecto**
  - **Explotación**
    - Una vez se ha liberado el software, se instala en el entorno real de producción
    - Se deben recopilar las **incidencias** (fallos del software) y las **necesidades de mejora**
    - Quizás surja la necesidad de **nuevas funcionalidades** (también hay que documentarlas)

# 5. Documentación

## 5.3 Tipos de documentación

- **Documentación en las diferentes fases del proyecto**
  - **Mantenimiento**
    - Se documentan los procedimientos correctivos y las actualizaciones del software
    - Es preciso recurrir a la documentación técnica generada anteriormente
    - Se debe documentar cada labor de mantenimiento (quién, qué, cómo y porqué)

# 5. Documentación

## 5.4 Generación automática de documentación

- Existen herramientas que permiten generar documentación externa a partir del código fuente
- En Java, se puede utilizar Javadoc para transformar los comentarios del código en HTML
- **Javadoc** genera una página web a partir de todos los comentarios generados
- A continuación vamos a ver un **ejemplo práctico** de cómo podemos usar Javadoc
- Puedes conocer más sobre Javadoc en este [enlace](#)

# Información complementaria

- Tutorial sobre refactorización desde cero: [enlace](#)
- Tutorial sobre refactorización en Eclipse: [enlace](#)
- Tutorial sobre patrones de diseño desde cero: [enlace](#)
- Descargar GIT: [enlace](#)
- Introducción a GIT (vídeo): [enlace](#)
- Tutorial de GIT y GitHub desde cero (vídeo): [enlace](#)
- GIT Markdown: [enlace](#)
- ¿Qué es Javadoc y cómo insertarlo en Eclipse? (vídeo): [enlace](#)

# Créditos de las imágenes y figuras

## Cliparts e iconos

- **Obtenidos mediante la herramienta web [IconFinder](#)** (según sus disposiciones):
  - Diapositivas 1
  - Según la plataforma IconFinder, dicho material puede usarse libremente (free commercial use)
  - A fecha de edición de este material, todos los cliparts son free for commercial use (sin restricciones)

## Resto de diagramas, gráficas e imágenes

- Se han desarrollado en **PowerPoint** y se han incrustado en esta presentación
- Todos estos materiales se han desarrollado por el autor
- Para el resto de recursos se han especificado sus fabricantes, propietarios o enlaces
- Si no se especifica copyright, el recurso es de desarrollo propio