

Asignatura

# Entornos de desarrollo



**UNIVERSAE**  
Instituto Superior de FP

Asignatura

Entornos de desarrollo

## UNIDAD 4

Optimización y documentación



UNIVERSAE  
Instituto Superior de FP

# Introducción

## Índice

1. Refactorización.
2. Patrones de diseño.
3. Control de versiones.
  1. Almacenamiento de las distintas Versiones.
  2. Tipos de colaboración en un SCV.
4. Documentación.
  1. Escritura de documentación de calidad.
  2. Tipos de documentación.
  3. Generación automática de documentación.





# Refactorización

## Ventajas

- ☐ Descubrir errores ocultos en el código.
- ☐ Aumentar la velocidad de ejecución del programa.
- ☐ Simplificar el código.
- ☐ Código más robusto y de mayor calidad.
- ☐ Evitar redundancias y duplicados.

# *Extract method o* reducción lógica

## Ventajas

- ❑ Concretar las tareas en funciones más específicas.
- ❑ Emplear otros métodos de complejidad mayor.

```
public void construirCuerpo() {  
    construirCabeza();  
    construirBrazo();  
    construirPierna();  
    construirPie();  
}  
  
private void construirPie() {  
    // Código para construir un pie  
}  
  
private void construirPierna() {  
    // Código para construir una pierna  
}  
  
private void construirBrazo() {  
    // Código para construir un brazo  
}  
  
private void construirCabeza() {  
    // Código para construir una cabeza  
}
```

# Métodos *inline* o código embebido

## Ventajas

- ❑ Redacción de código más simple.
- ❑ Mejorar la eficiencia en la ejecución de los procesos.
- ❑ Reducir el uso de variables temporales.

```
public boolean esNumeroMayor(int num1, int num2) {  
    boolean resultado;  
    if (num1 > num2)  
        resultado = true;  
    else  
        resultado = false;  
    return resultado;  
}
```

Antes

```
public boolean esNumeroMayor(int num1, int num2) {  
    return num1 > num2 ? true : false;  
}
```

Refactorizado



# Variables autoexplicativas

## Ventajas

- ❑ Redacción de código sencilla y legible.
- ❑ Facilita el mantenimiento.

```
public static void contarDinero(int dinero) {  
    int monedero = dinero;  
    if (monedero / 500 > 0) {  
        System.out.println("Billetes de 500: " + dinero / 500);  
        monedero -= ((monedero / 500) * 500);  
    }  
    if (monedero / 200 > 0) {  
        System.out.println("Billetes de 200: " + dinero / 200);  
        monedero -= ((monedero / 200) * 200);  
    }  
}
```

Antes

```
public static void contarDinero(int dinero) {  
    int monedero = dinero;  
    int cantidadBilletes500;  
    int cantidadBilletes200;  
  
    cantidadBilletes500 = monedero/500;  
    monedero-= cantidadBilletes500 * 500;  
  
    cantidadBilletes200 = monedero/200;  
    monedero-= cantidadBilletes200 * 200;  
  
    System.out.println("Billetes de 500: " +cantidadBilletes500);  
    System.out.println("Billetes de 200: " +cantidadBilletes200);  
}
```

Refactorizado

# Mal uso de variables temporales

## Ventajas

- ❑ Reducir el uso de memoria del programa.
- ❑ Aumentar la eficiencia del proceso.
- ❑ Hacer el código más fácil de mantener y modificar.

```
public static void sumarRestarNumeroXVeces(int numero, int cantidad) {  
    int j = 0;  
    int a = cantidad;  
  
    while(a>0) {  
        j += numero;  
        a--;  
    }  
    System.out.println("Suma: " + j);  
  
    j = 0;  
    a = cantidad;  
  
    while(a>0) {  
        j -= numero;  
        a--;  
    }  
  
    System.out.println("Resta: " + j);  
}
```

Antes

```
public static void sumarRestarNumeroXVeces(int numero, int cantidad) {  
    int resultadoSuma = 0;  
    int resultadoResta = 0;  
    int numeroVecesSuma = cantidad;  
    int numeroVecesResta = cantidad;  
  
    while(numeroVecesSuma>0) {  
        resultadoSuma += numero;  
        numeroVecesSuma--;  
    }  
  
    while(numeroVecesResta>0) {  
        resultadoResta -= numero;  
        numeroVecesResta--;  
    }  
  
    System.out.println("Suma: " + resultadoSuma);  
    System.out.println("Resta: " + resultadoResta);  
}
```

Refactorizado



# Cambiar algoritmos

## Ventajas

- ☐ Hacer el código más legible para el desarrollador.
- ☐ Más fácil de mantener y modificar.

```
public static int producto5PrimerosNumeros() {  
    int i = 0; int resultado = 1; while (i < 5) { if (i == 0) resultado =  
    resultado * 1; else resultado=resultado*(i+1); i++; } return resultado;  
}
```

Antes

```
public static int producto5PrimerosNumeros() {  
    int resultado = 1;  
    for (int i = 1; i <= 5; i++) {  
        resultado *= i;  
    }  
    return resultado;  
}
```

Refactorizado

# Autoencapsular campos (*getters* y *setters*)

## Ventajas

- ❑ Mejorar la gestión de las variables al ámbito de cada clase.
- ❑ Mejora la integración con *frameworks* de desarrollo posteriores.
- ❑ Permitir a las subclases modificar estos métodos según sus propias necesidades.

```
public class Coche {  
  
    public String tipo;  
    public String modelo;  
  
}  
  
public static void main(String[] args) {  
    Coche coche = new Coche();  
    coche.modelo = "modelo de coche";  
}
```

Antes

```
public class Coche {  
  
    private String tipo;  
    private String modelo;  
  
    public String getTipo() {  
        return tipo;  
    }  
  
    public void setTipo(String tipo) {  
        this.tipo = tipo;  
    }  
  
    public String getModelo() {  
        return modelo;  
    }  
  
    public void setModelo(String modelo) {  
        this.modelo = modelo;  
    }  
  
}  
  
public static void main(String[] args) {  
    Coche coche = new Coche();  
    coche.setModelo("modelo de coche");  
}
```

Refactorizado



# Foreign methods o métodos foráneos

## Ventajas

- ❑ Aprovechar el código ya escrito.
- ❑ Realizar operaciones muy concretas en cada método distribuyendo mejor la responsabilidad de tareas.

```
public static void operaciones(int tipo, float[] operandos) {  
    switch(tipo) {  
        case 1:  
            System.out.println("Raiz " + Math.sqrt(operandos[0]));  
            break;  
        case 2:  
            double a = operandos[0];  
            double b = operandos[1];  
            double c = operandos[2];  
  
            double resultadoPositivo = -b + Math.sqrt( Math.pow(b, 2) - (4 * a * c) ) / (2 * a);  
            double resultadoNegativo = -b - Math.sqrt( Math.pow(b, 2) - (4 * a * c) ) / (2 * a);  
            System.out.println("Ecuacion segundo grado : "  
                + "Resultado + "+ resultadoPositivo + "Resultado - "+resultadoNegativo);  
            break;  
    }  
}
```

Antes



# Foreign methods o métodos foráneos

## Ventajas

- ❑ Reaprovechar el código ya escrito.
- ❑ Realizar operaciones muy concretas en cada método distribuyendo mejor la responsabilidad de tareas.

```
public static void operaciones(int tipo, float[] operandos) {  
    switch(tipo) {  
        case 1:  
            System.out.println("Raiz " + Math.sqrt(operandos[0]));  
            break;  
        case 2:  
            double a = operandos[0];  
            double b = operandos[1];  
            double c = operandos[2];  
  
            double resultadoPositivo = calcularEcuacionSegundoGrado(a, b, c)[0];  
            double resultadoNegativo = calcularEcuacionSegundoGrado(a, b, c)[1];  
            System.out.println("Ecuacion segundo grado : "  
                + "Resultado + " + resultadoPositivo + "Resultado - " + resultadoNegativo);  
            break;  
    }  
}  
  
public static double[] calcularEcuacionSegundoGrado(double a, double b, double c) {  
    double[] resultado = new double[2];  
  
    resultado[0] = -b + Math.sqrt( Math.pow(b, 2) - (4 * a * c) ) / (2 * a);  
    resultado[1] = -b - Math.sqrt( Math.pow(b, 2) - (4 * a * c) ) / (2 * a);  
  
    return resultado;  
}
```

Refactorizado

# Empleo de constantes

## Ventajas

- ❑ Mejorar la eficiencia de ejecución del programa.
- ❑ Simplificar y reutilizar el código.

```
public class Celda {  
  
    public static final String COLOR_NEGRO = "NEGRO";  
    public static final String COLOR_BLANCO = "BLANCO";  
  
    //..  
  
}
```

# Reemplazar objetos con subclases

## Ventajas

- ❑ Mejorar el rendimiento del programa.
- ❑ Simplificar el código. Cada una de las clases implementa su propia lógica
- ❑ Hacer el programa más mantenible.

```
public class Animal {  
  
    private final String tigre = "Tigre";  
    private final String tortuga = "Tortuga";  
  
    public String getTipo(String animal) {  
        if (animal.equalsIgnoreCase(tigre)) {  
            return "Soy de tipo tigre";  
        } else if (animal.equalsIgnoreCase(tortuga)) {  
            return "Soy de tipo tortuga";  
        }  
  
        return "No soy de ningun tipo";  
    }  
}
```

Antes

```
public Animal getTipo(String animal) {  
    if (animal.equalsIgnoreCase("Tigre")) {  
        return new Tigre();  
    } else if (animal.equalsIgnoreCase("Tortuga")) {  
        return new Tortuga();  
    }  
  
    return null;  
}  
  
public class Tortuga extends Animal {  
    // Código tortuga  
}  
  
public class Tigre extends Animal {  
    // Código tigre  
}
```

Refactorizado



**Reutilización** de los mismos patrones tantas veces como sea necesario.

**Efectividad** probada.

**Vocabulario común** entre desarrolladores de software.

**Estandarización** del código.

**Facilitación de la comprensión** de estructuras complejas.

**Versatilidad.**  
Permite el empleo de otras estrategias simultáneamente.

# Patrones de diseño

Son soluciones genéricas y probadas a ciertos problemas de diseño relacionados típicamente con la programación orientada a objetos. Podemos entenderlas como plantillas sobre las que construir nuestro *software*.

## Patrones creacionales

Permiten la creación de nuevos objetos de forma sencilla y la reutilización del código.



## Patrones estructurales

Se centran en las relaciones entre objetos partiendo, generalmente, del concepto de herencia para la composición de interfaces o nuevas funcionalidades.



## Patrones de comportamiento

Se ocupan de la comunicación entre objetos de clase y cómo modificarla.







### ¿Qué son los Sistemas de Control de Versiones (SCV)?

Son herramientas (programas) que nos ayudan a recopilar todos los cambios de código que se realizan sobre un programa o aplicación.



### ¿Por qué son importantes?

Lo más habitual es que el *software* lo desarrolle un equipo de personas que pueden realizar modificaciones en distintas partes o funcionalidades de la aplicación. Tener el control de las modificaciones nos permite identificar de forma más ágil los posibles errores y restaurar el código a versiones estables anteriores.



### ¿Qué beneficios obtenemos con ellos?

Facilita la colaboración entre desarrolladores aumentando la productividad y eficiencia, reduce el riesgo de errores y permite restaurar el sistema a una versión estable en el caso de un error grave.

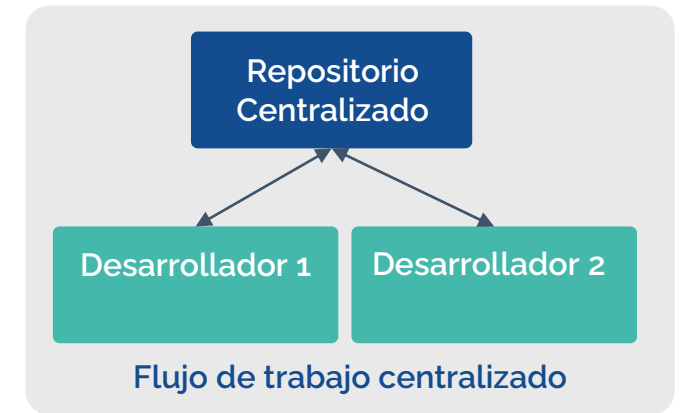
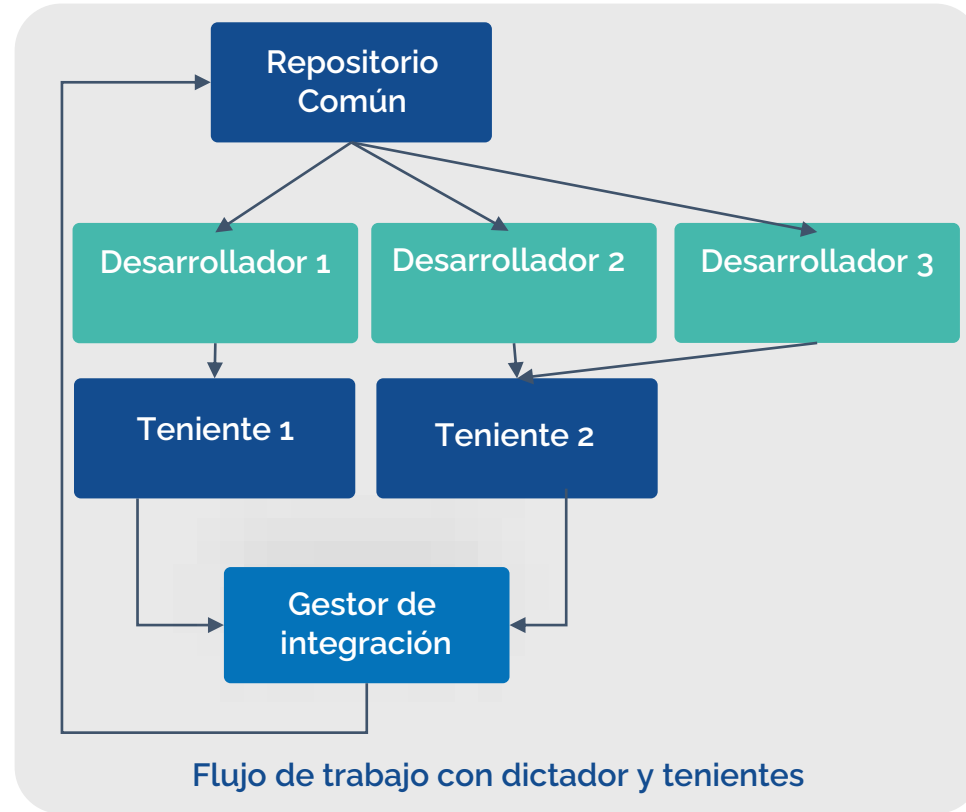
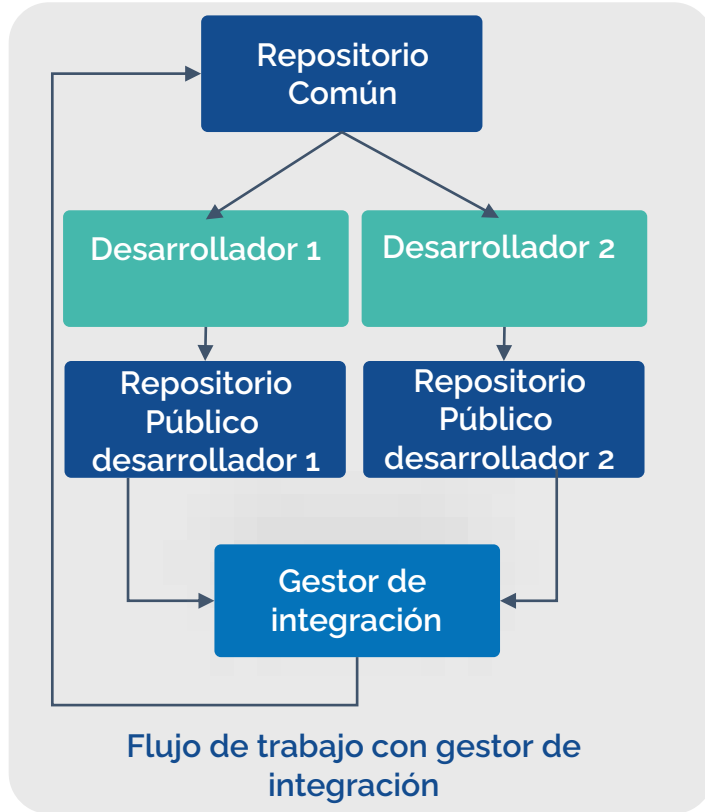


### ¿Qué tipos de SCV existen?

- ❑ **Centralizados:** se almacena todo el código en un único repositorio común.
- ❑ **Distribuidos:** Se crea un repositorio para cada usuario, por lo que otorga una gran seguridad por la existencia de diversas copias que servirán de respaldo.

# Control de versiones

# Tipos de colaboración en un SCV





# Documentación

Pautas que garantizan una documentación de calidad

- ☐ Realización y uso de esquemas.
- ☐ Clasificar la información.
- ☐ Síntesis de la información y la documentación.
- ☐ Empleo de estándares.
- ☐ Anticipación.
- ☐ Claridad.
- ☐ Empleo de las herramientas adecuadas.
- ☐ Empleo de los documentos apropiados.
- ☐ Tener muy presente el nivel del usuario.



# Tipos de documentación



## Documentos técnicos

Contienen información técnica sobre el producto: componentes, características, bases de datos, modelo de datos, etc.



## Documentos funcionales

Contienen información sobre la funcionalidad del producto: funcionamiento del programa/aplicación, modo de empleo, lógica de negocio, etc.



# Documentación. Fases del desarrollo

## Fase inicial

Planificación del proyecto centrada principalmente en rentabilidad y coste.

## Diseño

Creación del esquema del proyecto y de las aplicaciones que lo acompañan.

## Pruebas

Generación y ejecución de los casos de pruebas necesarios para validar el correcto funcionamiento del proyecto a nivel técnico y funcional.

## Mantenimiento

Modificación del código para corregir posibles errores detectados en la fase de explotación o realización de modificaciones posteriores.

## Análisis

Primera aproximación al proyecto centrado en el estudio de los objetivos a conseguir y el planteamiento de posibles problemas y el planteamiento de soluciones.

## Codificación/ Implementación

Basándose en el diseño, se realiza la codificación en el lenguaje o lenguajes de programación seleccionados.

## Explotación

Implementación del proyecto de software en el entorno real y puesta en funcionamiento en condiciones reales.

Manual técnico

Manual de usuario

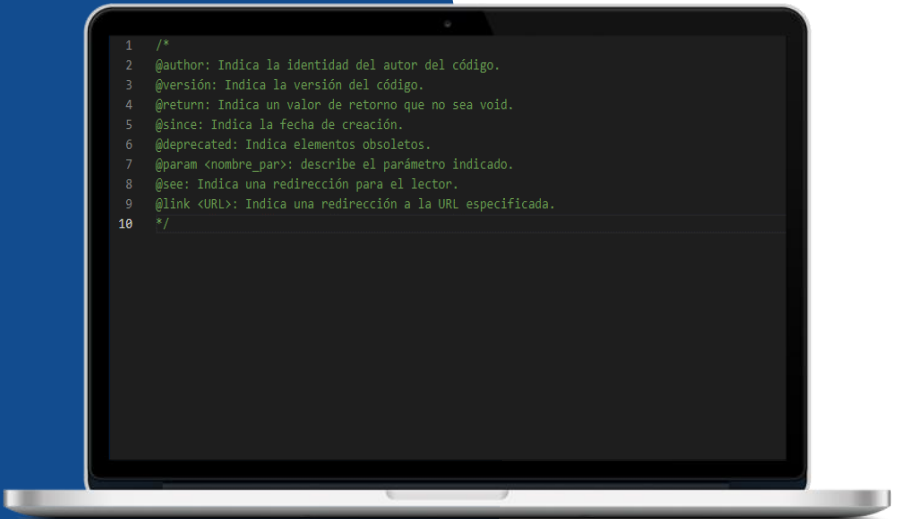
Manual de instalación

Documentación mínima a entregar

# Generación automática de documentación

El código suele estar en constante cambio por lo que es posible que la documentación quede obsoleta muy rápidamente. Por ello, es muy importante añadir comentarios en nuestro código.

Existen herramientas que generan automáticamente la documentación a partir de estos comentarios. Por ejemplo, en el caso de los proyectos en *Java*, contamos con *Javadoc* que genera documentación en HTML a partir de los comentarios incluidos en el código.



```
1  /*
2  @author: Indica la identidad del autor del código.
3  @versión: Indica la versión del código.
4  @return: Indica un valor de retorno que no sea void.
5  @since: Indica la fecha de creación.
6  @deprecated: Indica elementos obsoletos.
7  @param <nombre_par>: describe el parámetro indicado.
8  @see: Indica una redirección para el lector.
9  @link <URL>: Indica una redirección a la URL especificada.
10 */
```

The background is a solid blue color. Overlaid on this are several faint, light-blue geometric patterns. These include a grid of small squares that form larger, irregular shapes, and numerous small, light-blue arrows pointing in various directions, some of which are slightly blurred, giving a sense of motion or data flow.

# UNIVERSAE

— CHANGE YOUR WAY —