



Optimización y documentación



Introducción



En la presente unidad se estudiarán conceptos relacionados con la **refactorización**, el **control de versiones** y el proceso de **documentación**.

Refactorización: proceso el cual aporta **técnicas** para que el **código** de una aplicación quede más **claro** y libre de líneas indeseadas o que pueden ser sustituidas por otras más eficientes.

Control de versiones: técnica que permite a los miembros de un equipo de desarrollo **controlar** cada una de las **versiones** que se crean y se suben a un repositorio común.

Documentación: es una **tarea transversal** a todas las fases del ciclo de vida de una aplicación y es de suma importancia tener documentados **todos los procesos** desarrollados.



Ejemplo práctico

El equipo de desarrollo ha establecido un nuevo patrón de trabajo. Todos los miembros del equipo deben tener disponible toda la documentación generada, así como el código resultante de los requisitos implementados. Para ello, se decide llevar a cabo una buena práctica: instalar un software para el control de versiones.

Por otro lado, se decide llevar a cabo una refactorización del código del aplicativo. ¿Cuál sería la mejor decisión a tomar? El equipo de desarrollo se reúne y se les asigna a cada uno de ellos un requisito funcional ya programado, para que pueda ser revisado y refactorizado, repartiendo la carga de trabajo. Al estar el código bajo un software para el control de versiones, todos tendrán notificaciones de las novedades existentes en el código.



Refactorización



Refactorizar: es realizar una transformación al software preservando su comportamiento, modificando sólo su estructura interna para mejorarlo.

Puede verse como un tipo de mantenimiento preventivo, cuyo objetivo es **disminuir la complejidad** del software en anticipación a los incrementos de complejidad que los cambios pudieran traer.

Ayuda a **evitar** los **problemas** típicos que aparecen con el tiempo, como, por ejemplo, un mayor número de líneas para hacer las mismas cosas o código duplicado.

Otro aspecto importante es que ayuda a **aumentar** la **simplicidad** en el diseño.



Patrones de refactorización más usuales



Cualquier programador, al desarrollar una aplicación, siempre trata de conseguir que el código tenga **la mayor calidad posible**.

Es importante que el software **no falle**, pero es importante conocer que existen **otras características** que determinan si software es de calidad.

Han surgido estándares de codificación que permiten desarrollos más legibles y patrones para dar soluciones. La **simplicidad** y la **claridad** son características importantes de la calidad. Conseguir esa sencillez y claridad es complicado.

Hay que tener en cuenta también que los patrones, **en ocasiones, presentan** algunos **problemas**. En muchos casos enseñan a afrontar problemas de diseño generales, pero no son válidos para afrontar problemas más específicos.



List of Refactorings

Change Function Declaration (124)
Change Reference to Value (252)
Change Value to Reference (256)
Collapse Hierarchy (380)
Combine Functions into Class (144)
Combine Functions into Transform (149)
Consolidate Conditional Expression (263)
Decompose Conditional (260)
Encapsulate Collection (170)
Encapsulate Record (162)
Encapsulate Variable (132)
Extract Class (182)
Extract Function (106)
Extract Superclass (375)
Extract Variable (119)
Hide Delegate (189)
Inline Class (186)
Inline Function (115)
Inline Variable (123)
Introduce Assertion (302)
Introduce Parameter Object (140)
Introduce Special Case (289)
Move Field (207)
Move Function (198)
Move Statements into Function (213)
Move Statements to Callers (217)
Parameterize Function (310)
Preserve Whole Object (319)
Pull Up Constructor Body (355)
Pull Up Field (353)
Pull Up Method (350)
Push Down Field (361)
Push Down Method (359)
Remove Dead Code (237)
Remove Flag Argument (314)
Remove Middle Man (192)
Remove Setting Method (331)
Remove Subclass (369)
Rename Field (244)
Rename Variable (137)
Replace Command with Function (344)
Replace Conditional with Polymorphism (272)
Replace Constructor with Factory Function (334)
Replace Derived Variable with Query (248)
Replace Function with Command (337)
Replace Inline Code with Function Call (222)
Replace Loop with Pipeline (231)
Replace Nested Conditional with Guard Clauses (266)
Replace Parameter with Query (324)
Replace Primitive with Object (174)
Replace Query with Parameter (327)
Replace Subclass with Delegate (381)
Replace Superclass with Delegate (399)
Replace Temp with Query (178)
Replace Type Code with Subclasses (362)
Separate Query from Modifier (306)
Slide Statements (223)
Split Loop (227)
Split Phase (154)
Split Variable (240)
Substitute Algorithm (195)



Extraer método

Es una operación de refactorización que proporciona una manera sencilla para **crear** un **nuevo método** a partir de un fragmento de código existente.

El nuevo método que se ha extraído **contiene el código seleccionado** y el código seleccionado del miembro existente se reemplaza por una **llamada al nuevo método**.

Convertir un fragmento de código en su propio método permite **reorganizar el código** de forma rápida y precisa para que sea posible volver a utilizarlo (reutilización) y lograr una mejor legibilidad.



```
public .... metodoPrincipal(){  
    codigo que realiza una funcionalidad repetible a utilizar  
    .  
    .  
    .  
    más código  
    .  
    .  
    .  
}
```

Creo un método secundario
donde indicar el código

```
public .... metodoPrincipal(){  
    metodoSecundario();  
    más código  
    .  
    .  
    .  
}  
  
public .... metodoSecundario(){  
    codigo que realiza una funcionalidad repetible a utilizar  
    .  
    .  
    .  
}
```

Llamo al método secundario
en el principal



Actividad 1

Método que muestra la información de un cliente

Realizar refactorización mediante el patrón de **extracción de método**

```
public void mostrarInformaciónCliente(Cliente cliente){
    imprimirBanner();

    System.out.println("Nombre: " cliente.getNombre());
    System.out.println("Apellidos: "
cliente.getApellidos());
    System.out.println("DNI: " cliente.getDNI());
    System.out.println("Calle: " cliente.getCalle());
}

public void imprimirBanner(){
    System.out.println("//////////");
}
```



Actividad 1

Método que muestra la información de un cliente

Realizar refactorización mediante el patrón de **extracción de método**

```
No_devuelve_nada mostrarInformaciónCliente(cliente){
    imprimirBanner();

    muestra -> "Nombre: " nombre del cliente;
    muestra -> "Apellidos: " apellidos del cliente;
    muestra -> "DNI: " dni del cliente;
    muestra -> "Calle: " calle del cliente;
}

No_devuelve_nada imprimirBanner(){
    muestra -> "/////////"
}
```



```
public void mostrarInformaciónCliente(Cliente cliente){  
    imprimirBanner();  
    printInfoPersona(cliente);  
}
```

```
public void imprimirBanner(){  
    System.out.println("//////////");  
}
```

```
public void printInfoPersona(Cliente c){  
    System.out.println("Nombre: " c.getNombre());  
    System.out.println("Apellidos: " c.getApellidos());  
    System.out.println("DNI: " c.getDNI());  
    System.out.println("Calle: " c.getCalle());  
}
```



Actividad 2

Método que calcula el sumatorio de los números de una lista y muestra el sumatorio por consola.

Realizar refactorización mediante el patrón de **extracción de método**

```
public void imprimirSumatorio(List<Integer>
listaNumeros){
    Integer sumatorio=0;
    for (Integer numero:listaNumeros){
        sumatorio = sumatorio +numero;
    }
    imprimirValor(sumatorio);
}

public void imprimirValor(Integer valor) {
    System.out.println(valor);
}
```



Actividad 2

Método que calcula el sumatorio de los números de una lista y muestra el sumatorio por consola.

Realizar refactorización mediante el patrón de **extracción de método**

```
No_devuelve_nada imprimirSumatorio(listaNumeros){
    sumatorio comienza en 0;
    recorrer -> lista{
        sumatorio acumulado <- valor obtenido de
la lista en la posición por la que va + sumatorio
acumulado;
    }
    imprimirValor(sumatorio acumulado);
}

No_devuelve_nada imprimirValor(numero){
    muestra -> numero
}
```




```
public void imprimirSumatorio(List<Integer> listaNumeros){  
    Integer resultado=calcularSumatorio(listaNumeros)  
    imprimirValor(resultado);  
}
```

```
public Integer calcularSumatorio(List<Integer>  
listaNumeros) {  
    Integer sumatorio = 0;  
    for(Integer numero : listaNumeros) {  
        sumatorio = resultado + numero;  
    }  
    return sumatorio;  
}  
public void imprimirValor(Integer valor) {  
    System.out.println(valor);  
}
```



Actividad 3

Método que encuentra todos los números primos hasta el 100. Además muestra 10 primos por línea.

Realizar refactorización mediante el patrón de **extracción de método**

```
public void obtenerPrimos() {  
  
    int numero = 100;  
    int contador = 0;  
  
    for (int i = 2; i <= numero; i++) {  
        boolean esPrimo = true;  
        for (int j = 2; j <= Math.sqrt(i); j++) {  
            if (i % j == 0) {  
                esPrimo = false;  
            }  
        }  
  
        if (esPrimo) {  
            System.out.print(i + " ");  
            contador++;  
            if (contador % 10 == 0) {  
                System.out.println();  
            }  
        }  
    }  
}
```



Actividad 3

Método que encuentra todos los números primos hasta el 100. Además muestra 10 primos por línea.

Realizar refactorización mediante el patrón de **extracción de método**

```
No_devuelve_nada obtenerPrimos() {  
  
    Numero_limite_a_buscar = 100;  
    contador = 0;  
  
    Recorrer (i) -> (del 2 al Numero_limite_a_buscar; de 1 en 1) {  
        Recorrer (j) -> ( del 2 a la raiz cuadrada del indice  
        anterior (i); de 1 en 1) {  
            si (i % j == 0) {  
                No es primo  
            }  
        }  
  
        si (es Primo) {  
            mostrar -> i  
            si (he mostrado 10 números en la fila){  
                salto a la siguiente fila  
            }  
        }  
    }  
}
```



```
public void obtenerPrimos() {

    int numero = 100;
    int contador = 0;
    for (int i = 2; i <= numero; i++) {

        if (esPrimo(i)) {
            System.out.print(i + " ");
            contador++;
            if (contador % 10 == 0) {
                System.out.println();
            }
        }
    }

}

public boolean esPrimo(int i) {
    boolean esPrimo = true;
    for (int j = 2; j <= Math.sqrt(i); j++) {
        if (i % j == 0) {
            esPrimo = false;
        }
    }
    return esPrimo;
}
```



Métodos en línea: se presenta cuando se tiene uno o más métodos cuyos códigos son lo suficientemente autoexplicativos como para poder prescindir de ellos.



Actividad 4

Método que a través de una lista de clientes crea una lista con sus DNI y los muestra.

Realizar refactorización mediante el patrón de **métodos en línea**.

```
private void imprimirSalida(String x) {  
    System.out.println(x);  
}  
  
public List<String> listaDNI(List<Cliente> clientes) {  
    List<String> result = new List<String>();  
  
    for (Cliente cliente : clientes) {  
        result.add(cliente.getDNI());  
        imprimirSalida(cliente.getDNI());  
    }  
  
    return result;  
}
```



```
public List<String> listaDNI(List<Cliente> clientes){  
    List<String> result = new List<String>();  
  
    for (Cliente cliente : clientes) {  
        result.add(cliente.getDNI());  
        System.out.println(cliente.getDNI());  
    }  
  
    return result;  
}
```



Actividad 5

Método que otorga un 0 si ha realizado más de 5 entregas tardías y un 1 en caso contrario.

Realizar refactorización mediante el patrón de **métodos en línea**.

```
int obtenerPuntuacion() {  
    return (mas5PedidosTarde()) ? 0 : 1;  
}  
  
boolean mas5PedidosTarde() {  
    return _numeroPedidosTarde > 5;  
}
```




```
int obtenerPuntuacion() {  
    return _numeroPedidosTarde > 5 ? 0 : 1;  
}
```



Extraer variable: estamos frente a una expresión complicada. La mejor manera es poner el resultado de la expresión (o de las distintas partes de ésta) en una variable temporal con un nombre que explique su propósito.



Actividad 6

Devuelve el cálculo del precio del producto, a través del precio del producto, descuento y envío .

Realizar refactorización mediante el patrón de **extracción de variable**.

```
return order.cantidad * order.precioProducto -  
Math.max(0, order.cantidad - 500) *  
order.precioProducto * 0.05 + Math.min(order.cantidad  
* order.precioProducto * 0.1, 100);
```



```
double precioBase = order.cantidad * order.precioProducto;  
  
double cantidadDescuento = Math.max(0, order.cantidad - 500) *  
order.precioProducto * 0.05;  
  
double envio = Math.min(precioBase * 0.1, 100);  
  
return precioBase - cantidadDescuento + envio;
```



Actividad 7

Realiza una acción si cumple con una serie de requisitos.

Realizar refactorización mediante el patrón de **extracción de variable**.

```
public void renderBanner(){  
    if(plataforma.toUpperCase().indexOf("MAC") > -1 &&  
    navegador.toUpperCase().indexOf("IE") > -1 && fueInicializado())    {  
  
        // hacer algo  
    }  
}
```



```
public void renderBanner(){  
    final boolean esMAC = plataforma.toUpperCase().indexOf("MAC") > -1 ;  
    final boolean esIE = navegador.toUpperCase().indexOf("IE") > -1;  
    if(esMAC && esIE && fueInicializado()){  
        // hacer algo  
    }  
}
```



Inline Temp: Si una variable temporal solo se usa para almacenar el resultado de una llamada a un método, podemos sustituirla por la propia llamada y eliminar la variable.



Actividad 8

Devuelve verdadero si el precio base es mayor a 1000.

Realizar refactorización mediante el patrón **Inline Temp**.

```
public boolean precioBaseMayor1000() {  
    double precioBase = pedido.precioBase();  
    return (precioBase > 1000);  
}
```




```
public boolean precioBaseMayor1000() {  
    return (pedido.precioBase() > 1000);  
}
```



Renombrado: proporciona una manera fácil de cambiar el nombre de los identificadores, como campos, variables locales, métodos, espacios de nombres, propiedades y tipos, por un nombre más significativo.



Actividad 9

Clase que contiene un método para sumar y/o restar dos números enteros pasados por teclado.

Realizar refactorización mediante el patrón **Renombrado**.

```
import java.util.Scanner;

public class Matematicas {

    private int n;
    private int m;

    public void sumar() {
        Scanner scanner = new Scanner(System.in);

        System.out.println("digite el primer numero a sumar: ");
        n = scanner.nextInt();

        System.out.println("digite el segundo numero a sumar: ");
        m = scanner.nextInt();

        System.out.println("el resultado de la suma es: " + n + m);

    }

    public void restar() {
        Scanner scanner = new Scanner(System.in);

        System.out.println("digite el primer numero a restar: ");
        n = scanner.nextInt();

        System.out.println("digite el segundo numero a restar: ");
        m = scanner.nextInt();

        System.out.println("el resultado de la restar es: " + (n - m));

    }

}
```



U4: Optimización y documentación

```
import java.util.Scanner;

public class Matematicas {

    private int numero1;
    private int numero2;

    public void sumar() {
        Scanner scanner = new Scanner(System.in);

        System.out.println("digite el primer numero a sumar: ");
        numero1 = scanner.nextInt();

        System.out.println("digite el segundo numero a sumar: ");
        numero2 = scanner.nextInt();

        System.out.println("el resultado de la suma es: " + numero1 + numero2);

    }

    public void restar() {
        Scanner scanner = new Scanner(System.in);

        System.out.println("digite el primer numero a restar: ");
        numero1 = scanner.nextInt();

        System.out.println("digite el segundo numero a restar: ");
        numero2 = scanner.nextInt();

        System.out.println("el resultado de la restar es: " + (numero1 - numero2));

    }

}
```



Campos encapsulados: se aconseja crear métodos getter y setter, (de asignación y de consulta) para cada campo que se defina en una clase. Cuando sea necesario acceder o modificar el valor de un campo, basta con invocar al método getter o setter según convenga.



Actividad 10

Clase Persona que contiene los atributos nombre, apellidos y dni.

Realizar refactorización mediante el patrón **Campos encapsulados**.

```
public class Persona{  
    private String nombre;  
    private String apellidos;  
    private String dni;  
}
```



U4: Optimización y documentación

```
public class Persona{  
    private String nombre;  
    private String apellidos;  
    private String dni;  
  
    public String getNombre(){  
        return nombre;  
    }  
  
    public void setNombre(String nombreNuevo){  
        nombre= nombreNuevo;  
    }  
    public String getApellidos(){  
        return apellidos;  
    }  
    public void setApellidos(String apellidosNuevos){  
        apellidos= apellidosNuevos;  
    }  
  
    public String getDNI(){  
        return dni;  
    }  
    public void setDNI(String dniNuevo){  
        dni= dniNuevo;  
    }  
}
```



Mover la clase: si es necesario, se puede mover una clase de un paquete a otro, o de un proyecto a otro. La idea es no duplicar código que ya se haya generado. Esto impone la actualización en todo el código fuente de las referencias a la clase en su nueva localización.

Borrado seguro: se debe comprobar, que cuando un elemento del código ya no es necesario, se han borrado todas la referencias a él que había en cualquier parte del proyecto.

Cambiar los parámetros del proyecto: nos permite añadir nuevos parámetros a un método y cambiar los modificadores de acceso.



Analizadores de código



Cada IDE incluye **herramientas** de refactorización y analizadores de código. Además, existen analizadores de código que se pueden añadir como **complementos** a los entornos de desarrollo.

El **análisis estático de código**, es un proceso que tiene como objetivo, **evaluar** el software, sin llegar a ejecutarlo. Esta técnica se va a aplicar directamente sobre el código fuente, para poder obtener información que nos permita **mejorar la base de código**.

Los **analizadores de código**, son las **herramientas** encargadas de realizar esta labor. El analizador estático de código **recibirá** el código fuente de nuestro programa, lo **procesará** intentando averiguar la funcionalidad del mismo, y nos **dará** sugerencias, o nos mostrará posibles **mejoras**.



Las principales funciones de los analizadores es **encontrar** partes del código que puedan reducir el **rendimiento**, provocar **errores** en el software, tener una excesiva **complejidad**, complicar el flujo de **datos**, crear problemas de **seguridad**.

El análisis puede ser **automático** o **manual**. El automático, los va a realizar un programa, que puede formar parte de la funcionalidad de un entorno de desarrollo, y el manual cuando es una persona.




Ejemplos de herramientas:

- **PMD:** Esta herramienta basa su funcionamiento en detectar patrones, que son posibles errores en tiempo de ejecución, código que no se puede ejecutar nunca porque no se puede llegar a él, código que puede ser optimizado, expresiones lógicas que pueden ser simplificadas, malos usos del lenguaje, etc.
- **CPD.** Forma parte del PMD. Su función es encontrar código duplicado.




Instalar plugin en Eclipse desde market:

pmd-eclipse-plugin 4.30.0



PMD is a source code analyzer. It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth. It supports... [more info](#)

by [PMD](#), [BSD](#)
[PMD linter](#) [Source Code Analyzer](#) [code quality](#) [java](#)

★ 141  Installs: **68,5K** (1.114 last month) **Install**



Configuración:





Si vamos a PMD > Rule Configuration y activamos el “utilizar la gestión de reglas global” entonces podremos al seleccionar una regla ver una descripción de la misma.

Además, indica si es una regla de documentación, de estilo, de errores, de buenas prácticas, etc.



Realizar una ejecución de PMD en un proyecto de la asignatura de Programación que tengáis sin errores y ver que ocurre.



Refactorización con Eclipse



Muchos editores y entornos de desarrollo presentan **soporte automatizado** para la refactorización.

Eclipse incorpora desde versiones muy tempranas opciones para refactorizar nuestro código.

Se puede acceder a las operaciones de Refactorización a través de la **opción “Refactor”** en el menú principal.



Realizar una refactorización de **extracción de método** mediante Eclipse.

Usar “Extract Method”

```
import java.util.Random;

public class Prueba {

    public static void main(String[] args) {
        Random r = new Random();
        int valorDado = r.nextInt(10) + 1; // Valor
        aleatorio entre 1 y 10
        System.out.print(valorDado);
    }
}
```



U4: Optimización y documentación

```
import java.util.Random;

public class Prueba {

    public static void main(String[] args) {
        aleatorio();
    }

    /**
     * Método que calcula valor aleatorio entre 1 y 10
     */
    private static void aleatorio() {
        Random r = new Random();
        int valorDado = r.nextInt(10) + 1;
        System.out.print(valorDado);
    }
}
```



Realizar una refactorización de **método en línea** mediante Eclipse.

Usar “Inline”.

```
int obtenerPuntuacion() {  
    return (mas5PedidosTarde()) ? 0 : 1;  
}  
  
boolean mas5PedidosTarde() {  
    return _numeroPedidosTarde > 5;  
}
```



```
int obtenerPuntuacion() {  
    return _numeroPedidosTarde > 5 ? 0 : 1;  
}
```



Realizar una refactorización de **Inline Temp** mediante Eclipse.

Usar “Inline”

```
public boolean precioBaseMayor1000() {  
    double precioBase = pedido.precioBase();  
    return (precioBase > 1000);  
}
```



```
public boolean precioBaseMayor1000() {  
    return pedido.precioBase() > 1000;  
}
```




Realizar una refactorización de **extraer variable** mediante Eclipse.

Usar “Extract Local Variable”

```
return order.cantidad * order.precioProducto - Math.max(0, order.cantidad -  
500) * order.precioProducto * 0.05 + Math.min(order.cantidad *  
order.precioProducto * 0.1, 100);
```



```
double precioBase = order.cantidad * order.precioProducto;  
double cantidadDescuento = Math.max(0, order.cantidad - 500) *  
order.precioProducto * 0.05;  
double envio = Math.min(precioBase * 0.1, 100);  
  
return precioBase - cantidadDescuento + envio;
```



Realizar una refactorización de **renombrado** mediante Eclipse.

Usar “Rename”

```
import java.util.Scanner;

public class Matematicas {

    private int n;
    private int m;

    public void sumar() {
        Scanner scanner = new Scanner(System.in);

        System.out.println("digite el primer numero a sumar: ");
        n = scanner.nextInt();

        System.out.println("digite el segundo numero a sumar: ");
        m = scanner.nextInt();

        System.out.println("el resultado de la suma es: " + n + m);

    }

    public void restar() {
        Scanner scanner = new Scanner(System.in);

        System.out.println("digite el primer numero a restar: ");
        n = scanner.nextInt();

        System.out.println("digite el segundo numero a restar: ");
        m = scanner.nextInt();

        System.out.println("el resultado de la restar es: " + (n - m));

    }

}
```



U4: Optimización y documentación

```
import java.util.Scanner;

public class Matematicas {

    private int numero1;
    private int numero2;

    public void sumar() {
        Scanner scanner = new Scanner(System.in);

        System.out.println("digite el primer numero a
sumar: ");
        numero1 = scanner.nextInt();

        System.out.println("digite el segundo numero a
sumar: ");
        numero2 = scanner.nextInt();

        System.out.println("el resultado de la suma es: " +
numero1 + numero2);
    }

    public void restar() {
        Scanner scanner = new Scanner(System.in);

        System.out.println("digite el primer numero a
restar: ");
        numero1 = scanner.nextInt();

        System.out.println("digite el segundo numero a
restar: ");
        numero2 = scanner.nextInt();

        System.out.println("el resultado de la restar es: "
+ (numero1 - numero2));
    }
}
```



Actividad 6

Realizar refactorización mediante el patrón **Campos encapsulados** mediante Eclipse.

Usar Source > Generators Getters and Setters

```
public class Persona{  
    private String nombre;  
    private String apellidos;  
    private String dni;  
}
```



U4: Optimización y documentación

```
public class Persona{
    private String nombre;
    private String apellidos;
    private String dni;

    public String getNombre(){
        return nombre;
    }

    public void setNombre(String nombreNuevo){
        nombre= nombreNuevo;
    }
    public String getApellidos(){
        return apellidos;
    }
    public void setApellidos(String apellidosNuevos){
        apellidos= apellidosNuevos;
    }

    public String getDNI(){
        return dni;
    }
    public void setDNI(String dniNuevo){
        dni= dniNuevo;
    }
}
```



Refactorización y pruebas



Para poder refactorizar de forma satisfactoria es indispensable contar con un buen lote de **casos de prueba** que validen el correcto funcionamiento del sistema.

Los casos de prueba permiten **verificarse** repetida e incrementalmente si los cambios introducidos han alterado el comportamiento observable del programa.

Estos casos de prueba deben cumplir con los siguientes requisitos:

- Deben ser **automáticos** de tal manera que se puedan ejecutar todos a la vez con simplemente hacer clic un botón.
- Deben ser **auto verificables** de forma que no se invierta tiempo en la verificación de los resultados de los mismos. Estos errores deben ser reportados por cada caso de prueba.
- Deben ejecutarse de manera **independiente** uno del otro, para que los resultados de uno no afecten a los resultados del resto.



Pasos a seguir:

1. Ejecutar las pruebas antes de haber efectuado cualquier cambio, de forma que sepamos como actúa el sistema antes de los cambios.
2. Analizar los cambios a realizar.
3. Aplicar los cambios.
4. Volver a aplicar las pruebas para comprobar que el funcionamiento sigue siendo el mismo.

IMPORTANTE: una optimización de código no es una refactorización ya que, si bien tienen en común la modificación del código fuente sin alterar el comportamiento observable del programa, la diferencia radica en la forma de impactar en el código fuente: la optimización suele agregarle complejidad.



Control de versiones



Repositorio: es un **almacén** de datos. Almacena información en forma de un árbol de archivos, una jerarquía típica de archivos y directorios.

```
pruebas
├── capitales.txt
├── ejemplosort.txt
├── info.csv
├── libreria -> libreria1.2
├── libreria1.1
├── libreria1.2
├── misegundaprueba.txt
├── numeros.txt
├── paises.txt
├── paisesycapitales.txt.gz
├── procesos
│   ├── about.html
│   ├── clase.txt
│   ├── history.html
│   ├── listurls.txt
│   ├── manpage.html
│   ├── miprimeraprueba.txt
│   └── misegundaprueba.txt
├── prueba1.txt
├── prueba2.txt
└── users.sh
```



Sistema de control de versiones: es una combinación de **tecnologías** y **prácticas** para seguir y controlar los cambios realizados en los ficheros de un proyecto, en particular el código fuente, documentación, etc.

La razón por la cual el control de versiones es universalmente utilizado es porque ayuda virtualmente en todos los aspectos al **dirigir un proyecto**:

- Comunicación entre los desarrolladores
- Manejo de los lanzamientos
- Administración de fallos
- Estabilidad entre el código y los esfuerzos de desarrollo experimental
- Atribución y autorización en los cambios de los desarrolladores.

El núcleo del sistema es la **gestión de cambios**: identificar cada cambio a los ficheros del proyecto, anotar cada cambio con metadata como la **fecha** y el **autor** de la modificación y disponer esta información para quien sea y como sea.



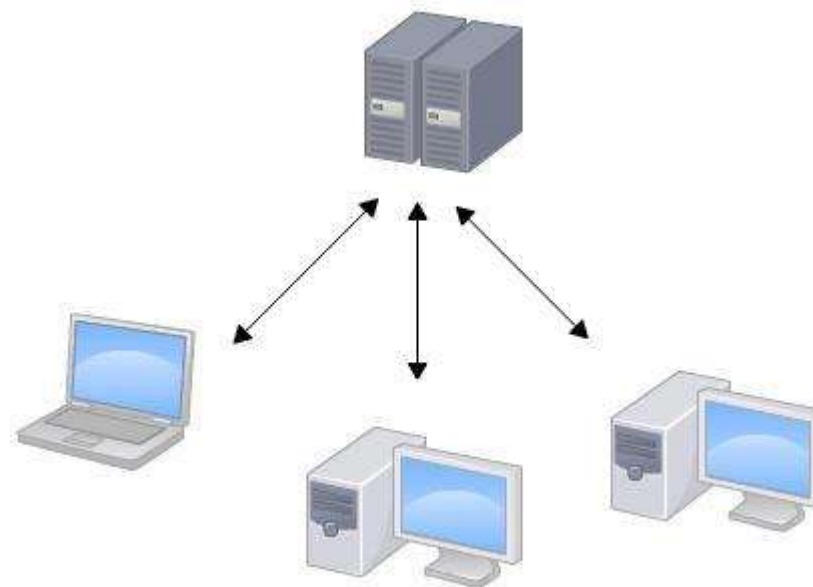
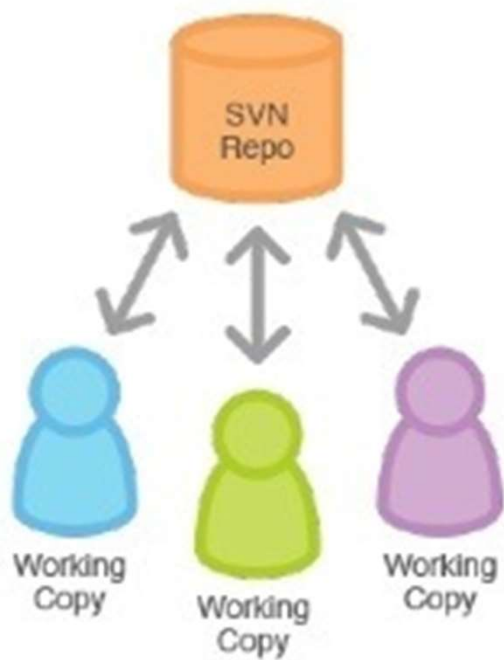
Los sistemas de control de versiones se pueden clasificar en 2 grandes grupos:

- Centralizados:
 - Todas las fuentes y sus versiones están almacenados en un **único repositorio de fuentes** de un servidor.
 - Todos los desarrolladores que quieran trabajar con esas fuentes, deben pedirle al sistema de control de versiones una **copia local para trabajar**.
 - En ella **realizan** todos sus **cambios**
 - Cuando están listos y funcionando, le piden al sistema de control de versiones que **guarde** las fuentes modificados como **una nueva versión**.

Algunos ejemplos son CVS y Subversión.



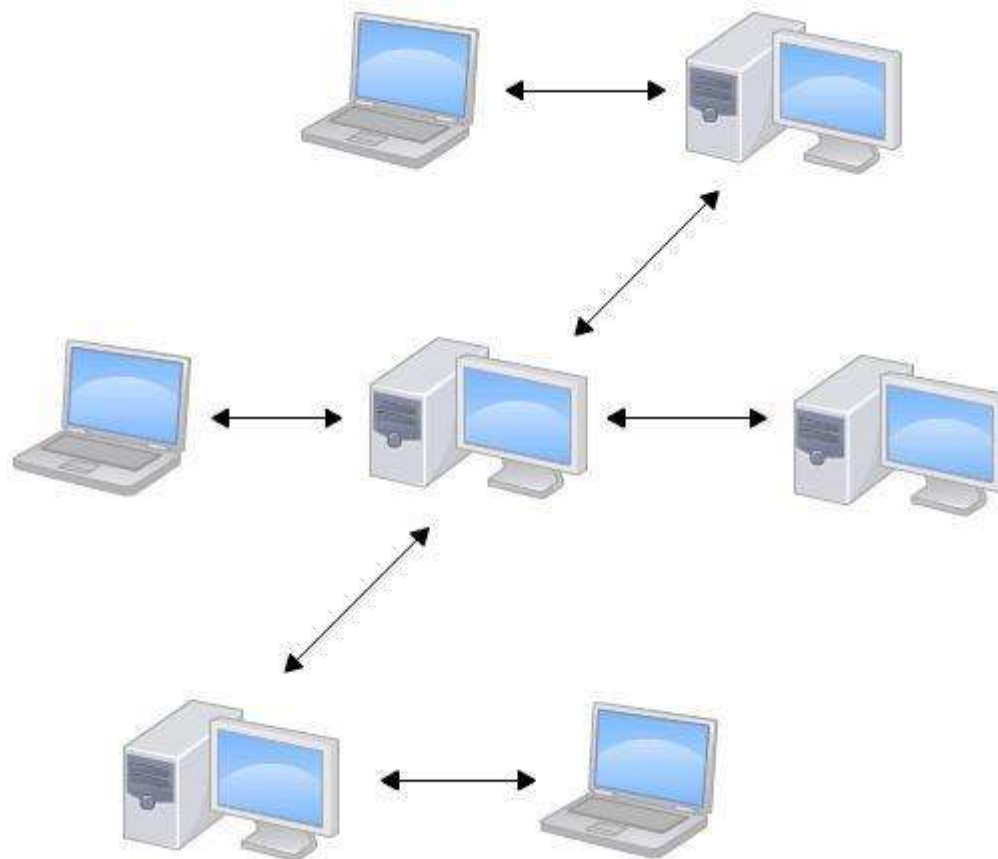
Central-Repo-to-Working-Copy
Collaboration





- Distribuidos:
 - **No** hay un repositorio central.
 - Todos los desarrolladores tienen **su propia copia del repositorio**, con todas las versiones y toda la historia.
 - Según van desarrollando y haciendo cambios, sus fuentes y versiones van siendo **distintas unas de otras**.
 - Sin embargo, permiten que en cualquier momento dos desarrolladores cualesquiera **puedan sincronizar sus repositorios**.
 - Generalmente también tiene una computadora que actúa como un "servidor central", pero la función de este servidor es solo **facilitar el "intercambio"** de las modificaciones de todos.
 - Si uno de los desarrolladores ha modificado determinados ficheros fuentes y el otro no, los modificados se convierten en la **versión más moderna**.

Ejemplos: Git y Mercurial.





GIT



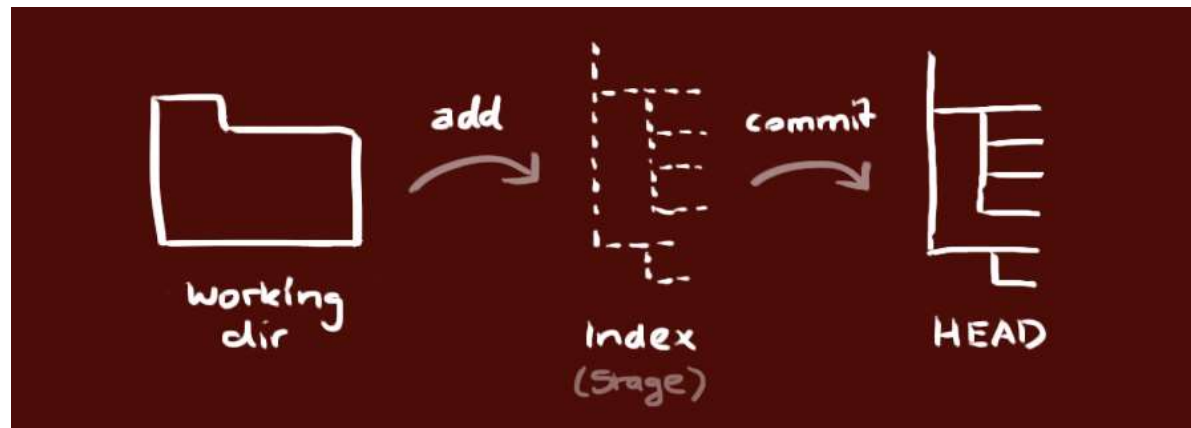
Git:

- Es un repositorio de código distribuido.
- Cada nodo tiene una copia completa del repositorio.
- Ejemplos de uso: github, bitbucket, gitlab, ...



3 repositorios:

- Directorio de trabajo: nuestra **capeta** en la que **trabajaremos**.
- Index: Sitio donde **añadiremos** nuestros **cambios**, es un espacio intermedio.
- Head: **Última versión** de nuestro repositorio.





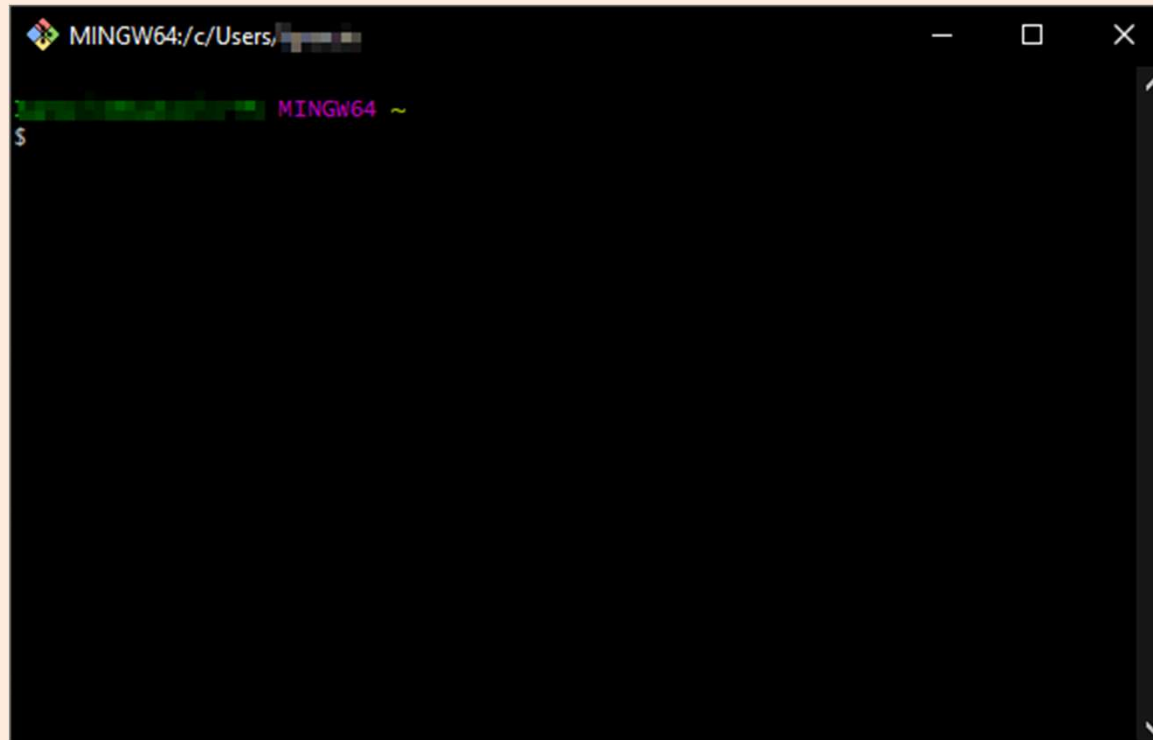
Instalar Git



<https://git-scm.com/downloads>



Abrir Git Bash





Crear un repositorio nuevo local: `git init`

Al ejecutar `git init`, se crea un subdirectorio de **.git** en el directorio de trabajo actual, que contiene todos los **metadatos** de Git necesarios para el nuevo repositorio.

Clonar un repositorio existente en nuestro repositorio local: `git clone urlrepositorio`

Este comando internamente llama primero a `git init` y a continuación copia los datos del repositorio existente.



git config: herramienta que te permite **obtener** y **establecer** variables de **configuración** que controlan el aspecto y funcionamiento de Git.

global vs local:

- Si indicamos la opción --global en el comando entonces la configuración será para **todos los usuarios y repositorios del ordenador**, por lo que solo tendremos que hacer la configuración una única vez.
- Si no indicamos nada o indicamos --local, la configuración será para el **repositorio de contexto** en el que se invoca git config.



Definir quienes somos en git:

- Indicar nombre usuario: `git config user.name "mi nombre"`
- Indicar email del usuario: `git config user.email "mi email"`
- Para comprobar la configuración (cerrar con la tecla "q"): `git config --list`

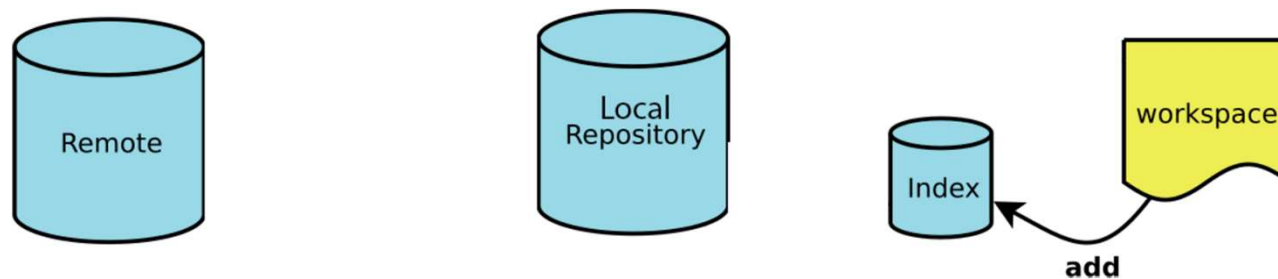
Esta configuración se encuentra en el archivo `.git/config`



Repositorio de trabajo local → Index:

- Añadir y actualizar todos los archivos del árbol de trabajo: `git add -A`
- Añadir y/o actualizar un único fichero: `git add nombrefichero`

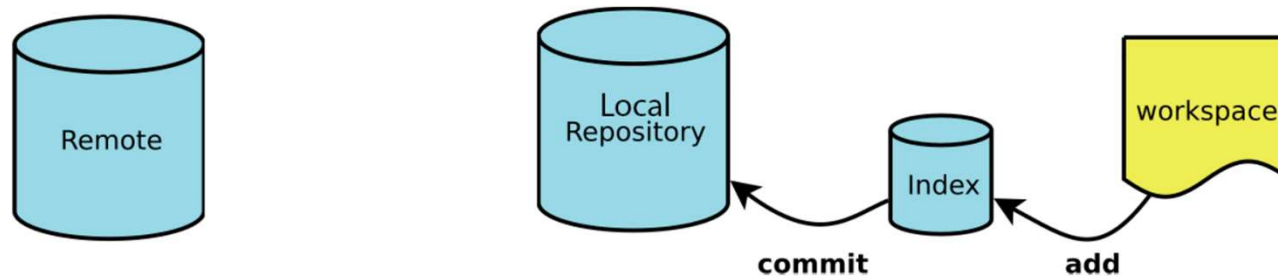
Los ficheros que indiquemos en el archivo .gitignore no se subirán a git





Index → HEAD:

Enviar cambios al HEAD: `git commit -m "mensaje a añadir"`





1. Crear un repositorio local
2. Configurar el usuario y email para dicho repositorio
3. Crear una carpeta en vuestro equipo que se llame “Entornos de desarrollo” (dentro del repositorio)
4. Añadir la carpeta al Index
5. Añadir los PDFs de las unidades a la carpeta en vuestro equipo.
6. Añadir todos los archivos de golpe al Index.
7. Finalmente, confirmar en el HEAD todos los archivos añadidos.