

# ENTORNOS DE DESARROLLO

## TEMA 4: OPTIMIZACIÓN Y DOCUMENTACIÓN

# Índice

- Refactorización
- Patrones de refactorización más habituales
- Documentación
- Control de versiones

## 4.1. Refactorización

3



# 4.1. Refactorización

- Consiste en realizar pequeñas transformaciones en el código de un programa, para mejorar la estructura **sin que cambie el comportamiento ni funcionalidad del mismo**.
- Su objetivo es mejorar la estructura interna del código.
- Es una tarea que pretender limpiar el código minimizando la posibilidad de introducir errores.
- *¿Podemos mejorar la estructura del código y que sea de mayor calidad, sin que cambie su comportamiento?  
¿Cómo hacerlo? ¿Qué patrones hay que seguir?*

# 4.1. Refactorización

- **Ventajas:**
  - Hace que el software sea más fácil de entender.
  - Hace que el mantenimiento del software sea más sencillo.
  - Ayuda a encontrar errores.
  - La acumulación de pequeños cambios pueden mejorar de forma ostensible el diseño.
- Ejemplos de refactorización es “Extraer Método” y “Encapsular Campos”.

# 4.1. Refactorización

- Hay que diferenciar la refactorización de la optimización:
  - En ambos procesos, se pretende mejorar la estructura interna de una aplicación o componente, sin modificar su comportamiento.
  - Sin embargo, cuando se optimiza, se persigue una mejora del rendimiento, por ejemplo mejorar la velocidad de ejecución, pero esto puede hacer un código más difícil de entender.

# 4.1. Refactorización

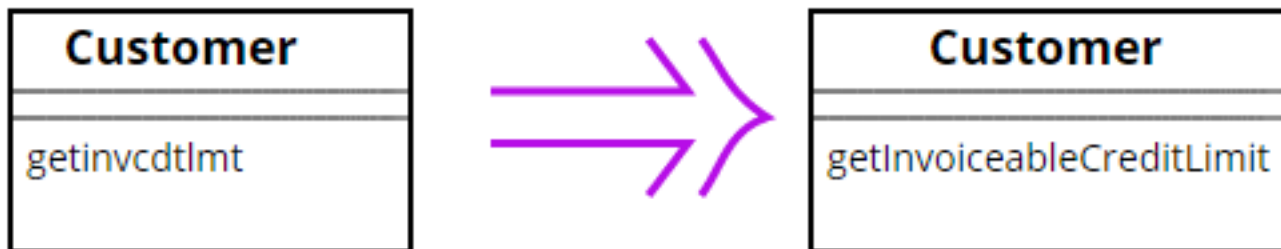
7

- Posibles limitaciones que supone la refactorización:
  - Puede suponer cambiar interfaces:
    - Cuando refactorizamos, estamos modificando la estructura interna de un programa o de un método. Ese cambio interno no afecta al comportamiento ni a la interfaz.
    - Sin embargo, por ejemplo, si renombramos un método → hay que cambiar todas las referencias que se hacen a él. Siempre que se hace esto se genera un problema si es una interfaz pública.
  - No es recomendable refactorizar, cuando la estructura a modificar es de vital importancia en el diseño de la aplicación.
  - Hay ocasiones en las que no se debería refactorizar en absoluto. Nos podemos encontrar con un código que, aunque se puede refactorizar, sería más fácil reescribirlo desde el principio → Si un código no funciona, no se refactoriza, se reescribe.

## 4.2. Patrones de refactorización más habituales

8

- A continuación, se exponen los patrones más habituales de refactorización, que vienen ya integrados en la mayoría de IDEs más extendidos en el mercado:
  1. **Tabulación:** Propiamente no es una técnica de refactorización, pero permite visualizar el código organizado jerárquicamente. (Source→Format) 😊
  2. **Renombrado (rename):** Este patrón nos indica que debemos cambiar el nombre de un paquete, clase, método o campo, por *un nombre más significativo*.





## 4.2. Patrones de refactorización más habituales

9

```
public void PrintAccountDetails(Account account)
{
    // print summary
    System.out.println("Account: " + account.getId());
    System.out.println("Balance: " + account.getBalance());
    // print history
    for (Iterator iterator = account.getTransactions().iterator(); iterator.hasNext();) {
        Transaction tx = (Transaction) iterator.next();
        System.out.println("Type: " + tx.getType() +
            "Date: " + tx.getDate().toString() +
            " Amount: " + tx.getAmount().toString());
    }
}
```

3. **Extraer método:** Este patrón nos aconseja sustituir un bloque de código, por un método. De esta forma, cada vez que queramos acceder a ese bloque de código, bastaría con invocar al método. Su nombre debe explicar el propósito del método.

```
public void PrintAccountDetails(Account account)
{
    printSummary(account);
    printHistory(account);
}

private void printSummary(Account account) {
    System.out.println("Account: " + account.getId());
    System.out.println("Balance: " + account.getBalance());
}

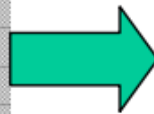
private void printHistory(Account account) {
    for (Iterator iterator = account.getTransactions().iterator(); iterator.hasNext();) {
        Transaction tx = (Transaction) iterator.next();
        System.out.println("Type: " + tx.getType() +
            "Date: " + tx.getDate().toString() +
            " Amount: " + tx.getAmount().toString());
    }
}
```

## 4.2. Patrones de refactorización más habituales

10

4. **Campos encapsulados (encapsulated field):** 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.

```
public class Course
{
    public List students;
}
```



```
public class Course
{
    private List students;

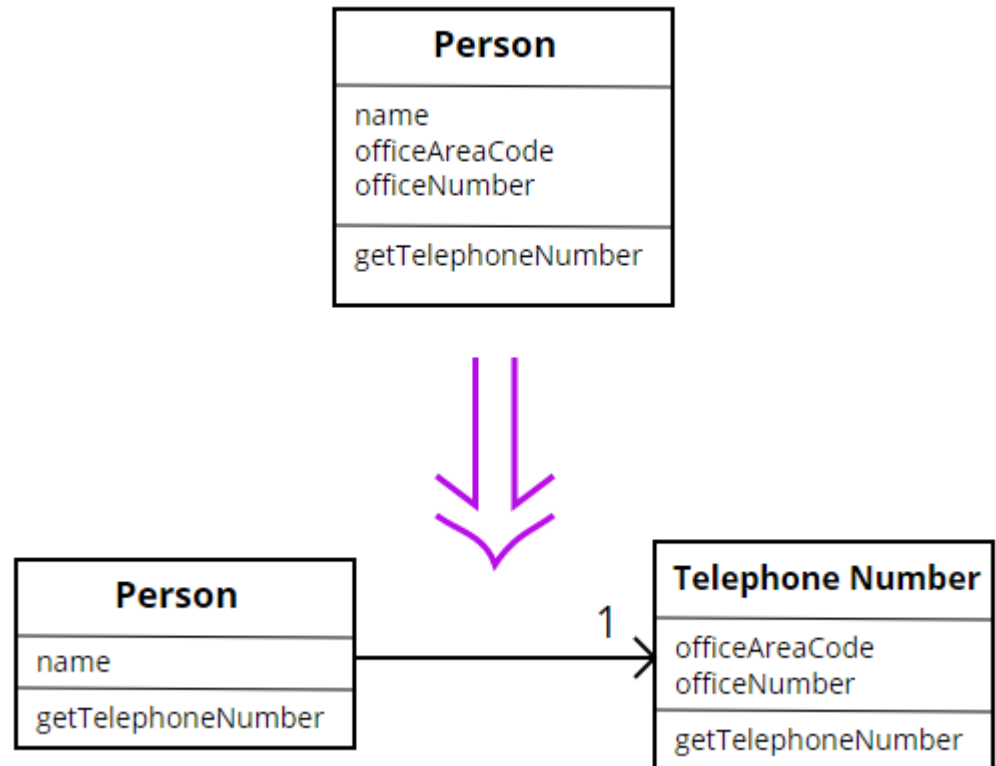
    public List getStudents() {
        return students;
    }
}
```

## 4.2. Patrones de refactorización más habituales

11

### 5. Extraer la clase

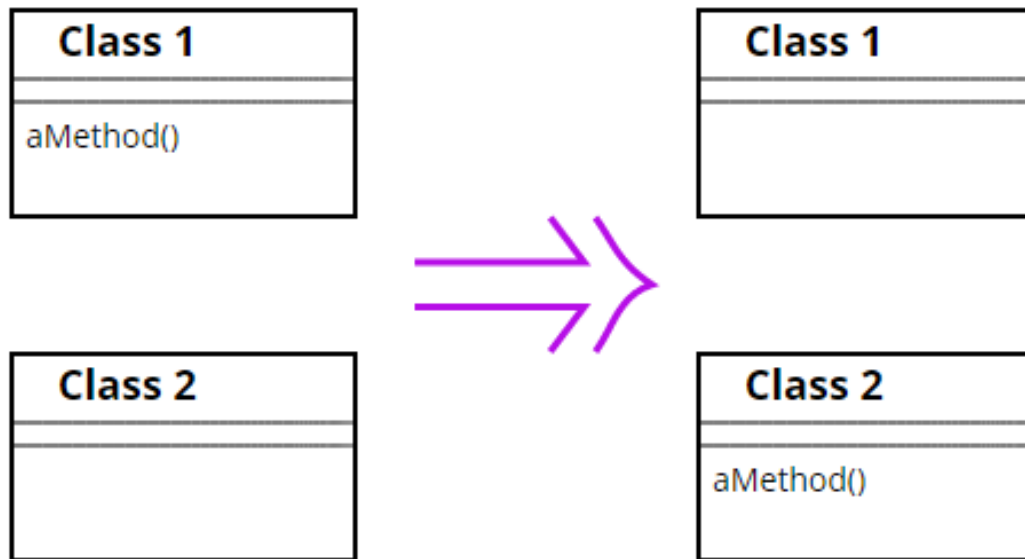
**(extract class):** En este caso se ha implementado una clase, cuando realmente se debería de descomponer en dos. Esto hace necesario crear una nueva clase moviendo los atributos y métodos correspondientes a la nueva clase. Esto impone la actualización en todo el código fuente de las referencias a su nueva localización.



## 4.2. Patrones de refactorización más habituales

12

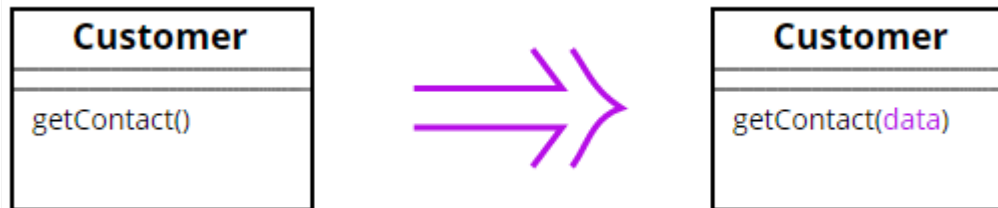
6. **Mover método:** Ocurre cuando, el método en cuestión se utiliza más por otra clase que en la que está definida.



## 4.2. Patrones de refactorización más habituales

13

7. **Borrado seguro:** Se debe comprobar, que cuándo un elemento del código ya no es necesario, se han borrado todas las referencias a él que había en cualquier parte del proyecto.
8. **Cambiar los parámetros de un método (*Change method signature*):** Nos permite añadir nuevos parámetros a un método y cambiar los modificadores de acceso.

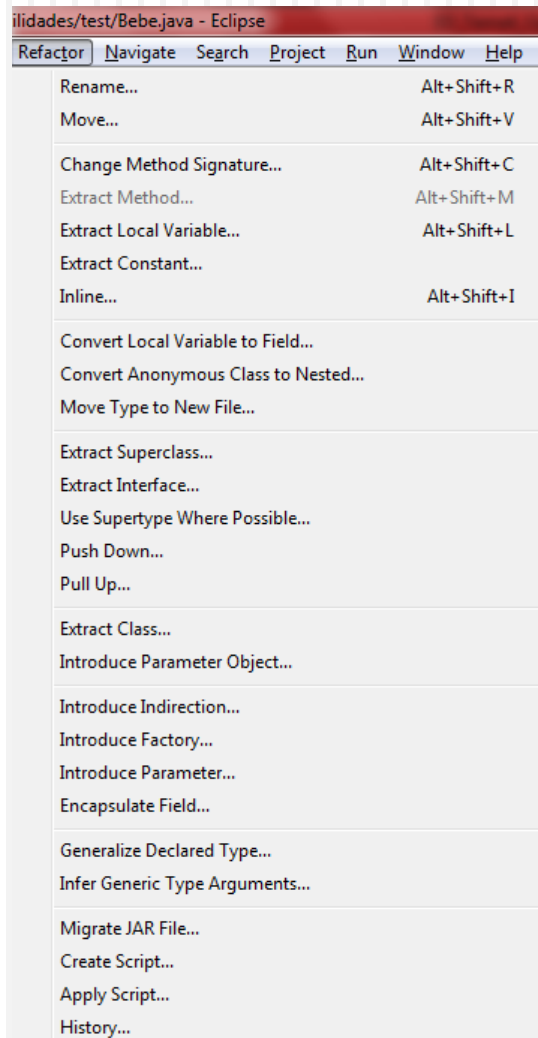


9. **Extraer la interfaz:** Crea una nueva interfaz de los métodos public non-static seleccionados en una clase o interfaz.

## 4.2. Patrones de refactorización más habituales

14

➤ Eclipse →




## A4.1 Refactorización:



```
public class RestauranteIsrael {  
  
    public int p;  
    public int c;  
  
    RestauranteIsrael(int a, int b) {  
        this.p = a;  
        this.c = b;  
    }  
  
    int cantidadPapas() {  
        int a = p * 3;  
        return a;  
    }  
  
    int cantidadChocos() {  
        int ch = c * 6;  
        return ch;  
    }  
  
    public void addChocos(int c) {  
        this.c = this.c + c;  
    }  
  
    public void addPapas(int d) {  
        this.p = this.p + d;  
    }  
}
```

```
public static void main(String[] args) {  
  
    RestauranteIsrael r1 = new RestauranteIsrael(50, 60);  
    System.out.println("Numero de chocos: " + r1.c);  
    System.out.println("Numero de papas: " + r1.p);  
  
    if (r1.cantidadPapas() <= r1.cantidadChocos()) {  
        System.out.println("Israel puede dar de comer a "  
            + r1.cantidadPapas() + " personas");  
    } else {  
        System.out.println("Israel puede dar de comer a "  
            + r1.cantidadChocos() + " personas");  
    }  
  
    r1.addChocos(40);  
    r1.addPapas(20);  
    System.out.println(r1.c);  
    System.out.println(r1.p);  
  
    if (r1.cantidadPapas() <= r1.cantidadChocos()) {  
        System.out.println("Israel puede dar de comer a "  
            + r1.cantidadPapas() + " personas");  
    } else {  
        System.out.println("Israel puede dar de comer a "  
            + r1.cantidadChocos() + " personas");  
    }  
  
}
```

- 
- 1) Copiar el código anterior en una clase llamada RestaurantIsrael.
  - 2) Cambiar el nombre de la atributo “p”, por “papa”. Y el de “c” por “choco”.
  - 3) Encapsula los atributos.
  - 4) Introduce el método calculaComensales que sustituya al código que calcula el número de comensales.



## 4.3. Documentación

17

- El proceso de documentación de código, es uno de los aspectos más importantes de la labor de un programador.
- Para qué sirve:
  - Para explicar su funcionamiento, de forma que cualquier persona que lea el comentario, puede entender la finalidad del código. Debe tratar de explicar todo lo que no resulta evidente. Su objetivo no es repetir lo que hace el código, sino explicar por qué se hace.
    - Cúal es la finalidad de un clase, de un paquete, qué hace un método, para que sirve una variable, qué se espera del uso de una variable, qué algoritmo se usa, qué se podría mejorar en el futuro, etc.
  - Para la detección de errores y para su mantenimiento posterior, que en muchos casos, es realizado por personas diferentes a las que intervinieron en su creación. Todos los programas tienen errores y todos los programas sufren modificaciones a los largo de su vida.

## 4.3.1 Uso de comentarios

18

- Uno de los elementos básicos para documentar código, es el **uso de comentarios**.
- Un comentario es una anotación que se realiza en el código, pero que el compilador va a ignorar, sirve para indicar a los desarrolladores de código diferentes aspectos del código que pueden ser útiles.
- Dos propósitos diferentes comunes:
  1. Explicar el objetivo de las sentencias. De forma que el programador, sepa en todo momento la función de esa sentencia, tanto si lo diseñaron como si son otros los que quieren entenderlo o modificarlo.
  2. Explicar qué realiza un método, o clase, no cómo lo realiza. En este caso, se trata de explicar los valores que va a devolver un método, pero no se trata de explicar cómo se ha diseñado.

## 4.3.1 Uso de comentarios

19

- En el caso del lenguaje Java, C# y C, los comentarios, se implementan de forma similar.
  - Cuando se trata de explicar la función de una sentencia, se usan los caracteres `//` seguidos del comentario, o con los caracteres `/*` y `*/`, situando el comentario entre ellos: `/* comentario */`.
- Otro tipo de comentarios que se utilizan en Java, son los que se utilizan **para explicar qué hace un código**, se denominan **comentarios JavaDoc** y se escriben empezando por `/**` y terminando con `*/`, estos comentarios pueden ocupar varias líneas. Este tipo de comentarios tienen que seguir una estructura prefijada.
- Con JavaDoc:
  - Los comentarios se deben incorporar al principio de cada clase, al principio de cada método y al principio de cada atributo de clase.
  - No es obligatorio, pero en muchas situaciones es conveniente, poner los comentarios al principio de un fragmento de código que no resulta lo suficientemente claro, a la largo de bucles, o si hay alguna línea de código que no resulta evidente y pueda llevarnos a confusión.
- Hay que tener en cuenta, que **si el código es modificado, también se deberán modificar los comentarios.**

## 4.3.2 Alternativas

20

- En la actualidad, el desarrollo rápido de aplicaciones, en muchos casos, va en detrimento de una buena documentación del código.
- Si el código no está documentado, puede resultar bastante difícil de entender, y por tanto de solucionar errores y de mantenerlo.
- La primera alternativa que surge para documentar código, son los comentarios.
- Existen diferentes herramientas que permiten automatizar, completar y enriquecer nuestra documentación → JavaDoc, SchemeSpy y Doxygen.
- De forma que, insertando comentarios en el código más difícil de entender, y utilizando la documentación generada por alguna de las herramientas citadas, se genera la suficiente información para ayudar a cualquier nuevo programador.

## 4.3.3 Documentación de clases

21

- **Las clases que se implementan en una aplicación, deben de incluir comentarios.**
- En el lenguaje Java, los criterios de documentación de clases, son los establecidos por JavaDoc.
  - Los comentarios de una clase deben comenzar con `/**` y terminar con `*/`. Entre la información que debe incluir un comentario de clase: las etiquetas **@autor** y **@version**.
  - **Con el uso de los entornos de desarrollo, las etiquetas se añaden de forma automática.** También se suele añadir la etiqueta **@see**, que se utiliza para referenciar a otras clases y métodos.
  - Dentro de la la clase, también se documentan los **constructores y los métodos**. Al menos se indican las etiquetas:
    - **@param**: seguido del nombre, se usa para indicar cada uno de los parámetros que tienen el constructor o método.
    - **@return**: si el método no es void, se indica lo que devuelve.
    - **@throws**: se indica el nombre de la excepción, especificando cuales pueden lanzarse.
  - Los campos de una clase, también pueden incluir comentarios, aunque no existen etiquetas obligatorias en JavaDoc.

## 4.3.4 Herramientas

22

- En concreto, los entornos de programación que implementan Java, como Eclipse o Netbeans, incluyen una herramienta (JavaDoc) que genera páginas HTML de documentación a partir de los comentarios incluidos en el código fuente.

## 4.3.4 Herramientas

23

- Para que JavaDoc pueda generar las páginas HTML es necesario seguir una serie de normas de documentación en el código fuente, estas son:
  - Los comentarios JavaDoc deben empezar por `/**` y terminar por `*/`.
  - Los comentarios pueden ser a nivel de clase, a nivel de variable y a nivel de método.
  - La documentación se genera para métodos `public` y `protected`.
  - Se pueden usar tags para documentar diferentes aspectos del código, como:

Tipo de tag	Formato	Descripción
<b>Todos.</b>	<code>@see.</code>	Permite crear una referencia a la documentación de otra clase o método.
<b>Clases.</b>	<code>@version.</code>	Comentario con datos indicativos del número de versión.
<b>Clases.</b>	<code>@author.</code>	Nombre del autor.
<b>Clases.</b>	<code>@since.</code>	Fecha desde la que está presente la clase.
<b>Métodos.</b>	<code>@param.</code>	Parámetros que recibe el método.
<b>Métodos.</b>	<code>@return.</code>	Significado del dato devuelto por el método
<b>Métodos.</b>	<code>@throws.</code>	Comentario sobre las excepciones que lanza.
<b>Métodos.</b>	<code>@deprecated.</code>	Indicación de que el método es obsoleto.

# 4.3.4 Herramientas

24

```
/**
 * Java class example The class illustrates how to write comments used to
 * generate JavaDoc documentation
 *
 * @author Catalin
 * @version 2.00, 23 Dec 2010
 */
```

```
public class MyClass {
    /**
     *
     * Simple method.
     *
     * The method prints a received message on the Console
     *
     * @param message
     *         String variable to be printed
     * @see MyClass
     * @deprecated
     * @since version 1.00
     */
    public void myMethod(String message) {
        System.out.printf(message);
    }
}
```

1

```
/**
 *
 * Simple method example. The method prints a received message on the
 * Console
 *
 * @param message
 *         String variable to be printed
 * @since version 1.00
 */
public void printMessage(String message) {
    System.out.printf(message);
}
```

```
/**
 *
 * Simple method example.
 *
 * The methods adds 2 numbers and return the result
 *
 * @param val1
 *         the first value
 * @param val2
 *         the second value
 * @return sum between val1 and val2
 * @since version 2.00
 */
public int add(int val1, int val2) {
    return val1 + val2;
}
```

2



# 4.3.4 Herramientas

25

All Classes

MyClass

Package

Class

Use

Tree

Deprecated

Index

Help

Prev Class

Next Class

Frames

No Frames

Summary: Nested | Field | Constr | Method

Detail: Field | Constr | Method

## Class MyClass

java.lang.Object  
MyClass

---

```
public class MyClass  
extends java.lang.Object
```

Java class example The class illustrates how to write comments used to generate JavaDoc documentation

**Version:**  
2.00, 23 Dec 2010

**Author:**  
Catalin

### Constructor Summary

Constructors

Constructor and Description

MyClass ()

### Method Summary

#### A4.2 Documentacion:

- Sobre la clase Restaurante de la actividad A4.1 inserta comentarios JavaDoc.
- Genera la documentación.

#### E2.3 Refactorización y documentación

## 4.4. Control de versiones

27

- Cuando estamos desarrollando software, el código fuente está *cambiando continuamente*, siendo esta particularidad vital → Sistemas de control de versiones
- Ventajas:
  - *Facilita al equipo de desarrollo su labor*, permitiendo que varios desarrolladores trabajen en el mismo proyecto (incluso sobre los mismo archivos) de forma simultánea, sin que se que pisen unos a otros.
  - *Proveen de un sitio central donde almacenar el código fuente* de la aplicación, así como el *historial de cambios* realizados a lo largo de la vida del proyecto.
  - También permite a los desarrolladores *volver a un versión estable* previa del código fuente si fuera necesario.

## 4.4. Control de versiones

28

- Ejemplos de controladores de código abierto: CVS y Subversion.
- Estructura de los controladores de versiones:
  - Utilizan una *arquitectura cliente-servidor*: un servidor guarda la versión actual del proyecto y su historia, y los clientes conectan al servidor para sacar una copia completa del proyecto, trabajar en esa copia y entonces ingresar sus cambios.
  - Los clientes pueden también comparar diferentes versiones de ficheros, solicitar una historia completa de los cambios, etc.
  - Los clientes también pueden utilizar el comando de actualización con el fin de tener sus copias al día con la última versión que se encuentra en el servidor.

## 4.4. Control de versiones

- Componentes de un controlador de versiones:
  - **Repositorio:** Es el lugar de almacenamiento de los datos de los proyectos. Suele ser un directorio en algún ordenador.
  - **Módulo:** En un directorio específico del repositorio. Puede identificar una parte del proyecto o ser el proyecto por completo.
  - **Revisión:** Es cada una de las versiones parciales o cambios en los archivos o repositorio completo. La evolución del sistema se mide en revisiones. Cada cambio se considera incremental.
  - **Etiqueta:** Información textual que se añade a un conjunto de archivos o a un módulo completo para indicar alguna información importante.
  - **Rama:** Revisiones paralelas de un módulo para efectuar cambios sin tocar la evolución principal. Se suele emplear para pruebas o para mantener los cambios en versiones antiguas.

## 4.4. Control de versiones

30

- Las órdenes que se pueden ejecutar son:
  - **Update:** actualiza la copia local con los últimos cambios del repositorio.
  - **Checkout:** obtiene una copia del trabajo para poder trabajar con ella, por parte de un usuario/desarrollador.
  - **Commit:** almacena la copia modificada en el repositorio.
  - **Abort:** abandona los cambios realizados en la copia de trabajo.
- Cuando usamos un sistema de control de versiones, trabajamos de forma local, sincronizándonos con el repositorio, haciendo los cambios en nuestra copia local, realizando el cambio, y después almacenando la copia modificada de nuevo en el repositorio.

### A.4.3 Control de versiones (Moodle)

