Tema 5: Optimizar y documentar el código

¿Qué aprenderás?

- Refactorizar el código.
- Identificar los patrones de refactoring más comunes.
- Aplicar el funcionamiento de las herramientas de control de versiones.
- Reconocer la estructura de los programas de control de versiones.
- Localizar la importancia de documentar el código.

¿Sabías que...?

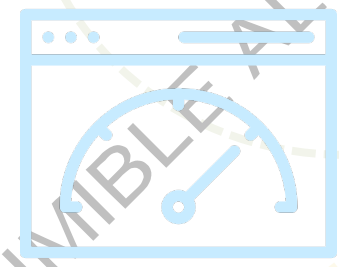
- El refactoring es una técnica de la ingeniería de software.
- Gracias al control de versiones el trabajo en equipo es más sencillo.
- Documentar el código puede ser una de las tareas más tediosas para un desarrollador. Sin embargo, una buena documentación puede mejorar el desarrollo de una aplicación.

5.1. Optimización del código

5.1.1. Refactoring

Para hablar del refactoring, partiremos de que nuestro código fuente puede que funcione, pero su codificación no sea óptima.

Por ejemplo podemos tener un programa cuyo objetivo sea una hacer de calculadora. Imaginate que tiene una función SUMA que realiza una operación a priori muy simple. Si esta función internamente tiene un proceso demasiado complejo y un uso de métodos y variables poco óptimizado. Aunque el resultado que devuelve sea el correcto seguramente no sea la mejor solución a nivel de desarrollo.



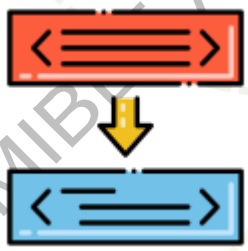
En definitiva, podemos tener un código que funciona, pero tal vez no sea el más óptimo. En este escenario es en el que entra en acción el **refactoring**.

5.1.1.1. Qué es el refactoring

El refactoring (refactorización) son los pequeños cambios que podemos hacer en el código para que sea más visible, más flexible, más fácil y modificable.

- **Flexible:** será más fácil de modificar o adaptar otro código de una forma más fácil y segura.
- **Modificable:** nos permitirá hacer cambios de forma más sencilla.
- **Visible** (hablamos de legibilidad) será más fácil de comprender. Si escribimos nuestro código y conseguimos que sea más visible se lo podremos pasar a otro programador y le resultará más fácil de entender.

El refactoring se debe aplicar a un código que ya funciona. Una vez aplicado se debe verificar que una vez refactorizado siga funcionando.



¿Para qué refactorizar si ya funciona el código?

Pues para tener un código funcional pero mucho más fácil de mantener reutilizar y de entender.

Refactorizar el código nos aporta una serie de ventajas, la principal es que aumenta la calidad del código.

Nos referimos la calidad del código teniendo en cuenta que se mantendrá mejor a medida que se vaya modificando, ampliando o manteniendo. De esta manera facilitamos la tarea de trabajar con nuestro código y disminuimos la probabilidad de generar errores.

Ventajas del refactoring

- Evitar problemas derivados de los cambios de los mantenimientos posteriores
- Simplificar el diseño
- Ayuda a entenderlo mejor
- Será más fácil detectar errores
- Agiliza la programación

Inconvenientes del refactoring

- No es fácil de aplicar si se tiene poca experiencia
- Si se llega a un exceso de querer optimizar el código de una forma obsesiva
- Dedicar mucho tiempo a la refactorización
- Posible impacto de la refactorizacion al resto del software

Por ejemplo, si en un código fuente hacemos servir un valor en concreto, por ejemplo el número pi y cada vez que se usa se crea una variable para almacenarlo.

```
var pi = 3.14;
```

La mejor práctica sería tener una constante a la que se puede acceder de forma global cada vez que se necesite su valor para realizar un cálculo.

De esta manera estamos ahorrando la instrucción de crear la variable y asignarle valor cada vez que la necesitamos.

Otro escenario que se podría dar es el de modificar el valor para ganar más precisión en los calculos, por ejemplo, pasar del valor 3.14 a 3.1415. Pensad si se hubiera declarado esta variable en muchas funciones diferentes. Tendríamos que ir una a una y modificar el valor.

¿Cuando se realiza el refactoring?

No existe una fase específica para aplicar el refactoring, aún así es recomendable hacerlo durante la fase de mantenimiento.

5.1.1.2. Patrones de *refactoring* más usuales

Muchos de los problemas que se repiten en el código ya le han ocurrido anteriormente a muchos programadores. También alguien ha encontrado la solución mucho antes que nosotros. Por lo tanto, no tendremos que reinventar la rueda, eso si, debemos conocer los patrones de refactorización más usuales para poder identificarlos en nuestro código fuente.

Al ser problemas recurrentes a los que se le ha encontrado una solución, esta se estandariza y se convierte en un patrón de refactorización, se le da un nombre y de esa manera podemos usarlos.

Estos patrones de refactoring, nos ofrecen una solución durante el proceso de desarrollo a nuestro código y normalmente pueden usarse en diferentes contextos.

Existen muchos patrones y cada patrón es distinto.

- Cada uno aporta una serie de reglas o casos de uso se deben aplicar en el código.
- No siempre se pueden aplicar estos patrones código que estamos trabajando. Por ejemplo estamos programando con un lenguaje de programación orientado a objetos vamos a necesitar patrones de refactorización específicos de ese paradigma programación.

Podéis encontrar diferentes patrones de refactorización en las versiones diferentes de Eclipse independientemente del sistema operativo.

Cómo existen muchos tipos de patrones de refactorización, nosotros vamos a ver la aplicación de algunos de los patrones existentes aplicados a la programación orientada a objetos.

- **Extraer Método**

¿Cuándo se puede aplicar?

Cuando se tiene un fragmento de código que puede agruparse.

¿Qué hay que hacer?

Convertir el fragmento en un método cuyo nombre explique el propósito del método.

```
void imprimir() {
      imprimirBanner();
      //detalles de impresión
      Console.WriteLine ("nombre: " + _nombre);
      Console.WriteLine("cantidad " + getCargoPendiente());
}
// Refactorizamos
void ImprimirTodo() {
      imprimirBanner();
      imprimirDetalles(getCargoPendiente());
}
void imprimirDetalles(double cargoPendiente) {
      Console.WriteLine ("nombre: " + _nombre);
      Console.WriteLine("cantidad " + cargoPendiente);
}
```

- **Separar variables temporales**

¿Cuándo se puede aplicar?

Si tienes una variable temporal que usas más de una vez, pero no es una variable de bucle ni una variable temporal de colección.

¿Qué hay que hacer?

Creamos una variable temporal diferente para cada asignación.

```
double temp = 2 * (_alto + _ancho);
Console.WriteLine (temp);
temp = _alto * _ancho;
Console.WriteLine (temp);


//Refactorizamos
final double perimetro = 2 * (_alto + _ancho);
Console.WriteLine (perimetro);
final double area = _alto * _ancho;
Console.WriteLine (area);
```

- **Eliminar asignaciones a parámetros**

¿Cuándo se puede aplicar?

Cuando tenemos un parámetro que es usado para recibir una asignación.

¿Qué hay que hacer?

Usar una variable temporal en su lugar para no modificar el valor del parámetro.

```
int descuento (int entradaValor, int cantidad, int año) {
      if (entradaValor > 50)
            entradaValor -= 2;
      […]
}

//Refactorizamos

int descuento (int entradaValor, int cantidad, int año) {
      int resultado = entradaValor;
      if (entradaValor > 50)
            resultado -= 2;
      […]
}
```

- **Mover método**

¿Cuándo se puede aplicar?

Cuando un método es, o será, usado por más características de otra clase que en aquella donde está definido.

¿Qué hay que hacer?

Crear un nuevo método con un cuerpo similar en la clase que se use más. Convertiremos el cuerpo del método antiguo en una delegación simple o lo removeremos por completo.

```
class Proyecto {
      Persona[] participantes;
}
class Persona {
      int id;
      boolean participante(Proyecto p) {
      for(int i=0; i<p.participantes.length; i++)
      if (p.participantes[i].id == id)
      return(true);
      return(false);
}
[…] if (x.participante(p)) […]
```

- **Consolidar fragmentos duplicados en condiciones**

¿Cuándo se puede aplicar?

Si el mismo fragmento de código está en todas las ramas de una expresión condicional.

¿Qué hay que hacer?

Sacamos dicho fragmento fuera de la expresión.

```
if (esAcuerdoEspecial()) {
      total = precio * 0.95;
      enviar();
}else {
      total = precio * 0.98;
      enviar();
}
Refactorizamos

if (esAcuerdoEspecial())
      total = precio * 0.95;
else
      total = precio * 0.98;
enviar();
```

- **Descomponer un condicional**

¿Cuándo se puede aplicar?

Cuando tenemos una complicada declaración en el condicional.

¿Qué hay que hacer?

Extraer los métodos de la condición y del cuerpo del condicional.

```
if (fecha.antes (EMPIEZA_VERANO) || fecha.despues(FIN_VERANO))
      cargo = cantidad * _tasaInvierno + _cargoServicioInvierno;
else
      cargo = cantidad * _tasaVerano;

//Refactorizamos

if (noEsVerano(fecha))
      cargo = cargoInvierno(cantidad);
else
      charge = cargoVerano (cantidad);

double cargoInvierno(int cantidad) {
      return cantidad * _tasaInvierno + _cargoServicioInvierno;
}
double cargoVerano(int cantidad) {
      return cantidad * _tasaVerano;
}
```

- **Consolidar expresiones condicionales**

¿Cuándo se puede aplicar?

Cuando tenemos una secuencia de condicionales con el mismo resultado.

¿Qué hay que hacer?

Los combinamos en una sola expresión y lo extraemos.

```
double cuantiaPorDiscapacidad() {
      if (_antiguedad < 2)
            return 0;
      if (_mesesDiscapacitado > 12)
            return 0;
      if (_esTiempoParcial)
            return 0;
}
Refactorizamos

double cuantiaPorDiscapacidad () {
      if (esNoElegibleParaDiscapacidad())
            return 0;
}
```

- **Reemplazar número mágico con método constante simbólica**

¿Cuándo se puede aplicar?

Tenemos un literal con un significado particular.

¿Qué hay que hacer?

Creamos una constante, la nombramos significativamente y la sustituimos por el literal.

```
double energiaPotencial(double masa, double altura) {
      return masa * altura * 9.81;
}
Refactorizamos

double energiaPotencial(double masa, double altura) {
      return masa * CONSTANTE_GRAVITACIONAL * altura;
}
static final double CONSTANTE_GRAVITACIONAL = 9.81;
```

- **Reemplazar número mágico con método constante**

¿Cuándo se puede aplicar?

Cuando tenemos un literal con un significado particular.

¿Qué hay que hacer?

Crear un método que nos devuelve el literal, lo nombramos significativamente y lo sustituimos por el literal.

```
double energiaPotencial(double masa, double altura) {
	return masa * altura * 9.81;
}
Refactorizamos

double energiaPotencial(double masa, double altura) {
	return masa * constanteGravitacional() * altura;
}
public static double constanteGravitacional(){
	return 9.81;
}
```

- **Encapsular atributo**

¿Cuándo se puede aplicar?

Tenemos un atributo público

¿Qué hay que hacer?

Lo convertimos a privado y le creamos métodos de acceso.

```
public String _nombre;

Refactorizamos

private String _nombre;
public String getNombre()
{
  return _nombre;
}
public void setNombre(String arg)
{
  _nombre = arg;
}
```

- **Reemplazar subclases por atributos**

¿Cuándo se puede aplicar?

Tenemos subclases que sólo varían en métodos que devuelven información constante

¿Qué hay que hacer?

Cambiamos los métodos por atributos de la superclase y eliminamos las subclases.

- **Extraer clase**

¿Cuándo se puede aplicar?

Tenemos una clase que hace el trabajo que debería ser hecho por dos

¿Qué hay que hacer?

Creamos una nueva clase y movemos los atributos y métodos relevantes de la vieja a la nueva clase.

5.1.1.3. Los malos olores (*bad smells*)

Los malos olores o *bad smells* se refieren al código que está mal hecho y el cuál podemos corregir, mejorar o refactorizar.

Es importante conocer algunos casos para ser capaces de mejorar la capacidad de detectar estos malos olores. Si conseguimos localizarlos en el código, probablemente podamos mejorarlos y obtener un código mejorado.

Cuando se detecte un mal olor, tendremos un indicador de que se ha codificado con malas prácticas. Tocará aplicar alguno de los patrones de refactorización. Eso si, no siempre podremos solucionarlo o no conseguimos mejorarlo de una forma fácil.

Siempre tendremos la opción de dejarlo como estaba aunque no sea la mejor práctica. ¿Te suena la expresión: "*si funciona, no lo toques*"?

A continuación, vemos algunos de los sintomas que puedes encontrar en el código.

Para los siguientes malos olores, es recomendable programar pensando en aquella frase de **divide y vencerás**.

Método largo

- Si encontramos una función de 30 o 40 líneas que podríamos **dividir** este método tan largo en otros métodos más cortos. Mejor tener **pequeñas funciones que se expliquen por sí mismas.**

Clases largas

- Cuando tenemos una clase demasiado grande con sus parámetros y métodos. Piensa cuando nos toque modificar nuestro software. Todo el impacto que puede tener realizar un cambio será mucho mayor para una clase grande que no para **clases más pequeñas**.

Lista de parámetros largas

- Cuando un método tiene muchos parámetros de entrada la dificultad de acoplarlo con otras funciones aumenta. Siempre **será mejor tener métodos con pocos parámetros de entrada** con tal de mejorar la compatibilidad. Si nuestra función necesita muchos parámetros tal vez debamos preguntarnos si la podemos dividir en diferentes funciones (divide y vencerás).

Para los siguientes malos olores, es recomendable codificar pensando en **mejorar la legibilidad del código**.

Estructuras con muchos condicionales anidados

- Cuando empezamos a poner sentencias *switch* o cláusulas *if* anidadas puede ser que si profundizamos demasiado el código resulte difícil de comprender y mantener. Tal vez sea mejor tener una llamada a una función dentro de *if* con tal de simplificar la función y de hacerla más entendible.

Demasiados comentarios

- Los comentarios nos ayudan a entender el código y un código bien documentado siempre ayuda al entendimiento. Si el código requiere comentarios con demasiadas explicaciones eso va a significar no es fácil de entender y no se explica por sí mismo.

Para los siguientes malos olores, es necesario aplicar patrones que permitan **optimizar los recursos.**

Generalidad especulativa

- Son jerarquías difíciles de mantener y de comprender. Suele pasar cuando añadimos nuevas jerarquías o clases que en la actualidad no se hacen servir pero que pensamos que tal vez en un futuro nos puedan ser de utilidad. Esto hace que sea más difícil de mantener el código.

Jerarquías paralelas

- Cada vez que se añade una subclase jerarquía y toca añadir otra nueva clase en otra jerarquía distinta.

Intermediario

- Clases cuyo trabajo es la delegación y hacer de intermediarios, solo pasan un valor o asignan un valor. Tal vez tenga más sentido llamar a la clase qué queremos lugar de una clase intermedia.

Legado rechazado

- Clases heredan muy poco de su herencia. En este caso es conveniente plantearse la relación de herencia.

Intimidad inadecuada

- Clases que tratan con la parte privada de otras. Toca restringir el acceso interno de la clase.

Cadena de mensajes

- Cuando una función pide algo a un objeto que a su vez lo pide a otro y así sucesivamente. Mejor evitar este tipo de situaciones.

Clase perezosa

- Verificar que no existan clases que no hagan nada o casi nada.

Cambios en cadena

- Cuando modificar una clase implica cambiar muchas otras clases.

Duplicación de código

- Vigilar cuando copiamos y pegamos código y hacerlo de forma inteligente. revisar que no quede código duplicado.

Atributo temporal

- Es normal que usemos variables temporales o auxiliares en ciertas circunstancias. No pensamos en darles nombre, las usamos y no recurrimos más a ellas. Si estamos usando demasiadas variables temporales o auxiliares puede ser síntoma de que algo no estamos haciendo bien.

5.2. Documentación del código

5.2.1. La importancia de documentar el código

Documentar el código de un programa consiste en dotar al código fuente de toda explicaciones con información que explica en lenguaje natural lo que hace el código de ese mismo código fuente.

La documentación es una de las fases del desarrollo del software. Es importante documentar el código no sólo para el usuario final sino para los desarrolladores.

El objetivo de documentar es que los desarrolladores y cualquier colaborador puedan entender más facilmente que hace el código y el porqué.

¿Por qué debemos documentar?

- Es una buena práctica
- Para mejorar la legibilidad del código
- Para facilitar el trabajo colaborativo con otros desarrolladores

¿Cuándo documentar el código?

Es recomendable hacerlo cuando estamos codificando, es la opción más sencilla y la que nos permite facilitar su entendimiento.

5.2.2. Los comentarios

Son frases cortas que pretenden aportar información sobre una parte del código. Se pueden encontrar intercalados con las sentencias del código fuente, de hecho los comentarios se consideran que forman parte del código fuente.

Se añaden normalmente con los caracteres // o /* .... */ y se pueden escribir en bloque o en línea.

*calculadora.java

```
1
2  /**
3   * Comentarios en formato JavaDoc
4   * @author Cristian González
5   *
6   */
7
8
9  // comentarios en línea
10
11
12 /*
13  * Comentarios en bloque
14  *
15  */
16
17 public class calculadora {
18     // Función suma
19     public static int suma(int a,int b) {
20
21         return a + b;
22
23     }
24     // Función resta
25     public static int resta(int a,int b) {
26
27         return a - b;
28     }
29
30
```

Dependiendo del lenguaje de programación, los comentarios se hacen de diferentes maneras.

Estos son sólo algunos tipos de comentarios que se pueden hacer en diferentes lenguajes de programación. Incluso muchos lenguajes aceptan diferentes formatos de comentarios.

Visual Basic	•‘ Comentario en •‘ Visual basic
Delphi y Pascal	•(* Comentario *)
Python, Pearl	•# Comentario
HTML	•<!-- Comentario en HTML -->
Cobol, PLI/1	•* Comentario
Java, JavaScript, Delphi, SQL	•//comentario de línea

5.2.3. Herramientas de documentación

Existen herramientas que ayudan a documentar proyectos. Proporcionando unos comentarios en un determinado formato nos generan la documentación.

Javadoc es una utilidad para la generación de documentación en formato HTML a partir de código fuente Java. Se podria decir que Javadoc es el estándar para documentar clases de Java. Muchos IDEs utilizan javadoc para generar de forma automática documentación de clases.

Para documentar una clase se deben añadir los siguientes comentarios en el código fuente:

Nombre de la clase, la descripción general, el número de versión, el nombre de autores. También la documentación de cada constructor o método (especialmente los públicos) incluyendo: nombre del constructor o método, tipo de retorno, nombres y tipos de parámetros si los hay, descripción general, descripción de parámetros (si los hay), descripción del valor que devuelve.

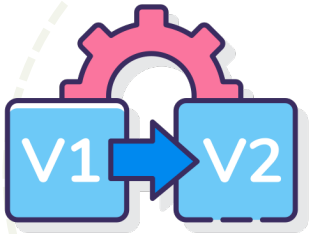
calculadora.java

```
1  package calculadora;
2  /**
3   * Descripción de la clase.
4   * For example:
5   * <pre>
6   *      Añadimos texto con etiquetas HTML
7   * </pre>
8   *
9   * @author  SNathan Drake
0   * @version %I%, %G%
1   * @see     java.awt.BaseWindow
2   * @see     java.awt.Button
3   */
4
5  class calculadora {
6      public static int suma(int a, int b) {
7          return a + b;
8      }
9
0      public static int resta(int a, int b) {
1          return a / b;
2      }
3
```

5.3. Control de versiones

Cuando un grupo de desarrolladores trabaja con el mismo software es muy importante disponer de una herramienta que permita tener un control de las tareas que se realizan sobre el código.

Un sistema de control de versiones es una herramienta de ayuda al desarrollo de software que se encarga de ir almacenando el estado del código fuente en momentos determinados.



El uso de estas herramientas consigue facilitar la tarea a los desarrolladores que trabajan en equipo y en consecuencia permite aumentar la productividad.

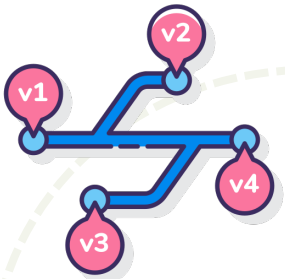
Lo consigue registrando los cambios realizados en un archivo o conjunto de archivos a lo largo del tiempo, de modo que es posible recuperar versiones específicas más adelante.

Dicho de otra manera más metafórica, es como una herramienta que va haciendo fotos del código fuente cada cierto tiempo y las va guardando por si las necesitaramos consultar para ver como estaban antes.

Se ha de tener en cuenta que en el caso que el software se desarrolle por un único programador, será cuestionable que tengamos un software de control de versiones. Aún así, si contamos con un control de versiones, el programador podría volver a recuperar una versión anterior del código.

¿Para qué sirve un programa de control de versiones?

- **Compartir** archivos de diferentes programadores.
- **Bloquear** archivos que se están editando.
- **Fusionar** archivos con diferentes cambios.



¿Qué funcionalidades tiene un programa de control de versiones?

- **Comparar** cambios en el código fuente.
- **Coordinar** las tareas entre diferentes programadores.
- **Guardar versiones** anteriores del código fuente.
- **Seguimiento de los cambios** realizados en el código: con un historial de cambios realizados en el código fuente pudiendo conocer el momento del cambio y el autor.
- **Restaurar** a una versión de código anterior.
- Control de los **usuarios.**
- **Crear ramas** (forks) del proyecto que permiten desarrollar varias versiones de un mismo programa a la vez.

5.3.1. Partes de un sistema de control de versiones

Los sistemas de control de versiones, constan de diferentes partes:

Repositorio: es donde se almacenan los datos actualizados e históricos de cambios. Normalmente es un servidor.

Módulo: Conjunto de directorios y/o archivos dentro del repositorio que pertenecen a un proyecto común.

Revisión: es una versión determinada de la información que se almacena.

Etiqueta (Tag): Darle a cada uno de los ficheros del módulo en desarrollo en un momento preciso un nombre común para asegurarse de reencontrar ese estado de desarrollo posteriormente bajo ese nombre.

Rama (branch): cuando se genera un duplicado del código fuente sobre el que se va a trabajar dentro de los directorios y/o archivos dentro del repositorio.

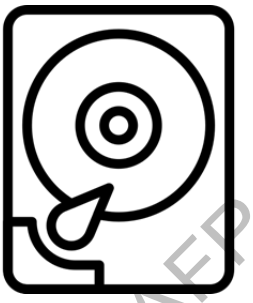
Trunk: Rama principal de código fuente.

Merge: Operación de fusión de diferentes ramas.

Commit: Operación de confirmación de cambios en el sistema de control de versiones.

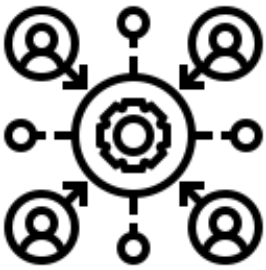
Changeset: Conjunto de cambios que hace un usuario y sobre los que se realiza una operación "Commit" de manera simultánea y que son identificados mediante un número único en el sistema de control de versiones.

5.3.2. Tipos de sistemas de control de versiones



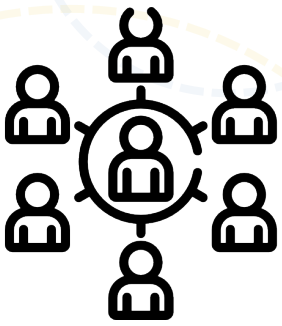
Locales: Cuando se copian los archivos a otro directorio en el mismo equipo. Es sencillo, pero también es tremendamente propenso a errores.

- Fácil olvidar en qué directorio te encuentras.
- Guardar accidentalmente en el archivo equivocado.
- Sobrescribir archivos que no querías.



Centralizados: Tienen un único servidor que contiene todos los archivos versionados y varios clientes que descargan los archivos desde ese lugar central. El responsable es un único usuario. Esto hace que todas las acciones importantes necesiten la aprobación del responsable.

- Más control sobre los cambios



Distribuidos: Cuando cada usuario tiene su propio repositorio en local. Los distintos repositorios se mezclan entre ellos de forma sincronizada para mantener la coherencia.

- Mayor autonomía
- Más rápido
- Aunque caiga el servidor, se puede seguir trabajando
- El servidor

5.3.3. Operaciones que se pueden hacer en un sistema de versiones

Es posible realizar muchas acciones sobre el código del repostorio. No las veremos todas aquí. De todos modos si que hemos de tener claro las acciones más importantes que podemos hacer son las siguientes:

- **Commit** (subir): Sube una copia de los cambios hechos en local que se integra sobre el repositorio.
- **CheckOut** (bajar o desplegar): Cuando crea una copia de trabajo local desde el repositorio.
- **Update** (actualización): cuando se integran los cambios que se han hecho en el repositor en la copia de trabajo local.

5.3.4. GIT

El sistema de control de versiones que más está siendo utilizado es Git (https://git-scm.com), con una cuota de mercado del 70% más o menos.

Es un modelo de repositorio distribuido compatible con sistemas y protocolos como HTTP, FTP, SSH y es capaz de manejar proyectos pequeños y grandes.

Recursos y enlaces

- Manual JavaDoc

- Más información sobre el refactoring

- Más información sobre Bad smells

- Web oficial de GIT

Conceptos clave

- **VCS (Version Control System):** Sistema de control de versiones.
- **Refactoring:** Refactorización.
- **Ingenieria del software:** una de las ramas de las ciencias de la computación que estudia la creación de software confiable y de calidad.
- **JavaDoc:** utilidad de Oracle para la generación de documentación de APIs en formato HTML a partir de código fuente Java.
- **GIT**: es un software de control de versiones pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones.

Test de autoevaluación

1. ¿En qué consiste la refactorización?
   a) En mejorar la estructura interna de un software para mejorarlo y manteniendo las funcionalidades.
   b) En aprovechar la estructura base de un software para crear uno nuevo con distintas funcionalidades.
   c) En mejorar un software volviendo a crearlo utilizando un lenguaje mejor.
   d) En cambiar el comportamiento del programa para conseguir facilitar la interacción con el usuario.

2. El control de versiones...
   a) no permite tener un control de las modificaciones que han hecho los diferentes desarrolladores.
   b) una de sus finalidades es permitir un desarrollo colaborativo en el que muchos programadores trabajan de manera simultánea en un proyecto.
   c) no está pensado para permitir un desarrollo colaborativo en el que muchos programadores trabajan de manera simultánea en un proyecto.
   d) una de sus finalidades es permitir un desarrollo colaborativo en el que muchos programadores trabajan de manera que se van alternando en un proyecto.

3. ¿Para quién es muy importante la documentación de un proyecto software?
   a) Solo para el usuario final.
   b) Para el usuario final y para los propios desarrolladores.
   c) Solo para los proprios desarrolladores.
   d) Para nadie, en general la documentación no sirve.

.

Ponlo en práctica

Refactorizar el código

A partir del código dado, trata de refactorizarlo con alguno de los patrones que conoces.

Aplicar el patrón Extraer método:

```
void imprime() {
    imprimePrimeraLinea();

    //imprimir
    System.out.println ("Nombre:        " + getNombre );
    System.out.println ("Edad          " + getEdad());
}
```

SOLUCIONARIOS

Test de autoevaluación

1. ¿En qué consiste la refactorización?

a) En mejorar la estructura interna de un software para mejorarlo y manteniendo las funcionalidades.

b) En aprovechar la estructura base de un software para crear uno nuevo con distintas funcionalidades.

c) En mejorar un software volviendo a crearlo utilizando un lenguaje mejor.

d) En cambiar el comportamiento del programa para conseguir facilitar la interacción con el usuario.

2. El control de versiones...

a) no permite tener un control de las modificaciones que han hecho los diferentes desarrolladores.

b) una de sus finalidades es permitir un desarrollo colaborativo en el que muchos programadores trabajan de manera simultánea en un proyecto.

c) no está pensado para permitir un desarrollo colaborativo en el que muchos programadores trabajan de manera simultánea en un proyecto.

d) una de sus finalidades es permitir un desarrollo colaborativo en el que muchos programadores trabajan de manera que se van alternando en un proyecto.

3. ¿Para quién es muy importante la documentación de un proyecto software?

a) Solo para el usuario final.

b) Para el usuario final y para los propios desarrolladores.

c) Solo para los proprios desarrolladores.

d) Para nadie, en general la documentación no sirve.

Ponlo en práctica

Refactorizar el código

A partir del código dado, trata de refactorizarlo con alguno de los patrones que conoces.

a) Aplicar el patrón Extraer método:

```
void imprime() {
    imprimePrimeraLinea();

    //imprimir
    System.out.println ("Nombre:        " + getNombre );
    System.out.println ("Edad          " + getEdad());
}
```

Solución:

```
// código refactorizado
void imprime() {
    imprimePrimeraLinea();
    imprimeDetalles(getCantidad(),getEdad());
}

void imprimeDetalles (string nombre, double edad) {
    System.out.println ("Nombre:       " +nombre );
    System.out.println ("Edad:         " +edad );
}
```