5️⃣

Optimización y documentación

▼ Refactorización

La 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.

Con la refactorización se mejora el diseño del software, hace que el software sea más fácil de entender, hace que el mantenimiento del software sea más sencillo, la refactorización nos ayuda a encontrar errores y a que nuestro programa sea más rápido.

▼ ¿Por qué refactorizar?

- 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.

Refactorizar no cambia la funcionalidad del código ni el comportamiento del programa

▼ Patrones de refactorización

El patrón dará una resolución a este problema, que ya ha sido aceptada como solución buena, y que ya ha sido bautizada con un nombre.

No siempre se pueden aplicar estos patrones al código sobre el que estamos trabajando, depende en gran medida del lenguaje que estemos utilizando.

▼ Extraer método

Tenemos un fragmento de código que puede agruparse.

```
void imprimir() {
  mostrarBanner();
  //detalles de impresión
  Console.WriteLine(“nombre: “ + nombre);
  Console.WriteLine(“cantidad “ + getCantidad());
}
```

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

```
//Refactorizamos
void imprimir() {
  mostrarBanner();
  mostrarDetalles(getCantidad));
}
void mostrarDetalles(doublecantidad) {
  Console.WriteLine(“nombre: “ + nombre);
  Console.WriteLine(“cantidad “ + cantidad);
}
```

▼ Separar variables temporales

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

```
double temp = 2 * (base + altura);
Console.WriteLine (temp);
temp= base * altura;
Console.WriteLine (temp);
```

Creamos una variable temporal diferente para cada asignación.

```
//Refactorizamos
final double perímetro = 2 * (base + altura);
Console.WriteLine (perímetro);
```

```
final double area = base * altura;
Console.WriteLine (area);
```

▼ Consolidar fragmentos duplicados en condicionales

El mismo fragmento de código se repite en todas las ramas de una expresión condicional.

```
If (aplicarDescuentoSocio ()) {
  total = precio * 0.75;
  enviar();
}else{
  total = precio * 0.9;
  enviar();
}
```

Sacar el fragmento fuera de la expresión.

```
//Refactorizamos
If (aplicarDescuentoSocio())
  total = precio * 0.75;
else
  total = precio * 0.90;
enviar();
```

▼ Descomponer un condicional

Tenemos una complicada declaración en el condicional.

```
if (fecha.inicio(TEMPORADA_VERANO) || fecha.final(FINAL_VERANO))
  cargo = cantidad * precioInvierno;
else
  cargo = cantidad * precioVerano;
}
```

Extraemos métodos de la condición y del cuerpo del condicional.

```
if (noEsVerano(fecha))
  cargo = cargoInvierno(cantidad);
else
  cargo = cargoVerano(cantidad);
double cargoInvierno(intcantidad) {
```

```
  return cantidad * precioInvierno;
}
double cargoVerano(intcantidad) {
  return cantidad * precioVerano;
}
```

▼ Consolidar expresiones condicionales

De una secuencia de condicionales con el mismo resultado.

```
double precioSegunAntiguedad() {
if (antigüedad < 2)
  return 0;
if (mesesBaja > 12)
  return 0;
if (aTiempoParcial)
  return 0;
}
```

Combinamos en una sola expresión y lo extraemos.

```
//Refactorizamos
double precioSegunAntiguedad() {
  if (noAplicaAntiguedad())
    return 0;
}
```

▼ Reemplazar número mágico con constante simbólica

Tenemos un literal con un significado particular.

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

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

```
//Refactorizamos
double energiaPotencial(doublemasa, doublealtura) {
  return masa * GRAVEDAD * altura;
```

```
}
static final double GRAVEDAD = 9.81;
```

▼ Encapsular atributo

A partir de un atributo público.

```
public String nombre;
```

Se convierte a privado y le creamos métodos de acceso.

```
//Refactorizamos
private String nombre;
public String getNombre(){
  return nombre;
}
Public void setNombre(Stringarg){
  nombre = arg;
}
```

▼ Encapsular atributo como propiedad

Un atributo público.

```
public String_nombre;
```

Se convierte en una propiedad el objeto.

```
//Refactorizamos
public virtual string Nombre {get; set;}
```

▼ Encapsular colección

Un método devuelve una colección.

Hacemos que devuelva una colección de solo lectura y le facilitamos métodos de adición y eliminación de elementos.

▼ Extraer subclase

Propiedades que solo son usadas en determinadas instancias.

Crear subclase para dicho subset de propiedades.

▼ Extraer clase

Hay una clase que hace el trabajo que debería ser hecho por dos.

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

▼ Eliminar asignaciones a parámetros

Un parámetro es usado para recibir una asignación.

```
int descuento (int entrada, int cantidad, int año) {
  if (entrada > 50)
    entrada -= 10;
  [...]
}
```

Usamos una variable temporal en su lugar.

```
//Refactorizamos
int descuento (in tentrada, int cantidad, int año) {
  int resultado = entrada;
  if (entrada > 50)
    resultado -= 10;
  [...]
}
```

▼ Reemplazar número mágico con método constante

Tenemos un literal con un significado particular.

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

Crear método devuelve el literal con un nombre representativo.

```
//Refactorizamos
double energiaPotencial(doublemasa, doublealtura) {
```

```
  return masa * constanteGravitacional() * altura;
}
public static double constanteGravitacional(){
  return9.81;
}
```

▼ Reemplazar datos y valores por objetos

Tenemos un atributo que necesita información o comportamiento adicional.

Convertir el atributo en un objeto.

▼ Reemplazar array con objeto

Un array en el que hay elementos con un significado diferente.

```
String[] jugada = new String[3];
fila[0] = "NathanDrake";
fila[1] = "15";
```

Reemplazar array por un objeto que tenga un atributo para cada elemento.

```
//Refactorizamos
Partida jugada = new Partida();
jugada.setNombre("NathanDrake");
jugada.setBajas("15");
```

▼ Reemplazar subclases por atributos

Subclases que solo varían en métodos que devuelven información constante.

Cambiar métodos por atributos de la superclase y eliminar las subclases.

▼ Malos olores (”Bad Smells”)

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

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 sí, no siempre podremos solucionarlo o no conseguimos mejorarlo de una forma fácil.

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

- Método largo: mejor crear métodos cortos, son más reutilizables y aportan mayor semántica.
- Clases largas: son muy vulnerables al cambio.
- Lista de parámetros largas: los métodos con muchos parámetros complican el acoplamiento, difíciles de comprender y cambian con frecuencia.

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

- Estructuras con muchos condicionales anidados: Switch con muchas cláusulas o muchos If anidados no es una buena idea.
- Demasiados comentarios: un comentario largo puede ocultar muchas veces a otro “mal olor”.

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 único trabajo es la delegación y ser intermediarias.
- Legado rechazado: subclases que usan solo un poco de lo que sus padres les dan. Si las clases hijas no necesitan lo que heredan, generalmente la herencia está mal aplicada.
- Intimidad inadecuada: clases que tratan con la parte privada de otras. Se debe restringir el acceso al conocimiento interno de una clase.
- Cadena de mensajes: un cliente pide algo a un objeto que a su vez lo pide a otro y este a otro, etc.
- Clase perezosa: una clase que no está haciendo nada o casi nada deberían eliminarse.

- 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.

▼ Documentación

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

El objetivo de documentar es que los desarrolladores y cualquier colaborador puedan entender más fácilmente 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.

▼ 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.

▼ ¿Qué es JavaDoc?

Documentación en formato HTML generada automáticamente a partir de los comentarios escritos en el código.

▼ Control de versiones

¿Qué es un repositorio?

> Espacio de almacenamiento donde se guardan archivos y documentos relacionados con un proyecto o software específico.

¿Qué es GitHub?

> GitHub es una plataforma en línea que permite el alojamiento y la gestión de repositorios de código fuente.

¿Qué es Git?

> Git es un sistema de control de versiones.

¿Qué es un sistema de control de versiones?

> Es una herramienta que permite a los desarrolladores gestionar diferentes versiones de un mismo proyecto a lo largo del tiempo.

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.

¿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.

▼ Partes de un sistema de control de versiones:

- 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.

▼ Git

El sistema de control de versiones que más está siendo utilizado es Git (https://git-cm.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.

▼ Eclipse Git

Crear repositorio

1. Ir a www.github.com
2. Crear una cuenta gratuita. Si ya tenéis entrad en vuestra cuenta.
3. Crear un nuevo repositorio haciendo clic en el botón `NUEVO`.
4. Ingresar datos e información inicial del repositorio:
    a. Elegir un nombre para el repositorio.
    b. Opcionalmente, ingrese una descripción.
    c. Indicar si desea que sea público o privado (en este tutorial usaremos uno público).
    d. No marcamos ninguna opción porque queremos un vacío.
    e. Botón `Crear Repositorio`.
5. Aparece esta pantalla que nos confirma que el repositorio se ha creado.
6. Invitar a nuestros compañeros de equipo como colaboradores en el repositorio:
    a. Seleccionar pestaña `Configuración`
    b. En el menú de la izquierda elegir la opción `Administrar el acceso`
    c. Pinchar en el botón `Invita a un colaborador`
    d. Introducir el nombre de los colaboradores
7. Una vez completada esta acción, los colaboradores recibirán un email para aceptar la invitación que les hemos enviado.

Uso de Git en IDE

▼ Crear, inicializar y cargar (Push) una versión inicial de nuestro proyecto

1. Para este caso he creado usando Eclipse un proyecto llamado Tutorial, cuenta con una clase Hola que muestra por consola “Hola, soy Ceinmark”.
2. Crear e inicializar el repositorio:
    a. Haced clic botón derecho sobre el proyecto (Tutorial)
    b. Opción `Team` > `Share Project`

3. Seleccionar la ubicación de nuestro proyecto para el proyecto de Git y creamos un repositorio allí.
    a. En esta ventana seleccionamos la opción `Use or create repository in parent folder of Project`
    b. Seleccionar en la lista el proyecto y su ubicación
    c. Haced clic en `Create Repository`
    d. Clic en `Finish`
4. Vemos en el `Package Explorer` que junto a los íconos del proyecto se muestra un pequeño cilindro amarillo y junto a la carpeta `src`, el package tutorial y el archivo Hola.java se muestra un ícono de `?`. Esto indica que el proyecto tiene un repositorio git asociado, pero existen cambios sin agregar al index de git.
5. Agregar los cambios en index:
    a. Clic con el botón derecho en el proyecto Tutorial
    b. Opción `Team` > `Add to Index`
    - Vemos en el Package explorer que los íconos de antes ahora cambiaron por un `*` blanco (significa cambios en la estructura) en el proyecto, carpeta source y package y un `+` verde (significa agregado) en el archivo Hola.java
6. A continuación, commitear los cambios a git:
    a. Haced clic botón derecho sobre el proyecto Tutorial
    b. Vamos a la opción `Team` > `Commit`
    c. Aparece la pestaña `Git Staging`

En este punto podemos ver a la izquierda en la parte inferior los archivos modificados que se encuentran en el index (`Staged Changes`) y en la parte superior los archivos modificados que NO se han agregado al index (`Unstaged Changes`). Al hacer commit solo se incluirán los cambios que si se hayan agregado al index.

A su vez, en `Commit Message` DEBEMOS agregar un comentario que nos dé una idea de qué cambio se realizó.

- Luego de esto podemos hacer clic en `Commit and Push` o `Commit`.
- Una vez que presionamos `Commit` vemos en el `Project/Package explorer`.

7. Ahora tenemos que hacer push para llevar los cambios del proyecto a GitHub.
   a. Nos dirigimos a la página del repositorio, allí seleccionamos la opción para la url del repo. Usaremos https por simplicidad, si tienen configurado una key ssh puede usar la opción ssh.
   b. Seleccionamos la opción `HTTPS` y hacemos clic en el botón a la derecha para copiar la url o la copiamos a mano.
   c. Hacemos clic derecho sobre el proyecto.
   d. Seleccionamos la opción `Team` > `Push Branch 'master'...`
   Nota: Solo la primera vez, las siguientes veces , debemos usar `Push to Upstream`

Aparece una ventana para configurar el repositorio remoto:

- En el `Remote name` dejamos origin por defecto.
- Pegamos allí la url del repositorio que copiamos antes, se autocompletará casi toda la información.
- Ingresamos usuario y contraseña de github.
- Hacer clic en `Preview >`
- Configurar como `Branch` (rama) remota para sincronizar el mismo nombre (en
  este caso master)
- Seleccionar la opción `Configure upstream for push and pull`.
- Elegir la acción default a realizar el pull como `Merge`
- NO seleccionar el `Force overwrite branch in remote if it exists and has diverged` para evitar perder los cambios realizados por otros.
- Pulsar en `Preview`
- Nos aparece la confirmación del push.

- Clic en el botón en `Close`.

▼ Clonar el repositorio, realizar cambios y cargarlos (Push)

Para este caso abrimos eclipse en un workspace diferente para simular otro usuario.

1. Entrar en el correo electrónico para aceptar la invitación como colaborador.
2. Ir a www.github.com, acceder al repositorio y copiar la url del repositorio para clonar.
   - Clic en el botón `Code`
   - Elegimos la opción `HTTPS`.
   - Copiamos el enlace.
3. Clonamos el repositorio en Eclipse:
   - Ir al menú `Window` > `Perspective` > `Open Perspective` > `Other` > `Git`

   Haced clic en el botón de `Clone` a `Git repository`
   - Se abrirá la siguiente ventana
   - Pegar la url que copiamos del repositorio donde indica URI
   - Completar los datos de usuario y contraseña
   - Hacer clic en Next
   - Nos muestra la lista de branch que deseamos clonar.
   - Dejamos seleccionada master.
   - Clic en `Next`
   - En la siguiente ventana:
   - En `Destination` > `Directory` reemplazamos la ubicación propuesta por el directorio donde se creará el proyecto, es decir, la ubicación del workspace que utilizamos en eclipse
   - Seleccionamos `master` como la rama inicial

- Para la definición del nombre del repositorio remoto (el que se encuentra en GitHub) usaremos origin
- IMPORTANTE: seleccionamos la opción `import all existing Eclipse projects after clone finishes`, para asegurarnos que el proyecto se agregue al `Project/Package explorer`
- Clic en `Finish`

Nos aparece el repositorio agregado a la lista de repositorios de la pestaña `Git Repositories`

En la parte superior derecha, elegimos la perspectiva de `Java` o `Java EE` para ver él proyecto y el código fuente

4. Realizar cambios:
    - Editar y modificar el código. Hasta que no guardemos los cambio git no muestra cambios en los archivos del `Package explorer` ni la pestaña de `git staging`
    - Guardar los cambios en el `Package Explorer` y aparece el signo de `>`.
    - Realizar los pasos siguientes (visto anteriormente)
        a. Clic con el botón derecho en el proyecto Tutorial
        b. Opción `Team` > `Add to Index`
        c. Haced clic botón derecho sobre el proyecto Tutorial
        d. Vamos a la opción `Team` > `Commit`
        e. Aparece la pestaña `Git Staging`

           A su vez, en `Commit Message` DEBEMOS agregar un comentario que nos dé una idea de qué cambio se realizó.

        f. Presionar `Commit`
           En `Package explorer` el archivo y la estructura pasan a tener nuevamente un cilindro amarillo junto al ícono y junto al proyecto vemos una flecha en dirección ascendente con un 1 a la derecha. Esto indica que localmente hemos hecho un commit que no hemos pusheado aún.

g. Clic opción `Team` > `Pull`
   Nota: no es necesario realizar un pull antes de push, pero en caso de que haya cambios en el repositorio remoto hecho por otro compañero si no hacemos pull antes del push veremos un mensaje de error y deberemos hacer pull antes del push.

h. Clic opción `Team` > `Push to upstream`

Si accedemos al GitHub, vemos que ha sido modificado:

▼ Obtener cambios realizados por otro miembro del equipo

Accedo a Eclipse en el workspace original para simular nuestro primer usuario y compruebo qué el código está sin modificar.

1. Sobre el proyecto botón derecho `Team` > `Pull`

Al pinchar en `Close`, nos aparece el código modificado

▼ Crear una nueva rama

Es útil para no trabajar con la rama maestra cuando estamos desarrollando nuevas funcionalidades de nuestro código. Para ello crearemos una nueva rama para cada una de las funcionalidades que estemos desarrollando, una vez superadas las fases de pruebas esta rama podrá ser unida a la rama maestra.

1. Partimos de nuestro proyecto en Eclipse
2. Botón derecho sobre nuestro proyecto `Team` > `Switch To` > `New Branch`
3. Ponemos un nombre a la nueva rama
4. Aparece en la parte de `Package Explorer` el nombre de la nueva rama

   La rama Maestra y la nueva rama contiene el mismo código

   Nota: Si queremos volver a la rama maestra: `Team` > `Switch to` > `master`

   Si vamos a GitHub vemos que sigue apareciendo solo una rama que la Master.

5. Para sincronizar la nueva rama tengo que hacer un `Push`:
6. Damos en `Next`
7. Introducimos el origen y el destino y pinchamos en `+Add spec`. Botón `Next`

8. Vamos al Repositorio Remoto GitHub y comprobamos si se ha subido

▼ Modificar la nueva rama

1. Ahora puedo trabajar sobre la nueva rama y hacer modificaciones en el código desde Eclipse y luego lo subiremos a GitHub (commit y push)

   Nota: Para ver todos los commit que hemos realizado: `Team` > `Show in History`

2. En GitHub, comprobamos que se han realizado los cambios en la nueva rama

▼ Unir ramas (Merge)

Si queremos unir la Rama Maestra con otra rama

1. En el repositorio GitHub, elegir la rama a unir
2. Y pulsamos al botón `Solicitudes de extracción`
3. Clic `Comparar` y `Nueva solicitar extracción`
4. Crear solicitud de extracción, y nos pide introducir un comentario, sobre lo que hace nuestra rama y `Crear solicitud de extracción`.
5. Todavía no hemos realizado la unión (el Merge). Pinchamos en `Solicitud de extracción (1)`
6. Seleccionamos la solicitud y `combinar solicitar de extracción`
7. Merge Pull, `Confirmar fusión`
8. Si nos vamos a la Rama Maestra, ya lo tenemos actualizado con la otra rama

   Nota: Si nos vamos a `Insights` (Estadística) y en el menú de la izquierda vamos a `Red`, nos muestra como de la rama maestra hemos creado una nueva rama, hemos modificado la nueva rama y la hemos vuelto a unir con la maestra.

Si volvemos a Eclipse, comprobamos que nuestra rama no está unida con la maestra.
Para eso tendremos que hacer:

1. `Team` > `Switch to Master`
2. `Team` > `Pull`
3. Al pinchar en `Close`, nos aparece ya el código