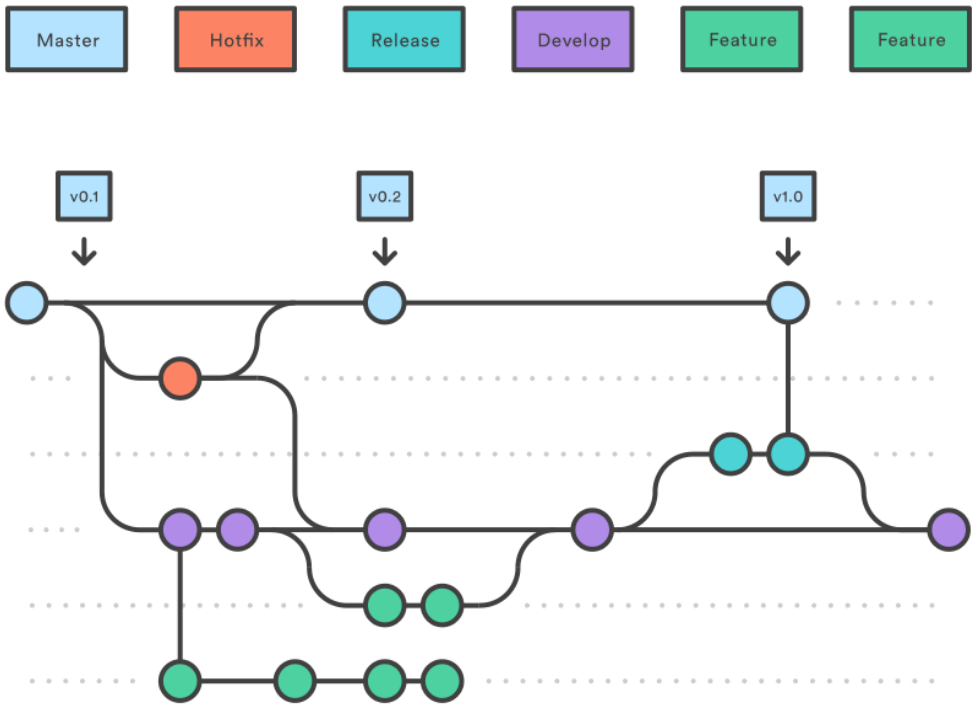
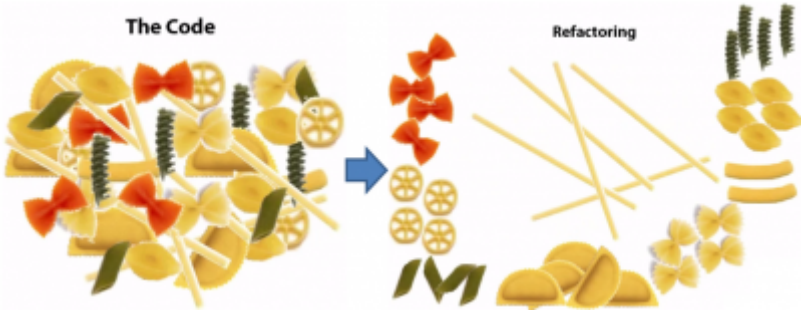
UD03: Refactorización y herramientas de control de versiones



1. Refactorización

El término refactorizar dentro del campo de la Ingeniería del Software hace referencia a la modificación del código sin cambiar su funcionamiento. Se emplea para crear un código más claro y sencillo, facilitando la posterior lectura o revision de un programa. Se podría entender como el mantenimiento del código, para facilitar su comprensión, pero sin añadir ni eliminar funcionalidades. **Refactorizar código consiste en crear un código más limpio.**



La refactorización debe ser un paso aislado en el diseño de un programa, para evitar introducir errores de código al reescribir o modificar algunas partes del mismo. Si después de refactorizar hemos alterado el funcionamiento del código, hemos cometido errores al refactorizar.

Se refactoriza para:

- Limpieza del código, mejorando la consistencia y la claridad.
- Mantenimiento del código, sin corregir errores ni añadir funcionalidades.
- Elimina el código “muerto”, y se modulariza.
- Facilita el futuro mantenimiento y modificación del código.

Los siguientes apartados están inevitablemente relacionados entre si, ya que todas las técnicas o reglas persiguen el mismo fin.

1.1. Convenciones de escritura de Java

Las convenciones de código existen debido a que la mayoría del coste del código de un programa se usa en su mantenimiento, (casi ningún programa se mantiene toda su vida con el código original), y mejoran la lectura del código permitiendo entender código nuevo mucho más rápido y a fondo. En la web de *Sun Microsystems* se recogen dichas convenciones en una guía en ingles, o una traducción al castellano cortesía de *javaHispano*. Por su parte Google también ha creado recientemente una Guía de estilo para Java.

Para que las convenciones funcionen, cada programador debe tratar de ser lo más fiel posible a estas.

1.1.1. Ficheros

Todos los ficheros fuente de java son ficheros de texto plano cuyo nombre termina con la extensión *.java* Dentro de cada fichero *.java* tenemos 4 partes en el siguiente orden:

1. Posibles comentarios sobre la clase (autor, fecha, licencias, etc)
2. Sentencia *package*. Toda clase debe estar en un paquete.
3. Sentencias *import*. Importar cada clase en una linea separada.
4. La definición de una única clase o interface **cuyo nombre es idéntico al nombre del fichero sin la extensión**.

```java
/* Correcto */
import java.awt.Frame;
import java.awt.Graphics;
```

```java
/* Incorrecto */
import java.awt.*;
```

Posteriormente dentro de la definición de la clase, aplicamos el siguiente orden:

1. Sentencia *class o interface*
2. Variables de clase (static)
3. Variables de instancia (Atributos de la clase)
4. Constructores (Si hay sobrecarga deben ir seguidos)
5. Métodos (Si hay sobrecarga deben ir seguidos)

1.1.2. Declaraciones de variables

- Una sola declaración por linea.

```java
int edad;
int cantidad;
```

- Las variables locales se deben inicializar en el momento de declararlas o justo después. Se declaran justo antes de su uso, para reducir su ámbito.
- Las variables de instancia o de clase se declaran al comienzo de la definición de la clase.
- Los arrays se pueden inicializar en bloque:

```java
int[] array =
{
 0, 1, 2, 3
};
// ó
int[] array = { 0, 1, 2, 3 };
```

- Los arrays tienen los corchetes [ ] unidos a su tipo de datos:

```java
String[] nombres; //correcto
String nombres[]; //incorrecto
```

> Por tanto la forma correcta de los argumentos en el `main` debería ser: `String[] args` y NO: ~~`String args[]`~~

1.1.3. Nombres de identificadores

Para los identificadores podemos usar las letras anglosajonas y números de la tabla ASCII. No se debe usar caracteres con tilde ni la (ñ). Las barras bajas o guiones tampoco se usan. **Los nombres de los identificadores deben ser siempre lo más descriptivos posible, ya sea variable, método o clase**. Solo se usan identificadores de un solo carácter para representar los contadores del bucle for, y comienzan en la letra *i*.

- Nombre de **Package**: siempre en minúsculas.
- Nombre de las **clases o interfaces**: *UpperCamelCase*.
- Nombre de los **métodos**: *lowerCamelCase*. Suelen ser verbos o frases.

- Nombres de **constantes**: *CONSTANT_CASE*. Todo el mayúsculas, separando con barra baja.
- **Variables locales, atributos de la clase, nombres de parámetros**: *lowerCamelCase*.

1.1.4. Magic Numbers

Se conoce bajo este nombre a cualquier valor literal ("texto" o numérico) empleado en el código sin ninguna explicación. Se deben sustituir siempre que se pueda por una constante que identifique su finalidad.

```
1  //incorrecto
2  int precioConIva = precioBase + (0.21 * precioBase);
3
4  //correcto
5  //Se define en la clase
6  final static double IVA = 0.21;
7  //Se utiliza en un método
8  int precioConIva = precioBase + (IVA * precioBase);
```

1.1.5. Estructura del código

- Debemos usar la codificación **UTF-8**
- En las sentencias de control de flujo (`if`, `else`, `for`, `do-while`, `try-catch-finally`) se incluyen llaves { }, incluso si no contienen código o es una sola instrucción. Se alinean las llaves {} al inicio de línea.

```
1  //correcto
2  if (final < indice) {
3      filaInicial = indice - numeroFilas;
4  } else if (indice < filaInicial) {
5      filaInicial = indice;
6  }
7
8  //incorrecto
9  if (final < indice)
10     filaInicial = indice - numeroFilas;
11 else if (indice < filaInicial)
12     filaInicial = indice;
```

- Una sola instrucción por linea.
- Las líneas de código no deben superar los 100 caracteres. Si no, se deben **romper** antes de algún operador.
- Si la declaración del método es demasiado larga, o una expresión aritmética es demasiado larga, o en una sentencia *if*, debo romper.
- Si una operación aritmética o lógica se compone de distintos tipos de operaciones con distinta jerarquía, se deben usar paréntesis para facilitar su legibilidad.

```
public void ejecutarAccion(
    TipoParametro parametro1, TipoParametro parametro2, TipoParametro parametro3){
...
}

if ((condicion1 && condicion2)
      || (condicion3 && condicion4)
      || !(condicion5 && condicion6)) {
  llamarMetodo();
}

longName1 = longName2 * (longName3 + longName4 - longName5)
+ (4 * longname6);            //Siempre con el operador al principio de línea
```

- Los espacios en blanco mejoran la legibilidad. Se deben colocar entre operadores, después de los puntos y coma de los bucles for, después de los operadores de asignación, etc.

```
cantidadTotal = cantidadInicial + cantidadFinal;

for(int i = 0; i < cantidadTotal; i++){
...
}

public String getItem(int fila, int columna) {
...
}

getItem(cantidadInicial, cantidadFinal);
```

Debemos estar familiarizados y poner en práctica las convenciones recogidas en alguna de las guías de estilo indicadas.

1.2. Bad Smells

Se conoce como **Bad Smell o Code Smell** (mal olor) a algunos indicadores o sintomas del código que posiblemente ocultan un problema más profundo. Los *bad smells* no son errores de código, bugs, ya que no impiden que el programa funcione correctamente, pero son indicadores de fallos en el diseño del código que dificultan el posterior mantenimiento del mismo y aumentan el riesgo de errores futuros. Algunos de estos síntomas son:

- **Código duplicado** (*Duplicated code*). Si se detectan bloques de código iguales o muy parecidos en distintas partes del programa, se debe extraer creando un método para unificarlo.
- **Métodos muy largos** (*Long Method*). Los método de muchas lineas dificultan su comprensión. Un método largo probablemente está realizando distintas tareas, que se podrían dividir en otros métodos. Las funciones deben ser los más pequeñas posibles (3 lineas mejor que 15). Cuanto más corto es un método, más fácil es reutilizarlo. *Un método debe hacer solo una cosa, hacerla bien, y que sea la única que haga.*
- **Clases muy grandes** (*Large class*). Problema anterior aplicado a una clase. Una clase debe tener solo una finalidad. Si una clase se usa para distintos problemas tendremos clases con demasiados métodos, atributos e incluso instancias. Las clases deben el menor numero de responsabilidades y que esté bien delimitado.

- **Lista de parámetros extensa** (*Long parameter list*). Las funciones deben tener el mínimo número de parámetros posible, siendo 0 lo perfecto. Si un método requiere muchos parámetros puede que sea necesario crear una clase con esa cantidad de datos y pasarle un objeto de la clase como parámetro. Del mismo modo ocurre con el valor de retorno, si necesito devolver más de un dato.
- **Cambio divergente** (*Divergent change*). Si una clase necesita ser modificada a menudo y por razones muy distintas, puede que la clase esté realizando demasiadas tareas. Podría ser eliminada y/o dividida.
- **Cirugía a tiros** (*Shotgun surgery*). Si al modificar una clase, se necesitan modificar otras clases o elementos ajenos a ella para compatibilizar el cambio. Lo opuesto al *smell* anterior.
- **Envidia de funcionalidad** (*Feature Envy*). Ocurre cuando una clase usa más métodos de otra clase, o un método usa más datos de otra clase, que de la propia.
- **Legado rechazado** (*Refused bequest*). Cuando una subclase extiende (hereda) de otra clase, y utiliza pocas características de la superclase, puede que haya un error en la jerarquía de clases.

En la siguiente página web tenemos la mayoría de Bad Smells agrupados en 5 tipos.

1.3. Buenas prácticas

- **Manejo de Strings**: Los Strings son objetos, por lo que crearlos es costoso. Es mucho más rápido instanciarlos con una asignación, que con el operador *new*.
    - Concatenar `String` con el operador '+' también genera mucha carga, ya que crea un nuevo `String` en memoria (Los objetos `String` son inmutables). Se debe tratar de evitar siempre las concatenaciones (+) dentro de un **bucle**, o usar otras clases en ese caso (p.e. `StringBuilder`)

```
//instanciación lenta
String lenta = new String("objeto string");

//instanciación rápida
String rapida = "objeto string";
```

- **Tipos primitivos mejor que clases wrapper (envoltorio)** : Las clases wrapper al ser objetos, proveen de métodos para trabajar mejor con ellas, pero al igual que los Strings, son más lentos que los tipos primitivos.
- **Comparación de objetos**: Recordar que tanto los `Strings` como las tipos `Wrapper` son objetos y sus variables solo contienen sus referencias (direcciones). **Los objetos no se comparan con ==**.
- Evitar la creación innecesaria de objetos. Como se ha dicho, generan mucha carga.

```java
int x = 10;
int y = 10;

Integer x1 = new Integer(10);
Integer y1 = new Integer(10);

String x2="hola";
String y2 = new String("hola");

System.out.println(x == y);    //TRUE
System.out.println(x1 == y1);  //FALSE, ya que son 2 objetos distintos
System.out.println(x2 == y2);  //FALSE, ya que son 2 objetos distintos
```

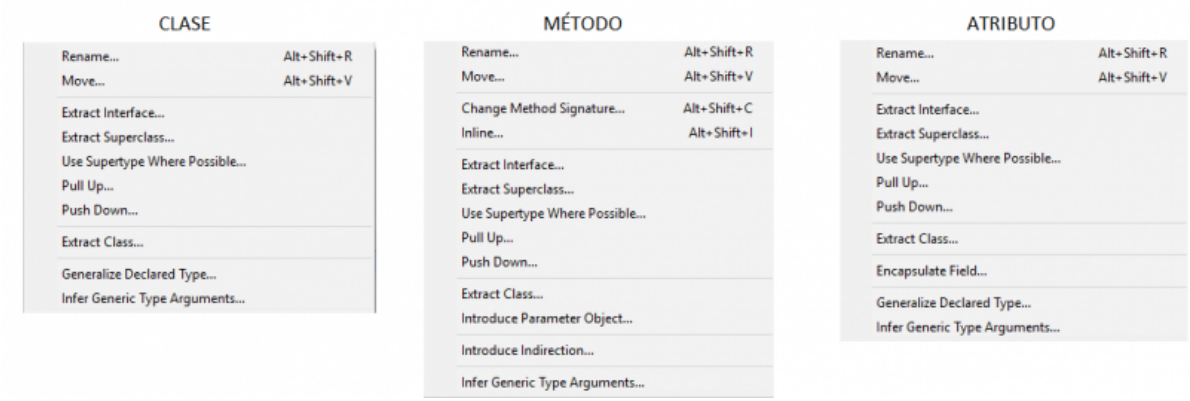
- **Visibilidad de atributos**: Los campos de una clase 'estándar' no deben declararse nunca como public, ni mucho menos no indicarle un modificador de visibilidad. Se usan sus *setters y getters* para su acceso.
- **Limitar siempre el alcance de una variable local**. Crear la variable local e inicializarla lo más cerca posible de su uso.
- **Usar siempre una variable para un único propósito**. A veces sentimos la tentación de reutilizar una variable, pero complica la legibilidad.

```java
...
int resultadoTotal = resultadoInicial - resultadoFinal;
...
```

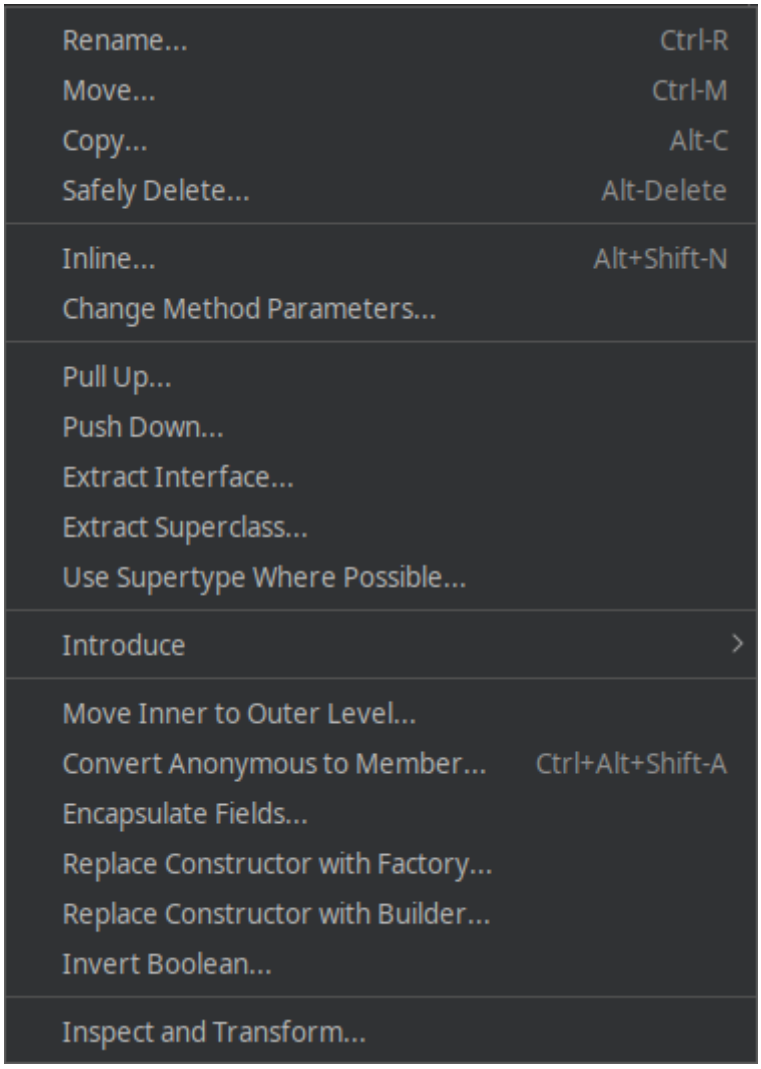
- **Bucle for**. Optar por el `for` siempre que se pueda (frente a `while`, `do-while`). Las ventajas son que reúne todo el control del bucle en la misma linea (inicio, fin, e incremento), y la variable de control ('`i`') no es accesible desde fuera de él. Si se necesita modificar su variable de control, usar otro bucle.
- **Constantes**: Cualquier valor literal debe ser definido como constante, excepto 1, -1, 0 ó 2 que son usados por el bucle for.
- **Switch**: Siempre debe llevar un `break` despues de cada caso, y tambien el caso *default* que ayudará a corregir futuros aumentos del número de casos.
- El *copiado defensivo* es salvador. Cuando creamos un constructor que recibe el mismo tipo de objeto de la clase, debemos tener cuidado y crear un nuevo objeto a partir del recibido.

1.4. Refactorización de Eclipse y NetBeans

Eclipse tiene distintos métodos de refactorización. Dependiendo de sobre qué mostremos el menú de refactorización, nos ofrecerá unas u otras opciones. Para refactorizar pulsaremos click derecho sobre el nombre del elemento deseado, y deplegaremos la opción **Refactor** del menu contextual.



NetBeans no muestra diferentes opciones, solo debemos hacer click derecho sobre cualquier zona del código y elegir el menú **Refactor**:



Los **métodos de refactorización**, también llamados **patrones de refactorización**, nos permiten plantear casos y previsualizar las posibles soluciones que se nos ofrecen. Podemos seleccionar diferentes elementos para mostrar su menú de refactorización (una clase, una variable, método, bloque de instrucciones, expresion, etc). A continuación se muestras algunos de los métodos más comunes:

- **Rename**: Es la opción empleada para cambiar el identificador a cualquier elemento (nombre de variable, clase, método, paquete, directorio, etc). Cuando lo aplicamos, se cambian todas las veces que aparece dicho identificador.
- **Move**: Mueve una clase (fichero .java)de un paquete a otro y se cambian todas las referencias. También se realiza la misma operación arrastrando la clase de un paquete a otro en el explorador de eclipse.

- **Extract Constant**: Convierte un número o cadena literal en una constante. Se puede ver donde se realizarán los cambios, y también el estado antes y después de refactorizar. Después, todas las apariciones de esa cadena se sustituyen por el nombre de la constante. Esto se utiliza modificar el valor en un solo lugar.
- **Extract Local Variable**: Convierte un número o cadena literal en una variable de ámbito local. Si esa misma cadena de texto existe fuera del bloque o del método, no se aplica el cambio. Parecido al patrón anterior, pero para aplicar dentro de método o bloques de código entre llaves { }.
- **Convert Local Variable to Field**: Convierte una variable local en un atributo privado de la clase. Después de aplicar el patrón de refactorización, todos los usos de la variable local se sustituyen por el atributo.
- **Extract Method**: Convierte un bloque de código en un método, a partir de un bloque cerrado por llaves { }. Eclipse ajusta las parámetros y el retorno del método. Es muy útil cuando detectamos *bad smells* en métodos muy largos, o en bloques de código que se repiten.
- **Change Method Signature**: Permite cambiar el nombre del método y los parámetros que recibe. Se actualizarán todas las dependencias y llamadas al método dentro del proyecto actual.
- **Inline**: Nos permite ajustar una referencia a una variable o método en una sola linea de código.

```
File fichero = new File("datos.dat");
PrintWriter escritor;
escritor = new PrintWriter(fichero);
PrintWriter escritor = new PrintWriter(new File("datos.dat"));
```

- **Extract Interface**: Este patrón de refactorización nos permite seleccionar los métodos de una clase para crear una *Interface*. Una *Interface* es una plantilla que define los métodos acerca de lo que puede hacer una clase. Define los métodos de una clase (Nombre, parámetros y tipo de retorno) pero no los desarrolla.
- **Extract Superclass**: Permite crear una superclase (clase padre) con los métodos y atributos que seleccionemos de una clase concreta. Lo usamos cuando la clase con la que trabajamos podría tener cosas en común con otras clases, las cuales serían también subclases de la superclase creada.

Eclipse también nos permite ver un histórico de la refactorización que se ha hecho en un proyecto, abriendo el menu **Refactor** → **History**.

2. Herramientas para el control de versiones

2.1. Introducción

Se llama control de versiones a la gestión de los diversos cambios que se realizan sobre los elementos de algún producto o una configuración del mismo.

Una versión, revisión o edición de un producto, es el estado en el que se encuentra dicho producto en un momento dado de su desarrollo o modificación. Los sistemas de control de versiones facilitan la administración de las distintas versiones de cada producto desarrollado, así como las posibles especializaciones realizadas (por ejemplo, para algún cliente específico).

El control de versiones se realiza principalmente en la industria informática para controlar las distintas versiones del código fuente. Sin embargo, los mismos conceptos son aplicables a otros ámbitos como documentos, imágenes, sitios web, etc.

Aunque un sistema de control de versiones puede realizarse de forma manual, es muy aconsejable disponer de herramientas que faciliten esta gestión
(CVS(concurrent versions system), Subversion, SourceSafe, ClearCase, Darcs, Bazaar , Plastic SCM, Git, Mercurial, Perforce...).

2.1.1. ¿Qué debe proporcionar?

Un sistema de control de versiones debe proporcionar:

- Mecanismo de almacenamiento de los elementos que deba gestionar (ej. archivos de texto, imágenes, documentación...).
- Posibilidad de realizar cambios sobre los elementos almacenados (ej. modificaciones parciales, añadir, borrar, renombrar o mover elementos).
- Registro histórico de las acciones realizadas con cada elemento o conjunto de elementos (normalmente pudiendo volver o extraer un estado anterior del producto).

Aunque no es estrictamente necesario, suele ser muy útil la generación de informes con los cambios introducidos entre dos versiones, informes de estado, marcado con nombre identificativo de la versión de un conjunto de ficheros, etc.

2.1.2. Clasificación

La principal clasificación que se puede establecer está basada en el almacenamiento del código:

- **Centralizados**: existe un repositorio centralizado de todo el código, del cual es responsable un único usuario (o conjunto de ellos). Se facilitan
  las tareas administrativas a cambio de reducir flexibilidad, pues todas las decisiones fuertes (como crear una nueva rama) necesitan la aprobación
  del responsable. Algunos ejemplos son CVS y Subversion.
- **Distribuidos**: cada usuario tiene su propio repositorio. No es necesario tomar decisiones centralizadamente. Los distintos repositorios pueden intercambiar y mezclar revisiones entre ellos. Ejemplos: Git y Mercurial.

2.1.3. Funcionamiento

Todos los sistemas de control de versiones se basan en disponer de un repositorio, que es el conjunto de información gestionada por el sistema. Este
repositorio contiene el historial de versiones de todos los elementos gestionados. Cada uno de los usuarios puede crearse una copia local duplicando el contenido del repositorio para permitir su uso. Es posible duplicar la última versión o cualquier versión almacenada en el historial. Este proceso se suele conocer como check out o desproteger. Para modificar la copia local existen dos semánticas básicas:

- **Exclusivos**: para poder realizar un cambio es necesario marcar en el repositorio el elemento que se desea modificar y el sistema se encargará de impedir que otro usuario pueda modificar dicho elemento.
- **Colaborativos**: en el que cada usuario se descarga la copia, la modifica, y el sistema automáticamente combina las diversas modificaciones. El principal problema es la posible aparición de conflictos que deban ser solucionados manualmente o las posibles inconsistencias que surjan al modificar el mismo fichero por varias personas no coordinadas. Además, esta semántica no es apropiada para ficheros binarios.

Tras realizar la modificación es necesario actualizar el repositorio con los cambios realizados. Habitualmente este proceso se denomina publicar, commit, check in o proteger.

2.1.4. Vocabulario común

La terminología empleada puede variar de sistema a sistema, pero a continuación se describen algunos términos de uso común.

Repositorio: El repositorio es el lugar en el que se almacenan los datos actualizados e históricos, a menudo en un servidor. A veces se le denomina depósito o depot. Puede ser un sistema de archivos en un disco duro, un banco de datos, etc..

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

Rotular ("tag"): Darle a alguna versión de cada uno de los ficheros del módulo en desarrollo en un momento preciso un nombre común ("etiqueta" o "rótulo") para asegurarse de reencontrar ese estado de desarrollo posteriormente bajo ese nombre. En la práctica se rotula a todos los archivos en un momento determinado. Para eso el módulo se "congela" durante el rotulado para imponer una versión coherente. Pero bajo ciertas circunstancias puede ser necesario utilizar versiones de algunos ficheros que no coinciden temporalmente con las de los otros ficheros del módulo.

Revisión ("version"): Una revisión es una versión determinada de un archivo.

Línea base ("Baseline"): Una revisión aprobada de un documento o fichero fuente, a partir del cual se pueden realizar cambios subsiguientes.

Abrir rama ("branch") o ramificar: Un módulo puede ser branched o bifurcado en un instante de tiempo de forma que, desde ese momento en adelante, dos copias de esos ficheros puedan ser desarrolladas a diferentes velocidades o de diferentes formas, de modo independiente. El módulo tiene entonces 2 (o más) "ramas". La ventaja es que se puede hacer un "merge" de las modificaciones de ambas ramas, posibilitando la creación de "ramas de prueba" que contegan código para evaluación, si se dedice que las modificaciones realizadas en la "rama de prueba" sean preservadas, se hace un "merge" con la rama principal.

Desplegar ("Check-out") ("checkout", "co"): Un despliegue crea una copia de trabajo local desde el repositorio. Se puede especificar una revisión concreta, y por defecto se suele obtener la última.

"Publicar" o "Enviar"("commit", "check-in", "ci", "install", "submit"): Un commit sucede cuando una copia de los cambios hechos a una copia
local es escrita o integrada sobre repositorio.

Conflicto: Un conflicto ocurre en las siguientes circunstancias:

1. Los usuarios X e Y despliegan versiones del archivo A en que las líneas n1 hasta n2 son comunes.
2. El usuario X envía cambios entre las líneas n1 y n2 al archivo A.
3. El usuario Y no actualiza el archivo A tras el envío del usuario X.
4. El usuario Y realiza cambios entre las líneas n1 y n2.
5. El usuario Y intenta posteriormente enviar esos cambios al archivo A.

   El sistema es incapaz de fusionar los cambios. El usuario Y debe resolver el conflicto combinando los cambios, o eligiendo uno de ellos para descartar el otro.

Resolver: El acto de la intervención del usuario para atender un conflicto entre diferentes cambios al mismo documento.

Cambio ("change", "diff", "delta"): Un cambio representa una modificación específica a un documento bajo control de versiones. La granularidad de la modificación considerada un cambio varía entre diferentes sistemas de control de versiones.

Lista de cambios ("changelist", "change set", "patch"): En muchos sistemas de control de versiones con commits multi-cambio atómicos, una lista de cambios identifica el conjunto de cambios hechos en un único commit. Esto también puede representar una vista secuencial del código fuente, permitiendo que el fuente sea examinado a partir de cualquier identificador de lista de cambios particular.

Exportación ("export"): Una exportación es similar a un check-out, salvo porque crea un árbol de directorios limpio sin los metadatos de control de versiones presentes en la copia de trabajo. Se utiliza a menudo de forma previa a la publicación de los contenidos.

Importación ("import"): Una importación es la acción de copia un árbol de directorios local (que no es en ese momento una copia de trabajo) en el repositorio por primera vez.

Integración o fusión ("merge"): Una integración o fusión une dos conjuntos de cambios sobre un fichero o un conjunto de ficheros en una revisión unificada de dicho fichero o ficheros.

- Esto puede suceder cuando un usuario, trabajando en esos ficheros, actualiza su copia local con los cambios realizados, y añadidos al repositorio, por otros usuarios. Análogamente, este mismo proceso puede ocurrir en el repositorio cuando un usuario intenta check-in sus cambios.
- Puede suceder después de que el código haya sido branched, y un problema anterior al branching sea arreglado en una rama, y se necesite incorporar dicho arreglo en la otra.
- Puede suceder después de que los ficheros hayan sido branched, desarrollados de forma independiente por un tiempo, y que entonces se haya requerido que fueran fundidos de nuevo en un único trunk unificado.

Integración inversa: El proceso de fundir ramas de diferentes equipos en el trunk principal del sistema de versiones.

Actualización ("sync" ó "update"): Una actualización integra los cambios que han sido hechos en el repositorio (por ejemplo por otras personas) en la copia de trabajo local.

Copia de trabajo: La copia de trabajo es la copia local de los ficheros de un repositorio, en un momento del tiempo o revisión específicos. Todo el trabajo realizado sobre los ficheros en un repositorio se realiza inicialmente sobre una copia de trabajo, de ahí su nombre. Conceptualmente, es un cajón de arena o sandbox.

Congelar: congelar significa permitir los últimos cambios (commits) para solucionar las fallas a resolver en una entrega (release) y suspender cualquier otro cambio antes de una entrega, con el fin de obtener una versión consistente. Si no se congela el repositorio, un desarrollador podría comenzar a resolver una falla cuya resolución no esta prevista y cuya solución dé lugar a efectos colaterales imprevistos.

2.2. Repositorio

Un repositorio es básicamente un servidor de archivos típico, con una gran diferencia: lo que hace a los repositorios especiales en comparación con esos
servidores de archivos es que recuerdan todos los cambios que alguna vez se hayan escrito en ellos, de este modo, cada vez que actualizamos el repositorio, éste recuerda cada cambio realizado en el archivo o estructura de directorios. Además, permite establecer información adicional para cada actualización, pudiendo tener un ejemplo un changelog de las versiones en el propio repositorio.

Cada herramienta de control de versiones tiene su propio repositorio, y, por desgracia, no son interoperables, es decir, no puedes obtener los datos del repositorio o actualizarlo si el repositorio y el control de versiones no coinciden. Realmente, al ser un servidor de archivos(especial, pero en esencia es lo
mismo), siempre se podría acceder directamente a los archivos almacenados en el repositorio y obtener el código sin mayor problema, pero no tendremos un control de versiones sobre dicho código hasta que lo asociemos a dicho repositorio con el control de versiones adecuado.

También podríamos asociar una copia de trabajo (la que se encentra en nuestro disco duro) a varios repositorios del mismo control de versiones siempre y cuando éste soporte replicación de repositorios; no obstante, si intentásemos realizar esa operación con repositorios de controles de versiones diferentes, es impredecible cómo responderá el sistema ante ello.

2.3. Herramientas de control de versiones.

Podemos encontrar una buena oferta de sistemas de control de versiones disponibles como software propietario y libre. Vemos algunos de los principales sistemas de control de versiones existentes en el mercado

CVS (concurrent versions system)
Era uno de los programas más utilizado en el mundo del software libre para el control de versiones de software. Está basado en el modelo cliente-servidor, existen versiones del programa para multitud de plataformas. Su solidez y su probada eficacia, tanto en grupos pequeños de usuarios como en grandes, le convirtió en la herramienta que utilizaban proyectos de software libre de éxito como Mozilla, OpenOffice.org, KDE o GNOME, entre otros.

RCS (revision control system)
El sistema RCS es extremadamente simple y fácil de instalar, pero no puede cubrir las necesidades de proyectos o equipos medianos. Entre las limitaciones más notables de RCS, destaca que sólo puede trabajar en un único directorio cuando la mayoría de proyectos tienen

múltiples directorios. RCS utiliza un sistema de bloqueo de archivos que impide a varios desarrolladores trabajar sobre el mismo archivo.

BitKeeper

BitKeeper es un producto propietario desarrollado por la empresa BitMover. Es probablemente el producto más sofisticado de su categoría. Entre las características que lo diferencian del resto de productos, destacan la posibilidad de trabajar con repositorios distribuidos y un sistema muy avanzando para integrar diferentes versiones de un mismo archivo.

Microsoft Visual Source Safe

Es uno de los productos más utilizados para desarrollo de aplicaciones Windows. Principalmente porque se integra en el entorno de trabajo de Visual Studio y con el resto de herramientas de desarrollo de Microsoft. Tiene funciones de comparación visual de archivos realmente avanzadas, en su modo básico de funcionamiento utiliza bloqueo de archivos.

Git

Es un software diseñado por Linus Torvalds, pensando en la eficiencia, la confiabilidad y compatibilidad del mantenimiento de versiones de aplicaciones cuando estas tienen un gran número de archivos de código fuente. Su propósito es llevar registro de los cambios en archivos de computadora incluyendo coordinar el trabajo que varias personas realizan sobre archivos compartidos en un repositorio de código. En cuanto a derechos de autor Git es un software libre distribuible bajo los términos de la versión 2 de la Licencia Pública General de GNU.

Mercurial

Esta herramienta funciona en Linux, Windows y Mac OS X, Es un programa de línea de comandos. Es una herramienta que permite que el desarrollo se haga distribuido, gestionando de forma robusta archivos de texto y binarios. Tiene capacidades avanzadas de ramificación e integración. Es una herramienta que incluye una interfaz web para su configuración y uso.

3. Fuentes de información

- Wikipedia
- Code&Coke (Fernando Valdeón)
- Apuntes IES El Grao (Mª Isabel Barquilla?)
- Apuntes IOC (Marcel García)
- Apuntes José Luis Comesaña
- Apuntes IES Luis Vélez de Guevara 17-18 (José Antonio Muñoz Jiménez)