

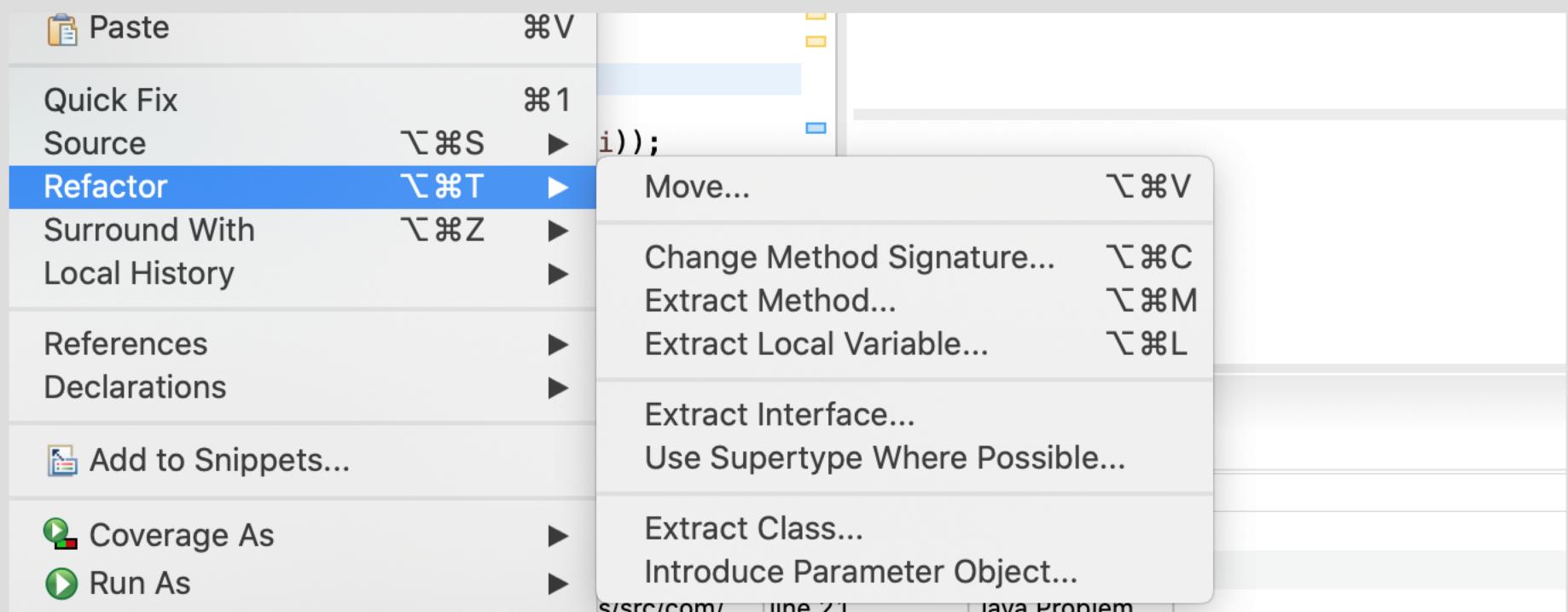
# **Optimización y documentación**

# Optimización y documentación

- Refactorización
  - Buenas prácticas
  - Refactorización con el IDE
  - Bad smells – malos olores
- SOLID
- Analizadores de código
- Control de versiones
- Documentación

# Refactorización

- Técnica de la ingeniería de software para reestructurar un código fuente, alterando su estructura interna sin cambiar su comportamiento externo.



# Refactorización

- Parte del mantenimiento del código que no arregla errores ni añade funcionalidad
  - Pero puede crear nuevos bugs si no se realiza correctamente
- Mantenimiento preventivo
  - Si no se refactoriza el software, con las continuas modificaciones se generarán bugs más rápidamente

# Refactorización

- Busca hacer el código más entendible y modificable
  - Siempre es un buen momento para refactorizar
- El software debe funcionar igual antes y después de cada refactorización
  - Comprobación de que no se han introducido bugs: tests

# Refactorización

- Limpieza de código
  - Refactorizar es tender a hacer el código más limpio
  - Eliminación de código duplicado
- Código limpio
  - “*Cualquier estúpido puede escribir un código que los ordenadores puedan entender. Son los buenos programadores los que escriben códigos que los humanos pueden entender*”. Martin Fowler

# Refactorización

- Buenas prácticas
  - Un fichero .java
    - Copyright (si necesario)
    - Declaración del package
    - Imports
    - Declaración de clase

# Refactorización

- Buenas prácticas
  - El nombre de las clases comienza en mayúsculas
  - Estructura de una clase
    - Atributos o campos
    - Constructores
      - Por orden creciente de número de argumentos
    - Métodos fábrica
    - Otros métodos

# Refactorización

- Buenas prácticas
  - Modificadores
    - 1) Modificador de acceso (public/protected/private)
    - 2) abstract
    - 3) static
    - 4) final
    - 5) default

# Refactorización

- Buenas prácticas. Llaves
  - Llave de apertura al final de la línea y no en otra línea por su cuenta.
  - Después de llave de cierre debe haber un salto de línea
  - Recomendadas incluso cuando son opcionales (*if - for* de una línea).

# Refactorización

- Buenas prácticas. Llaves.
  - Si uno de los bloques de un *if / else* tiene llaves, el otro bloque también debe llevarlas.
  - *else, catch* y *while* en un bucle *do...while* van en la misma línea que la llave que cierra el bloque precedente

# Refactorización

## Dos

```
void method() {  
    ...  
}
```

```
try {  
    something();  
} catch (AnException e) {  
    ...  
}
```

```
for (int[] row : matrix) {  
    for (int val : row) {  
        sum += val;  
    }  
}
```

# Refactorización

## Don'ts

```
// Wrong placement of opening brace
void method()
{
    ...
}
```

```
// Newline in front of catch should be avoided
try {
    something();
}
catch (AnException e) {
    ...
}
```

```
// Braces should be used
if (flag)
    // Restore x
    x = 1;
```

```
// Use braces if block comes last in enclosing block
// to avoid accidentally indenting the closing brace.
for (int[] row : matrix) {
    for (int val : row)
        sum += val;
}
```

# Refactorización

- Buenas prácticas
  - Tabulación vs Espacios
    - Una tabulación podrían ser un número diferentes de espacios dependiendo del entorno
    - Un espacio siempre es un único espacio/columna en un fichero
    - Se recomiendan 4 espacios en cada identación

# Refactorización

- Buenas prácticas

## Dos

```
switch (var) {  
    case TWO:  
        setChoice("two");  
        break;  
    case THREE:  
        setChoice("three");  
        break;  
    default:  
        throw new IllegalArgumentException();  
}
```

# Refactorización

- Buenas prácticas

## Don'ts

```
switch (var) {  
    case TWO:  
        setChoice("two");  
        break;  
    case THREE:  
        setChoice("three");  
        break;  
    default:  
        throw new IllegalArgumentException();  
}
```

# Refactorización

- Buenas prácticas. Variables
  - Nombrado en minúsculas
  - Una variable por declaración (y como máximo una línea por declaración)
  - Los corchetes de arrays deben estar en el tipo (`String[] args`) y no en la variable(`String args[]`).
  - Declarar una variable local justo antes de usarla e inicializarla lo más cerca posible a su declaración

# Refactorización



- Ejercicio
  - Dado el fichero java aportado por el profesor, realiza las correcciones necesarias teniendo en cuenta lo aprendido.

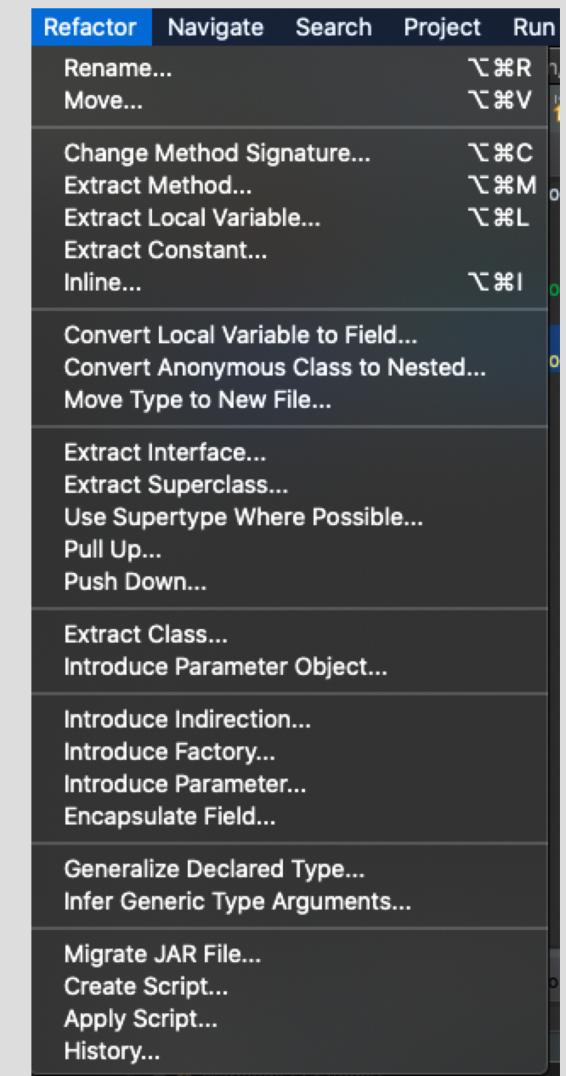
# Refactorización

- Ejercicio
  - Define:
    - Una enumeración MONTH que defina los meses del año con su nombre y su número de días
    - Una clase Year que tenga un método llamado getDays y reciba como argumento una variable de tipo MONTH. Retornará el número de días de ese mes



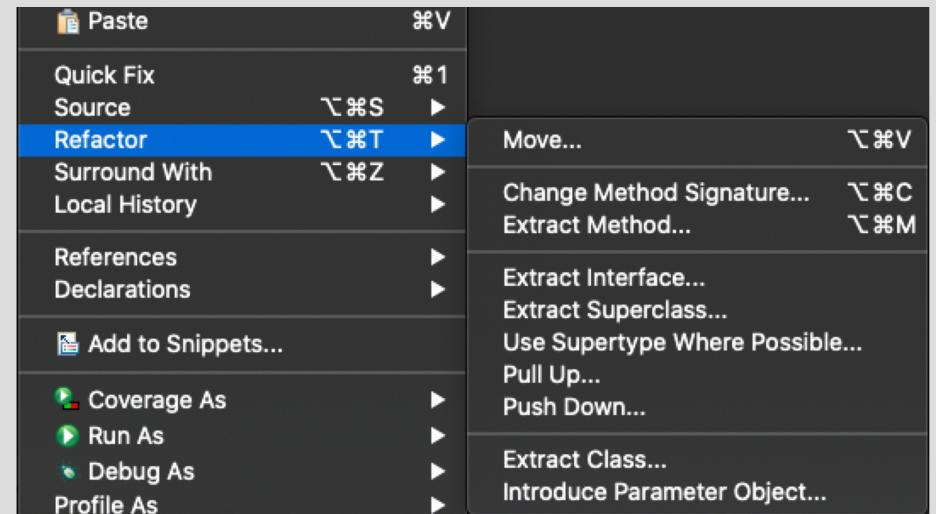
# Refactorización

- Otros patrones en Eclipse
  - Opción de menú “Refactor”
  - Seleccionar texto y opción “refactor” en el menú contextual



# Refactorización

- Extract Method
  - Crea un nuevo método a partir de la selección de las líneas de código
  - El fragmento de código debe poder agruparse
  - Se debe dar un nombre al nuevo método
  - Elección del tipo de retorno



# Refactorización

- Extract Method
  - Una forma habitual de extraer un método suele ser la condición de un if
  - Se busca que el código sea más legible

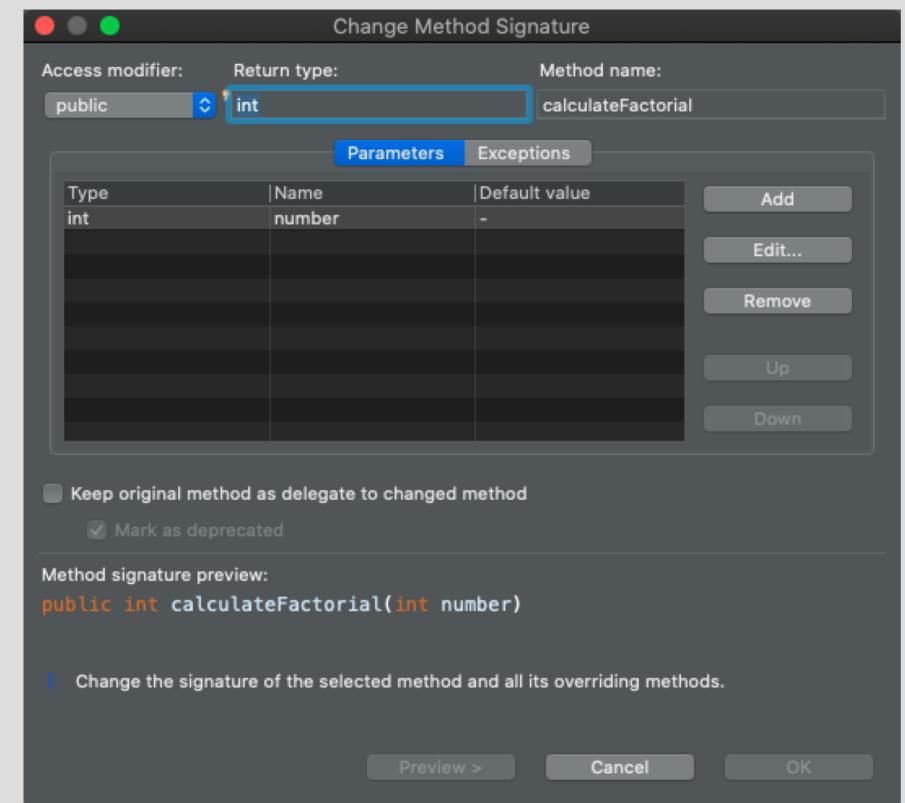
```
if (i > 0 && i%2 == 0) {  
    //do anything  
}
```

```
if (isEven(i)) {  
    //do anything  
}
```

```
boolean isEven(int number) {  
    return number > 0 && number%2 == 0;  
}
```

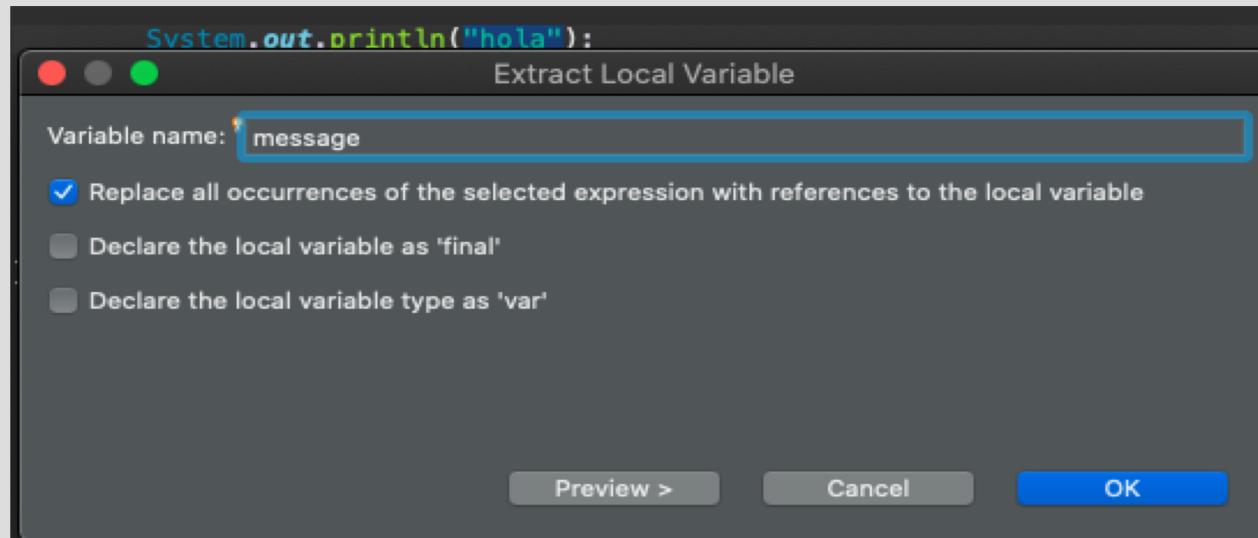
# Refactorización

- Change method signature
  - Cambia la firma de un método:  
*modifier tipo\_de\_retorno nombre\_método (argumentos)*
  - El cambio afectará a todas las referencias a ese método



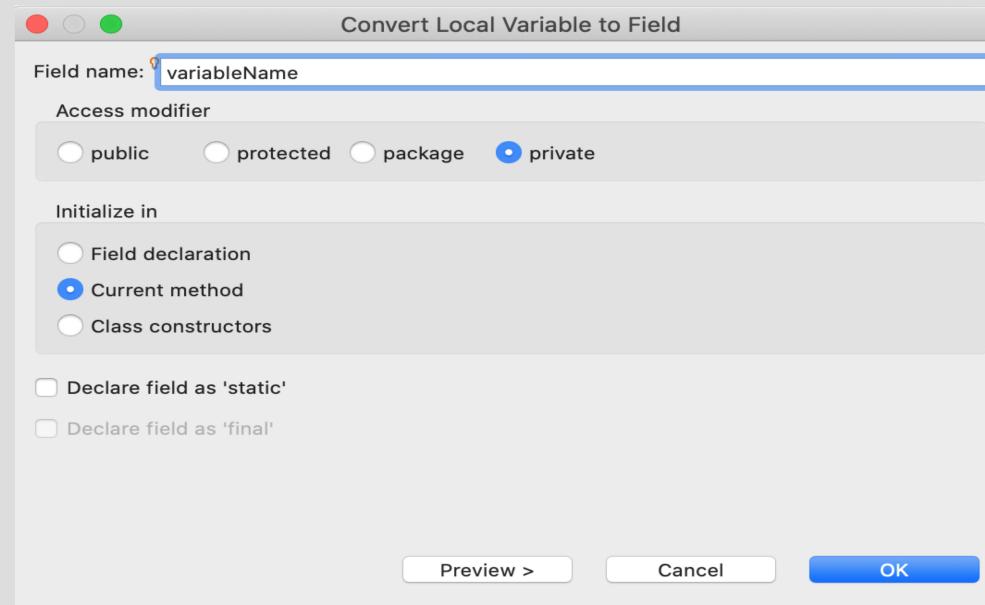
# Refactorización

- Extract local variable
  - Crea una nueva variable a partir de un valor mágico
  - Un valor mágico es aquel no asignado a una variable. Es escrito directamente en una fórmula o método como argumento



# Refactorización

- Convert local to field
  - Una variable utilizada en varios métodos de una misma clase se puede convertir en un atributo de la propia clase y ser accesible desde todos los métodos



# Refactorización

- Extraer interfaz
  - Genera una interfaz a partir de una clase con los métodos definidos en la clase origen
  - Se debe:
    - indicar el nombre de la interfaz que se va a crear,
    - Seleccionar los miembros que aparecerán en la intefaz

# Refactorización

- Bad Smells – Malos olores
  - **Método largo:** los métodos deberían ser cortos. Como medida estándar se puede tomar no codificar métodos más largos de 50 líneas
  - **Clases grandes:** similar a los métodos largos. Verificar si la clase realiza “demasiadas cosas”
  - **Lista de parámetros larga:** difíciles de mantener, aumentan el acoplamiento.

# Refactorización

- Bad Smells – Malos olores
  - **Agrupación condicional:** concatenación de varias estructuras if/else que complican la lectura del código
  - **Duplicación de código:** conjunto de líneas de código que se repiten en varias partes del programa
  - **Cambios en cadena:** el cambio en una clase obliga a cambiar otras consecutivamente.

# Refactorización

- Ejercicio
  - Descarga el ejercicio de refactorización del repositorio de la asignatura y realiza las modificaciones oportunas para mejorar el código y dejarlo lo más limpio posible
  - Ten en cuenta que tras cada acción de refactorización el/los tests deben seguir pasando correctamente



# Refactorización

- Ejercicio
  - Clase Evaluation
  - Fecha de entrega: martes 24 de marzo



# Optimización y documentación

- Refactorización
- **SOLID**
- Analizadores de código
- Control de versiones
- Documentación

# SOLID

- Conceptos básicos
  - **Acoplamiento:** grado de interdependencia que tienen dos unidades de software entre sí.
  - Se busca que haya bajo acoplamiento
  - Cuando dos unidades son independientes de la otra se dice que están desacopladas

# SOLID

- Conceptos básicos
  - **Cohesión:** grado en que elementos diferentes de un sistema permanecen unidos para alcanzar un mejor resultado que si trabajaran por separado
  - Agrupación de clases/componentes para realizar una función

# SOLID

- SOLID:
  - Single responsibility
  - Open-closed
  - Liskov substitution
  - Interface segregation
  - Dependency inversion
    - Robert C. Martin, uncle bob

# SOLID

- Principios básicos de la programación orientada a objetos y el diseño.
- Guías que pueden ser aplicadas en el desarrollo de software para eliminar código sucio
- Provoca refactorizaciones en el código fuente hasta que sea legible y extensible

# SOLID

- Single responsibility:
  - Principio de responsabilidad única
  - Un objeto, una acción
  - Aumenta la cohesión y disminuye el acoplamiento entre objetos
  - Una clase debería tener solo una razón para cambiar
  - Si una clase asume más de una responsabilidad, será más sensible al cambio.

# SOLID

- Open-close
  - abierta para su extensión, pero cerrada para su modificación
  - Basada en la herencia
  - Polimorfismo
    - Basado en clases abstractas o interfaces

# SOLID

- Liskov substitution
  - Los objetos de un programa deberían ser reemplazables por instancias de sus subtipos sin alterar el correcto funcionamiento del programa
  - Cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas.

# SOLID

- Interface Segregation Principle
  - los clientes de un programa dado sólo deberían conocer de éste aquellos métodos que realmente usan, y no aquellos que no necesitan usar
  - Mejor usar muchas interfaces pequeñas que una interfaz muy grande, con muchos métodos
  - Favorece bajo acoplamiento

# SOLID

- Dependency inversion principle
  - se debe “depender de abstracciones, no depender de implementaciones”
  - Inyección de dependencias (una dependencia es un objeto)
  - Las abstracciones no deberían depender de los detalles. Los detalles (implementaciones concretas) deben depender de abstracciones.

# Optimización y documentación

- Refactorización
- SOLID
- **Analizadores de código**
- Control de versiones
- Documentación

# Analizadores de código

- Permiten auditar/medir de una forma objetiva el estado del código del proyecto
- Las refactorizaciones están ligadas a los analizadores de código
- Los principios SOLID buscan mejorar el código

# Analizadores de código

- En todos los procesos de construcción de código se debería pasar por un analizador.
- Es labor del desarrollador comprobar y mejorar sus métricas (cobertura, complejidad ciclomática, ...)
- La más habitual es Sonar
  - SonarLint, plugin gratuito para eclipse

# Analizadores de código

- Otras herramientas
  - CheckStyle: pensada para Java
    - Plugin para Eclipse
  - PMD
    - Plugin para Eclipse
  - Kiuwan: Saas de pago
- Checkstyle y PMD son utilizados internamente por la plataforma de Sonarqube

# Analizadores de código

- Instalación del plugin de checkStyle
  - Desde el Marketplace dentro de Eclipse  
(Help -> Eclipse Marketplace...)
  - Buscar “Checkstyle” y hacer clic en Install
  - Concluir la instalación.
- Preferencias de configuración del plugin
  - Ejemplo de configuración
- Analizar una clase
  - Analiza la clase del ejercicio de Refactorización

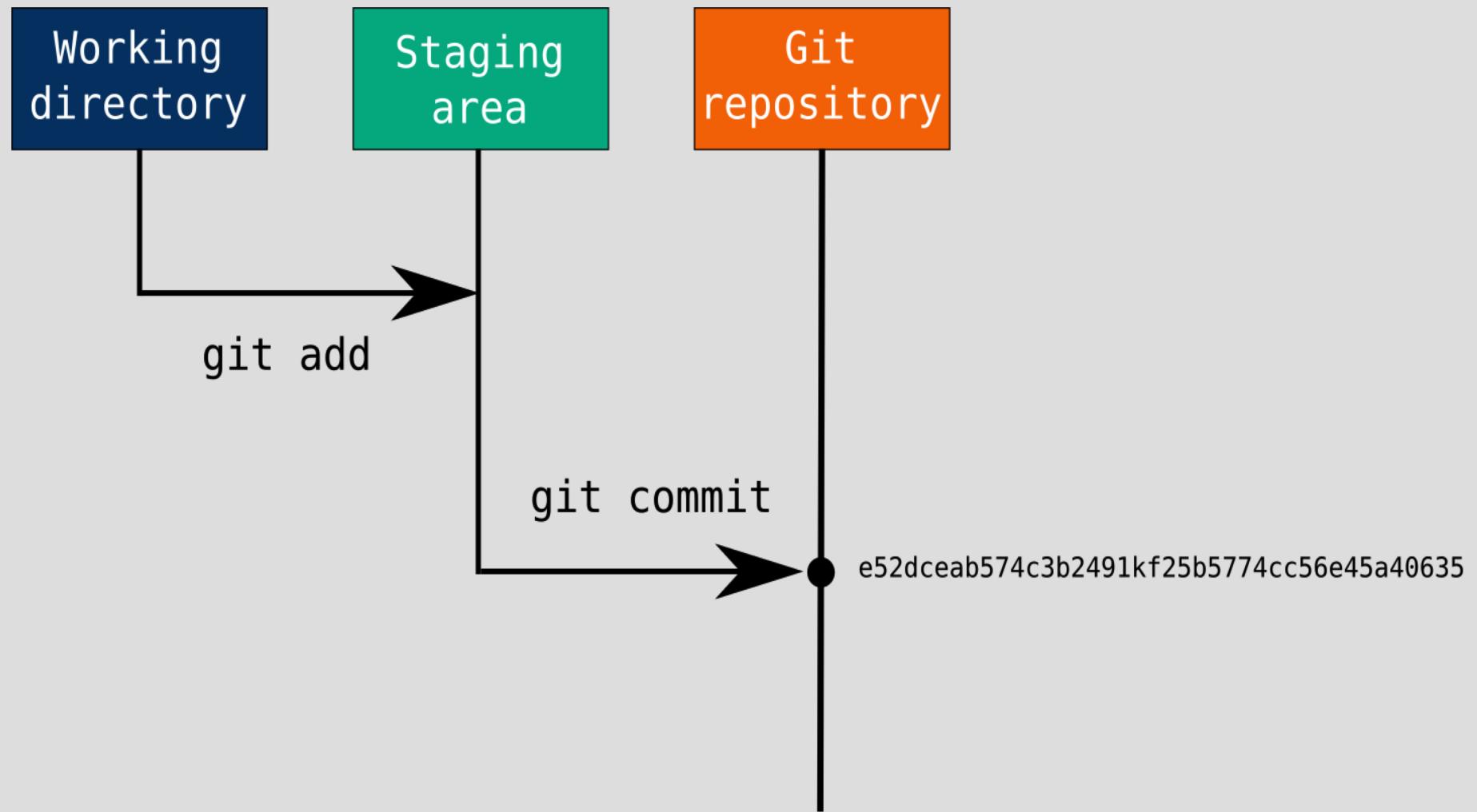
# Optimización y documentación

- Refactorización
- SOLID
- Analizadores de código
- **Control de versiones**
- Documentación

# Control de versiones

- Git es un sistema de control de versiones rápido, escalable y distribuido con un poco común conjunto de órdenes (commands)
- Remember Git
  - La guía fácil: [aquí](#)

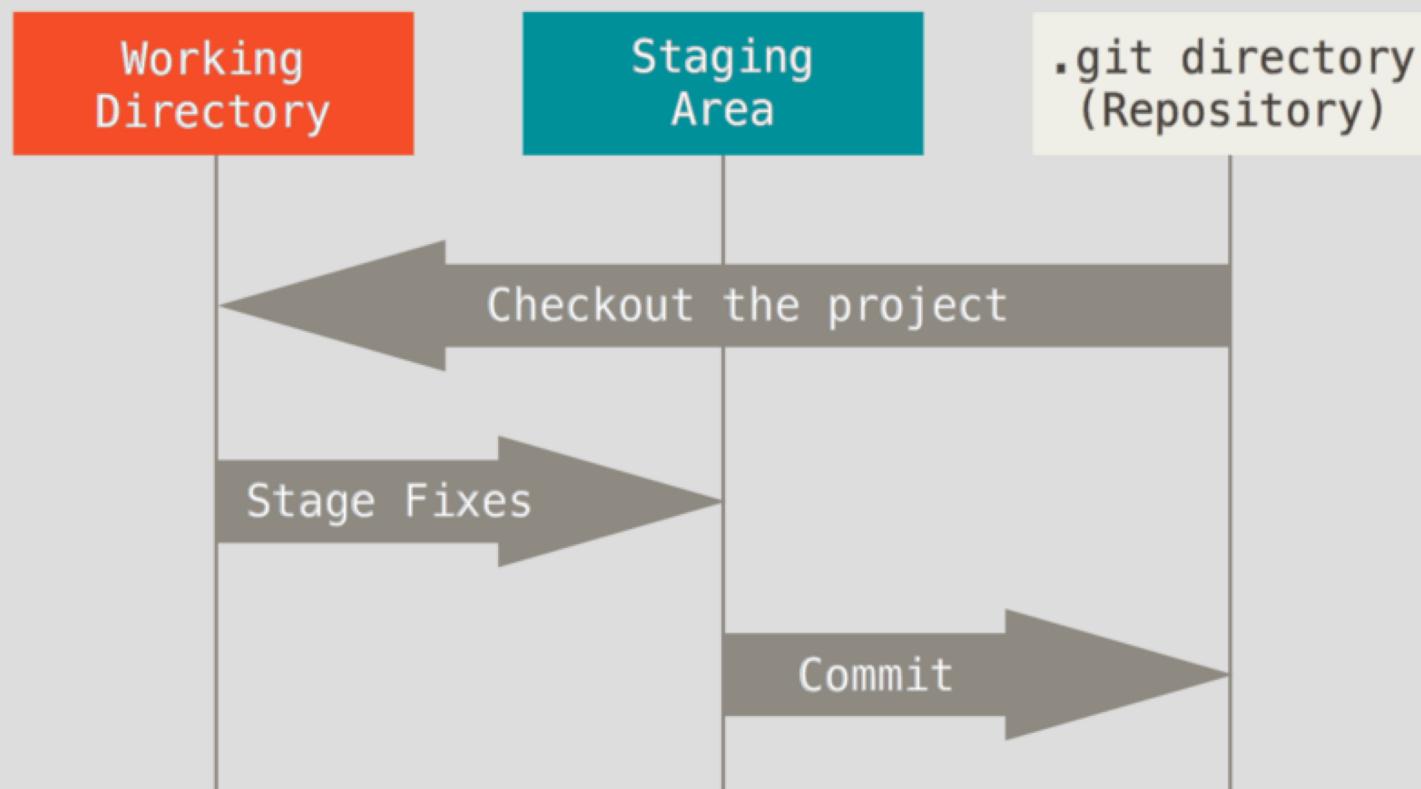
# Control de versiones



# Control de versiones

- El flujo básico en Git
  - El programador modifica ficheros en su proyecto
  - El programador selecciona qué ficheros pueden formar parte del siguiente commit y los añade al “staging area”.
  - El programador realiza el “commit” que tome los ficheros que se encuentran en el “staging area” y guarda un snapshot permanente en tu directorio Git

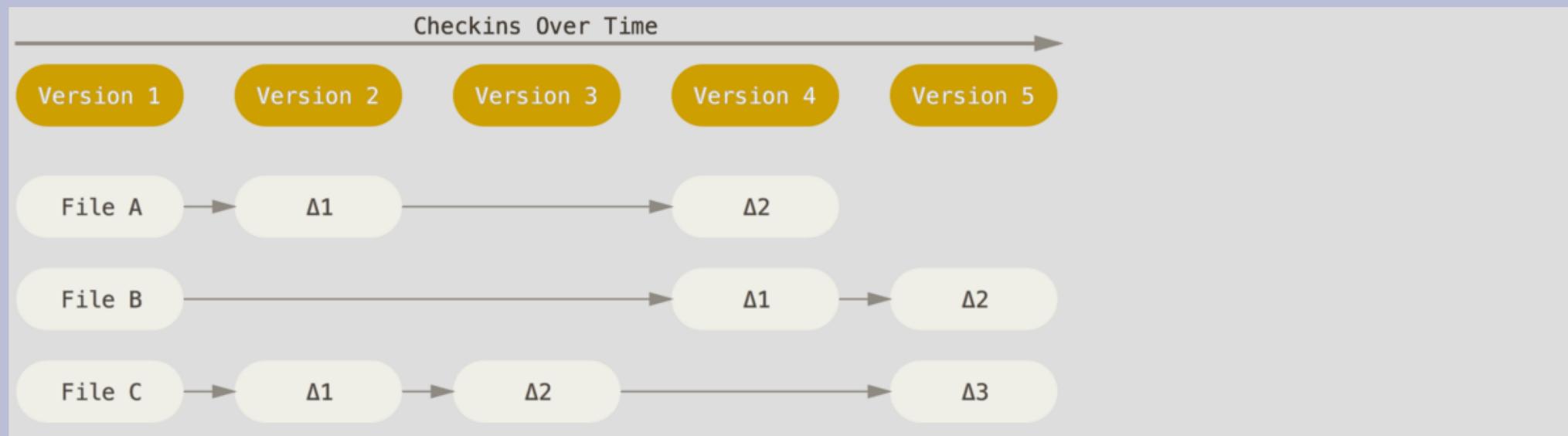
# Control de versiones



# Control de versiones

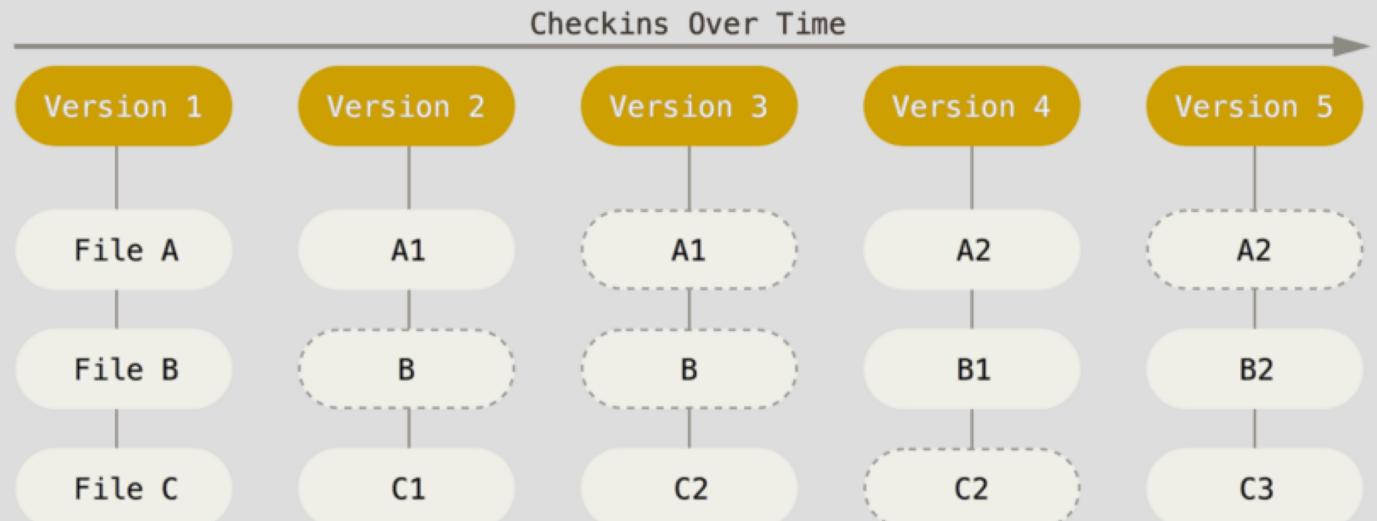
- Snapshot de git
  - Cada vez que realizamos “commit”, git guarda el estado de todos los ficheros: toma una foto del proyecto →SNAPSHOT
  - Cada commit (snapshot) es identificado con una cadena alfanumérica llamada *hash* o *identificador de commit*
  - Git trabaja como si los ficheros fueran un flujo de snapshots (un flujo de commits, de modificaciones)

# Control de versiones



SVN; Storing data as changes to a base version of each file.

GIT: Storing data as snapshots of the project over time.



# Control de versiones

- Git: tu identidad
  - Configurar git con tu nombre de usuario y tu dirección de correo electrónico.
    - Cada commit de git utiliza esta información y está ligada al mismo:

```
git config --global user.name "John Doe"
```

```
git config --global user.email johndoe@example.com
```

# Control de versiones

- Git: obtener ayuda
  - Git ofrece una ayuda a los usuarios explicando el funcionamiento de los comando

`git help`

- Para una acción concreta

`git help <action>`

`git help config`

# Control de versiones

- Git: comandos

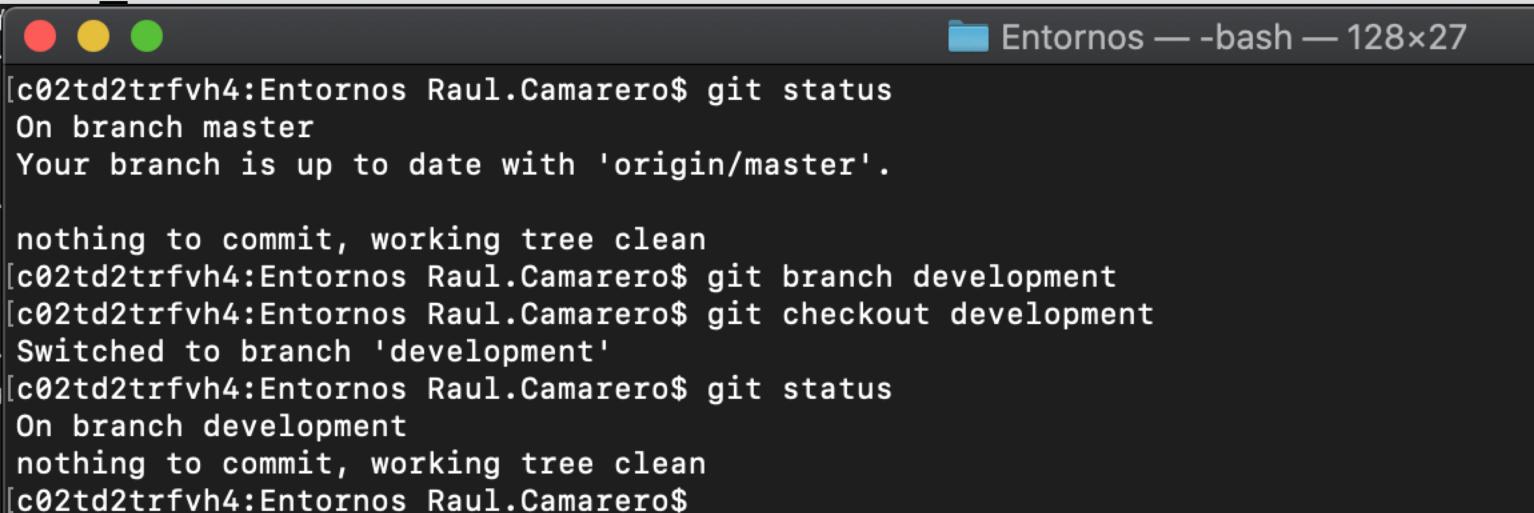
- *git clone <url\_repo>*: descarga el repositorio de la url
- *git status*: estado del repositorio actual
- *git fetch*: comprueba si hay cambios en el repo
- *git pull*: descarga los cambios del repo
- *git add <list\_files>*: añade los ficheros al staging area
- *git commit <list\_files> -m message*: añade list\_files al repositorio
- *git push*: empuja los últimos commits al repo remoto
- *git log*: lista los últimos commits
- *git diff <file>* : muestra los cambios de <file>
- *git branch <branch\_name>*: crea la rama branch\_name
- *git checkout <branch\_name>*: cambia a la rama branch\_name

# Control de versiones

- Git: ramas
  - Por defecto, rama *master*
  - Se crean ramas a partir de otra
    - Crear rama “development” a partir de master
  - Se suelen crear ramas por cada nueva funcionalidad
  - Cuando la funcionalidad está terminada, la rama creada se mezcla con la de origen para que contenga los cambios en el código de la nueva funcionalidad

# Control de versiones

- Git: ramas
  - Crea rama *development* a partir de *master*
  - Se cambia a la nueva rama *development*



```
[c02td2trfvh4:Entornos Raul.Camarero$ git status
On branch master
Your branch is up to date with 'origin/master'.

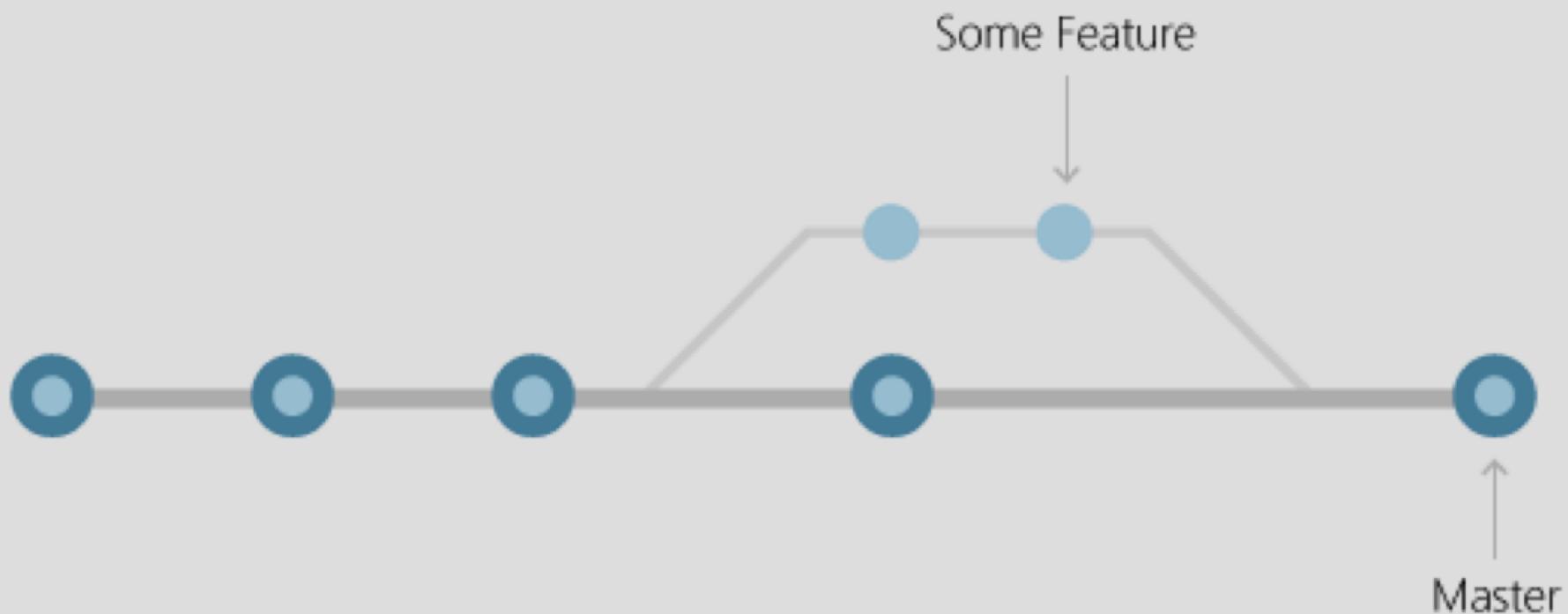
nothing to commit, working tree clean
[c02td2trfvh4:Entornos Raul.Camarero$ git branch development
[c02td2trfvh4:Entornos Raul.Camarero$ git checkout development
Switched to branch 'development'
[c02td2trfvh4:Entornos Raul.Camarero$ git status
On branch development
nothing to commit, working tree clean
[c02td2trfvh4:Entornos Raul.Camarero$
```

# Control de versiones

- Git: comandos
  - *git branch <branch\_name>*: crea la rama *branch\_name*
  - *git checkout <branch\_name>*: cambia a la rama *branch\_name*
  - *git merge <branch\_name>*: trae los cambios de *branch\_name* a la rama actual

# Control de versiones

- Git: ramas



# Control de versiones

- Ejercicio
  - 1. En tu repositorio local, desde master, crea una rama con el nombre que quieras (p.ej: *firstBranch*)
  - 2. Haz que tu repositorio local apunte a esa rama
  - 3. Crea un fichero y añádelo al `staging_area`
  - 4. Haz commit e inclúyelo a tu repo local
  - 5. Vuelve a tu rama inicial y haz merge.
  - 6. El fichero creado en el punto 3 deberá aparecer en tu rama master

# Optimización y documentación

- Refactorización
- SOLID
- Analizadores de código
- Control de versiones
- **Documentación**

# Documentación

- El objetivo de la documentación es clarificar el propósito de la clase
- Otros programadores podrán usar las clases
- API: interfaz de programación de aplicaciones
  - Conjunto de funciones, procedimientos que se publica para que pueda ser utilizado por otro software
- Javadoc: utilidad de Java para generar la documentación del API en formato HTML

# Documentación

- Oracle publica la documentación de todas las clases de cada versión de Java
  - Javadoc Java 8
- Para generar Javadoc se usan anotaciones (@author, @version, ..) o etiquetas (tags) de HTML.
- Se utiliza el comentario multilínea

# Documentación

- Los comentarios deben ser concretos y específicos
- Evitar redundancias y uso de expresiones que no añadan información
- Se escriben al principio de cada clase, miembro o método,
- Las variables de instancia o de clase no se suelen documentar a nivel de javadoc.

# Documentación

- Comentario de clase
  - Describir la responsabilidad de la clase
  - Añadir autor (opcional)
  - Referencias a
    - clases heredadas
    - Interfaces implementadas
    - subclases

# Documentación

- Comentario de clase

```
① /**
 * The Boolean class wraps a value of the primitive type
 * {@code boolean} in an object. An object of type
 * {@code Boolean} contains a single field whose type is
 * {@code boolean}.
 * <p>
 * In addition, this class provides many methods for
 * converting a {@code boolean} to a {@code String} and a
 * {@code String} to a {@code boolean}, as well as other
 * constants and methods useful when dealing with a
 * {@code boolean}.
 *
 * @author Arthur van Hoff
 * @since JDK1.0
 */
public final class Boolean implements java.io.Serializable,
    Comparable<Boolean>
{
    ...
}
```

# Documentación

- Comentario de método
  - Describir la intención del método
  - Valor de retorno
  - Descripción de parámetros
  - Excepciones (si las hay)
  - Referencias a otros métodos (si necesario)

# Documentación

- Comentario de método

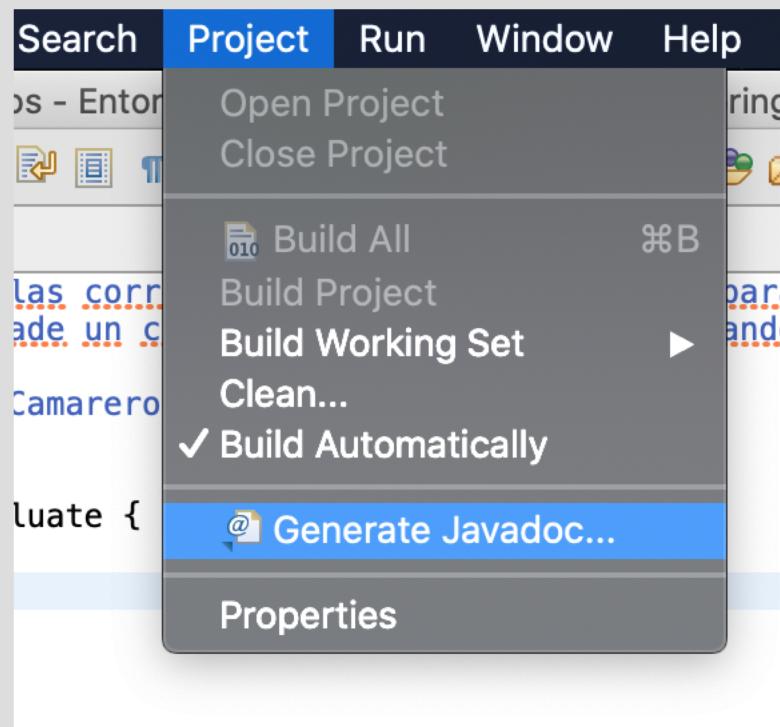
```
/*
 * Returns a {@code Boolean} with a value represented by the
 * specified string. The {@code Boolean} returned represents a
 * true value if the string argument is not {@code null}
 * and is equal, ignoring case, to the string {@code "true"}.
 *
 * @param s a string.
 * @return the {@code Boolean} value represented by the string.
 */
public static Boolean valueOf(String s) {
    return parseBoolean(s) ? TRUE : FALSE;
}
```

# Documentación

- Interfaces y herencia
  - Normalmente, el javadoc de una interfaz o superclase será heredado a las clases que la implementan/heredan
  - Tag: `@inheritDoc`
  - Se puede añadir alguna especificidad a la documentación heredada

# Documentación

- Javadoc en Eclipse



# Refactorización

- Ejercicio
  - Crea una clase llamada Vehicle con los métodos:
    - boolean accelerate(int kmh)
    - stop()
    - int powerCV(int kw)
  - Anádele la documentación pertinente y genera los ficheros javadoc en formato .html



# Bibliografía

- Wikipedia
- <http://cr.openjdk.java.net/~alundblad/styleguide/index-v6.html>