Introducción a git y gitHub

Control de versiones y repositorios distribuidos

¿Qué es el control de versiones...

y por qué debería importarme?

- Guardar un historial de los cambios que hago en mi trabajo
- Poder volver a un estado anterior de forma segura, coherente y fácil
- Ver las modificaciones entre diferentes versiones -o estados- del trabajo

—

Control de versiones

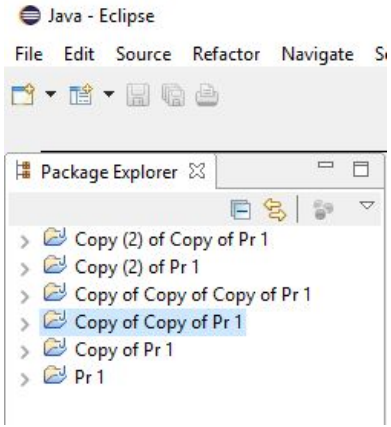
Para un proyecto Java:

- Cada vez que empiezo a trabajar en un cambio, me gustaría poder volver atrás
- O quizás no deshacer todos los cambios, sino ver que he cambiado

Control de versiones

Para un proyecto Java:

- Cada vez que empiezo a trabajar en un cambio, me gustaría poder volver atrás
- O quizás no deshacer todos los cambios, sino ver que he cambiado



Unos nombres muy descriptivos… Con suerte pondríamos fechas y alguna pista de los cambios.
Sin embargo… ¿qué archivos han cambiado?

Control de versiones

MiClaseSuperOriginal.java

```
1
2 public class MiClaseSuperOriginal {
3
4     public static void main(String[] args) {
5         System.out.println("Hola mundo!");
6     }
7
8 }
9
```

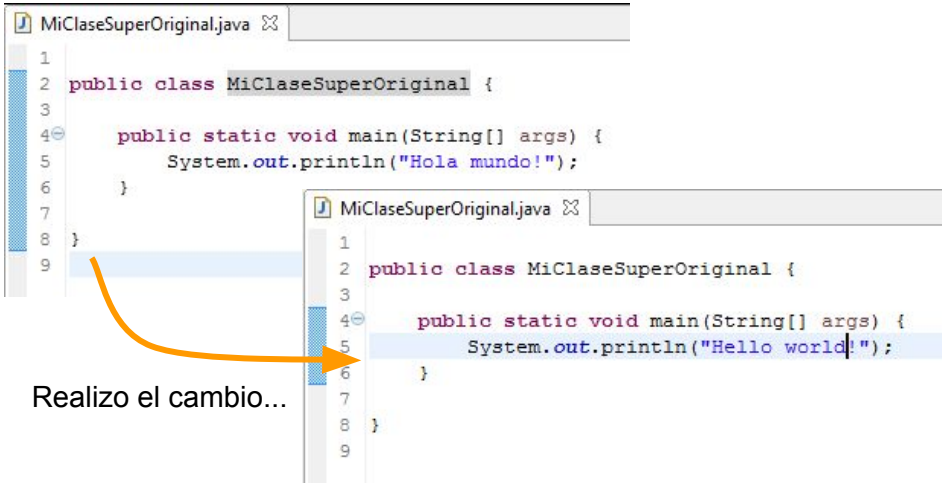
Para un proyecto Java:

- Cada vez que empiezo a trabajar en un cambio, me gustaría poder volver atrás
- O quizás no deshacer todos los cambios, sino ver que he cambiado

Control de versiones

Para un proyecto Java:

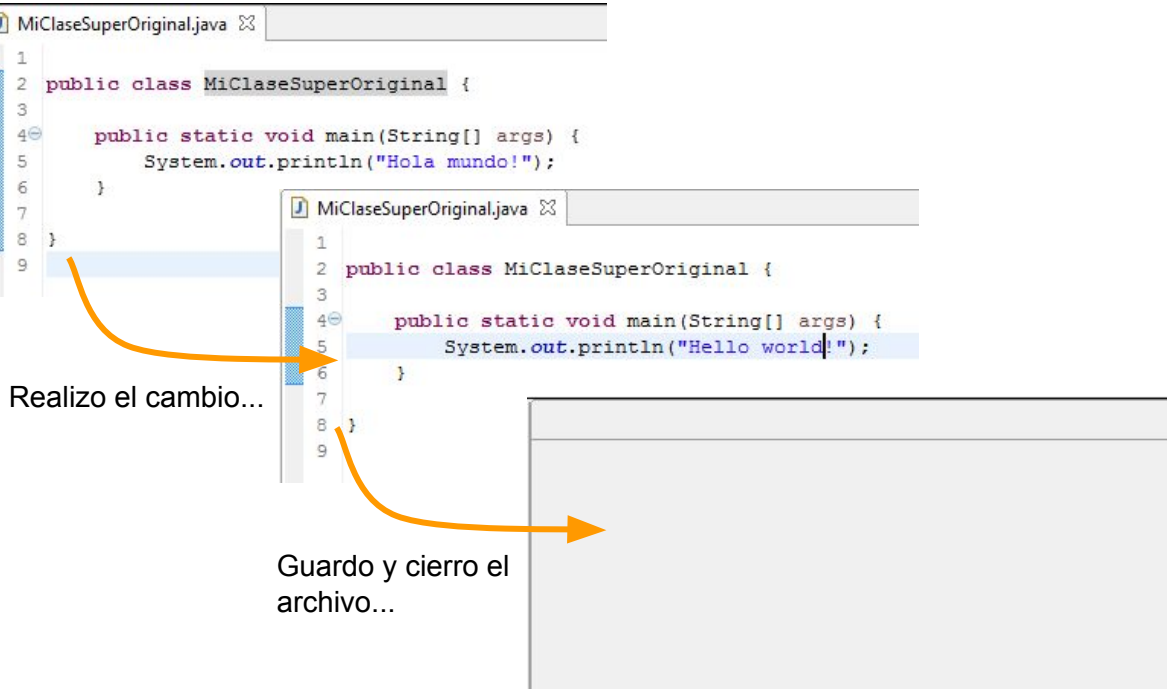
- Cada vez que empiezo a trabajar en un cambio, me gustaría poder volver atrás
- O quizás no deshacer todos los cambios, sino ver que he cambiado



Control de versiones

Para un proyecto Java:

- Cada vez que empiezo a trabajar en un cambio, me gustaría poder volver atrás
- O quizás no deshacer todos los cambios, sino ver que he cambiado



Control de versiones

Para un proyecto Java:

- Cada vez que empiezo a trabajar en un cambio, me gustaría poder volver atrás
- O quizás no deshacer todos los cambios, sino ver que he cambiado

MiClaseSuperOriginal.java

```
1
2  public class MiClaseSuperOriginal {
3
4      public static void main(String[] args) {
5          System.out.println("Hola mundo!");
6      }
7
8  }
9
```

Realizo el cambio...

MiClaseSuperOriginal.java

```
1
2  public class MiClaseSuperOriginal {
3
4      public static void main(String[] args) {
5          System.out.println("Hello world!");
6      }
7
8  }
9
```

Guardo y cierro el archivo...

Ahora quiero deshacer el cambio…

- ¡Control + z no está para ayudarme!
- Imagina eso en una clase de 300 líneas donde sólo cambiaba la comparación de un if… Acuérdate de cual era.

—

Parece que copiar y pegar proyectos, y hacer control + Z no es suficiente...

git para control de versiones

Nos ayuda en todo lo que hemos comentado

- Guarda un historial de los cambios producidos
- Puede volver a un estado anterior de forma segura, coherente y fácil
- Permite ver modificaciones entre diferentes versiones

git: control de versiones

git es una **herramienta** que trabaja directamente sobre un directorio, una carpeta de nuestro sistema.

Por lo tanto, puede ser utilizado para **cualquier tipo de proyecto** (una web, Java, C…) sin problemas.

Por supuesto, todo esto sin necesitar un IDE o una herramienta especial, simplemente desde la **línea de comandos** de git trabajamos sobre el directorio, aunque hay **IDEs** (como ecipse) lo integran.

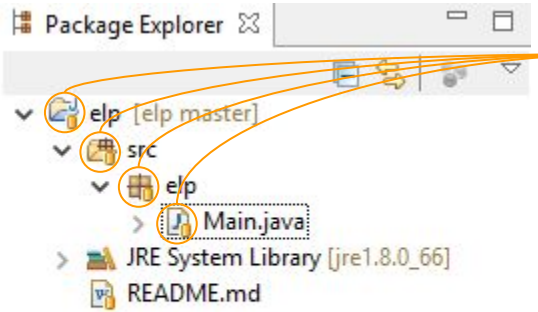
git en eclipse

Para mostrar el funcionamiento de git, en esta charla vamos a usar eclipse, un IDE que tiene integrado git.

En las últimas versiones viene integrado de base, en las más antiguas existía un plugin que se instalaba a parte.

– 

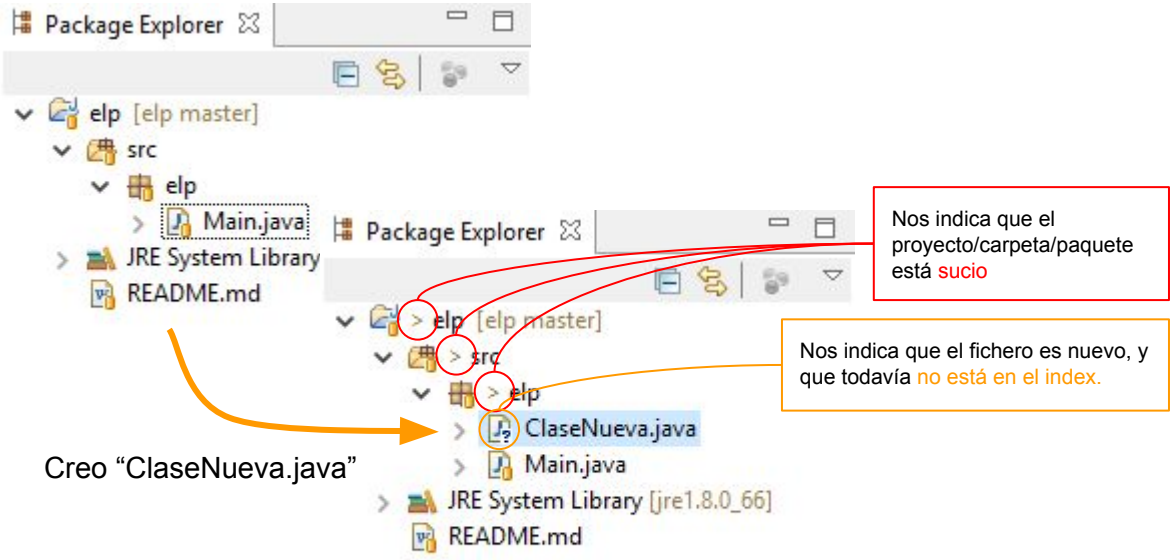
git en eclipse



Nos indica que el proyecto/carpeta/paquete/archivo están siendo “controlados” por git

git en eclipse: nuevo archivo

Cuando creamos un **archivo nuevo,** inicialmente git no lo considera parte de nuestro trabajo, y lo considera **suciedad:** o bien debemos especificar que lo **ignore**, o bien **añadirlo al index.**

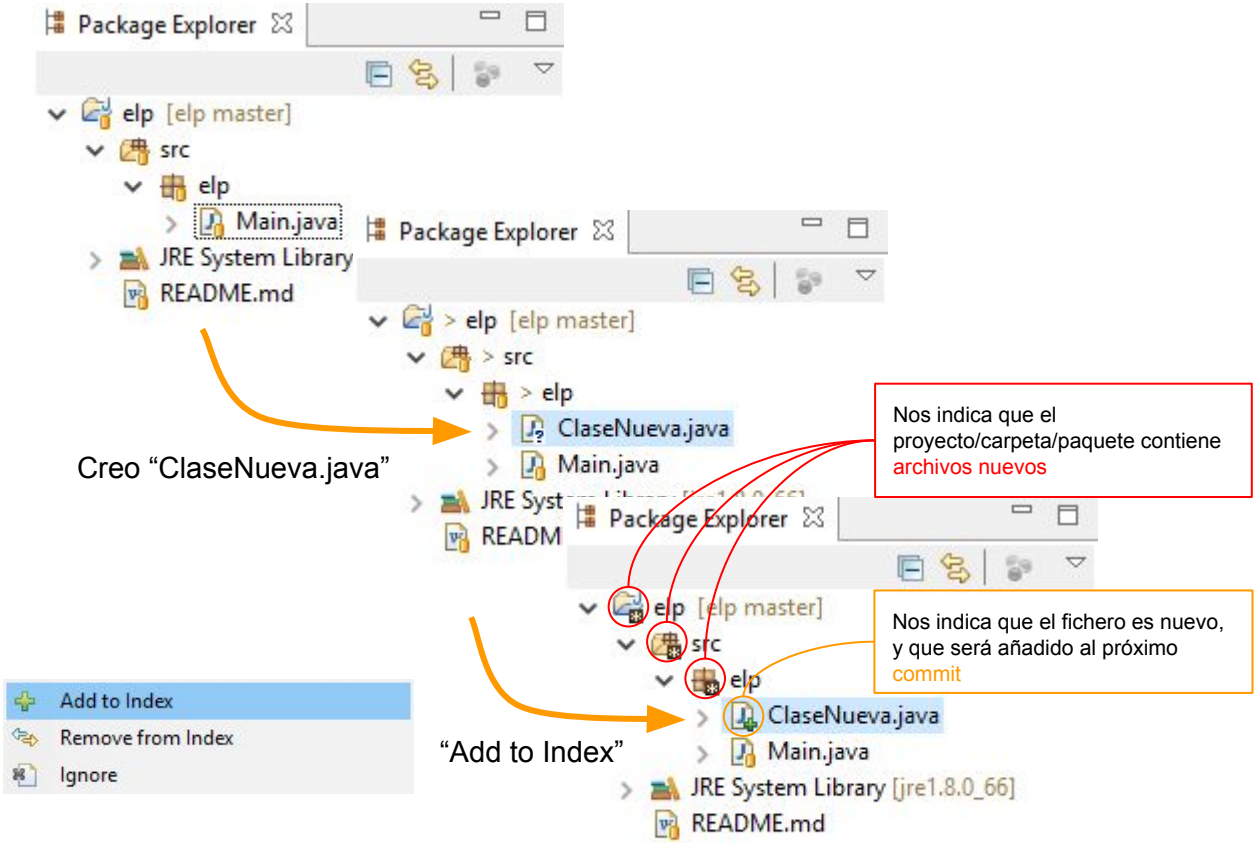


git en eclipse: add to index

Cuando creamos un archivo nuevo, inicialmente git no lo considera parte de nuestro trabajo, y lo considera suciedad: o bien debemos especificar que lo ignore, o bien añadirlo al index.

Para ello, **click derecho** sobre la nueva clase -> **team**, entre otras muchas opciones tendremos las siguientes ->

Add to index incluirá el archivo en el index de git para que **git trabaje con él** (lo mantenga en el control de versiones)

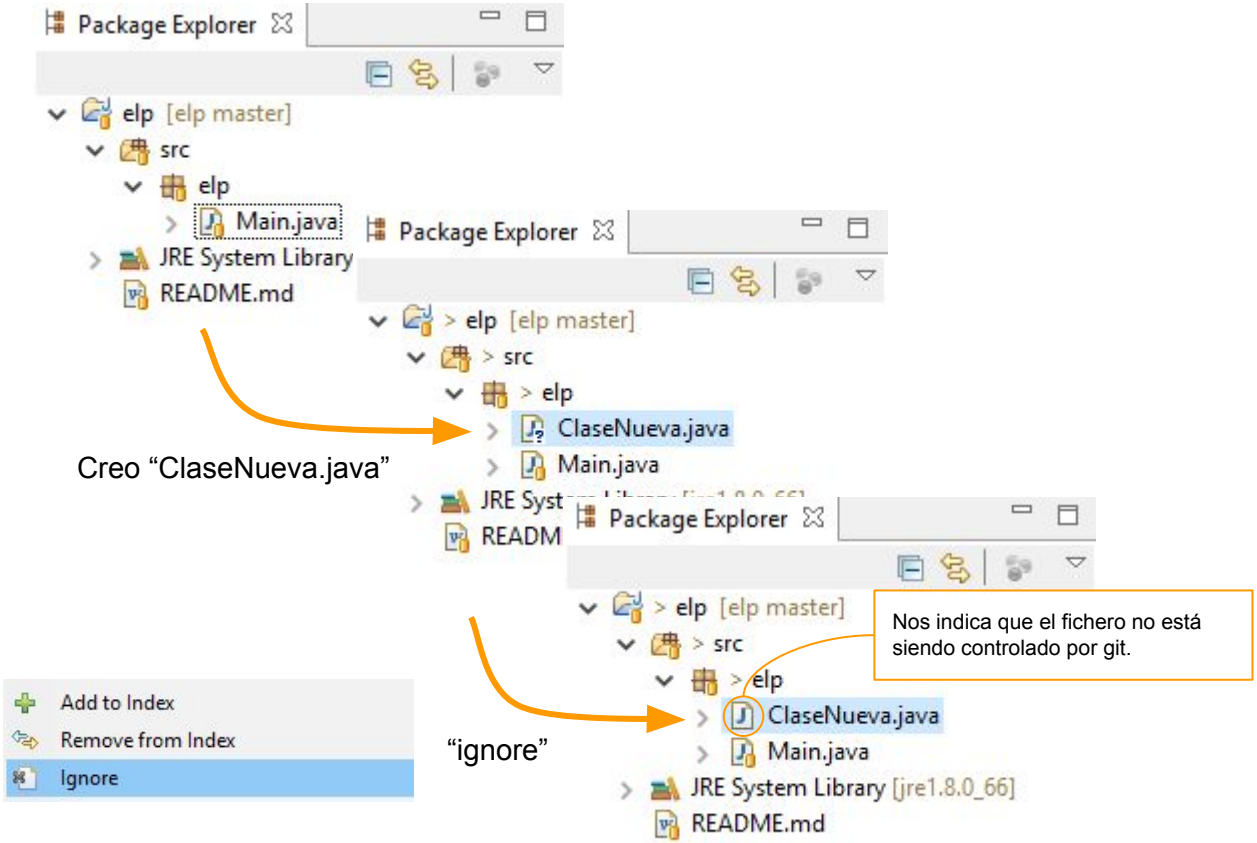


git en eclipse: ignore

Cuando creamos un archivo nuevo, inicialmente git no lo considera parte de nuestro trabajo, y lo considera suciedad: o bien debemos especificar que lo ignore, o bien añadirlo al index.

Para ello, click derecho sobre la nueva clase -> team, entre otras muchas opciones tendremos las siguientes ->

Ignore provocará que el archivo se añada al archivo de git .gitignore, que contiene la lista de archivos o carpetas que son ignorados por git.*



* esto provoca que se marquen como sucias algunas carpetas, pues está marcando el cambio en .gitignore. Aquí hay una lista de .gitignore para muchos lenguajes

git en eclipse: ignore

No es común ignorar archivos de código como en el ejemplo anterior, pues es bastante mala idea ya que cualquier cosa que dependa de ese archivo (por ejemplo hacer uso de la clase “ClaseNueva”), en el futuro si se realizan modificaciones en este archivo, no quedarán controlados por el sistema git.

Sí que es útil, por ejemplo, los archivos .class de java, son los archivos que se encuentran en la carpeta /bin del proyecto, y son regenerados cada vez que compilamos.

git en eclipse: commit

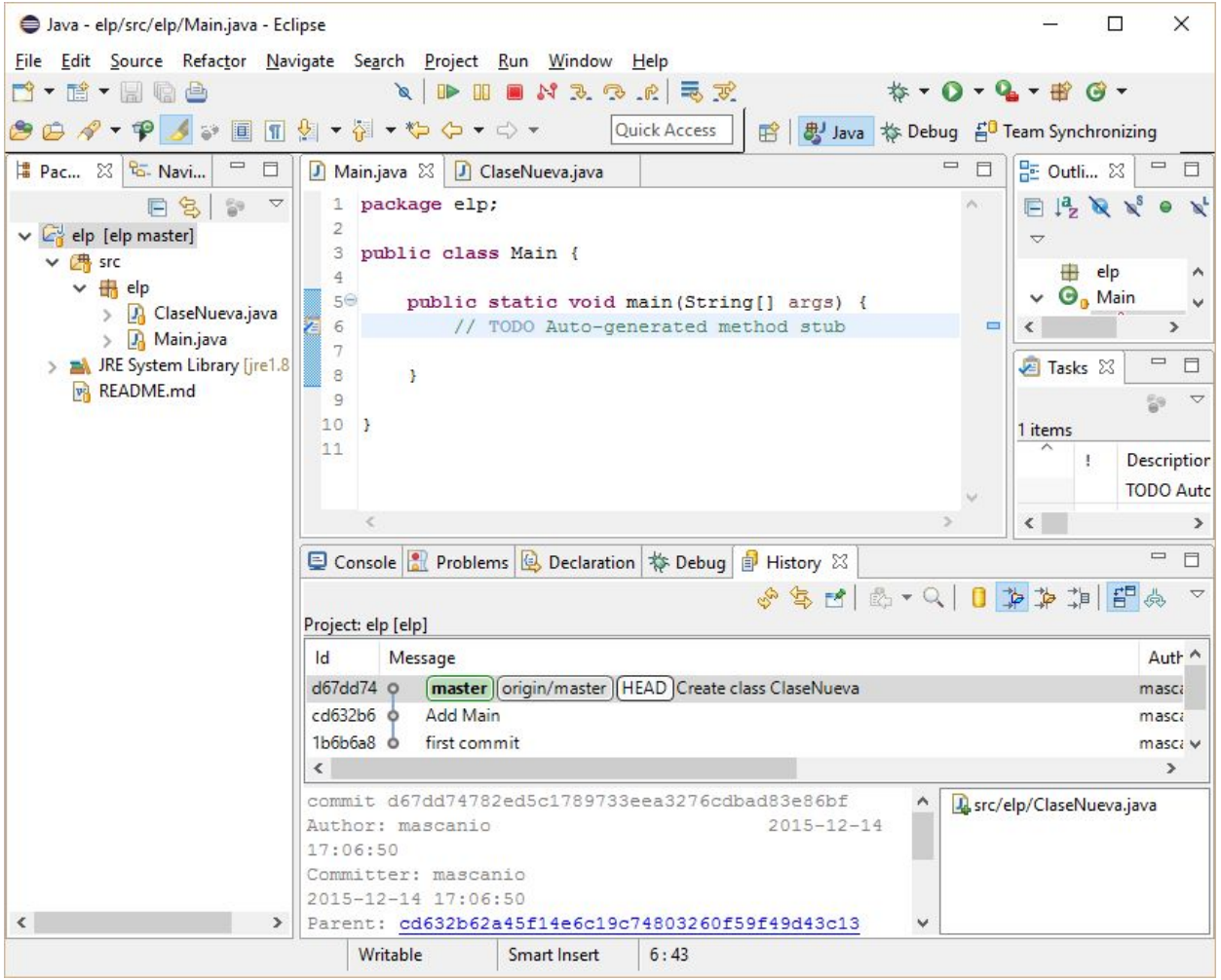
Cuando queremos guardar el estado actual, o realizar una *snapshot*, para poder revertir cambios o realizar comparaciones (como comentábamos antes), en git utilizamos commit

Realizar esto en eclipse es muy sencillo, simplemente hacemos click derecho sobre el proyecto -> team -> commit

git en eclipse: commit

Aquí ya hemos hecho el commit de la ceración de “ClaseNueva.java”. Se observa que el proyecto está limpio, con las clases controladas por git.

Como se puede ver en la “*History*” (click derecho sobre el proyecto -> team -> show in History), ya tenemos tres commits: el inicial (cuando se creó el proyecto), el de la creación de Main, y el de la creación de ClaseNueva.



git en eclipse: commit

Vamos a cambiar un poco la clase Main.

Java - elp/src/elp/Main.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Quick Access | Java | Debug | Team Synchronizing

Pac... | Navi...

> elp [elp master]
> src
> elp
ClaseNueva.java
> Main.java
JRE System Library [jre1.8
README.md

Main.java | ClaseNueva.java

```
1  package elp;
2
3  public class Main {
4
5      public static void main(String[] args) {
6          System.out.println("Hello world!");
7      }
8
9  }
10
```

Outli...
> Main
main(S

Tasks
0 items
! Description

Console | Problems | Declaration | Debug | History

Project: elp [elp]

Id	Message	Auth
d67dd74	master origin/master HEAD Create class ClaseNueva	masca
cd632b6	Add Main	masca
1b6b6a8	first commit	masca

```
commit d67dd74782ed5c1789733eea3276cdbad83e86bf
Author: mascanio                    2015-12-14
17:06:50
Committer: mascanio
2015-12-14 17:06:50
Parent: cd632b62a45f14e6c19c74803260f59f49d43c13
```

src/elp/ClaseNueva.java

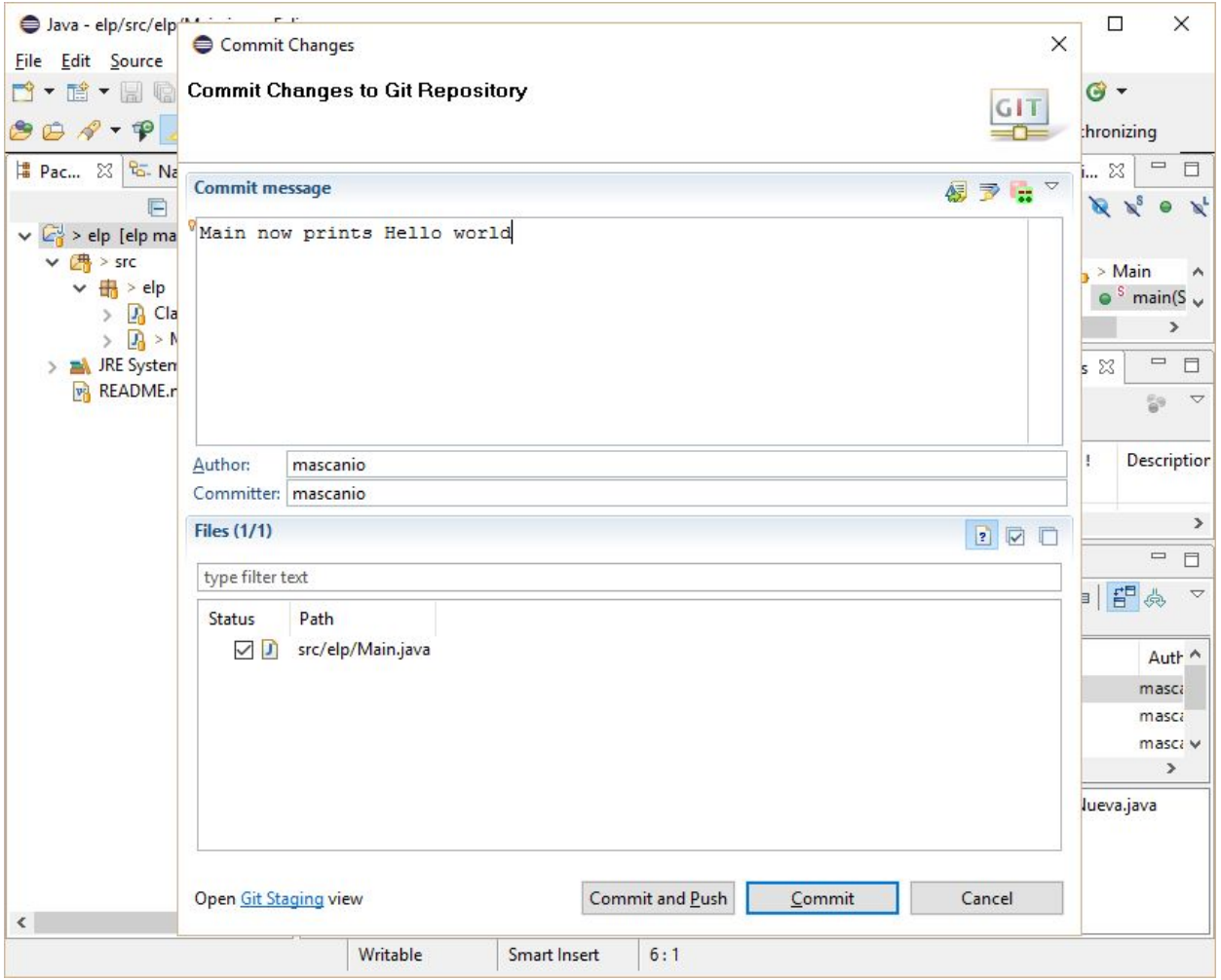
Writable | Smart Insert | 6 : 1

git en eclipse: commit

Vamos a cambiar un poco la clase Main.

Cuando hacemos commit, nos saldrá el asistente de commit de eclipse, donde tenemos un cuadro de texto.

La primera línea debe ser una descripción breve (max 80 caracteres) del commit, casi como un título. Después, separada por salto de línea, viene una descripción más detallada.



– git

git en eclipse: commit

Vamos a cambiar un poco la clase Main

Cuando hacemos commit, nos saldrá el asistente de commit de eclipse, donde tenemos un cuadro de texto.

La primera línea debe ser una descripción breve (max 80 caracteres) del commit, casi como un título. Después, separada por salto de línea, viene una descripción más detallada.

El commit queda reflejado en la historia.

Java - elp/src/elp/Main.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Quick Access | Java | Debug | Team Synchronizing

Pac... | Navi...

- elp [elp master ↑1]
  - src
    - elp
      - ClaseNueva.java
      - Main.java
  - JRE System Library [jre1.8
  - README.md

Main.java | ClaseNueva.java

```
1  package elp;
2
3  public class Main {
4
5      public static void main(String[] args) {
6          System.out.println("Hello world!");
7      }
8
9  }
10
```

Outli... | elp | Main

Tasks | 0 items | ! | Descriptior

Console | Problems | Declaration | Debug | History

File: elp/src/elp/Main.java [elp]

Id	Message	Author	Authored ...	Committer	Com
2daaa7e	master HEAD Main now prints Hello world	mascanio	8 minutes ago	mascanio	8 min
d67dd74	origin/master Create class ClaseNueva	mascanio	41 minutes ag	mascanio	41 mi
cd632b6	Add Main	mascanio	2 days ago	mascanio	2 day:

```
commit 2daaa7e8d42f0f6e8f6a621c0053ad96b1c45620
Author: mascanio                    2015-12-14
17:39:37
Committer: mascanio
2015-12-14 17:39:37
Parent: d67dd74782ed5c1789733eea3276cdbad83e86bf
```

src/elp/Main.java

Writable | Smart Insert | 10 : 1

git en eclipse: commits comp

Ahora vamos a comparar el commit que acabamos de realizar con “AddMain”

Para ello hacemos click derecho sobre el proyecto -> compare with -> commit, y seleccionamos el commit “AddMain”

Tras hacer doble click en Main.java, se nos abre esta vista de comparación de esa clase.

Team Synchronizing - Git (elp, elp): /elp/src/elp/Main.java - Eclipse

File Edit Navigate Search Project Run Window Help

Quick Access | Java | Debug | Team Synchronizing

Synchronize

Git (elp, elp)

elp [elp master ↑1]

src/elp

ClaseNueva.java

Main.java

Archivo nuevo

Archivo modificado

Main.java | ClaseNueva.java | Main.java

Java Structure Compare

Compilation Unit

Main

main(String [])

Java Source Compare

Zona de comparación

Local File Workspace

```
1 package elp;
2
3 public class Main {
4
5     public static void main(String
6         System.out.println("Hello
7     }
8
9 }
10
```

Remote File cd632b6... (mascanio)

```
1 package elp;
2
3 public class Main {
4
5     public static void main(St
6         // TODO Auto-generated
7
8     }
9
10 }
11
```

Task Repositories

Tasks

Local

Bugs

Eclipse.org

History | Tasks | Problems

Left: 6 : 9, Right: 9 : 1, outgoing change #1 (Left: 6 : 6, Right: 6 : 7)

Vale, todo esto es muy bonito y útil. Pero... ¿para qué le vale esto a un proyecto de 40 personas?

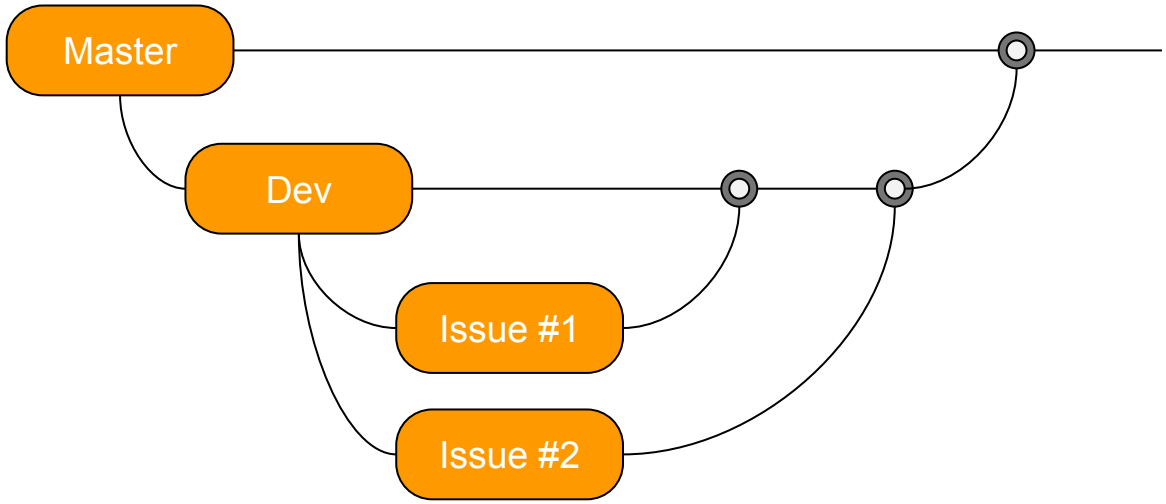
git distribuido

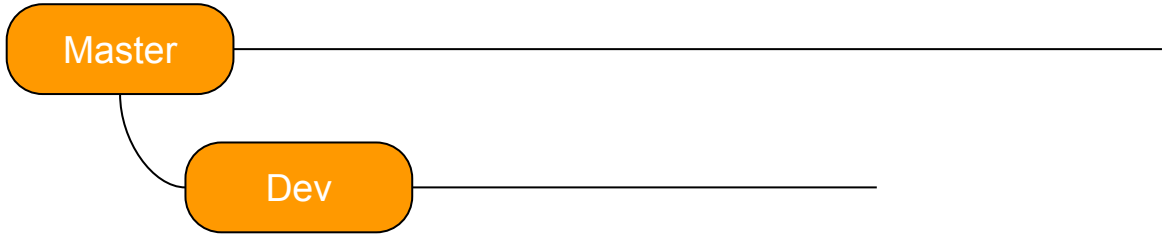
Es como lo que hemos visto hasta ahora, pero utilizándolo muchas personas a la vez.

git: branching

La clave para la programación distribuida en git son las “*branch*” (ramas).

Crear una rama podemos verlo conceptualmente como copiar todo lo que teníamos y seguir realizando modificaciones por otra parte: en el diagrama, crearemos la rama Dev a partir de Master (copiaremos todo lo que está en Master); a partir de Dev sacaremos Issue #1 e Issue#2

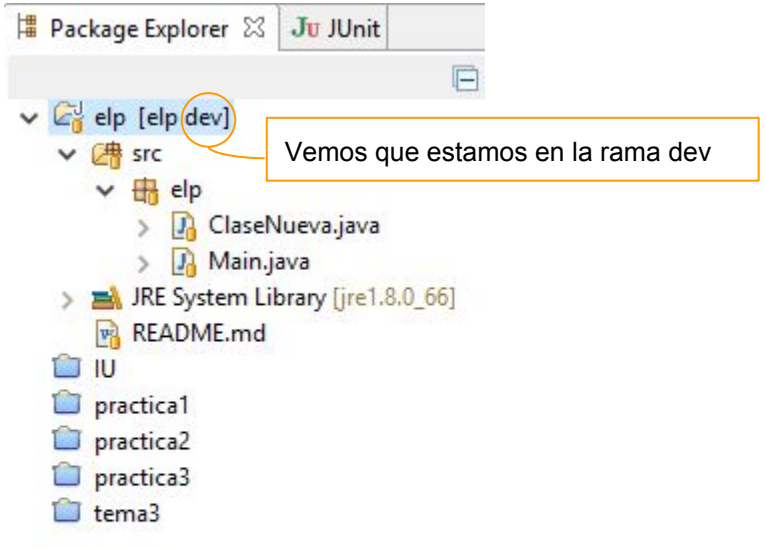


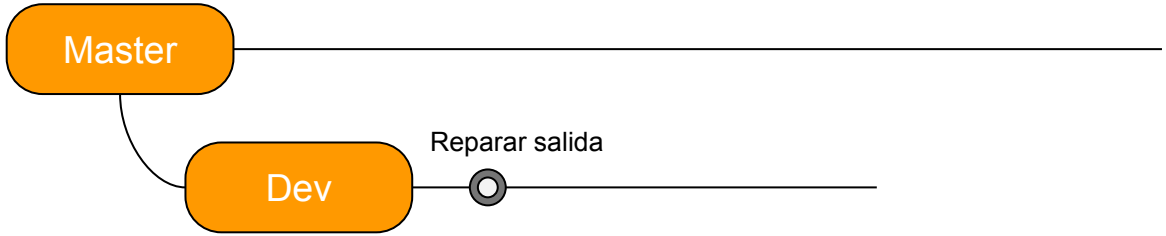


git: branching

Vamos a crear una nueva rama: Dev a partir de Máster.

Para ello, click derecho sobre el proyecto -> team -> switch to -> new branch; aquí creamos la branch Dev.



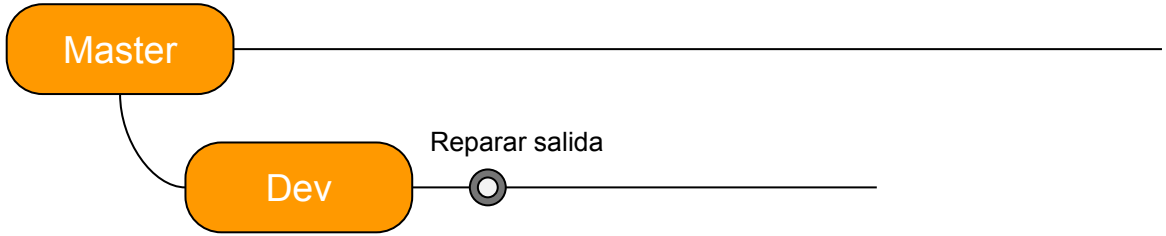


git: branching

Sigamos con este código en la rama dev, hemos cambiado el código anterior del “Hola mundo!” por un código para hacer divisiones, y hemos hecho un commit para arreglar un fallo en la salida por pantalla.

Main.java

```java
package elp;

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int a, b, c;

        System.out.println("Introduzca el dividendo: ");
        a = sc.nextInt();
        System.out.println("Introduzca el divisor: ");
        b = sc.nextInt();

        c = a / b;
        System.out.println("Resultado = " + c);

        sc.close();
    }

}
```



git: branching

Este código tiene un par de errores, o al menos es susceptible de mejora: por un lado, salta excepción al dividir por 0, y por otro, se podría mostrar el resto de la división.

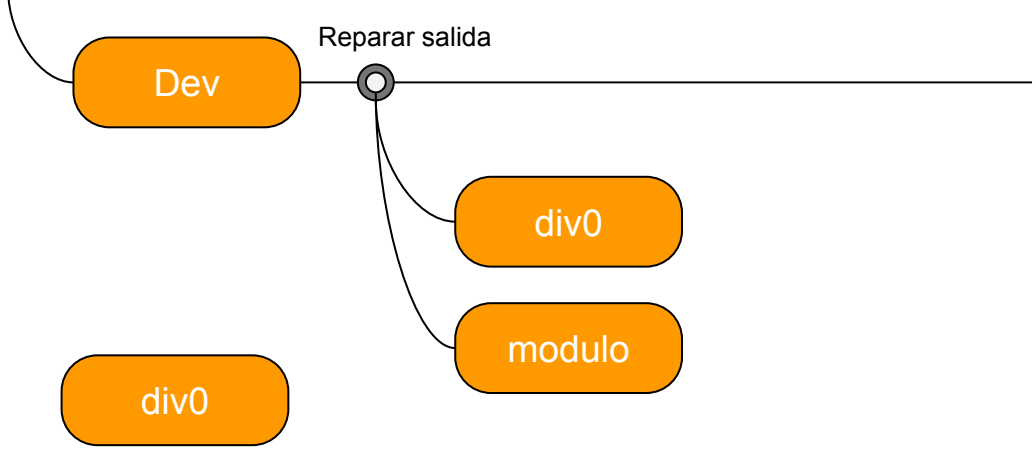
Para corregir lo anterior, planteamos estos dos problemas en nuestra organización, y tenemos dos programadores que se ofrecen a arreglarlos, **pero un problema cada uno, por separado.**

Main.java

```java
package elp;

import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int a, b, c;

        System.out.println("Introduzca el dividendo: ");
        a = sc.nextInt();
        System.out.println("Introduzca el divisor: ");
        b = sc.nextInt();

        c = a / b;
        System.out.println("Resultado = " + c);

        sc.close();
    }

}
```

git: branching

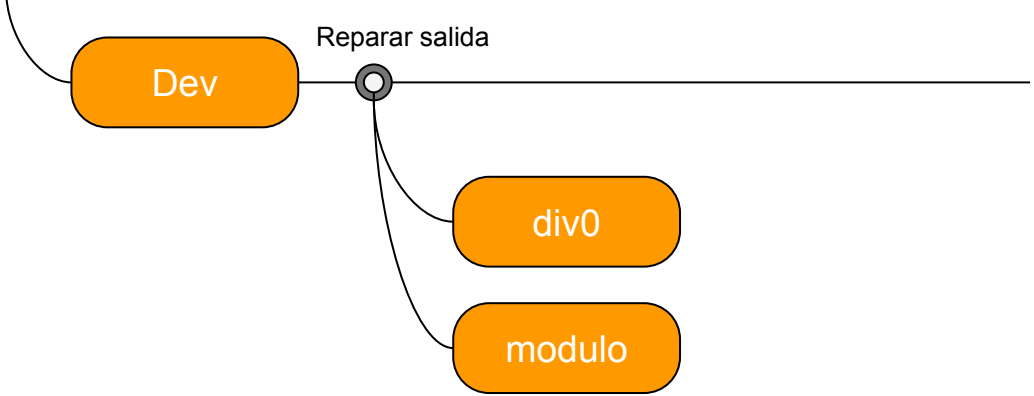
El programador 1 se encarga de la parte de la división por 0, y crea la rama div0; mientras que el programador 2 se encarga del módulo y crea la rama módulo; todo a partir del commit reparar salida.




```
if (b != 0) {
    c = a / b;
    System.out.println("Resultado = " + c);
} else {
    System.out.println("Error, división por 0");
}
```

git: branching

El programador 1 se encarga de la parte de la división por 0, y crea la rama div0; mientras que el programador 2 se encarga del módulo y crea la rama módulo; todo a partir del commit reparar salida.



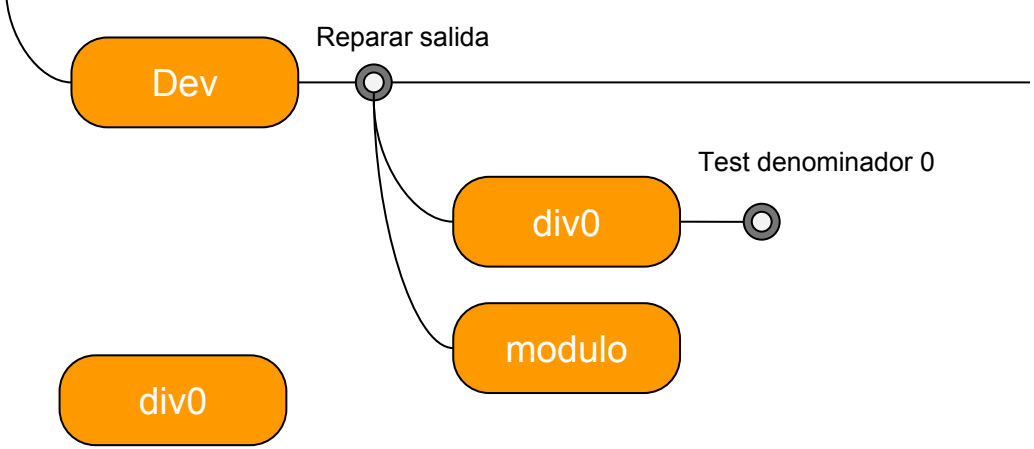
modulo

```
int a, b, c, r;

System.out.println("Introduzca el dividendo: ");
a = sc.nextInt();
System.out.println("Introduzca el divisor: ");
b = sc.nextInt();

c = a / b;
r = a % b;
System.out.println("Resultado = " + c + " resto: " + r);

sc.close();
```

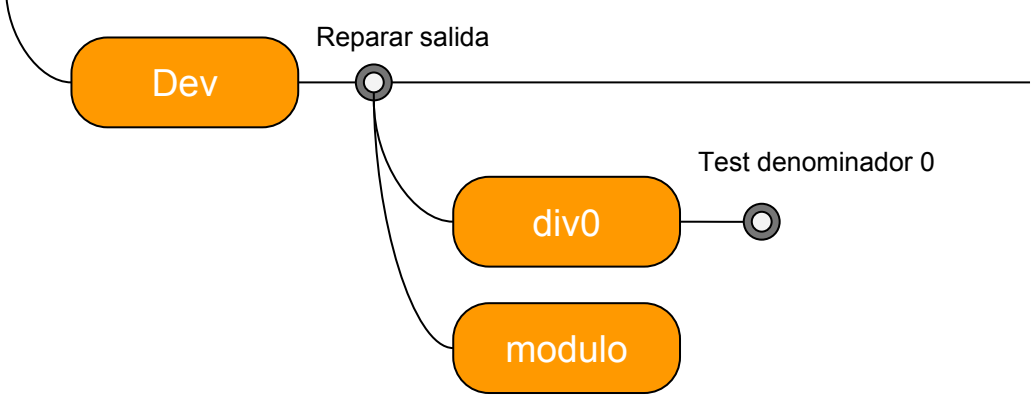
git: branching

Ahora el programador encargado del div0 hace commit de sus cambios llamado Test denominador 0 (sobre la rama div0).

Por supuesto, una vez implementado este cambio, lo interesante sería reflejar estos cambios sobre la rama dev.




```
if (b != 0) {
    c = a / b;
    System.out.println("Resultado = " + c);
} else {
    System.out.println("Error, división por 0");
}
```



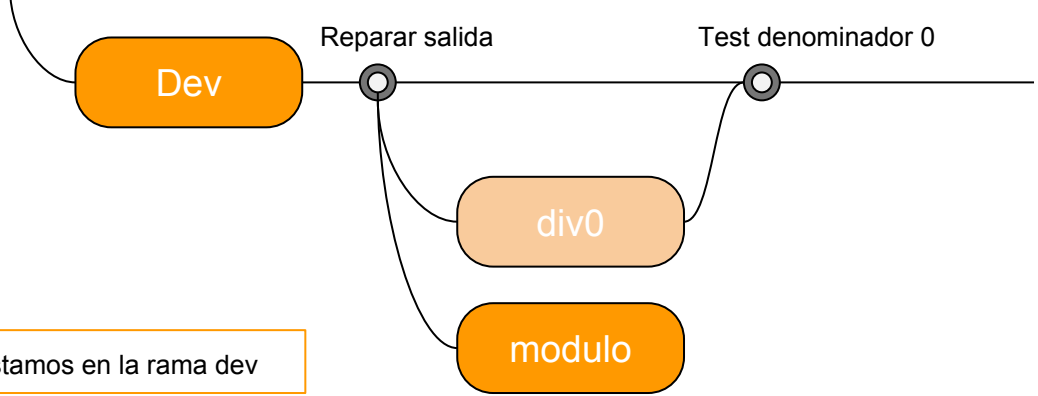
git: merge

Para esto tenemos la acción de git merge.

La operación merge nos permite juntar un commit de una rama sobre otra.

Para juntar el commit de div0 sobre la rama dev, primero cambiamos a la rama dev (team -> switch to -> dev).

Ahora, team -> merge y seleccionamos div0.



git: merge

En este caso, la operación de merge ha sido muy simple: conceptualmente, ha transladado el commit “Test denominador 0” a la rama Dev, realizando el proceso automáticamente.

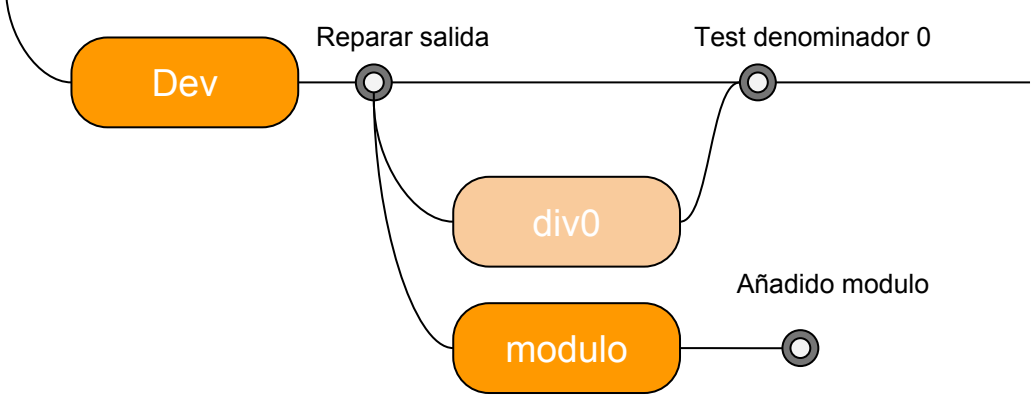
Vemos que estamos en la rama dev

Package Explorer

- elp [elp dev ↑1]
  - src
    - elp
      - ClaseNueva.java
      - Main.java
  - JRE System Library [jre1.8.0_66]
  - README.md

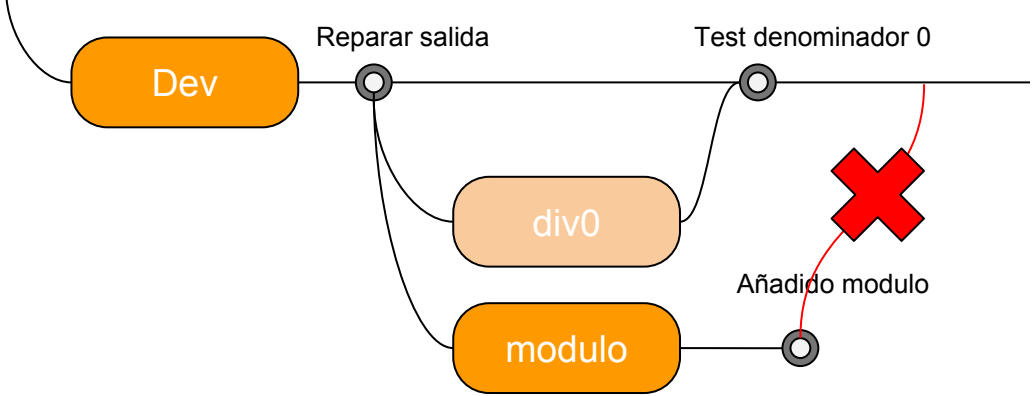
Main.java

```
7   public static void main(String[] args) {
8       Scanner sc = new Scanner(System.in);
9       int a, b, c;
10
11      System.out.println("Introduzca el dividendo: ");
12      a = sc.nextInt();
13      System.out.println("Introduzca el divisor: ");
14      b = sc.nextInt();
15
16      if (b != 0) {
17          c = a / b;
18          System.out.println("Resultado = " + c);
19      } else {
20          System.out.println("Error, división por 0");
21      }
22      sc.close();
23  }
```



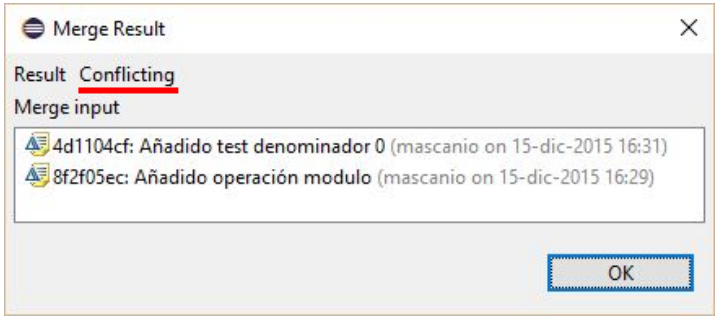
git: merge

Ahora, el programador 2 realiza el commit en la rama módulo. Por supuesto, todos los cambios que han sido realizados en la rama dev son totalmente ajenos a la rama módulo.



git: merge

Ahora, el programador 2 realiza el commit en la rama módulo. Por supuesto, todos los cambios que han sido realizados en la rama dev son totalmente ajenos a la rama módulo.



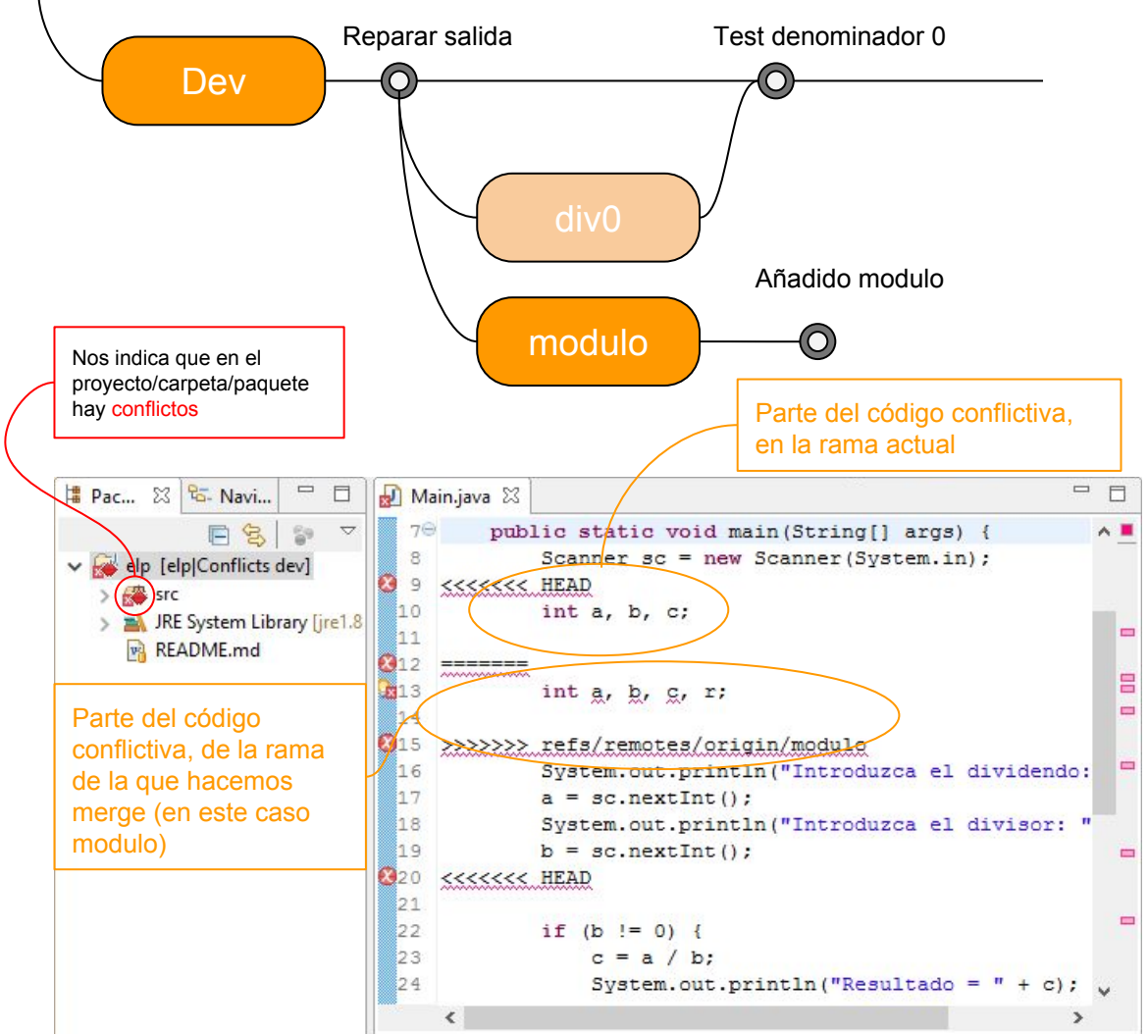
Ahora hacer el merge no es tan fácil, y git no lo hará automáticamente. Como vemos, git nos avisa de que existen conflictos.

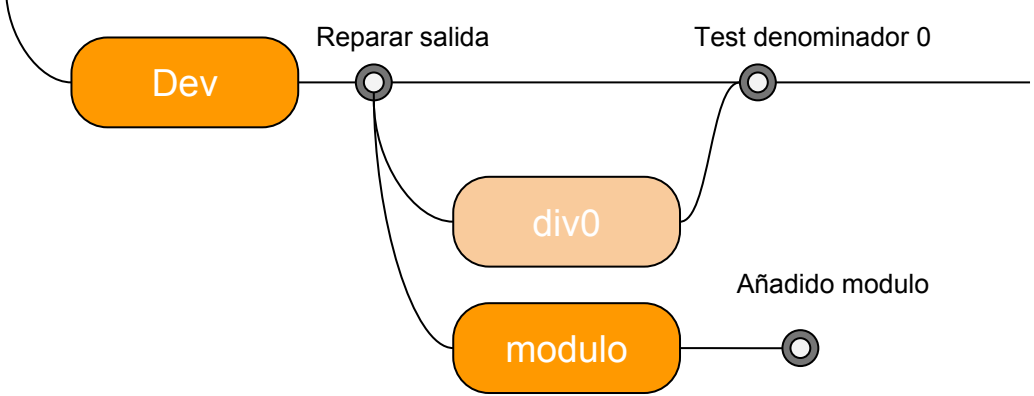
git: merge

Ahora, el programador 2 realiza el commit en la rama módulo. Por supuesto, todos los cambios que han sido realizados en la rama dev son totalmente ajenos a la rama módulo.

Ahora hacer el merge no es tan fácil, y git no lo hará automáticamente. Como vemos, git nos avisa de que existen conflictos.

Además, se muestran las zonas que han creado conflicto sobre el propio código.

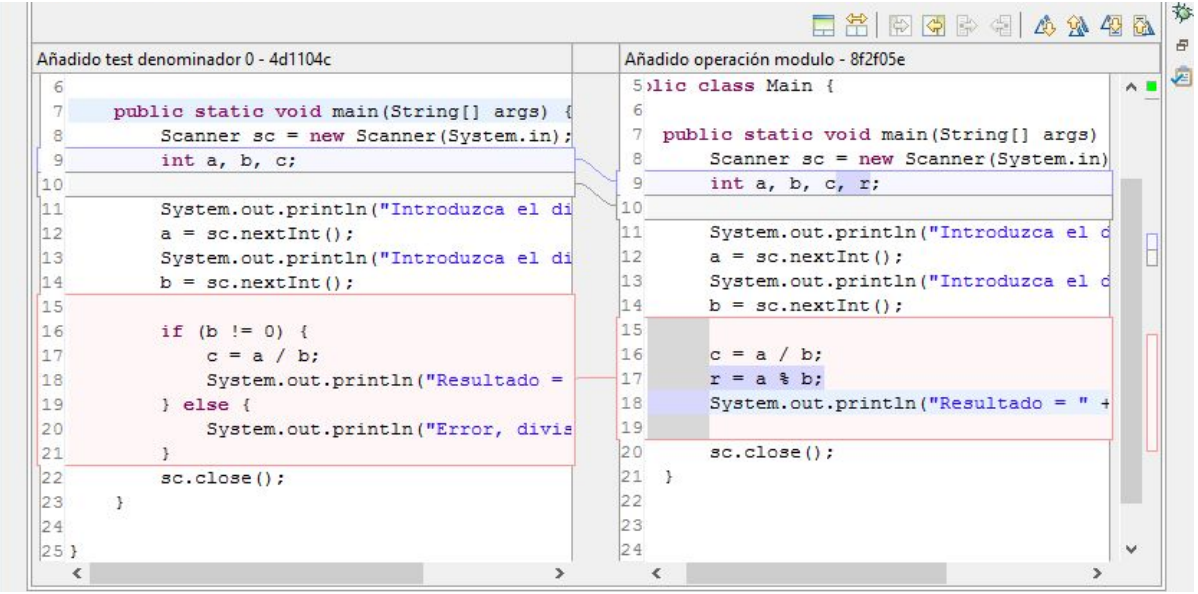


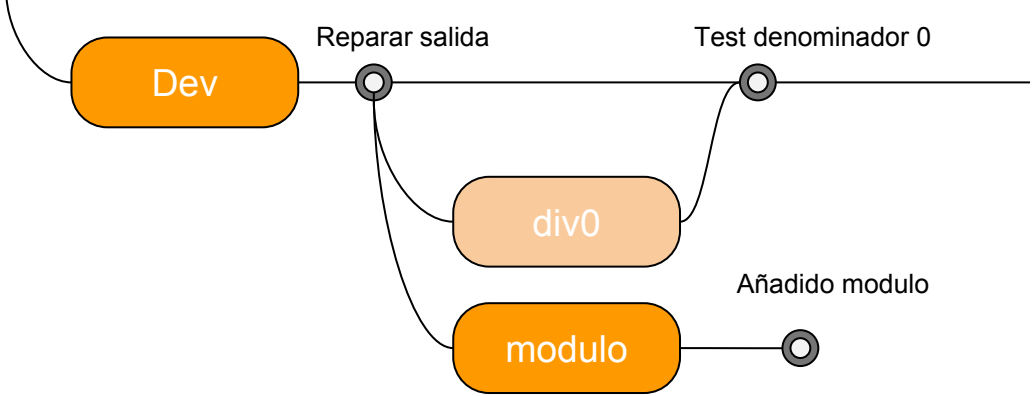


git: merge

Así presentado queda un poco feo, pero eclipse nos brinda la herramienta merge-tool. Para abrirla, hacemos click derecho sobre la clase Main.java (que está marcada como conflicto) -> team -> merge tool.

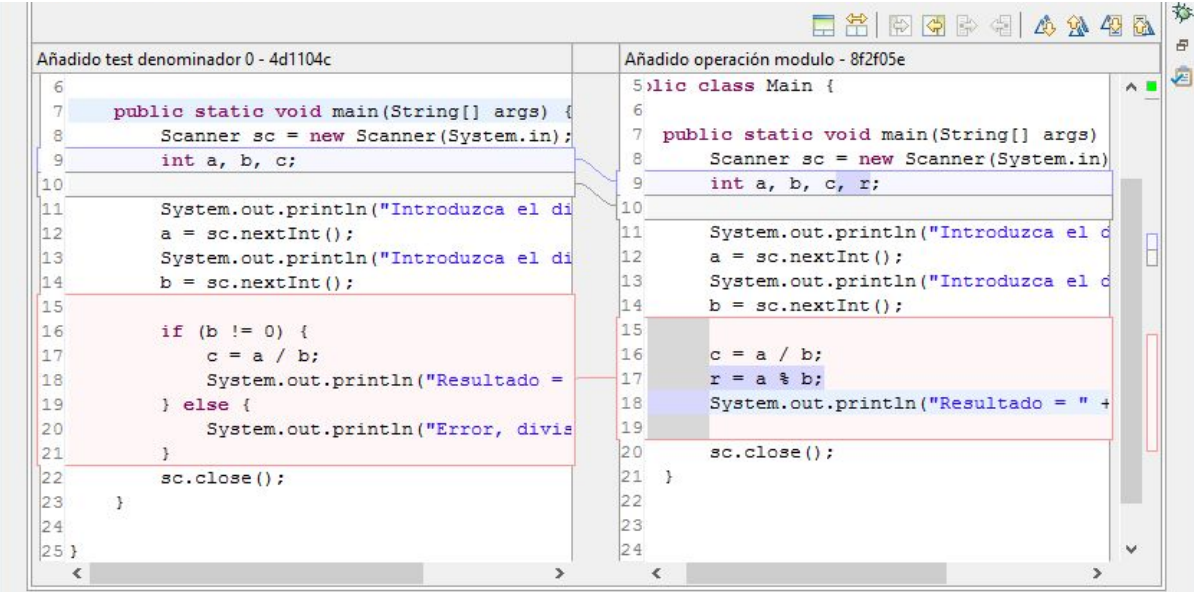
Aquí las diferencias se muestran de una manera más “amigable”, además que disponemos de herramientas para navegar por las distintas colisiones, y para aplicar los cambios “no conflictivos” automáticamente

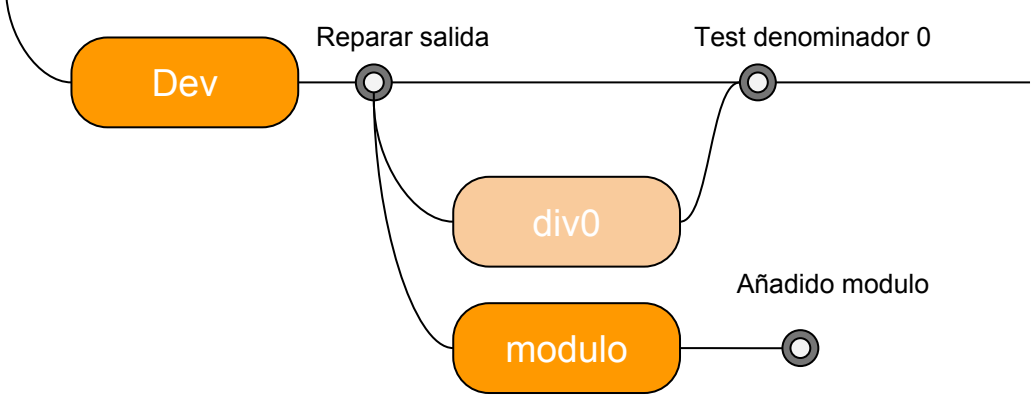




git: merge

Cómo aplicar los cambios: muy fácil, mientras que la ventana de la derecha nos muestra sólo el código de la branch de la que hacemos merge, la ventana de la izquierda es el código de la branch actual, y es editable: es el propio archivo.

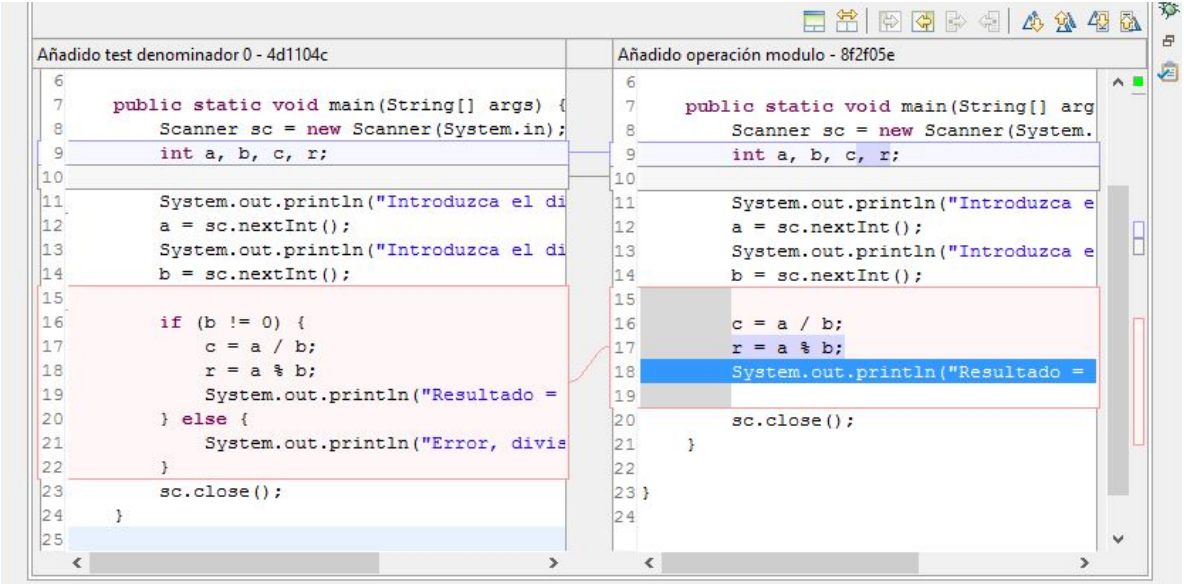




git: merge

Cómo aplicar los cambios: muy fácil, mientras que la ventana de la derecha nos muestra sólo el código de la branch de la que hacemos merge, la ventana de la izquierda es el código de la branch actual, y es editable: es el propio archivo.

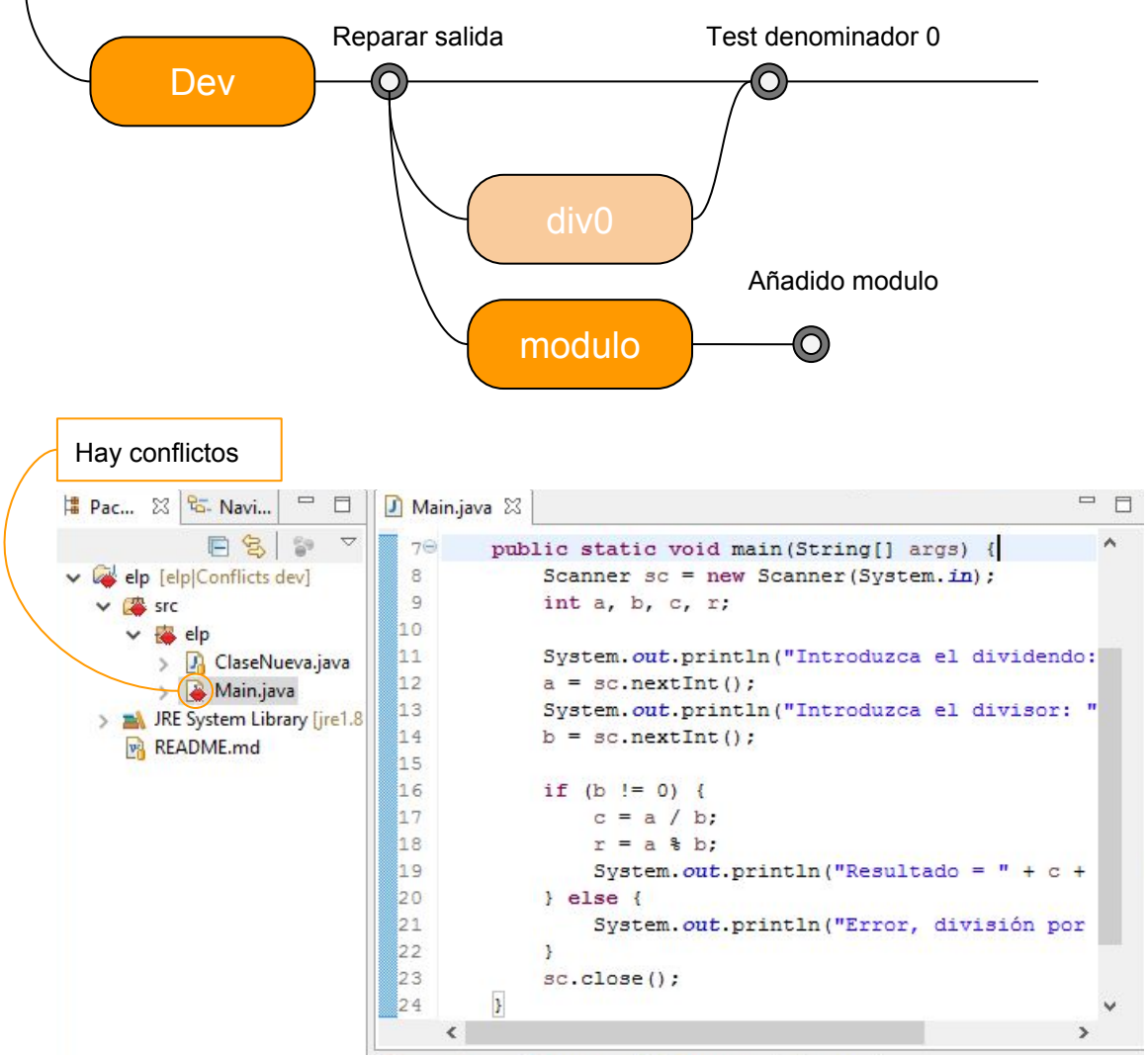
Aquí ya hemos realizado los cambios pertinentes según nuestro criterio, ya estamos listos para reflejar finalmente los cambios en dev.



git: merge

Primero, guardamos los cambios realizados en Main.java

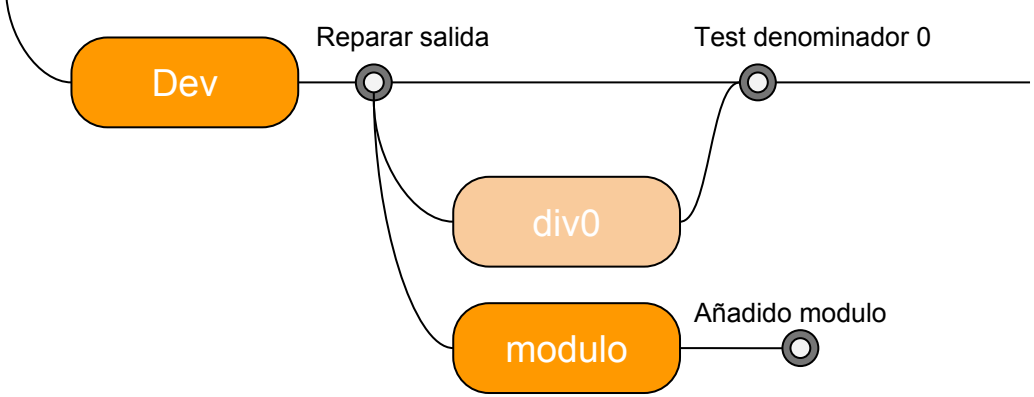
Segundo, corregimos el estado de “conflicto” de Main.java. Para ello, simplemente hacemos “Add to index” sobre Main.java

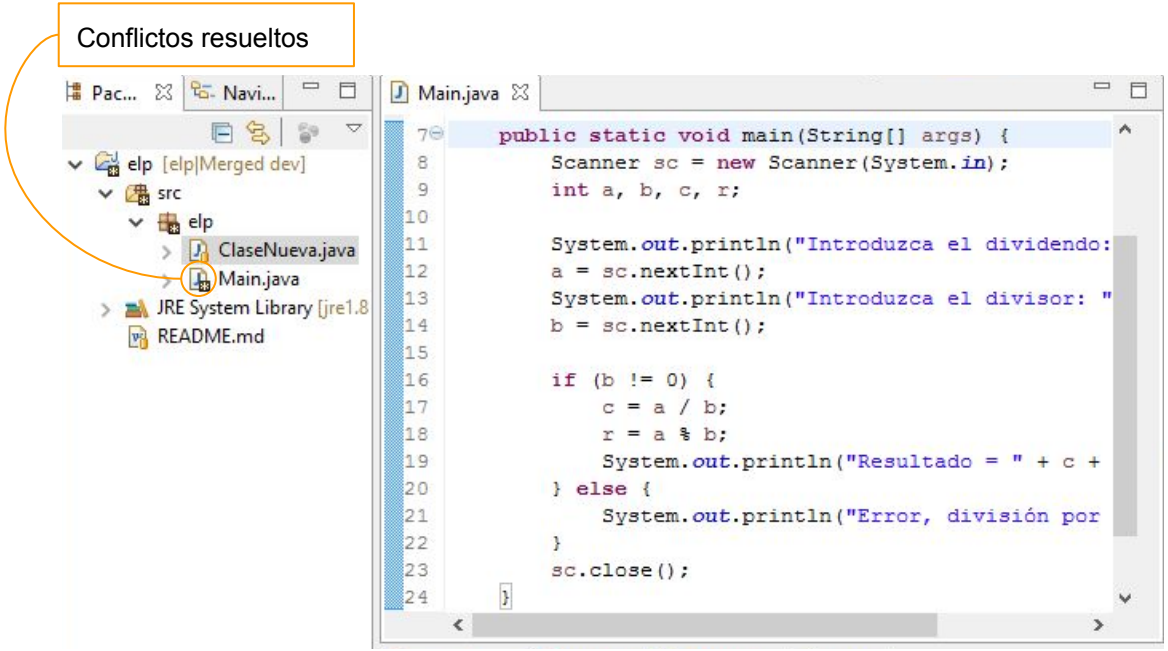


git: merge

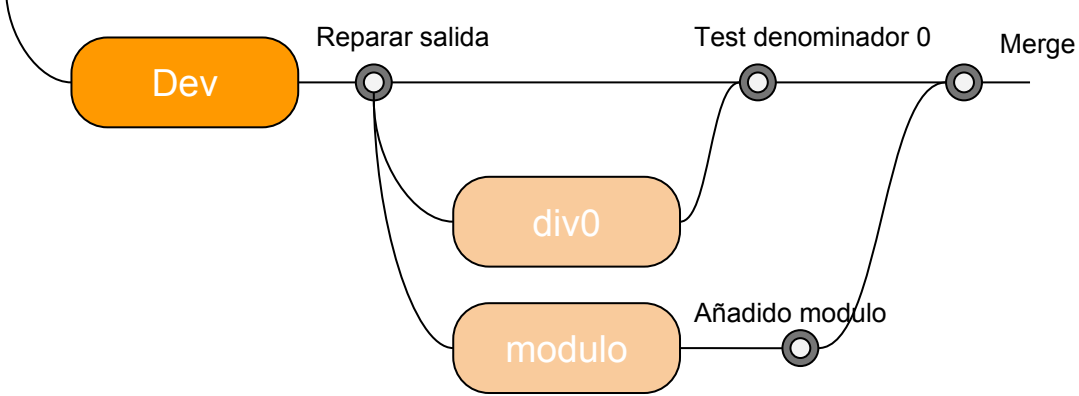
Primero, guardamos los cambios realizados en Main.java

Segundo, corregimos el estado de “conflicto” de Main.java. Para ello, simplemente hacemos “Add to index” sobre Main.java





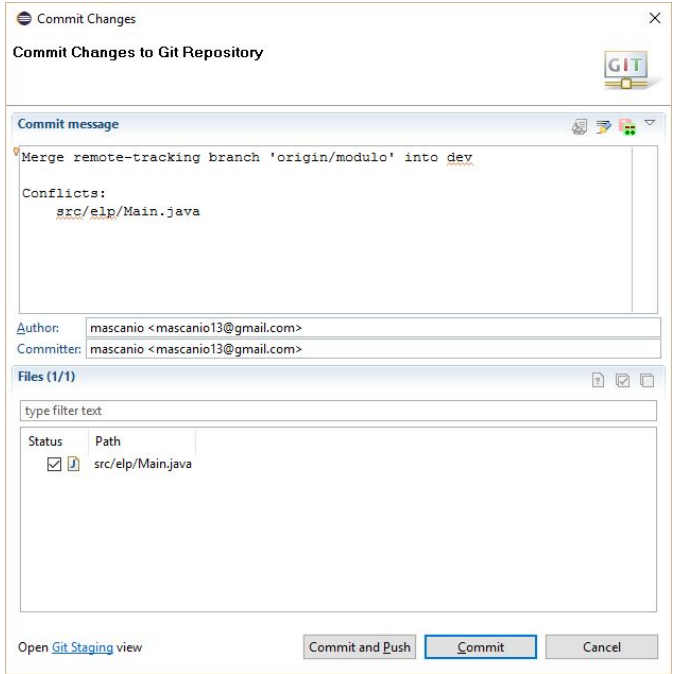
git: merge



Primero, guardamos los cambios realizados en Main.java

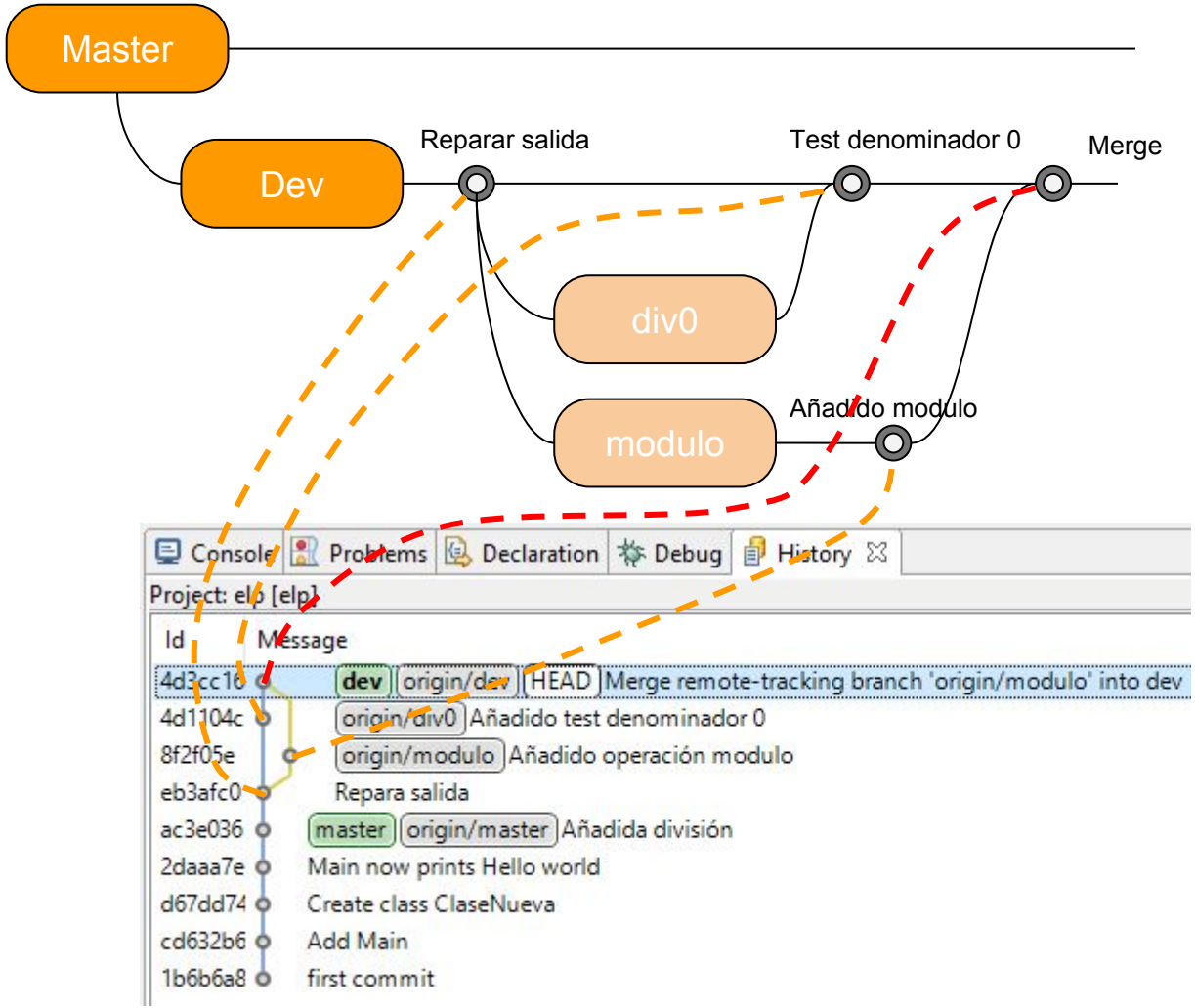
Segundo, corregimos el estado de “conflicto” de Main.java. Para ello, simplemente hacemos “Add to index” sobre Main.java

Finalmente, realizamos el commit. Por defecto, eclipse nos pone un mensaje predeterminado relacionado con la acción de merge.



git: merge

Si observamos la historia de la rama Dev, observamos que es un poco diferente



—

Bien, git vale para que varias personas trabajen sobre lo mismo, pero ¿cómo?

git: servidores

Para poder trabajar en git sobre el mismo proyecto, se necesita un servidor git.

Este servidor es ligeramente diferente a otros sistemas de control de versiones (como SVN).

Cualquiera nos podemos montar un servidor git, aunque en internet hay muchas opciones disponibles para todos los gustos: gratuitas y de pago, privadas o públicas.

GitHub

Servidor de git público y gratuito

- GitHub es uno de los servidores git más famosos y utilizados del mundo.
- En él se encuentran casi todos los proyectos de software libre que existen.
- GitHub no es sólo un servidor

GitHub como servidor de git

Como ya hemos comentado, GitHub hace de servidor de git: podemos subir nuestro git y que muchas personas trabajen con él.

Para esto, git nos proporciona pull y push, y por supuesto eclipse nos ayuda con ello.

Así, usando pull y push, y realizando las configuraciones necesarias, utilizamos GitHub

git pull y git push

El comando pull sirve para actualizar nuestro git local con la última versión del servidor.

Actualizará las branch que ya tengamos en local, pero no descargará el resto, esto siempre podemos hacerlo al cambiar a una branch remota.

Push la operación inversa al pull: sirve para subir commits o branches al servidor.

git y GitHub

En general, trabajar varias personas con git en un servidor como GitHub no es muy diferente a lo que vimos en la parte de branching.

GitHub es mucho más que un servidor de git.

Para verlo, mejor unos ejemplos...