Terminología general de Control de versiones

Terminología

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 de cambios, a menudo en un servidor. Puede ser un sistema de archivos en un disco duro, un banco de datos, etc..

Revisión (*"version"*)

Una **revisión** es una versión determinada de la información que se gestiona. Hay sistemas que identifican las revisiones con un contador (Ej. subversion). Hay otros sistemas que identifican las revisiones mediante un código de detección de modificaciones (Ej. Git usa SHA1). A la última versión se le suele identificar de forma especial con el nombre de **HEAD**. Para marcar una revisión concreta se usan los **rótulos** o *tags*.

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.
Los tags permiten identificar de forma fácil revisiones importantes en el proyecto. Por ejemplo se suelen usar tags para identificar el contenido de las versiones publicadas del proyecto.
En algunos sistemas se considera un tag como una rama en la que los ficheros no evolucionan, están congelados.

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 se tienen dos copias (ramas) que evolucionan de forma independiente siguiendo su propia línea de desarrollo. 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 contengan código para evaluación, si se decide que las modificaciones realizadas en la "rama de prueba" sean preservadas, se hace un "merge" con la rama principal. Son motivos habituales para la creación de ramas la creación de nuevas funcionalidades o la corrección de errores.

Desplegar (*"Check-out"*, *"checkout"*, "co")

Un despliegue crea una copia de trabajo local desde el repositorio. Se puede especificar una revisión concreta, y predeterminadamente se suele obtener la última. La copia local se queda asociada a la del repositorio.

"Publicar" o "Enviar" (*"commit"*, *"check-in", "submit"*)

Un ***commit*** sucede cuando una copia de los cambios hechos a una copia local es escrita o integrada sobre el repositorio.

Conflicto

Un conflicto ocurre cuando el sistema no puede manejar adecuadamente cambios realizados por dos o más usuarios en un mismo archivo. Por ejemplo, si se da esta secuencia de 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 archivo.

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.

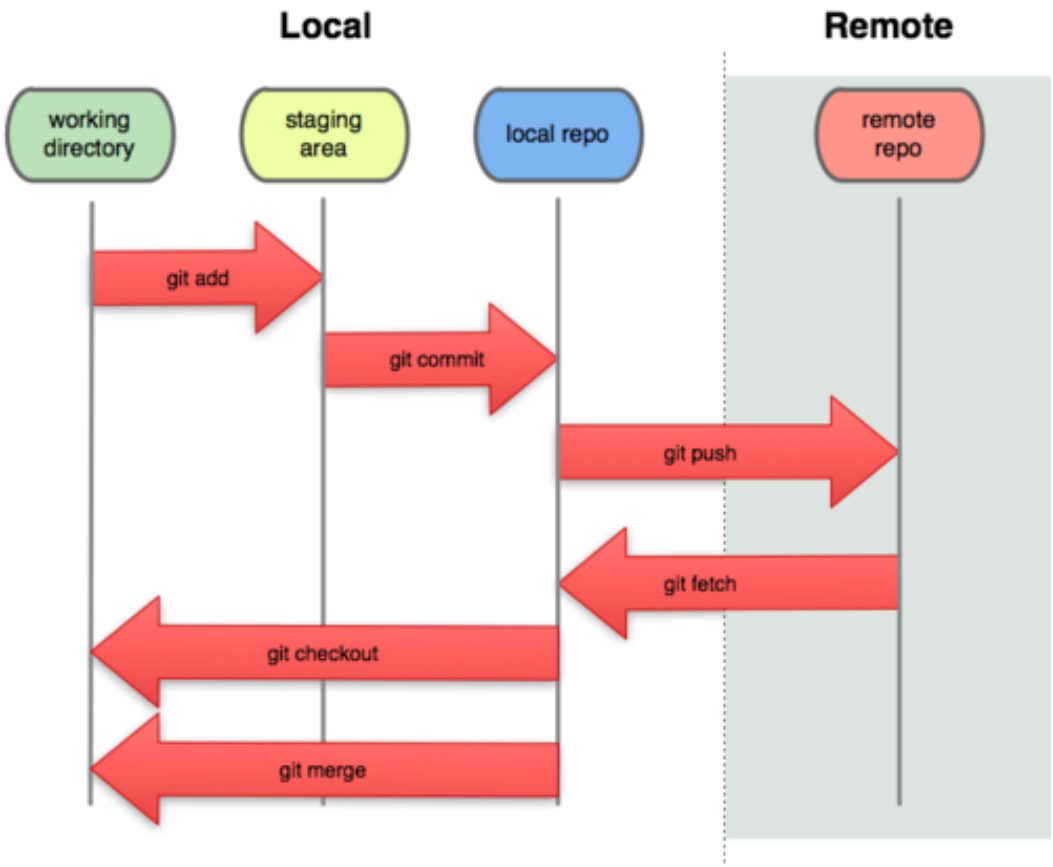
  - Esto puede suceder cuando un usuario, trabajando en esos ficheros, **actualiza** su copia local con los cambios realizados, y añadidos al repositorio, por otros usuarios. Análogamente, este mismo proceso puede ocurrir en el repositorio cuando un usuario intenta ***check-in*** sus cambios.
  - Puede 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.

Actualización (*"sync"* o *"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 (*"workspace"*)
: 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
: 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 está prevista y cuya solución dé lugar a efectos colaterales imprevistos.

Flujo de Git con Repositorio Remoto



Git / Github con Eclipse

¿Qué es GIT?

GIT es un software de control de versiones distribuido, que nos ofrece mayor versatilidad al tradicional software de control de versiones centralizado, como **SVN**.

Centralizado VS Distribuido

El software de control de versiones centralizado, únicamente tiene un servidor centralizado, donde se aloja el código. Todos los clientes trabajan sobre ese servidor de manera que si el servidor está caído, no es posible trabajar con el control de versiones.

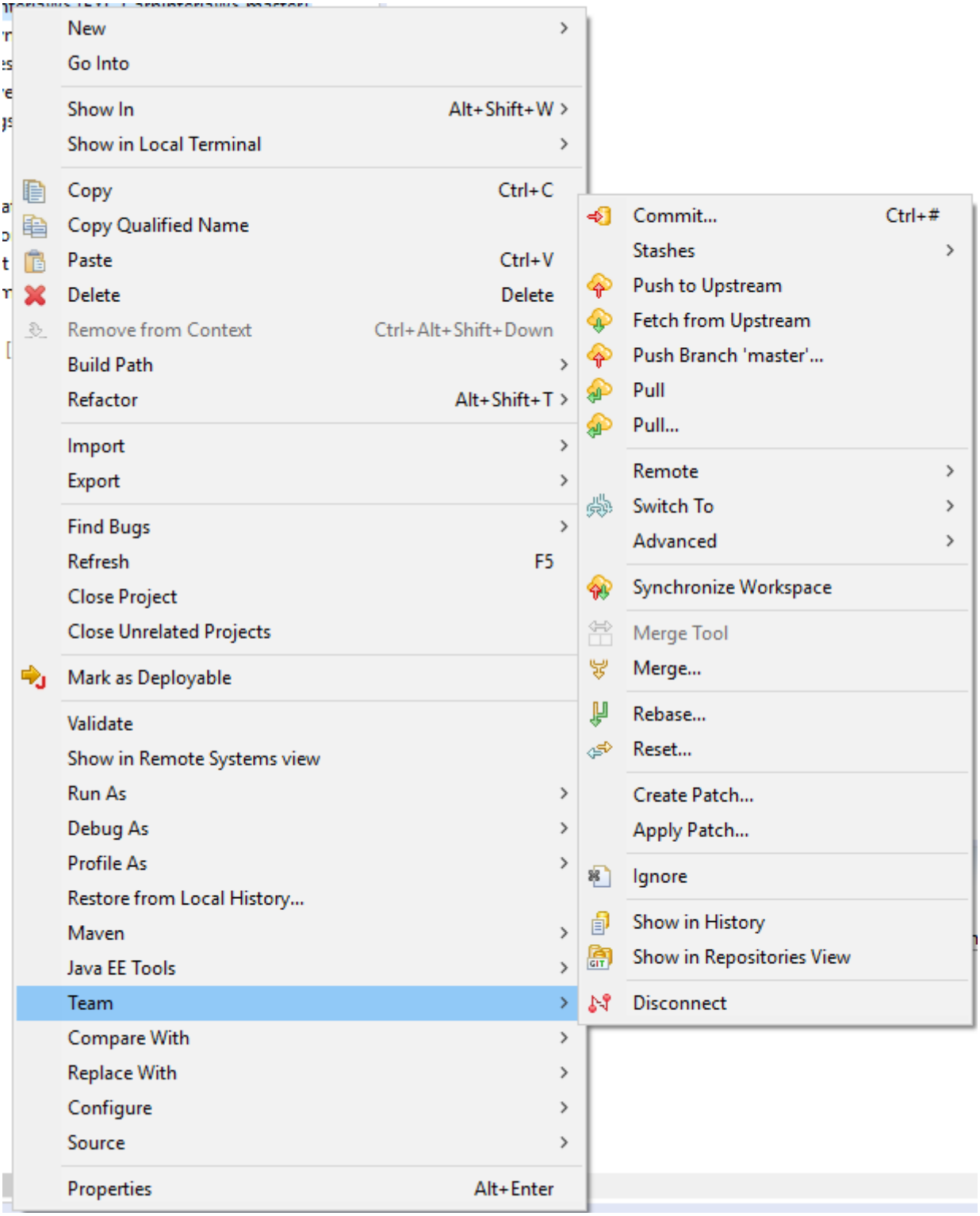
En el tipo de software distribuido, además de haber un servidor centralizado contra el que trabajan todos los clientes, cada uno tiene un servidor local, de manera que si el servidor centralizado falla, se puede seguir haciendo uso del control de versiones normalmente hasta que se restablezca.

GIT y ECLIPSE

La mayoría de los IDEs tienen integración con GIT, la versión para desarrolladores Java de Eclipse ya incluye el plugin de **GIT** por defecto para trabajar directamente con GitHub desde Eclipse, no obstante, si no lo tuviésemos instalado es tan sencillo como acceder al marketplace de ECLIPSE e instalarlo (egit).

Conceptos Básicos de GIT

Para acceder a todas las opciones de **GIT** desde Eclipse debemos pulsar el botón derecho del ratón sobre el proyecto o un archivo del mismo, acceder al menú **“TEAM”**, y aquí encontraremos todas las opciones para trabajar con **GIT**. Sólo aparecerán todas las opciones si ese proyecto ya está sincronizado con un repositorio git, sino sólo aparecerá la opción "share Proyect" para asociarlo.



Para sincronizar por primera vez un proyecto con un repositorio **GIT** tenemos dos posibilidades:

- **"Subir un proyecto a repositorio local y a GitHub", por primera vez:**

En este caso debemos pulsar sobre **"SHARE PROYECT"** y seguir los pasos indicados en pantalla.

- **"Importar un proyecto ya existente en GitHub a Eclipse":**

En este caso debemos acceder en el ECLIPSE al menu *File -> Import -> Projects from Git*.
Cuando importamos un proyecto **GIT** a nuestro workspace es importante marcar siempre la opción **"CLONE URI"** para que, además de importar el proyecto, nos genere un repositorio **GIT** local para ese proyecto.

Una vez realizado uno de estos pasos veremos que el menú **"TEAM"** estará mucho más poblado permitiéndonos realizar todas las operaciones propias de cualquier software de control de versiones.

Un repositorio **GIT** se compone de los siguientes elementos:

- **Rama Master:** Aquí se encuentra la rama principal del proyecto.
- **Branches:** Aquí se encuentran las ramas del proyecto, por ejemplo, cuando varias personas están realizando distintas modificaciones, se crearía un branch por cada funcionalidad y cuando estuviesen listos esos desarrollos en los branch, se reintegrarían a la rama master.
- **Tags:** Son "fotografías" en un instante de tiempo de la rama master o de algún branch. No debemos trabajar nunca contra los tags. Si necesitásemos hacer alguna modificación sobre algún tag se debe crear un branch a partir del tag y posteriormente crear un nuevo tag. Estas modificaciones además deberíamos incluirlas sobre el master para conseguir una integración continua.

Para movernos entre los elementos del repositorio antes mencionados debemos utilizar **"SWITCH TO"**.

Hay tres operaciones básicas que podemos realizar con **GIT**:

- **Pull:** Cuando hacemos pull de un proyecto, nos traemos todos los cambios existentes en el repositorio remoto (Github) a nuestro repositorio local. La primera vez deberemos usar "Pull ...". Si no funcionara, seguir los pasos de la Configuración manual del Pull (Fetch) en Eclipse.
- **Commit:** Con esta operación llevamos los cambios de nuestro workspace al repositorio **GIT** local.
- **Push:** Con esta operación llevamos los cambios de nuestro repositorio **GIT** local al repositorio **GIT** remoto (Github).

¿Cómo trabajar con GIT?

El uso habitual de **GIT** una vez tenemos sincronizado un proyecto es el siguiente:

Antes de empezar a trabajar **siempre haremos un pull** del proyecto para traernos todas las novedades del repositorio remoto (Github) y así evitar conflictos.

Cuando vayamos realizando modificaciones realizaremos commits de nuestro workspace sobre nuestro repositorio local.

Una vez terminado un bloque de desarrollo realizaremos un push sobre el repositorio remoto (también podemos hacer el push y el commit al mismo tiempo).

¿Qué hacer en caso de conflictos?

Hay tres tipos de conflictos, **al hacer push, al hacer pull, o al hacer merge (integrar)**:

Si tenemos algún conflicto a la hora de realizar algún **push**, los pasos que debemos hacer son los siguientes:

- Hacer un **commit** de nuestros cambios a nuestro repositorio local.
- Hacer un **pull** de las novedades en el repositorio central. EGit detectará el error y nos lo mostrará por pantalla (La mayoría de las veces **GIT** ya va a saber mergear las novedades del repositorio remoto con las de tu repositorio local y solucionará el problema), y en caso de que no pueda resolver el conflicto de manera automática, nos lo mostrará para que nosotros hagamos manualmente la integración de los cambios.
- Hacer un **commit**, y después **push** al repositorio central con "**PUSH TO UPSTREAM**".

Si el conflicto es al hacer **Merge** (fusionar dos desarrollos) una rama contra otra, por ejemplo, reintegrar un branch (rama) con un nuevo desarrollo en el master, debemos hacer lo siguiente:

- Situarnos en el master si no lo estamos ya con **"SWITCH TO"**.
- Mergear con la opcion **"MERGE"** , seleccionando la rama que queremos reintegrar. Revisar **Merge (fusionar) dos ramas**.
- Si al hacerlo surge un conflicto (lo mismo que si surge al hacer un **Pull**) que el software de control de cambios no sepa resolver por si solo, en los ficheros que generan el conflicto se marca el mismo de la siguiente forma:

<<<<<<<<<< Rama Origen

Código Rama Origen

===

Código Rama Destino

>>>>>>>> Rama Destino

Ejemplo:

public class Main {

public static void main(String[] args) {

//creamos un objeto de la clase ClaseRecursiva

ClaseRecursiva objeto= new ClaseRecursiva();

//Ejecutamos y sacamos por pantalla el mÃ©todo recursivo

System.out.println(objeto.potenciaR(2,3));

<<<<<<< HEAD

//Ejecutamos y sacamos por pantalla el método iterativo

System.out.println(objeto.potencial(2,7));

=======

//Ejecutamos y sacamos por pantalla el mÃ©todo iterativo

System.out.println(objeto.potencial(2,5));

>>>>>>> branch 'Prueba_rama' of https://github.com/mariluzfortea/Clase.git

}

}

De forma que debemos escoger si quedarnos con el código de la rama origen, destino, o hacer nosotros una mezcla.
Una vez tomada la decisión dejamos el código elegido resolviendo el conflicto.

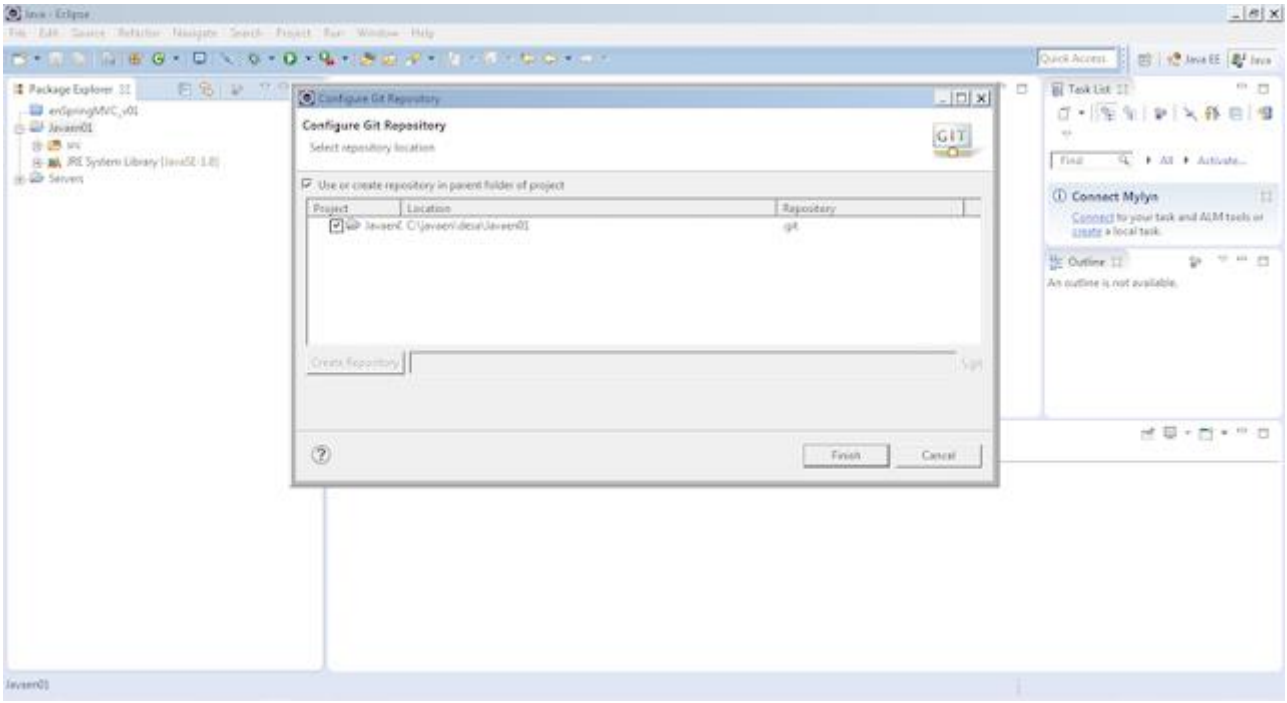
Cuando cerremos el fichero haremos "**add to index**", **commit**, y **push** para actualizar el repositorio.

Subir un proyecto a repositorio local y a Github

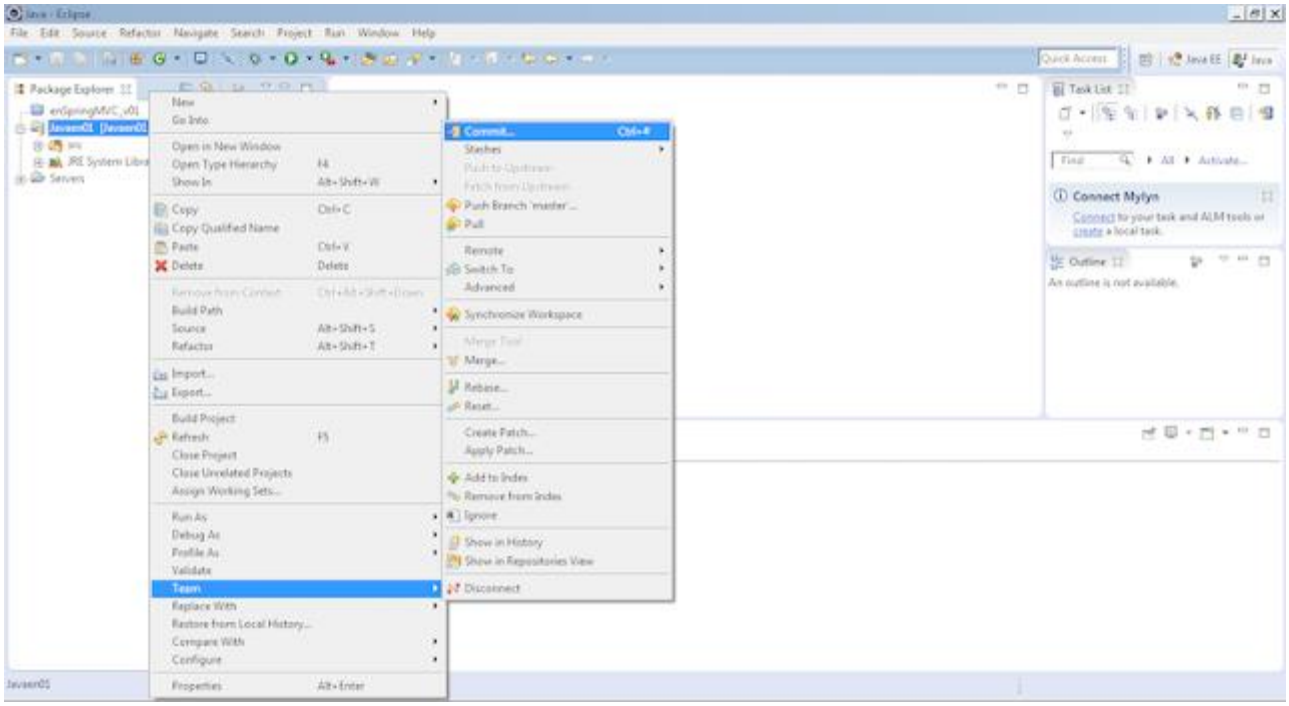
Lo conectamos primero a un **repositorio local** de **Git**, a diferencia de la conexión a **GitHub** que es un **repositorio remoto**.

El **plugin del eclipse** se encargará de **crear** este **repositorio local de Git**

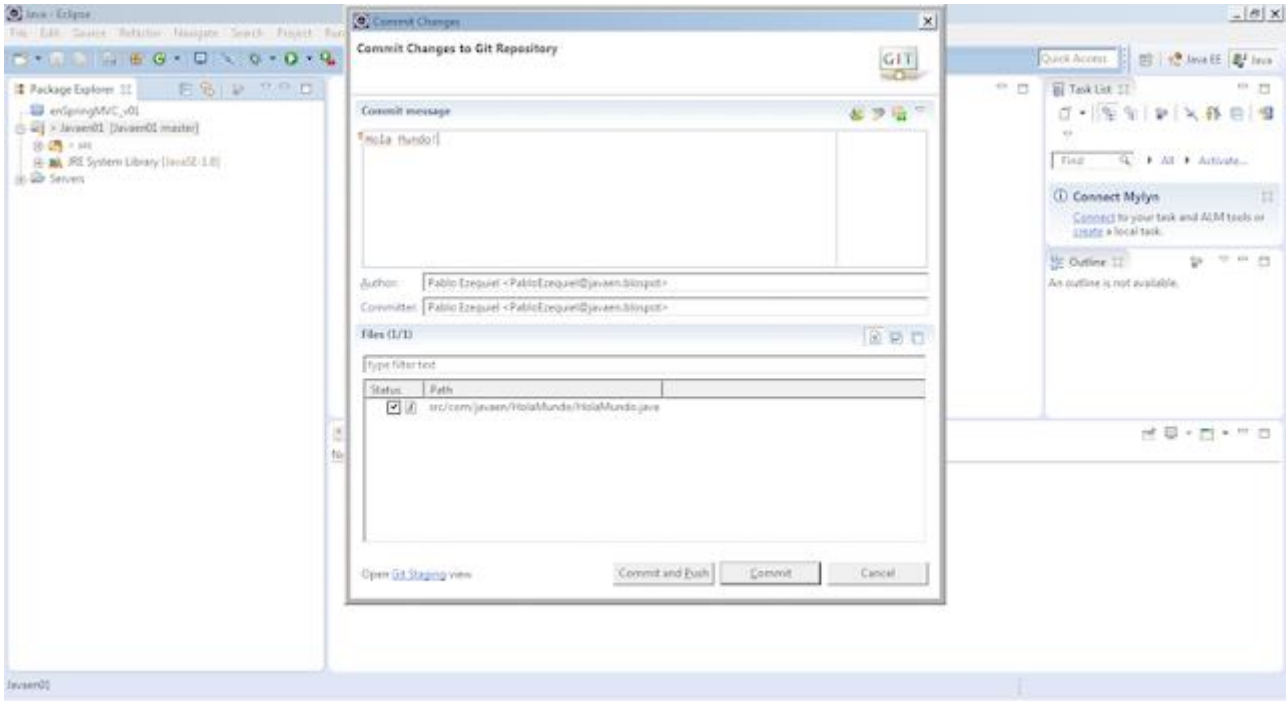
Pulsamos botón derecho encima del proyecto que queremos subir, elegimos **Team > Share Project** dentro del eclipse:



Al finalizar, veremos que todo lo que hay dentro de ese proyecto aparecerá con un signo de interrogación "?" por delante, que lo que nos indica es que no están aún ni el repositorio local, ni en remoto. Volvemos a pulsar botón derecho encima del proyecto (o el fichero que queramos subir a los repositorios), hacemos **Team > Commit**

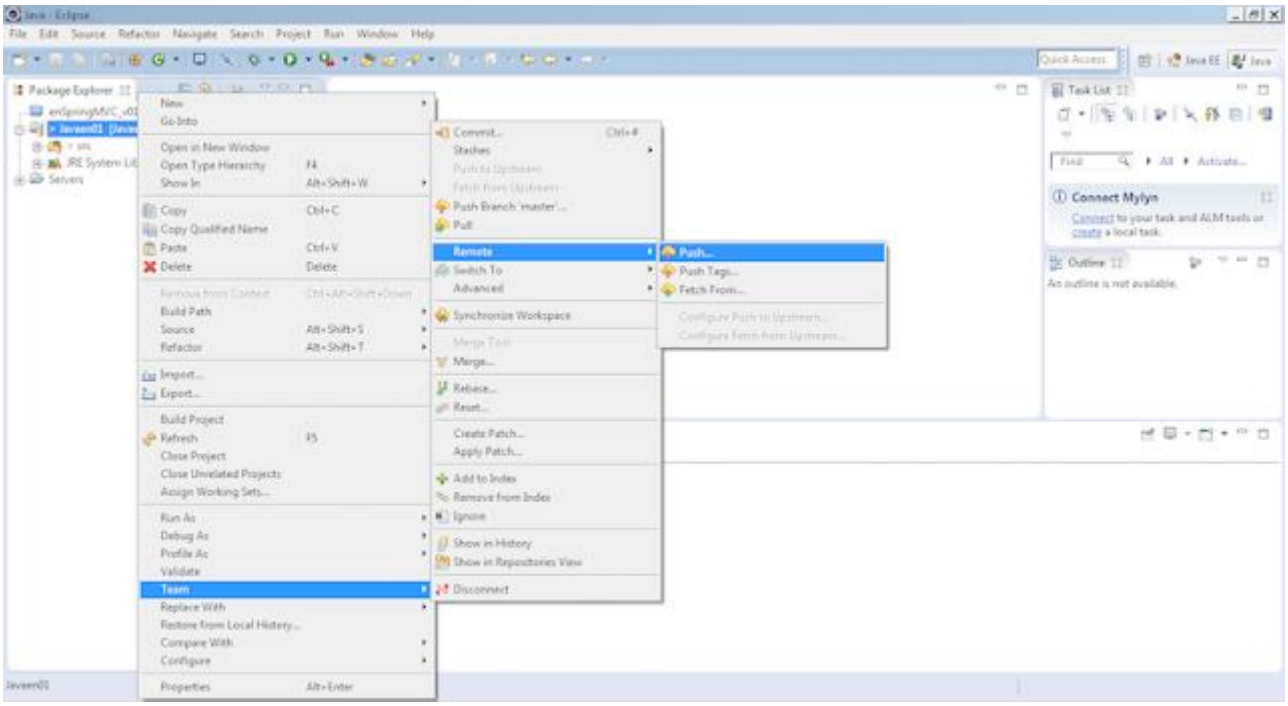


Agregando una descripción por el cambio para que quede registrado qué se hizo en esta versión de código:



Ahora ya está guardado en nuestro repositorio local, y habrán desaparecido los signos de interrogación de los archivos. Si queremos pasar esos cambios a nuestro repositorio remoto (Github), deberemos hacer un **push**.

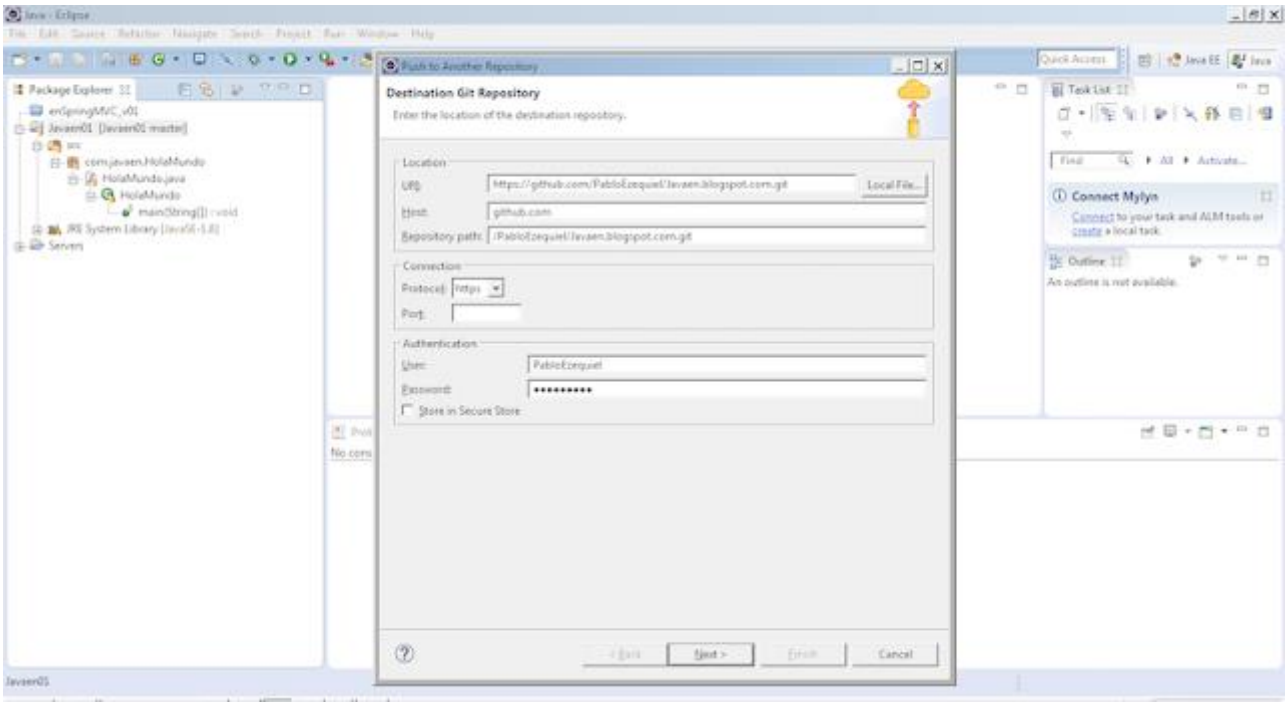
Hacemos el **push** para subir los cambios al **directorio remoto** de **GitHub**, pulsando botón derecho **Team > Remote > Push**



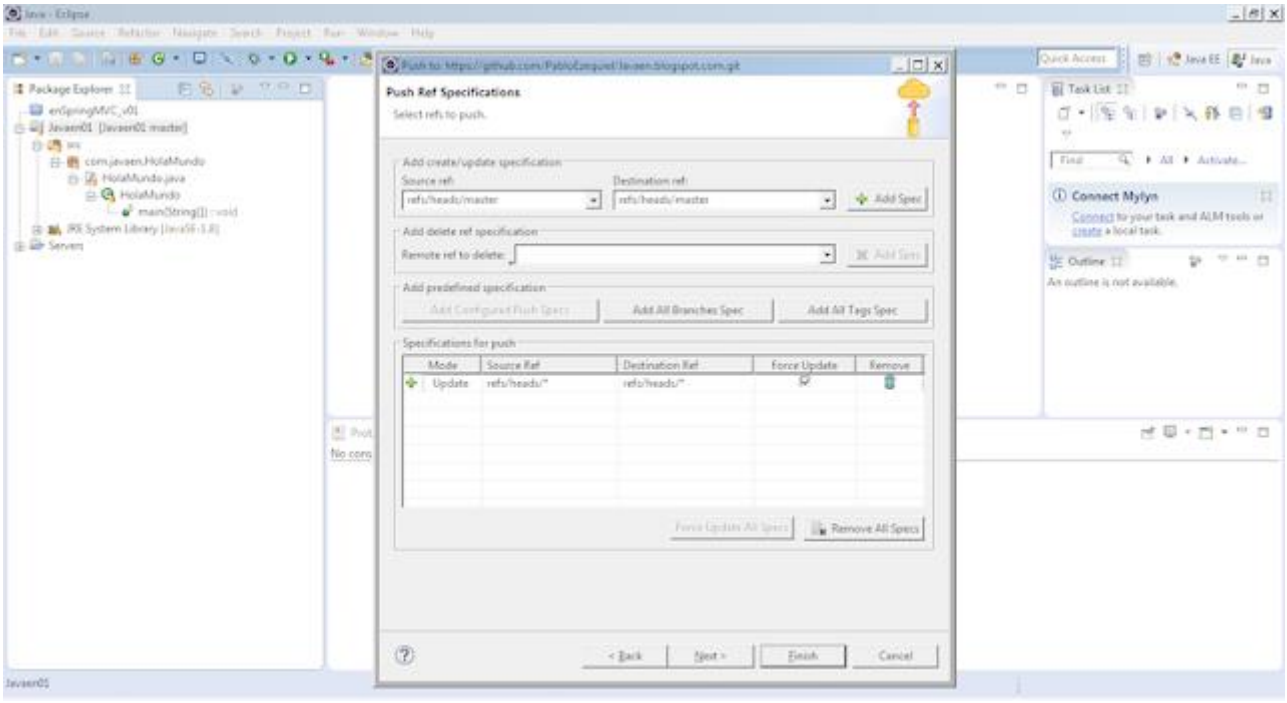
Y en la URL indicamos el de **nuestro repositorio** de **GitHub** (dentro de Github y de nuestro repositorio, pinchando en el botón **"clone or download"** copiamos la url del repositorio.

Y también debemos indicar nuestro **usuario y password** de nuestra cuenta de **GitHub** (si queremos que se conserve el usuario y pass deberemos pinchar el check que lo guarda permanentemente).

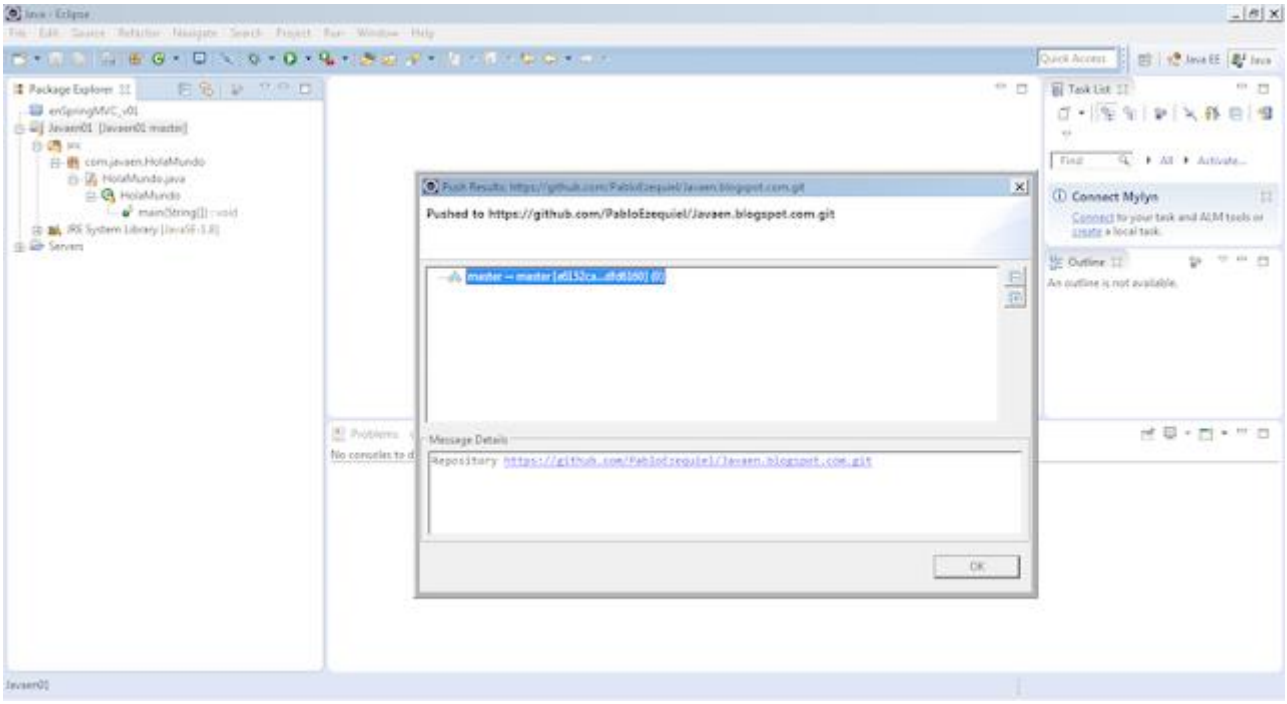
El plugin luego se encarga de completar el resto de los campos.



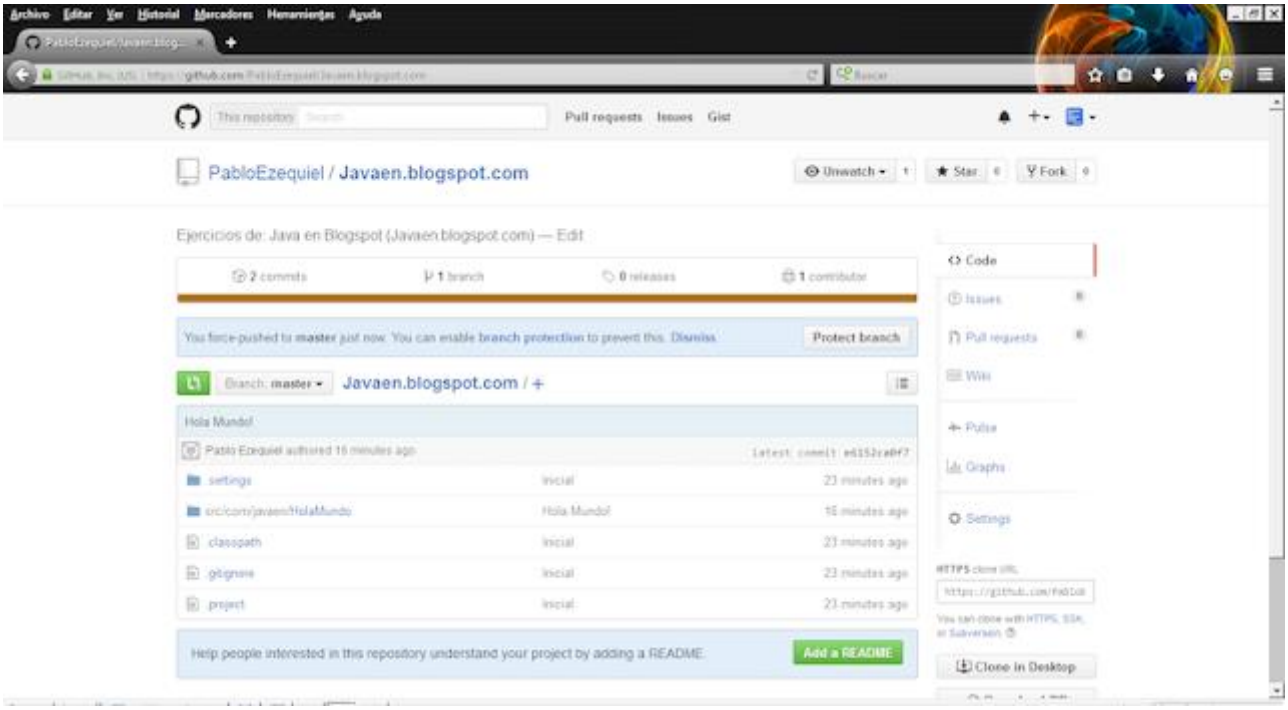
Seleccionamos la rama o ramas que queremos subir, o pulsamos en el botón que añade todas las ramas, y/o todos los tags:



Y presionando **Finish**, el proyecto se sube a GitHub:



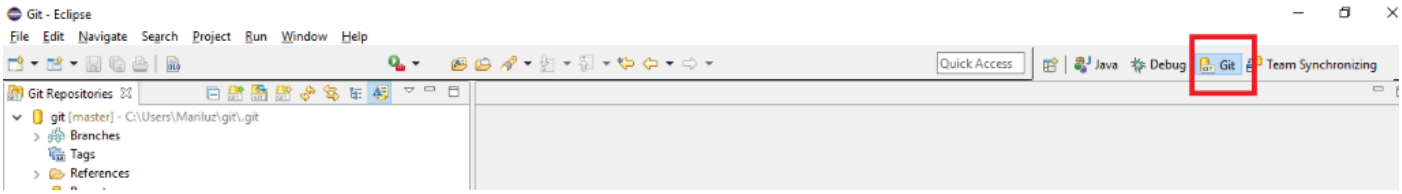
Si vamos a GIthub y refrescamos, veremos que ya aparece todo lo subido, y además con el comentario de commit que nosotros hayamos puesto (se verá el último comentario commit, si se hicieron varios commit y no se hizo push en todos, aparecerá el último, aunque todos se habrán quedado guardados).



Configuración manual del Pull (Fetch) en Eclipse

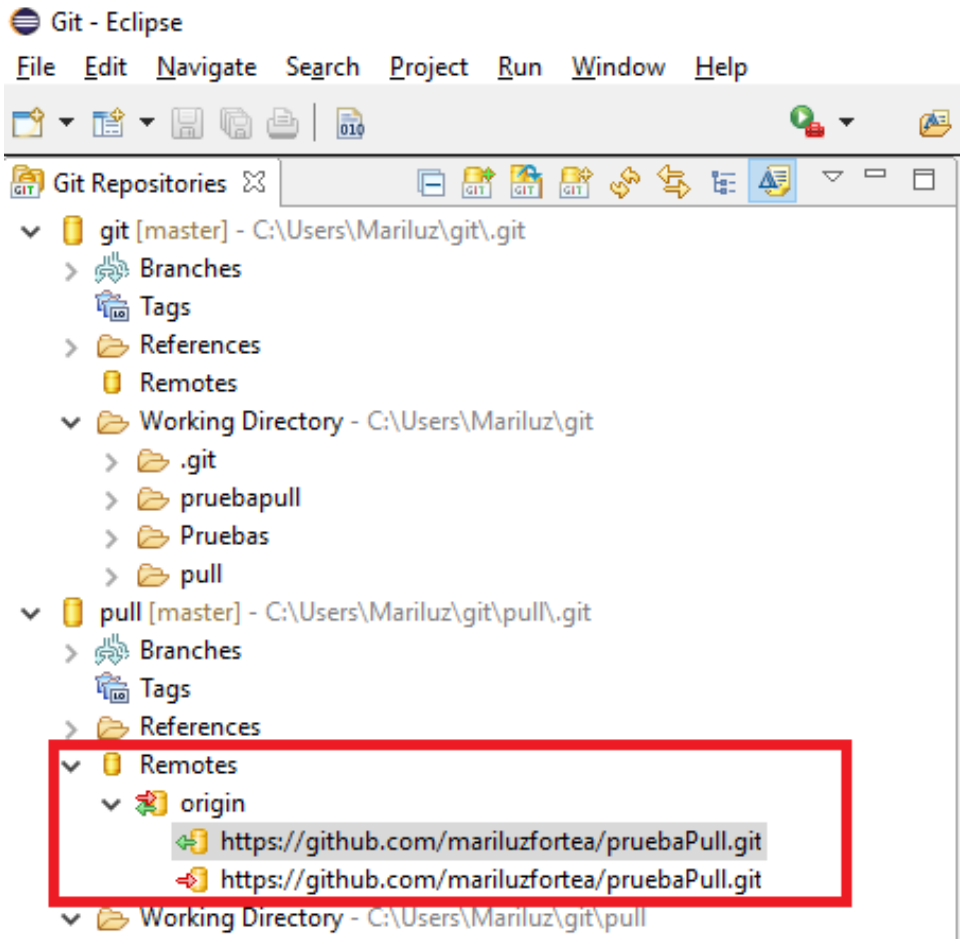
Para aquellos que no os funciona el "Pull" cuando vais a hacer pull de un repositorio por primera vez, hay que utilizar la opción **"Pull ..."** en lugar de **"Pull"**. Si aún así no os funciona, esta es la manera de configurarlo manualmente:

1. Vamos a la vista de Git en Eclipse

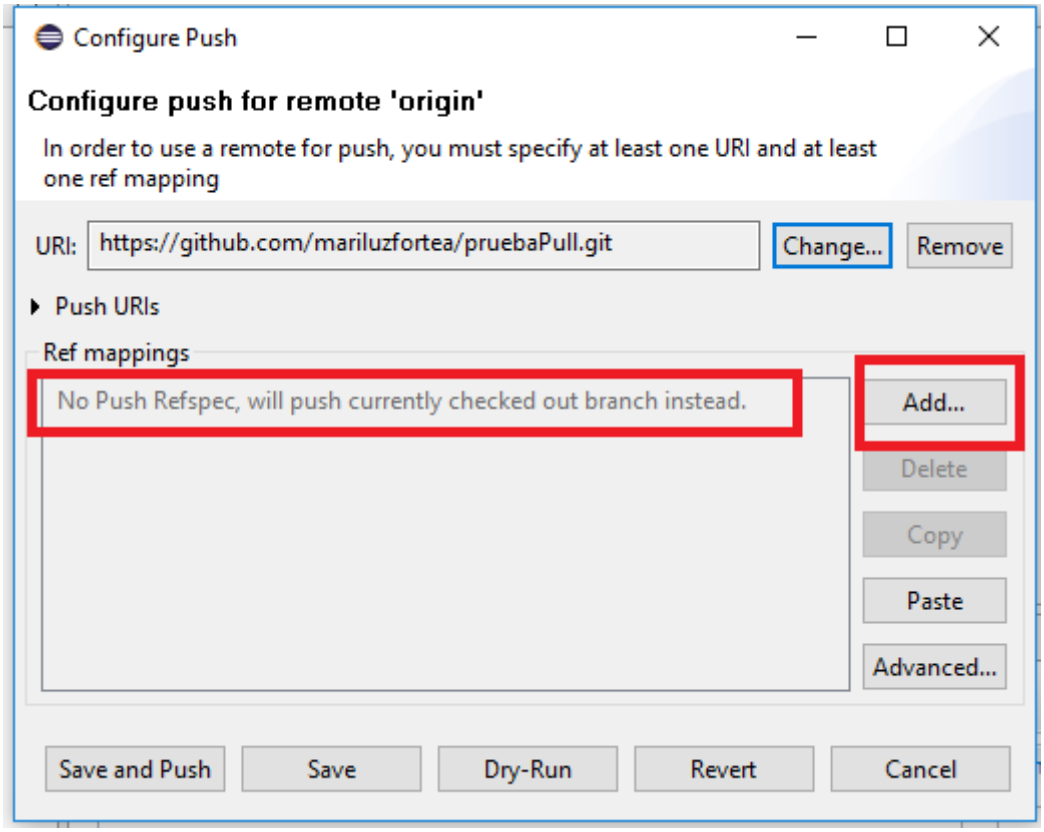


2. Vamos al proyecto que queremos hacer pull, y lo desplegamos. Uno de los componentes que aparecen es "Remotes", seguramente estará vacío, y ese es el problema, que no se ha quedado guardado el flujo de subida y bajada desde Github. **Si ese es el caso, pasa directamente al punto 3**.

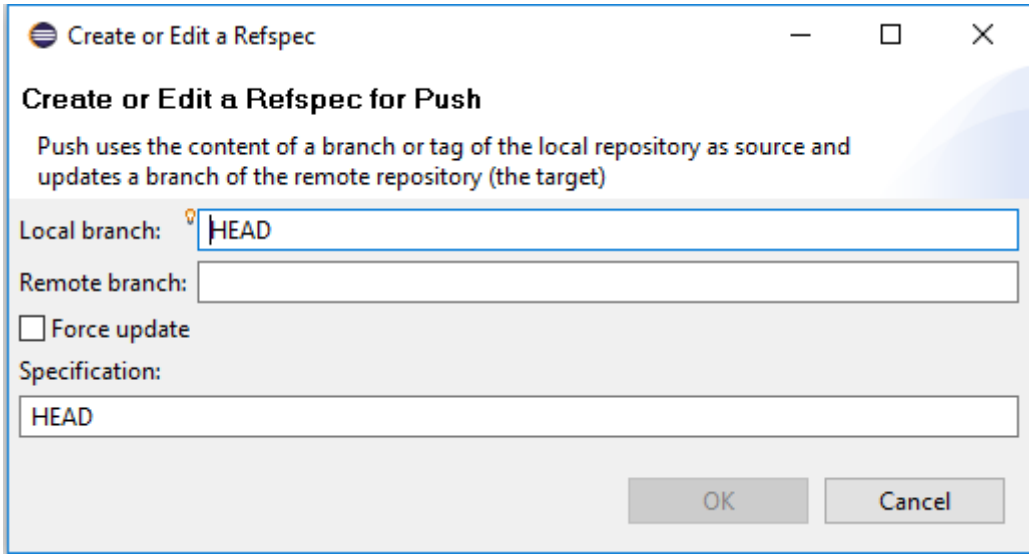
Si no es ese el problema, porque al desplegar “Remotes” aparece otro llamado "origin", desplegamos y aparecen los dos flujos de trabajo con Github, el primero el de bajada, y el segundo el de subida, entonces es que están mal configurados, y hacemos lo siguiente:



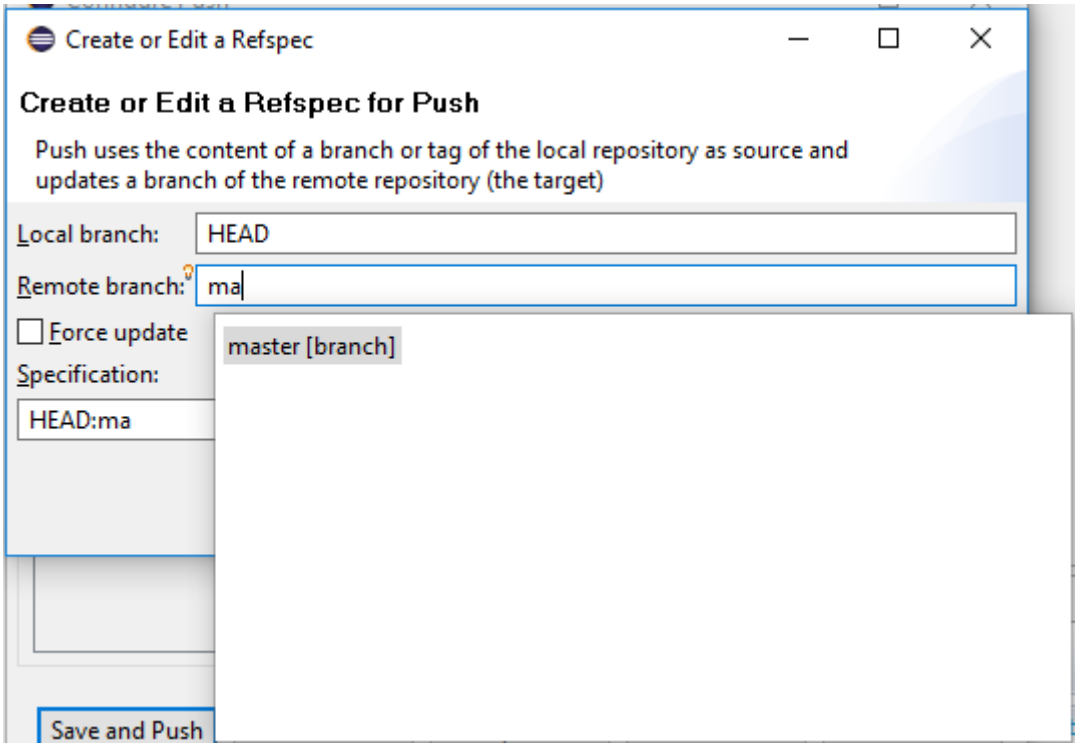
Click botón derecho encima del flujo de bajada, y seleccionamos “Configure Fetch”, vemos que el problema está en que está cogiendo correctamente el repositorio de Github, pero no la rama. Pulsamos en Add, para seleccionar la rama que debe usar para hacer el pull:

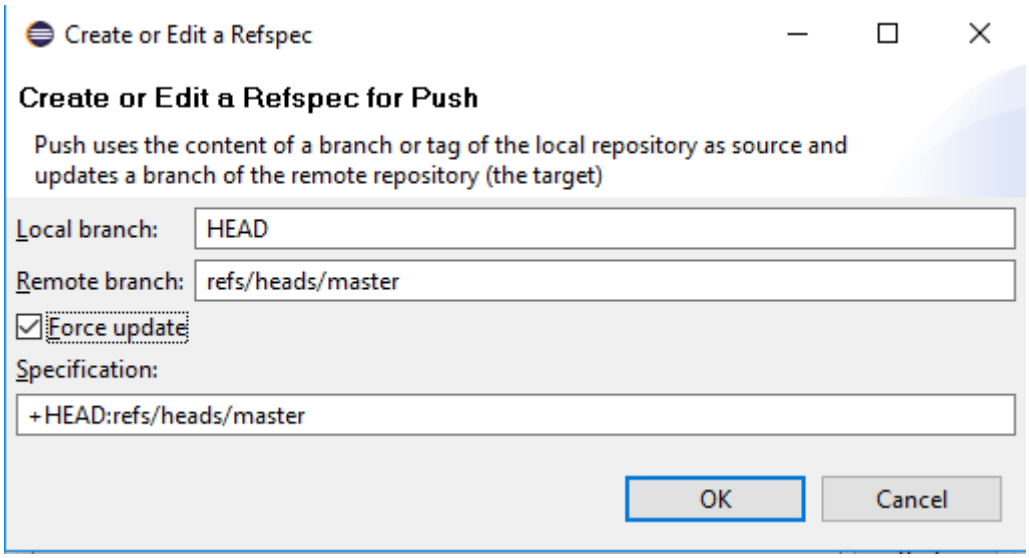


Nos aparecerá una pantalla como esta, en la que no hay nada seleccionado, ni tampoco se puede desplegar ninguna opción:

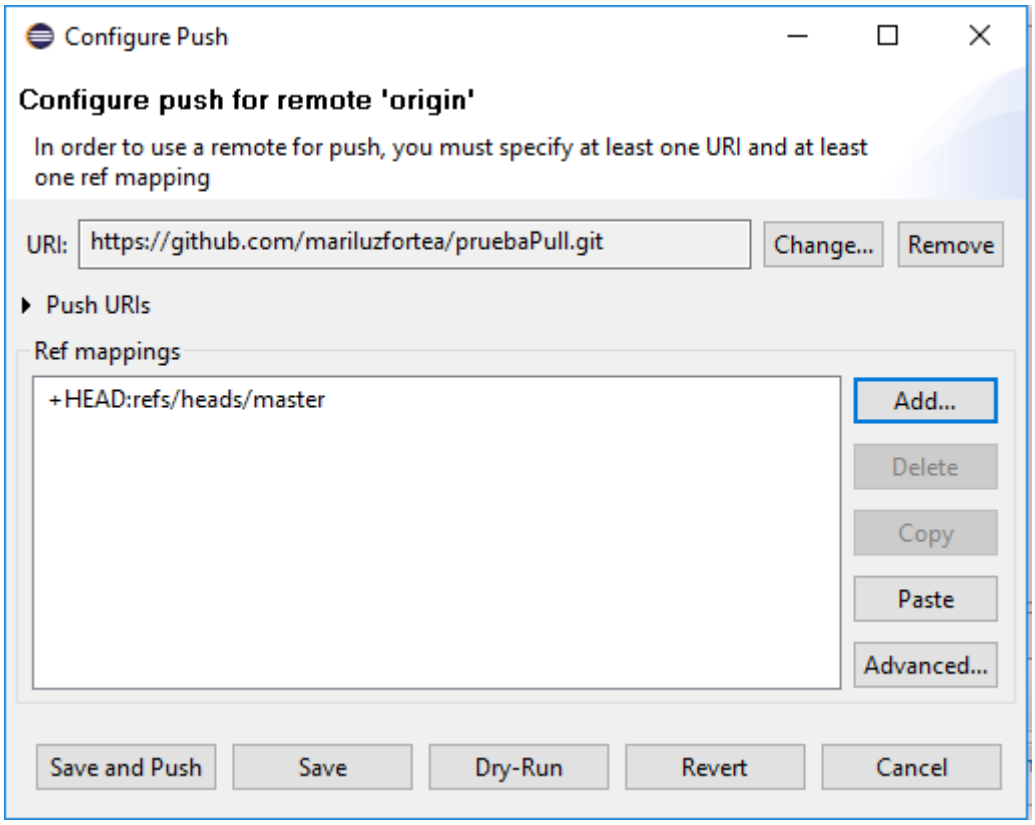


Tendremos que ponernos en el campo de Remote branch, y poner una “m”, y así nos aparecerá la opción de seleccionar “master [branch]”. La seleccionamos, seleccionamos también que el check de “Force update”:



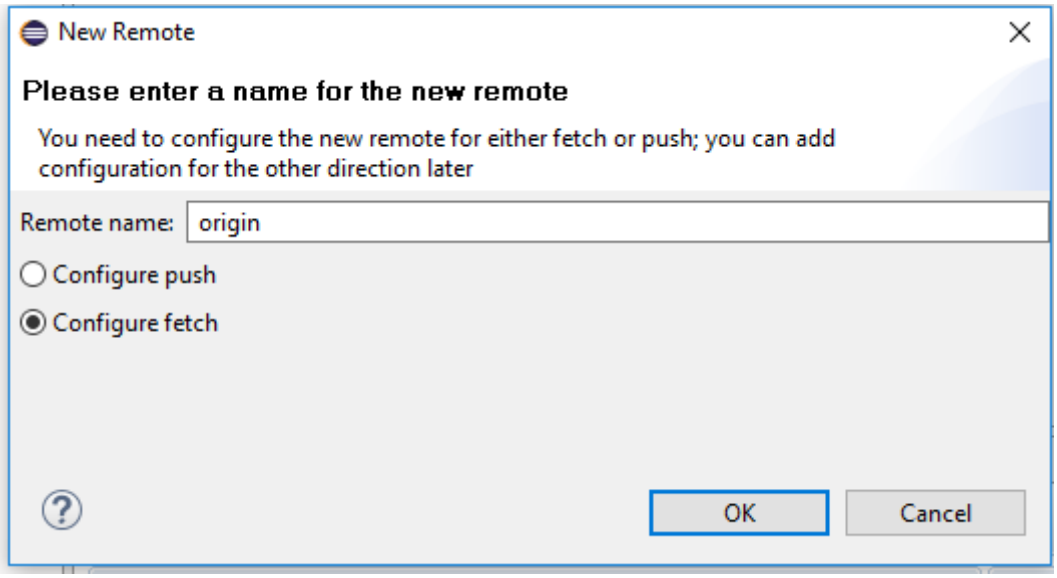


Al pulsar OK, ya nos aparecerá en la ventana anterior, seleccionada la rama para poder hacer el fetch (que nos configura el pull):

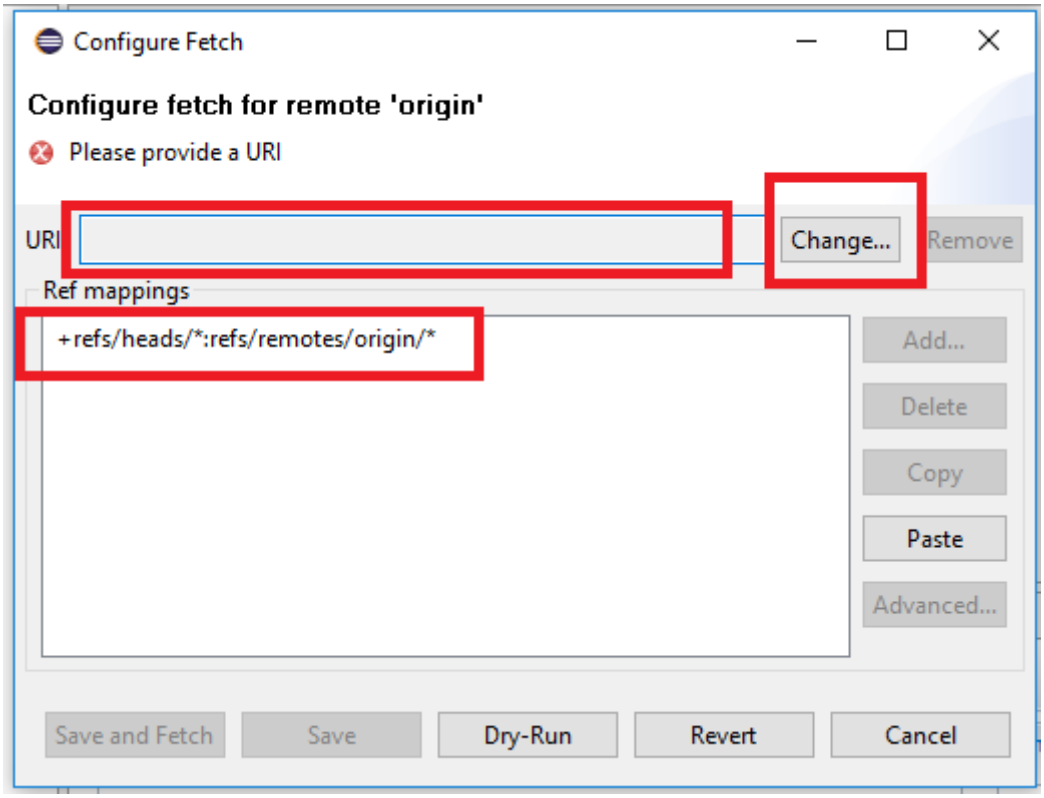


Pulsamos “Save”.

3. Botón derecho en “Remotes”, pulsamos en “Create Remote”, aparecerá la ventana para crear el flujo, seleccionamos Configure fetch:



Posiblemente nos aparecerá la pantalla de la siguiente manera, con la ruta branch seleccionada, pero no la url, pinchamos entonces en “Change” de la url:



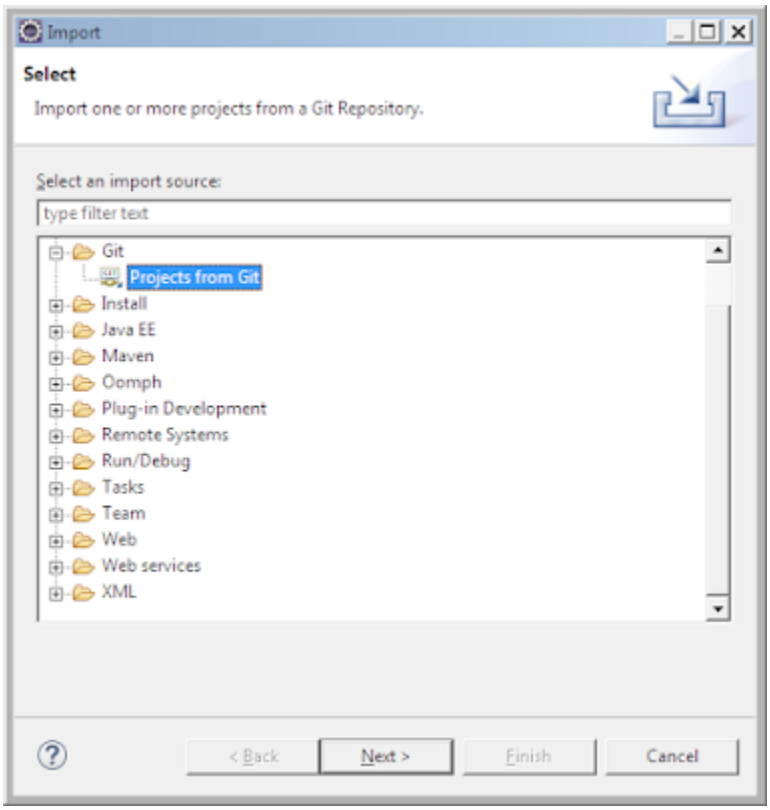
Nos abre la ventana de configuración de url, que debemos rellenar con todos los campos correctamente igual que en el push.

Ahora ya nos aparecerá correctamente toda la ventana, y pulsamos en “Save and Fetch”.

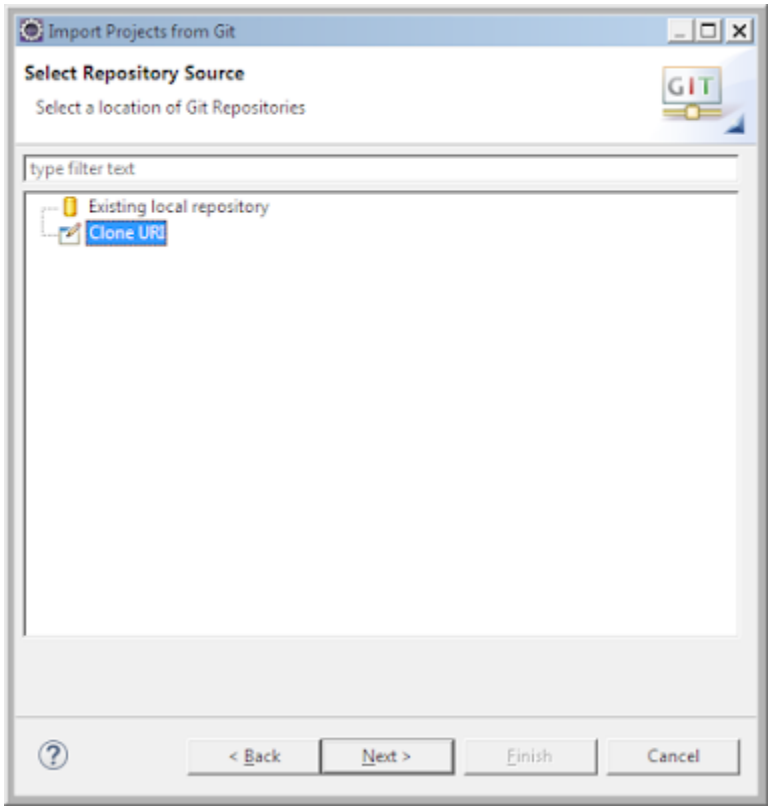
Nos aparecerá entonces la ventana con el resultado del Fetch (debería ser todo correcto), y ahora ya podemos usar el pull siempre que necesitemos actualizar desde Github.

Importar un proyecto ya existente en Github a Eclipse

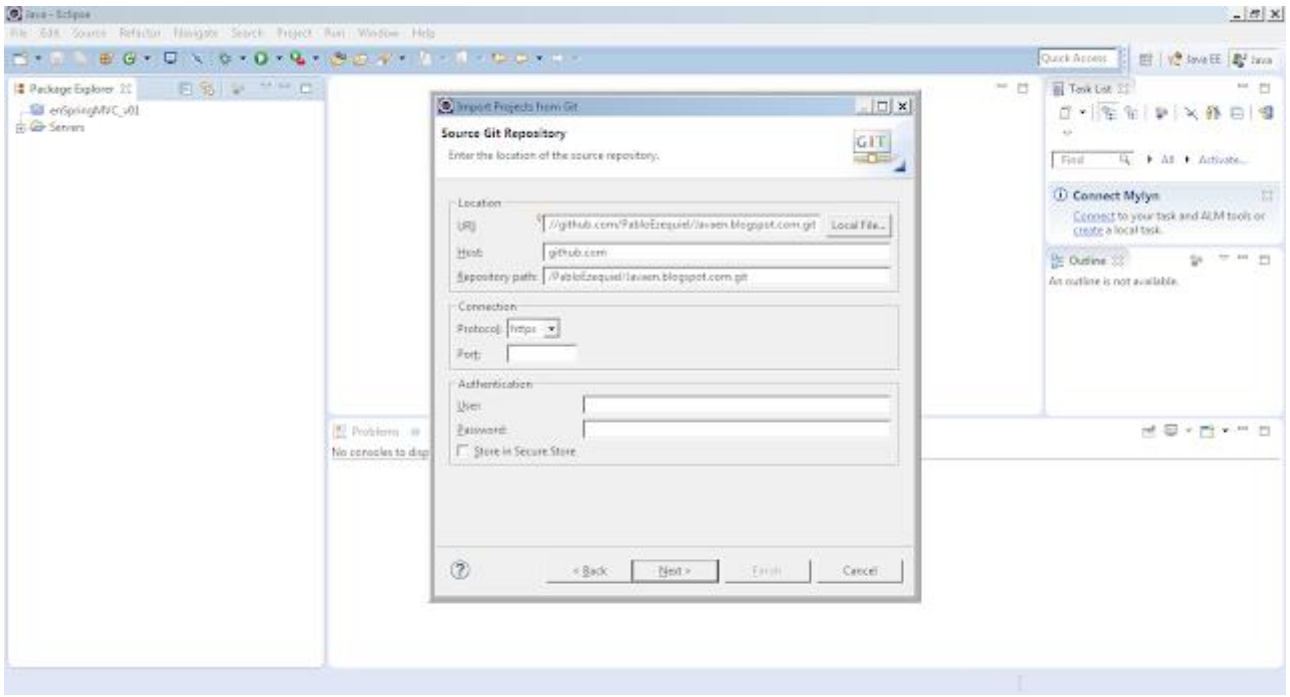
Desde el menú de Eclipse: **File > Import > Git > Project From Git**



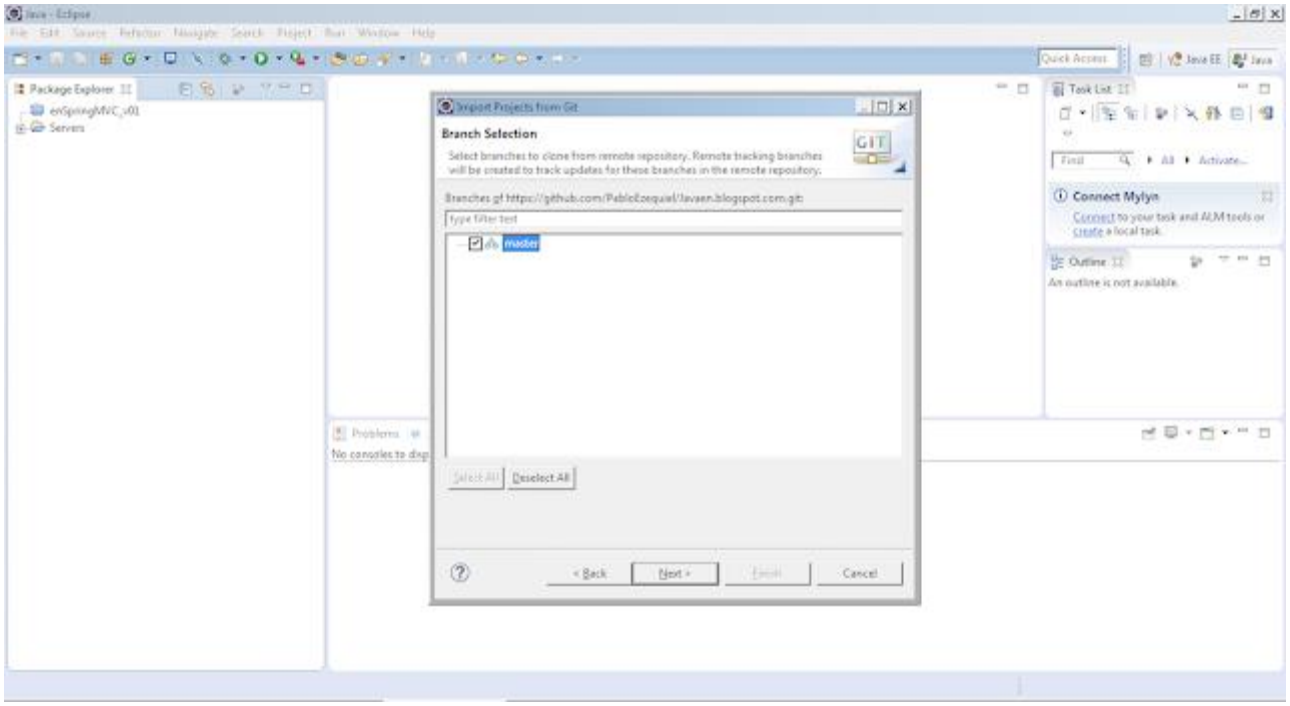
Presionamos Next y luego **Clone URL:**



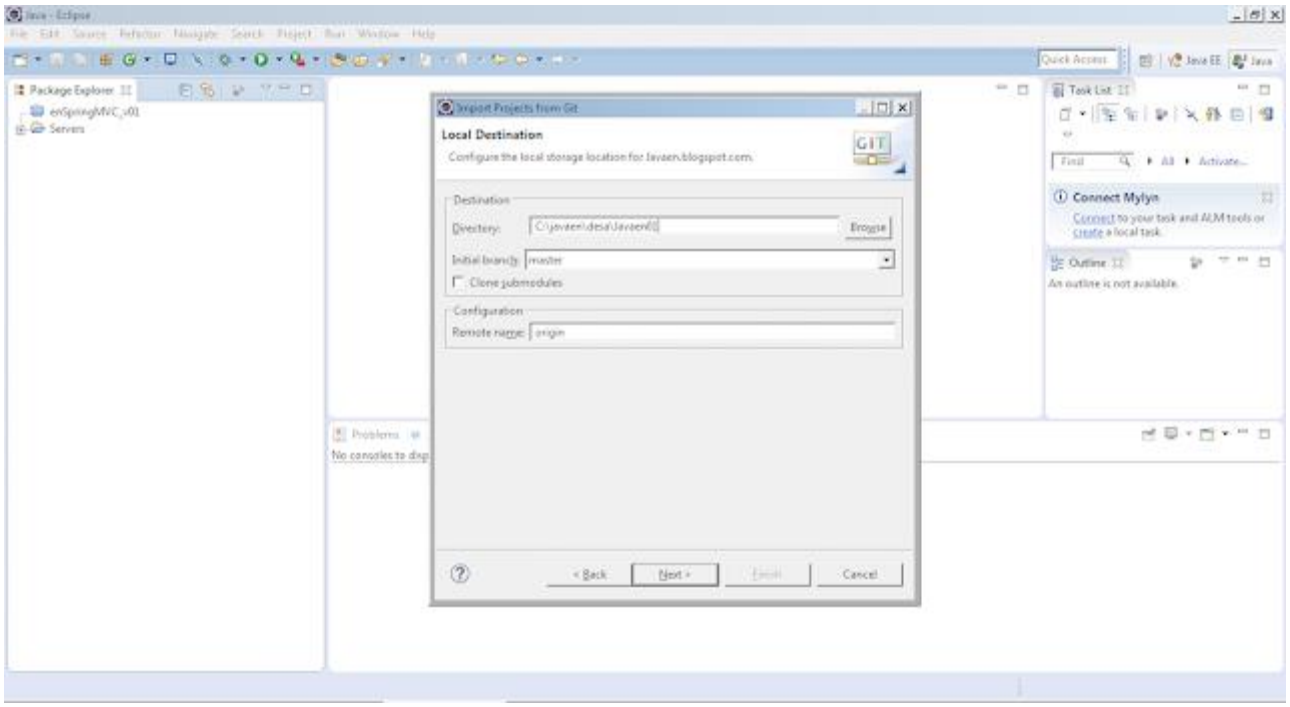
Pegamos la URL de nuestro repositorio Github (botón "**Clone or download**") y ponemos el usuario y contraseña de Github, y pulsamos Next:



seleccionamos **master**:

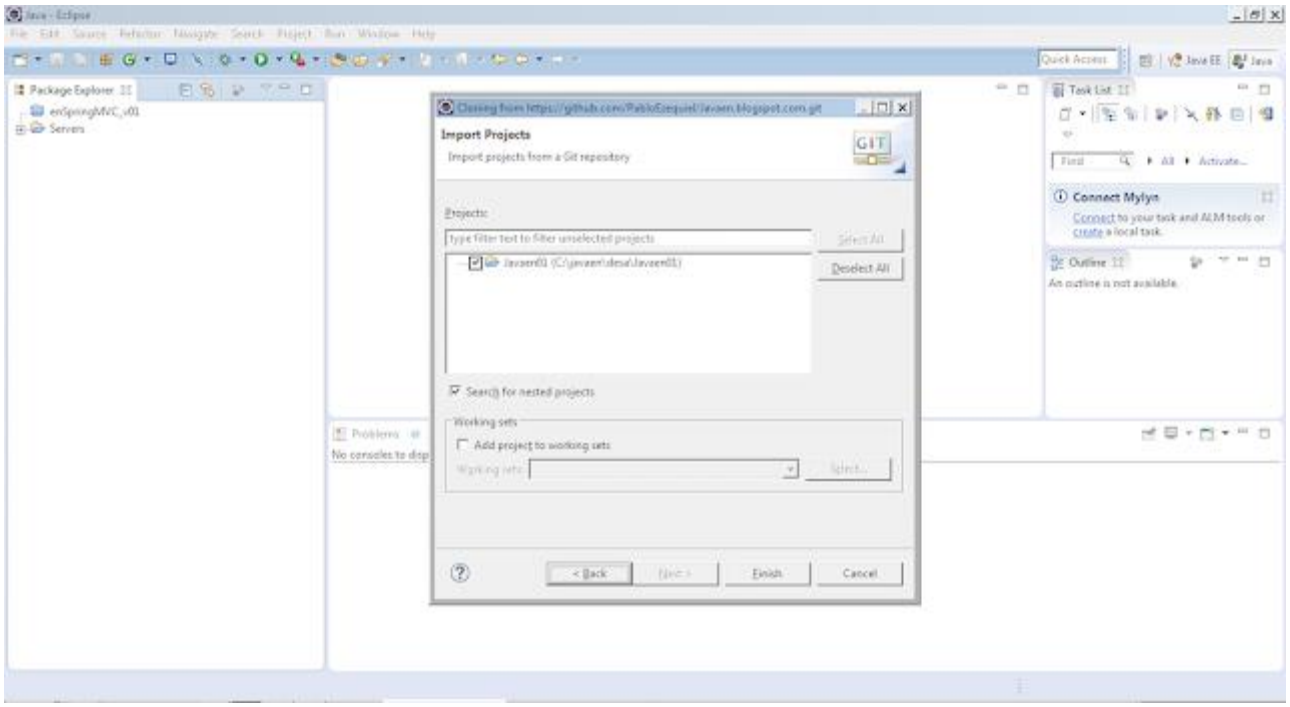


y elegimos el directorio donde se bajar el proyecto (**es importante que no haya un proyecto en nuestro workspace que se llame igual que el que estamos importando, porque sino causará un conflicto**):

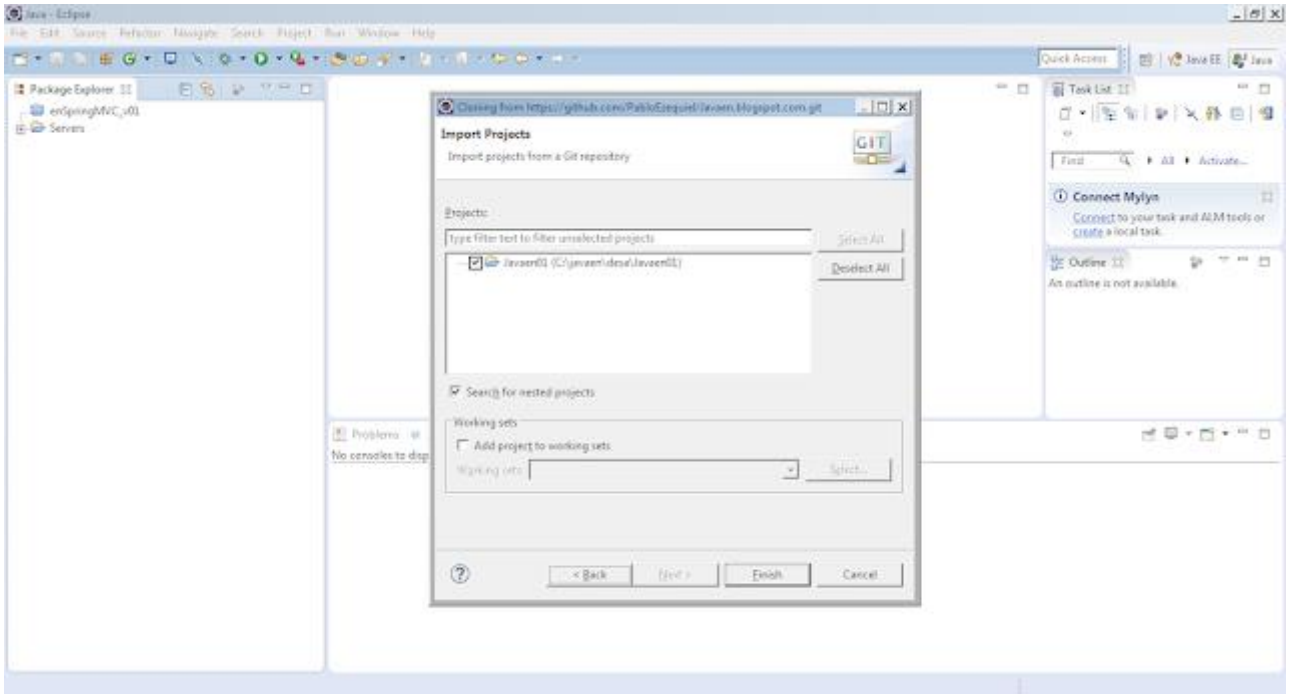


En la siguiente ventana seleccionamos la primera opción:

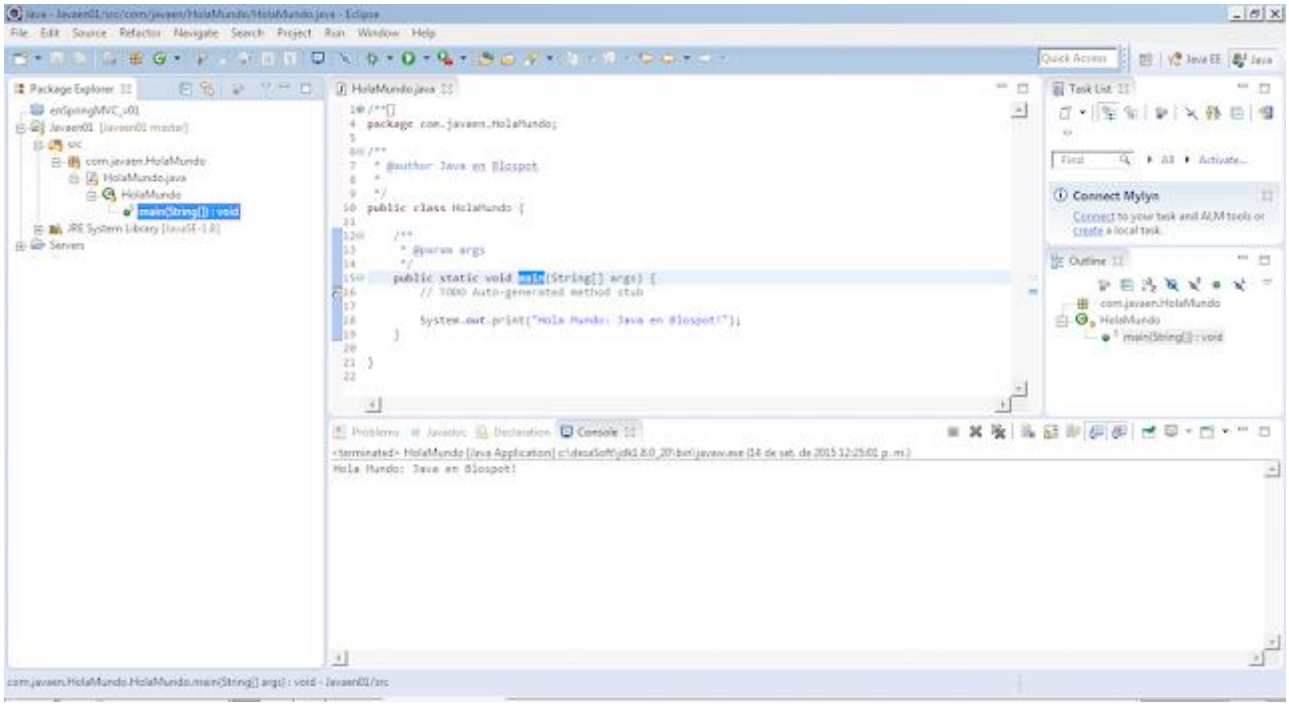
Presionamos Next:



Presionamos Next:



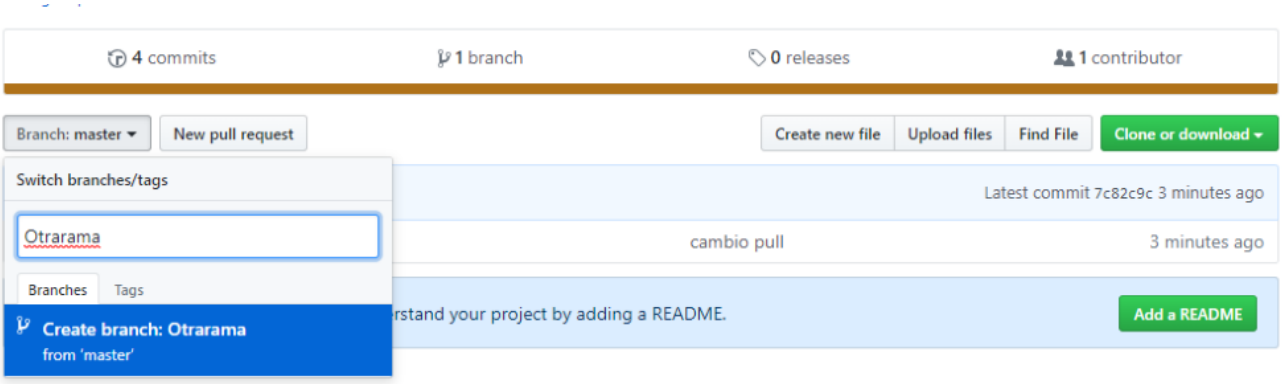
Y finalmente tenemos nuestro proyecto Java de Github descargado y conectado con el workspace y repositorio local.



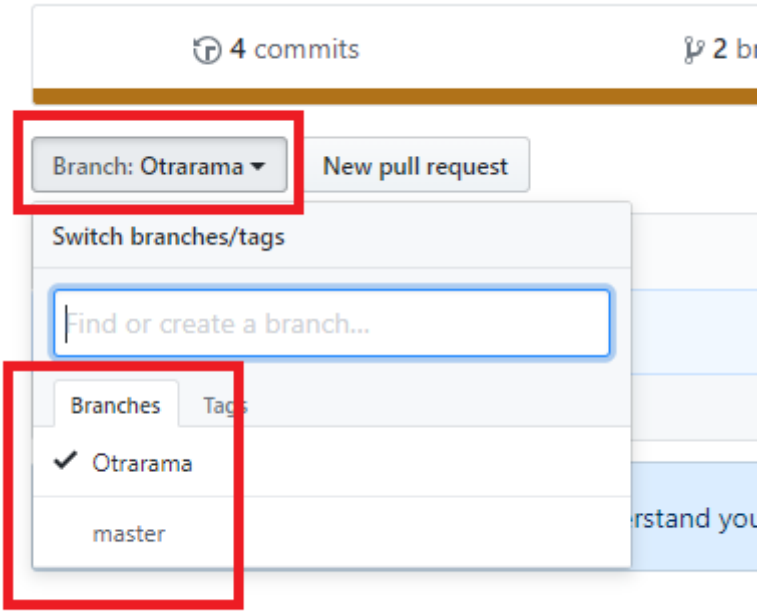
Abrir una rama y cambiar de rama (Github y Eclipse)

Hay dos maneras de abrir una rama trabajando con Github:

La primera es abrir la rama en Github, dentro de nuestro repositorio, pinchando en el desplegable Branch que nos indica en que rama estamos actualmente, podemos dar de alta otra rama de nuestro código:



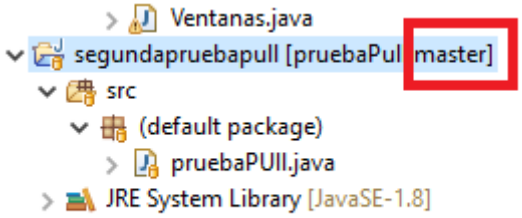
Una vez creada, veremos que estamos posicionados en la nueva rama, y que contiene lo mismo que contiene la rama principal (la ha clonado para poder trabajar con ella independientemente de la principal):



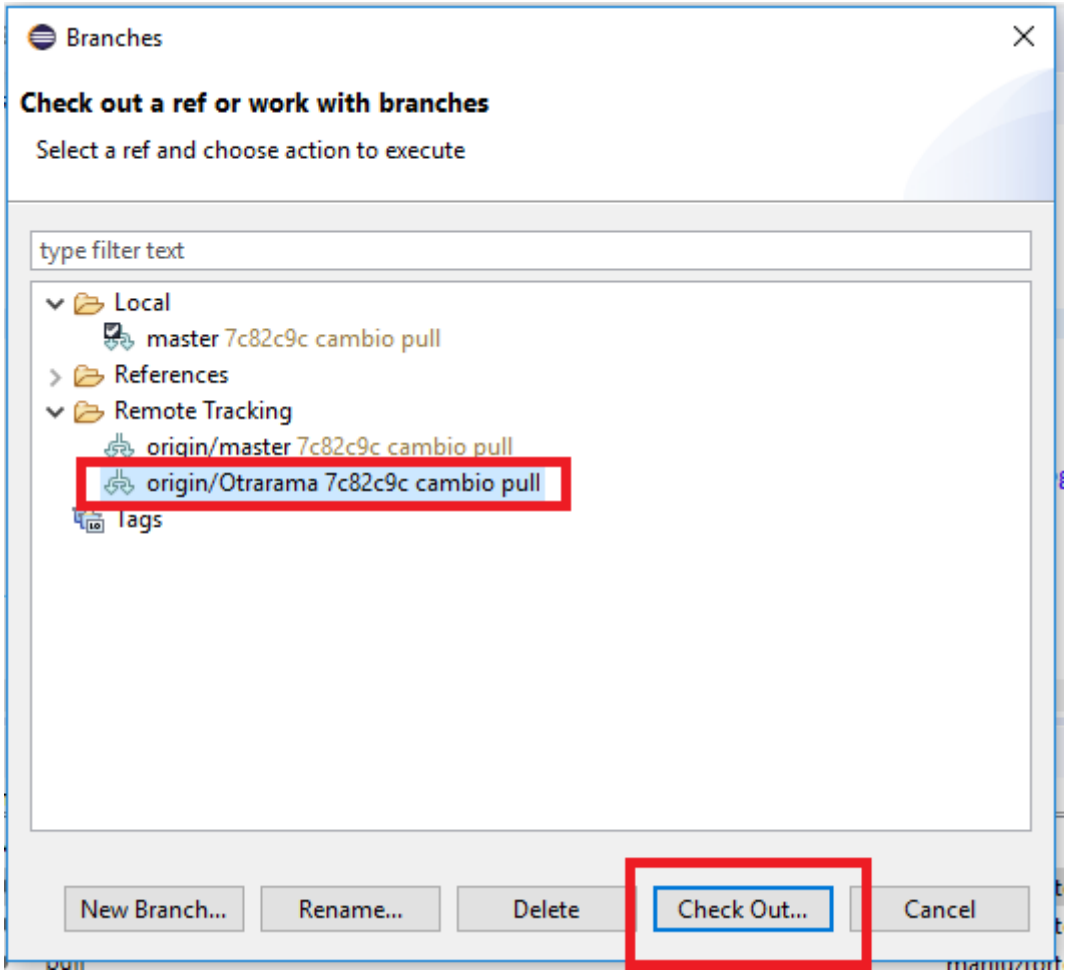
Si queremos cambiar de una rama a otra para visualizarla o hacer cambios sobre ella, tendremos que hacerlo siempre en ese desplegable seleccionando la rama que queremos ver.

Ahora tendremos que actualizar esa rama en nuestro repositorio local en Eclipse, y para ello haremos un Pull que como sabéis nos actualiza el repositorio con todas las novedades que haya habido sobre el mismo en GIthub.

Importante: Al hacer Pull, nos ha descargado la nueva rama al repositorio local, pero no la tenemos disponible para utilizar hasta que no hagamos checkout para crear esa nueva rama en nuestro repositorio:

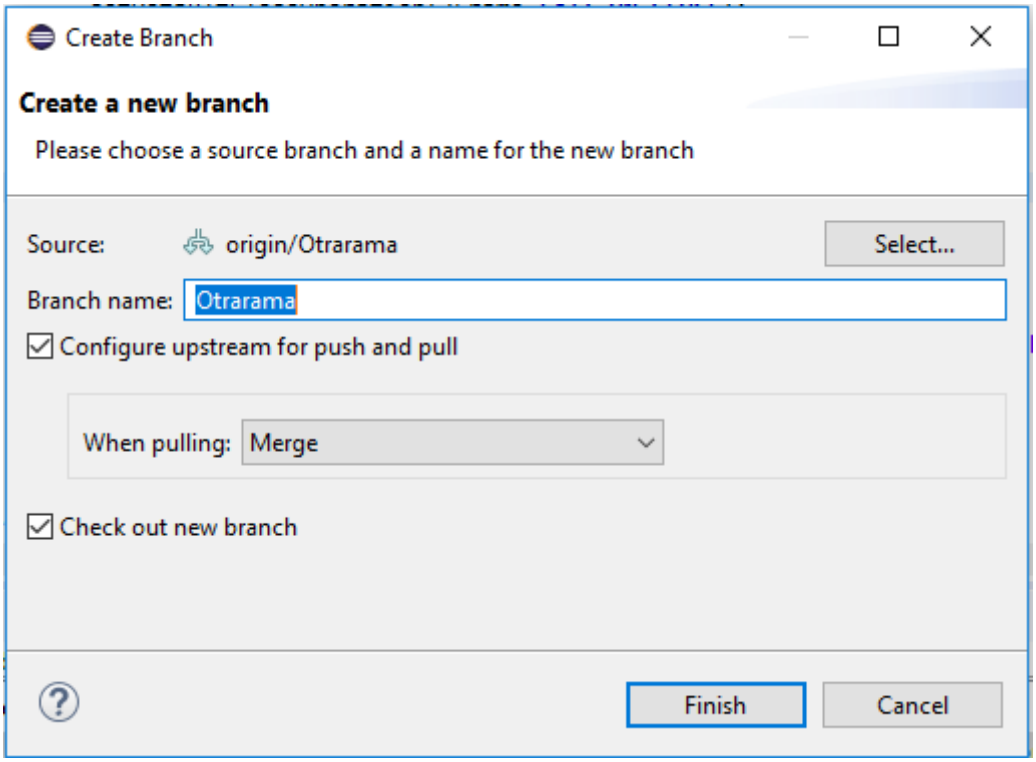


tendremos que pulsar en Team->Switch To->Other, y nos aparecerá una pantalla para seleccionar la rama:

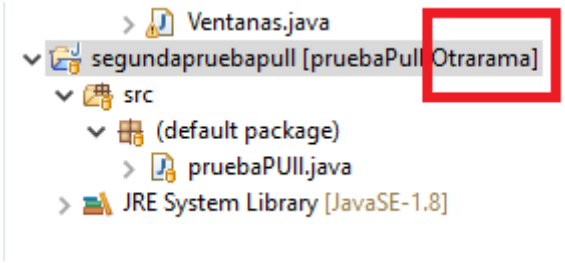


Al pinchar en chek out, nos pregunta si sólo vamos a querer visualizarla, o si vamos a trabajar sobre ella. Si trabajaremos sobre ella habrá que pulsar sobre “Check Out as New Local Branch”.

Aparecerá la pantalla con los datos de la nueva rama, y pulsamos sobre Finish:

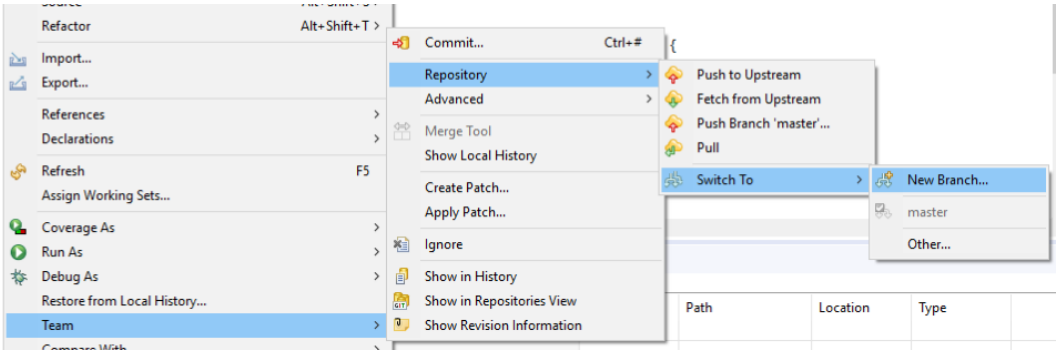


Ahora ya nos aparecer que estamos trabajando directamentemente sobre la rama nueva:

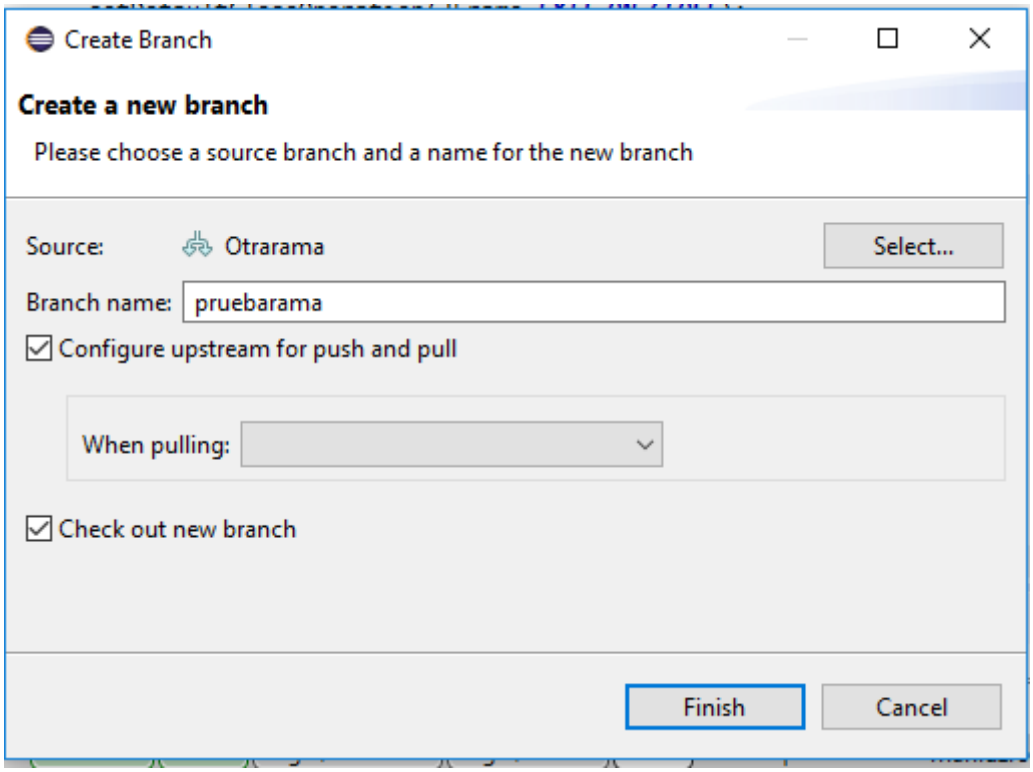


Para cambiar de una rama a otra (igual que en GIthub), tendremos que pulsar en Team->Switch To->(rama)

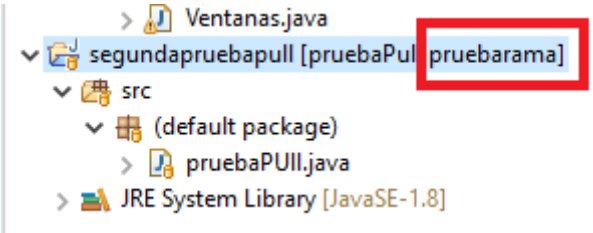
La segunda es abrir la rama en eclipse:



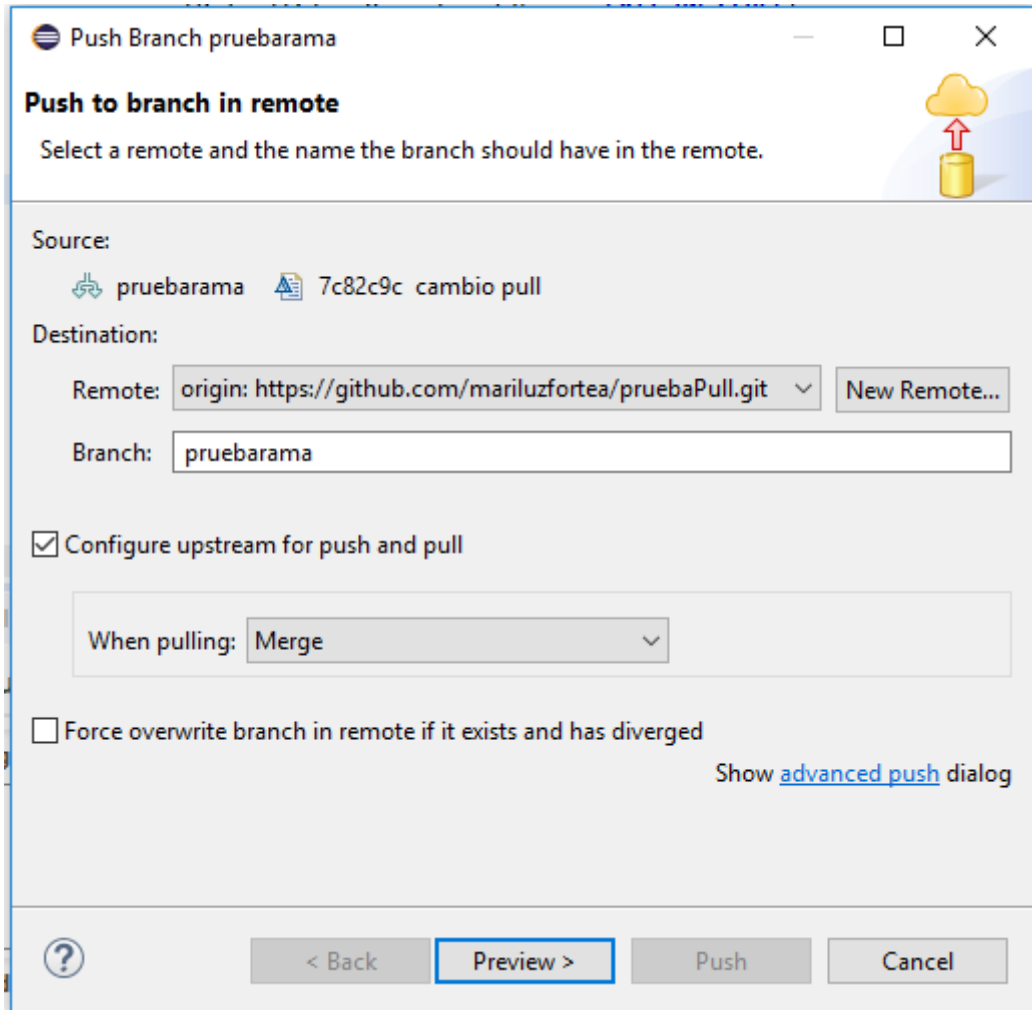
Le damos nombre a la rama y seleccionamos todos los campos:



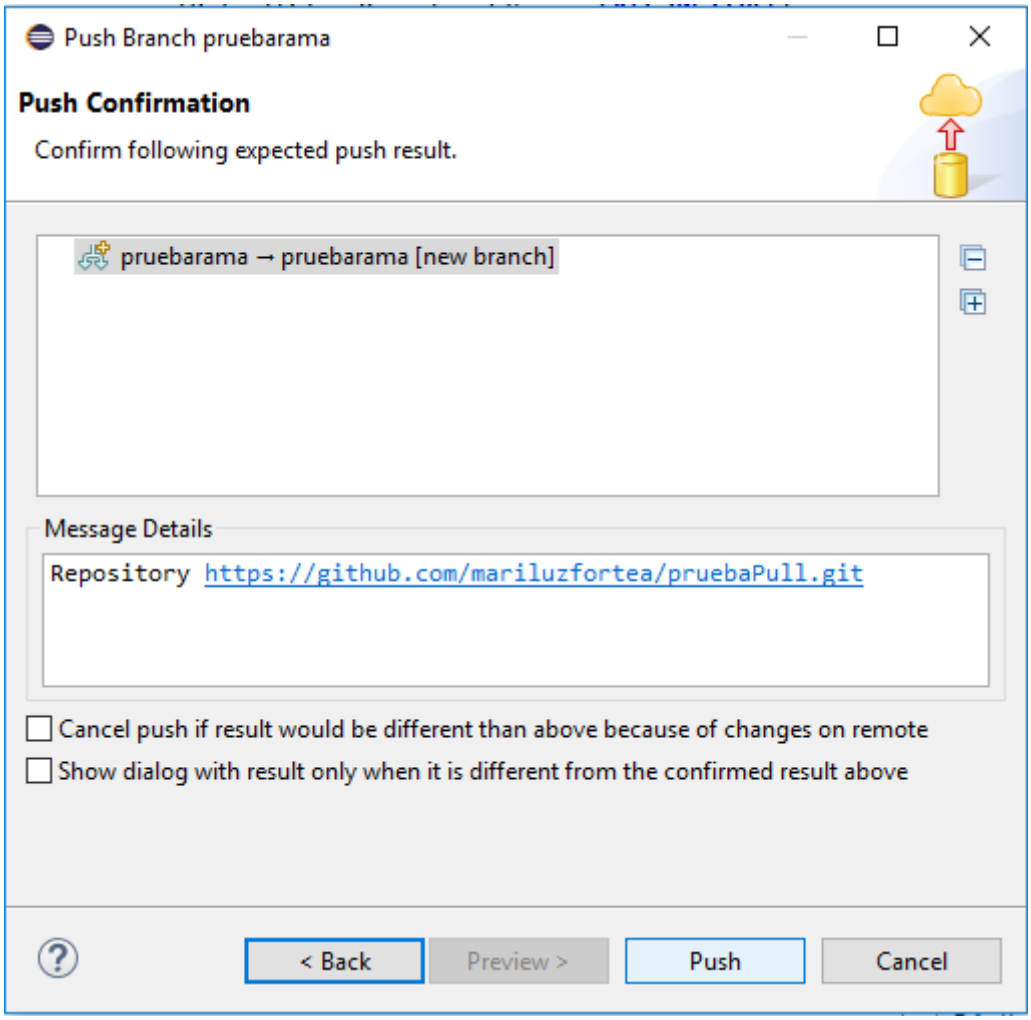
Al pulsar en Finish ya tenemos nuestra rama creada, y aparecerá esa rama como en la que estamos trabajando ahora en local:



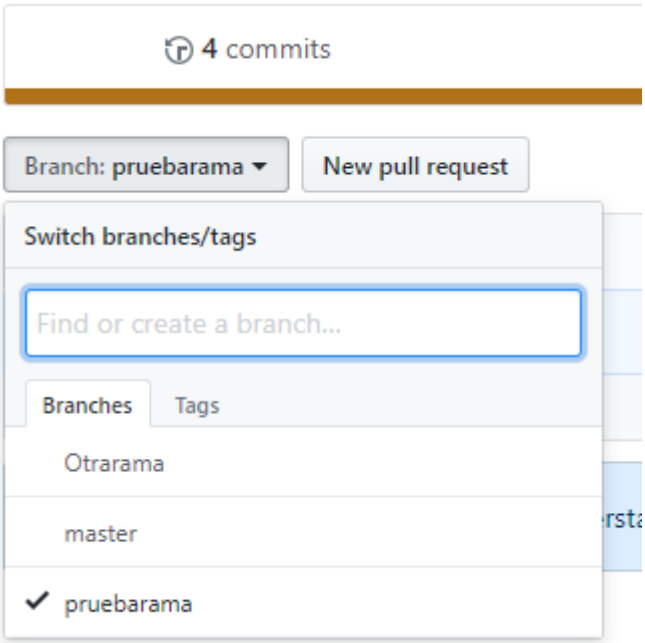
pero tendremos que replicarla en Github para que también se refleje en el repositorio remoto, para lo cual, tendremos que hacer un push especial para esta rama, pinchando en botón derecho, Team->Push Branch (nombre rama), y aparecerá la ventana de subida:



Pulsamos Preview, y nos aparecerá la ventana de confirmación de subida. Si todo es correcto sólo tendremos que pulsar en Push:



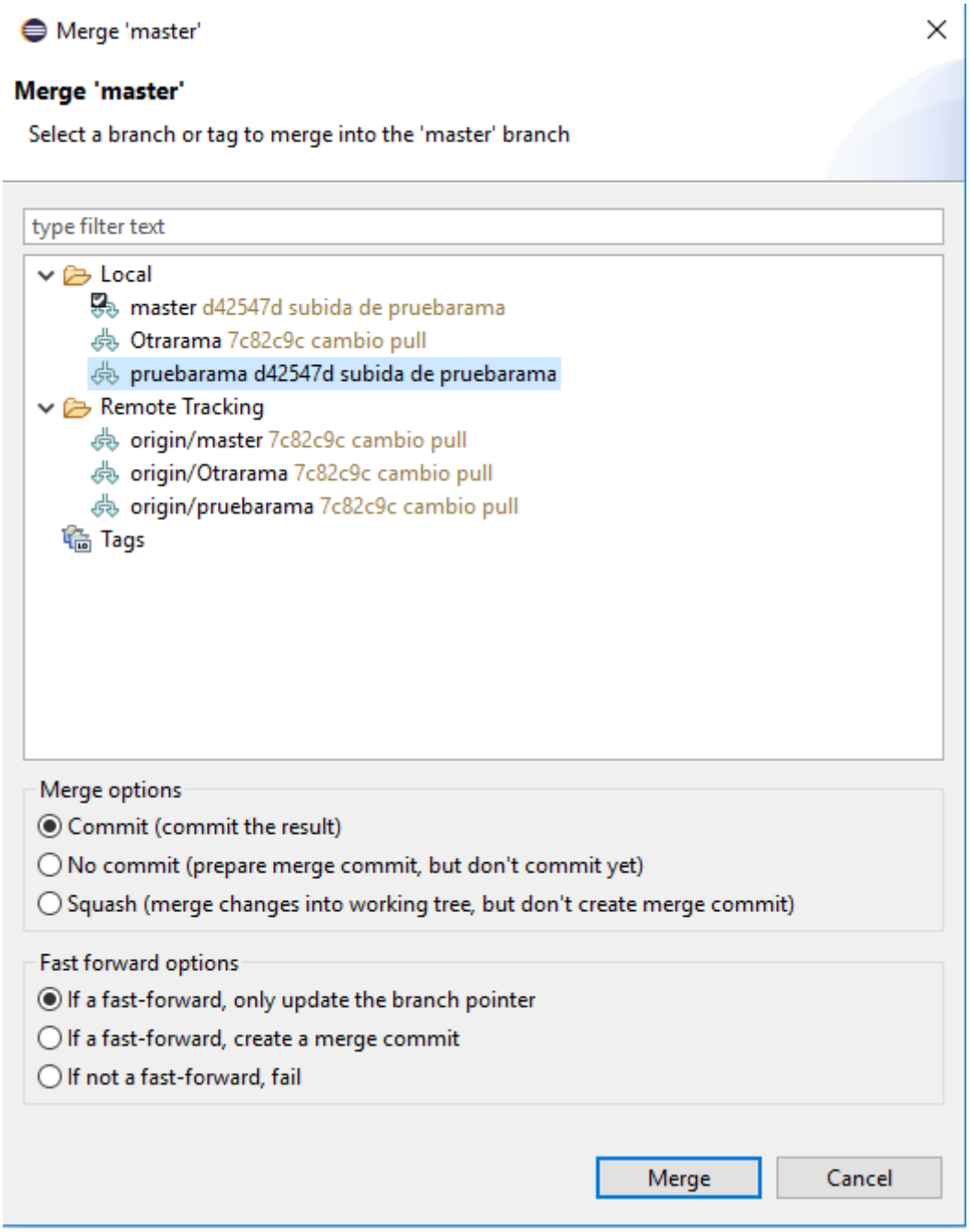
Si vamos a nuestro repositorio Github, observaremos que tenemos creada la nueva rama:


4 commits
Branch: pruebarama
New pull request
Switch branches/tags
Find or create a branch...
Branches
Tags
Otrarama
master
pruebarama

Merge (fusionar) dos ramas

Hacer un Merge en Eclipse

Si hacemos un cambio en la rama, hacemos commit, pero ahora quisiéramos fusionar (Merge) esa rama con la rama principal, deberemos irnos a la rama principal (master) con Team-> Switch to, y cuando estemos en la rama master, pinchar Team, Merge y nos aparecerá la ventana con las opciones para ese Merge:



Aquí seleccionamos con qué rama vamos a hacer el Merge (en el apartado "Local"), y también podremos decidir qué queremos que suceda en ese Merge, si queremos que directamente haga commit cuando lo haga, si queremos revisarlo primero antes de hacer un commit, si sólo queremos que se guarde el resultado en el workspace y luego ya decidiremos...

Pulsamos Merge, y nos habrá actualizado en nuestra rama Master la fusión con los cambios de la otra rama. Como siempre, para poder actualizar este cambio también en Github, habrá que hacer un “Push Branch ‘Master’”.

En el caso de que se produjese un conflicto al hacer el Merge (eclipse no es capaz de gestionar cómo fusionar los cambios), tendremos que hacerlo manualmente.

Ejemplo de Clase documentado con Javadoc

javadoc: documentación de programas

En Java, la documentación se hace por medio de *javadoc*.

A continuación se muestra un programa documentado, que se comenta más abajo:

```
 1 /**
 2  * Ejemplo: círculos.
 3  *
 4  * @author José A. Mañas - fprg5000
 5  * @version 23.9.2005
 6  */
 7 public class Circulo {
 8     private double centroX;
 9     private double centroY;
10     private double radio;

12     /**
13      * Constructor.
14      * @param cx centro: coordenada X.
15      * @param cy centro: coordenada Y.
16      * @param r radio.
17      */
18     public Circulo(double cx, double cy, double r) {
19         centroX = cx;
20         centroY = cy;
21         radio = r;
22     }

24     /**
25      * Getter.
26      * @return centro: coordenada X.
27      */
28     public double getCentroX() {
29         return centroX;
30     }

48     /**
49      * Calcula la longitud de la circunferencia (perímetro del círculo).
50      * @return circunferencia.
51      */
52     public double getCircunferencia() {
53         return 2 * Math.PI * radio;
54     }

64     /**
65      * Desplaza el círculo a otro lugar.
66      * @param deltaX movimiento en el eje X.
67      * @param deltaY movimiento en el eje Y.
68      */
69     public void mueve(double deltaX, double deltaY) {
70         centroX = centroX + deltaX;
71         centroY = centroY + deltaY;
72     }

74     /**
75      * Escala el círculo (cambia su radio).
76      * @param s factor de escala.
77      */
78     public void escala(double s) {
79         radio = radio * s;
80     }
```

```
81 }
```

Descripción del código anterior:

- Documentar todas las clases (ej. líneas 1-6), indicando
  - lo que hace la clase (ej. línea 2)
  - el autor (ej. línea 4)
  - la versión del programa, señalada (por ejemplo) por su fecha (ej. línea 5)
- Documentar todos y cada uno de los métodos
  (ej. líneas 12-17, 24-27, 48-51, 64-68 y 74-77),
  indicando
  - lo que hace el método (ej. línea 13, 25, 49, 65 y 75)
  - los parámetros de entrada (ej. líneas 14-16, 66-67 y 76).
    Para cada parámetro hay que escribir
    1. `@param`
    2. el nombre del parámetro
    3. una somera descripción de lo que se espera en el parámetro

    OJO: un método puede carecer de parámetros de entrada (ej. líneas 28 y 52)

  - el resultado que devuelve
    (ej. línea 26)
    1. `@return`
    2. una somera descripción de lo que devuelve

    OJO: un método puede no devolver nada (ej. líneas 18 y 78)

CÓDIGO:

```
package prueba01;

/**
 * Ejemplo: circulos.
 *
 * @author Jose A. Manias - fprg5000
 * @version 23.9.2005
 */
 public class Circulo {
  private double centroX;
  private double centroY;
  private double radio;

  /**
   * Constructor.
   * @param cx centro: coordenada X.
   * @param cy centro: coordenada Y.
   * @param r radio.
   */
   public Circulo(double cx, double cy, double r) {
    centroX = cx;
    centroY = cy;
    radio = r;
  }

  /**
   * Getter.
   * @return centro: coordenada X.
   */
   public double getCentroX() {
    return centroX;
```

```
  }

 /**
  * Calcula la longitud de la circunferencia (perimetro del circulo).
  * @return circunferencia.
  */
  public double getCircunferencia() {
   return 2 * Math.PI * radio;
  }

 /**
  * Desplaza el circulo a otro lugar.
  * @param deltaX movimiento en el eje X.
  * @param deltaY movimiento en el eje Y.
  */
  public void mueve(double deltaX, double deltaY) {
   centroX = centroX + deltaX;
   centroY = centroY + deltaY;
  }

 /**
  * Escala el circulo (cambia su radio).
  * @param s factor de escala.
  */
  public void escala(double s) {
    radio = radio * s;
  }
}
```