Práctica de Git

Gestión y desarrollo del software

Autores

David Lozano Jarque (NIU 1359958)
Carlos González Cebrecos (NIU 1212586)

Escuela de Ingeniería, Universidad Autónoma de Barcelona

Curso 2016-2017

Introducción

La práctica presente consiste en aprender a usar *Git*, el software de control de versiones diseñado por *Linus Torvalds*.

Para ello, realizaremos un proyecto sencillo de software y gestionaremos el desarrollo de su código fuente usando *Git*, lo que también nos permitirá gestionar las *releases* que entregaríamos al cliente o clientes.

El proyecto: un software de facturación

El proyecto que realizaremos y que gestionaremos con *Git* es un pequeño software de facturación implementado en *Java* que permite gestionar (sin persistencia de datos) un conjunto de clientes, productos y facturas.

Básicamente el software implementa operaciones *CRUD* (*Create, Read, Update, Delete*) sobre los elementos anteriormente mencionados a través de una interfície de línea de comandos.

El proyecto continuará su desarrollo a partir de una versión inicial funcional que contiene los ficheros `.java` con el código fuente de éste.

> No se incluye ningún proyecto de *Eclipse*, *Netbeans*... en la versión inicial a desarrollar del proyecto

Herramientas

Como herramientas durante el desarrollo, a parte de usar y aprender el software de control de versiones *Git*, también usaremos:

- *Java SE Development Kit 8u121* para el desarrollo del código y compilación de éste en lenguaje *Java*
- *Eclipse Neon.2* como *IDE* (Entorno de desarrollo integrado)
- Lenguaje *markdown* para la escritura de ficheros descriptivos en el repositorio como el fichero de descripción del repositorio *README.md* o este mismo documento.

Se asume se ha instalado y configurado las herramientas anteriormente mencionadas antes de seguir con la práctica (a excepción de *Git*).

Previo

Previo al desarrollo de la práctica, debemos instalar el software de *Git* que gestionará las versiones de nuestra aplicación.

Instalación de *Git*

En muchas distribuciones de Linux, el software de *Git* viene incluido por defecto. En otras, y en sistemas *Microsoft Windows* se requiere instalación de éste.

Podemos comprobar si disponemos de una instalación de *Git* bien configurada abriendo una terminal y escribiendo el comando
`git version`

Esto nos debería mostrar la versión del software de *Git* si éste se encuentra instalado. En caso de no reconocer el comando, quizás no hemos configurado bien la instalación (falta añadir el binario de *Git* a la variable de entorno `PATH`) o no tenemos el software instalado.

Para instalarlo en distribuciones *Debian* y derivadas como *Ubuntu*, podemos escribir en una terminal:

```
sudo apt install git
```

Para otros sistemas operativos, los binarios y guías de instalación se encuentran en el sitio oficial:

https://git-scm.com/downloads

Configuración del cliente *Git*

Configuración de usuario

Una vez tenemos instalado y configurado *Git* para su uso desde una terminal, procedemos a configurar el cliente de *Git*. Los parámetros básicos que necesitamos configurar es nuestro nombre y correo electrónico para asignarlos a nuestros *commits* o aportaciones al proyecto.

Para configurar estos parámetros de forma global para cualquier repositorio que gestionemos con *Git*, debemos escribir los siguientes comandos:
`git config --global user.name "John Doe"`

```
git config --global user.email "email@example.org"
```

La explicac

- `user.signingKey` : especifica el identificador corto (8 caracteres) de la llave que usaremos para firmar.
  Por defecto busca en el software que usemos las llaves disponibles y busca aquella que coincida con el correo establecido anteriormente (`user.email`)
- `commit.gpgSign` : establece que todos los *commit* se firmen digitalmente por defecto. Por defecto está deshabilitado.
- `push.gpgSign` : establece que todos los *push* se firmen digitalmente por defecto. Por defecto está deshabilitado.

> Los valores de configuración son *case-independent*, de manera que no afecta si las variables las escribimos con mayúsculas o minúsculas (luego `user.signingKey` es equivalente a `user.signingkey`)

Desarrollo

Una vez tenemos configurado el entorno de *Git* correctamente, procedemos a crear nuestro repositorio y comenzar con el desarrollo del proyecto.

Designación de ítems de configuración

Antes de todo, debemos crear un nuevo repositorio que contendrá nuestros ítems de configuración. Para ellos debemos definir cuales serán estos mismos.

Ítems del proyecto

Por esa razón, empezaremos por listar con que elementos contamos en nuestro proyecto. En nuestro proyecto contamos (técnicamente, contaremos cuando creemos el proyecto en *Eclipse*) con los siguientes ficheros:

- `src/*.java` : Ficheros de código fuente en *Java*
- `bin/*.class` : Binarios del código fuente compilado
- `.classpath` : Fichero con rutas a considerar para el compilado (ficheros a incluir, destino de los binarios)
- `.project` : Fichero de configuración de proyecto de *Eclipse*
- `.settings` : Directorio con preferencias para el proyecto de *Eclipse*, como la versión de *Java JRE* con la que ejecutar los binarios
- `GenVersiones.bat` : fichero con script para generar directorios con las versiones del proyecto para su posterior comprobación y corrección
- `Corrector.bat` : fichero con script para comprobar que las salidas del programa son correctas en función de las salidas correctas y ejecutadas
- `SalidaPractica` : directorio con ficheros de texto que contienen la salida de la práctica en diferentes versiones dados diferentes comandos. Generada por el script `Corrector.bat`
- `SalidaCorrecta` : directorio con ficheros de texto que contienen la salida de la práctica de referencia que debe proporcionar cada versión dados diferentes comandos
- `Comandos*.txt` : comandos a ejecutar para cada *release* del software para comprobar su salida con los ficheros del directorio `SalidaCorrecta` y verificar que el programa se ejecuta correctamente

Ítems de configuración

Una vez listados los elementos de nuestro proyecto y su funcionalidad, procedemos a listar aquellos que no serán parte de los ítems de configuración.

> Git en lugar de permitir especificar los ítems de configuración, permite especificar aquellos ítems que no serán de configuración como veremos más adelante y todo lo restante lo considerará ítem de configuración (una vez lo añadamos al índice de ficheros *tracked*)

> Dado que debemos especificar manualmente que ficheros entran al índice de seguimiento de Git para que los considere parte del repositorio, puede parecer que no sea necesario especificar los no-ítems de configuración, pero su especificación nos ayudará cuando realicemos comandos del tipo `git add *`, que añadirá totdos los ficheros al índice excepto aquellos que no sean ítems de configuración (que estén en la lista de excepciones)

Consideramos que los siguientes ítems no formaran parte de los ítems de configuración, por las siguientes razones:

- `GenVersiones.bat`, `Corrector.bat`: es de uso meramente académico y no forma parte del programa en sí. No se requiere para compilar y generar una versión del software ni proporciona ninguna ayuda a ello.

  Tampoco lo hemos considerado herramienta de test por su sencillez. Además los tests automatizados en *Git* se suelen almacenar o ejecutar en otros lugares, como en el caso de la herramienta *Travis CI*:

  https://travis-ci.org/
- `Comandos*.txt`, `SalidaCorrecta`, `SalidaPractica`: dado que no se requieren los ficheros anteriores, estos carecen de utilidad si los anteriores no se incluyen. En el caso de `SalidaCorrecta`, dado que su uso es académico y no realmente forma parte de un test de la aplicación, tampoco se incluye. Se supone los participantes de la práctica pueden descargarse esos ficheros manualmente si requieren de su uso.

Recordaremos estos ficheros cuando especifiquemos las excepciones de ítems de configuración más adelante.

Creación del repositorio

1. En primer lugar, nos situamos con una terminal abierta en el directorio donde deseamos crear nuestro proyecto.
2. Creamos el proyecto de *Java* en *Eclipse* en un nuevo directorio dentro de éste.
3. En la terminal, nos movemos al nuevo directorio creado por *Eclipse* que contendrá nuestro proyecto e inicializamos el directorio como contenedor de un repositorio de *Git* con la siguiente órden:

   ```
   cd facturacion && git init
   ```

4. Especificamos ítems de configuración. Para ello, recordemos debemos especificar las excepciones de ítems de configuración.

Debemos especificarlos en un fichero llamado `.gitignore` . Para ello, introducimos el siguiente comando:
`nano .gitignore` / `notepad .gitignore` (en *Windows*)

El fichero `.gitignore` nos permite especificar expresiones regulares para designar los ficheros que no formarán parte de los ítems de configuración. Para saber más sobre su sintaxis:

https://git-scm.com/docs/gitignore

Introducimos el siguiente contenido:

```
# Correctores
/Corrector.bat
/GenVersiones.bat

# Comandos
comandos*.txt
Comandos*.txt

# Ejecuciones del programa
/SalidaCorrecta/*
/SalidaPractica/*
```

Puesto que es un proyecto *Java*, hemos decidido eliminar las compilaciones del repositorio, pero quizás hay otros ficheros que no són del código fuente que *Java* genera y tampoco deseamos.

Hay un repositorio online, alojado en *Github* que contiene los ítems de configuración a excluir básicos según el lenguaje de programación:

https://github.com/github/gitignore

Buscamos el de *Java* y añadimos su contenido al fichero:

```
# Compiled class file
*.class

# Log file
*.log

# BlueJ files
*.ctxt

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear
*.zip
*.tar.gz
*.rar

# virtual machine crash logs, see http://www.java.com/en/download/hel
p/error_hotspot.xml
hs_err_pid*
```

Guardamos el fichero y ya tenemos el repositorio listo para albergar nuestro proyecto

> También existe otro fichero, llamado `.gitattributes` que nos permite especificar algunos atributos interesantes para el repositorio como su nombre indica. En nuestro caso, usaremos el fichero con el siguiente contenido:
>
> `* text eol=lf`
>
> Esto hará que *Git* convierta los fines de línea a `\n` (`LF`) en lugar de `\r\n` (`CRLF`) cuando se realice un *commit*. Esto nos permite trabajar cómodamente en *Microsoft Windows* y sistemas *UNIX*, puesto que no habrá errores con los fínes de línea y todos serán homogéneos en el repositorio, pese a que localmente puedan ser diferentes.
> Todo se debe a que nuestro equipo de desarrolladores tiene preferencias distintas por sus sistemas operativos favoritos

Realizamos el primer commit con dichos ficheros que definirán nuestro repositorio. Para ello, añadimos los ficheros al índice:

`git add *`

Y realizamos un *commit* con un mensaje

`git commit -m "Initial commit"`

> Si no especificamos `-m` y nuestro mensaje, se abrirá un editor de texto para especificarlo allí. En *Windows* dicho editor es `vim` . Pulsamos `Insert` si

> requerimos entrar en el modo edición, escribimos nuestro mensaje y a continuación `Esc` y `:wq` e `Intro` para guardar.

Añadiendo el código fuente existente

1. Descargamos código fuente del material
2. Copiamos código fuente de proyecto en el directorio destinado a ello: `src`
3. Comprobamos que *Eclipse* ejecuta bien el programa refrescando la carpeta `src` y añadiendo una *Run Configuration* que toma como parámetro el fichero de comandos de la primera versión.
4. Con la ejecución correcta, añadimos los ficheros al índice y realizamos el primer *commit* con el código fuente del programa.

```
git add *
git commit -m "Added Eclipse project & source files"
```

Añadiendo un repositorio remoto

Ahora necesitamos sincronizar nuestro repositorio local con uno remoto para permitir a los demás desarrolladores trabajar conjuntamente.

Para ello, añadiremos un remoto al repositorio local:

```
git remote add bitbucket \
git@bitbucket.org:uab-projects/facturacion_1212586_1359958
```

Con este comando añadimos un repositorio remoto al cual llamaremos `bitbucket` y que se encuentra en la siguiente URL:

```
git@bitbucket.org:uab-projects/facturacion_1212586_1359958
```

que hemos obtenido cuando lo creamos en la plataforma `Bitbucket`

A continuación debemos debemos establecer en la rama `master` (rama por defecto en la cual hemos realizado los previos commits) que el repositorio remoto al cual subir los cambios en ésta es el que acabamos de añadir. A continuación subiremos los *commits* realizados. Realizamos las dos acciones con el comando:

```
git push -u bitbucket master
```

Primera *release*

A continuación debemos marcar esta versión como la primera release, con la etiqueta `ReleaseMJ1.0`.

Para ello usamos el siguiente commando

```
git tag -s "ReleaseMJ1.0" -m "Port from source files into Git"
```

Que creará un *tag* firmado (argumento `-s`) llamado `ReleaseMJ1.0` con el mensaje `Port from source files into Git`. Si deseamos realizar un *tag* no firmado, usamos en lugar de `-s`, `-a`

Finalmente, subimos el *tag* al repositorio `bitbucket` con el comando:

```
git push --tags
```

Por defecto `git push` no sube las *tags*. El comando lo sube al remoto `bitbucket` puesto que es el único que encuentra.

Rama de desarrollo

A continuación debemos implementar el primer cambio, el listado de facturas. Para ello, antes creamos una nueva rama de desarrollo llamada `development` y actualizaremos el remoto con ésta.

```
git checkout -b development
```

Creamos una rama llamada `development` (que parte de la actual, `master`) y nos cambiamos a ella.

```
git push -u bitbucket development
```

Actualizamos el repositorio remoto `bitbucket`, añadiendo la nueva rama `development`

Listado de facturas (I)

Ahora, que ya estamos en la rama de desarrollo, creamos una nueva rama para desarrollar esta característica y nos situamos en ella.

```
git checkout -b list-invoice
```

Realizamos los cambios necesarios para imprimir facturas, hasta que nos damos cuenta que falta el precio del producto. Con la funcionalidad presente, que imprime facturas con precio 0, realizamos un *commit* en la rama.

```
git commit -a -m "Print invoices implemented"
```

La orden anterior, ya nos añade los ficheros modificados (que están presentes en el índice) al *commit* con el argumento `-a`.

Subimos los cambios al repositorio remoto, a la misma vez que especificamos el repositorio remoto de esta rama

```
git push -u bitbucket list-invoice
```

> Hemos realizado un *commit* con éstos cambios y no con la funcionalidad entera ya que debemos cambiar a otra rama para fijar un error en la siguiente sección y en

> *Git* no se permite cambiar a una rama habiendo ficheros modificados en el índice para prevenir errores.
> Si el desarrollador que fija la release fuera uno diferente, entonces podríamos realizar un sólo *commit* con todos los cambios realizados para el correcto listado de facturas puesto que el desarrollador que implementara el *bugfix* de la siguiente sección no estaría desarrollando esta característica y puede libremente cambiar de rama siempre y cuando no tenga él tampoco ningún fichero modificado.

Fijación de release `ReleaseMJ1.0`

El cliente nos informa de que ha aparecido un error en la *release* anterior que permite la existencia de clientes duplicados. Debemos volver a la `master`, donde se encuentra la `ReleaseMJ1.0` y realizar los cambios.

Para ello, volvemos a la `master`, que ya se encuentra en la `ReleaseMJ1.0` y abrimos una nueva rama llamada `hotfix` para fijar el error.

```
git checkout master
```

También podríamos haber usado para ser más precisos:

```
git checkout ReleaseMJ1.0
```

Abrimos la nueva rama para fijar el error

```
git checkout -b hotfix
```

Fijamos el error y realizamos un *commit* con los cambios

```
git commit -a -m "Bug fix: No repeated clients & ids"
```

A continuación, debemos juntar los cambios con la rama `master`, para ello, nos movemos a la rama `master` y realizamos una operación `merge`:

```
git checkout master && git merge hotfix
```

Git automáticamente ha combinado correctamente los cambios con la *release* anterior.

Creamos la nueva *release* con el cambio fijado y actualizamos las *tags* remotas

```
git tag -s ReleaseMJ1.1 -m "Bug fix: No repeated clients & ids" && git push --tags
```

La última orden ya sube los cambios con el *bugfix* y la nueva *tag* de la últi

> continua su implementaicón del listado de facturas en paralelo, puesto que el error no le produce un impedimento para continuar desarrollando esta característica. De esta forma, la presente sección y la anterior podrían ejecutarse simultáneamente por dos desarrolladores distintos, mientras que esta sección y la anterior a la previa serían desarrolladas por el mismo desarrollador.

Realizamos un *commit* con los cambios con el listado de facturas y la modificación de los precios ya implementadas.

```
git add * && git commit -m "Print invoices completed"
```

Hemos usado el comando `git add *` puesto que hay nuevos ficheros que hay que indicar que pertenezcan al índice (son ficheros de configuración).

Ahora juntamos los cambios con la rama `development`:

```
git checkout development && git merge list-invoice
```

Segunda release `ReleaseMJ2.0`

A continuación antes de lanzar la segunda *release* (*release* mayor), debemos actualizar la rama `development` para que tenga el error de la `ReleaseMJ1.1` fijado, ya que recordemos la rama `development` parte de `master` en la versión `ReleaseMJ1.0`. Para ello usamos el siguiente comando:

```
git rebase master
```

Git mezcla los cambios, pero detecta un conflicto que no puede solucionar en `src\CMain.java`, por lo que abrimos el fichero con *Eclipse* y solucionamos manualmente los cambios.

Esto sucede porque tanto la funcionalidad de listar facturas como el fijado del error han tocado el mismo fichero, `src\CMain.java`.

Nos fijamos en el fichero en los segmentos marcados con caracteres separadores del tipo:

```
====================== HEAD
----------------------
====================== Print invoices completed
```

Allí nos muestra las dos versiones con conflicto y nosotros debemos encargarnos de eliminar una u otra o realizar los cambios pertinentes para completar el *rebase* (incluyendo eliminar los separadores)

Cuando ya tengamos el fichero `src\CMain.java` fijado, añadimos el fichero al índice para indicar a *Git* que los conflictos ya se han solucionado y le indicamos a *Git* que debe

finalizar el *rebase*

```
git add src\CMain.java && git rebase --continue
```

Finalmente, juntamos la rama `development` con la `master` y creamos y anotamos la *release*

```
git checkout master && git merge development
git tag -s ReleaseMJ2.0 -m "Print invoices implemented"
```

Funcionalidad de listado de clientes

Continuamos el desarrollo esta vez implementando la funcionalidad de listar clientes. Dado que es una nueva funcionalidad, nos situaremos de nuevo en la rama `development` y crearemos una nueva rama llamada `list-client`:

```
git checkout development && git checkout -b list-client
```

Implementamos los cambios pertinentes en `src\CMain.java` y `src\CClientList.java` y una vez hemos finalizado, realizamos un *commit* con los cambios realizados:

```
git commit -a -m "Print clients completed"
```

> Antes de realizar la operación de *merge*, se sugiere realizar la lectura del siguiente apartado del informe, ya que mientras se implementa esta característica, la siguiente se desarrolla en paralelo.

Finalmente, juntaremos los cambios en la rama `development` realizando una operación *merge*.

```
git checkout development && git merge list-client
```

Subimos los cambios para que nuestro compañero desarrollador pueda juntar sus cambios con los nuestros una vez finalice

```
git push
```

Funcionalidad de listado de productos

Al mismo tiempo que se desarrolla la característica anterior, otro desarrollador decide desarrollar en paralelo una nueva característica que permita listar productos. Para ello, este nuevo desarrollador, que tiene su rama `development` en la última versión `ReleaseMJ2.0`, crea una nueva rama `list-product` para desarrollar su característica.

```
git checkout development && git checkout list-product
```

En este momento, tendremos que la característica de listado de clientes y productos se desarrollan en paralelo, cada una en su rama `list-client` y `list-product` respectivamente.

Una vez finalizamos nuestros cambios, que implican los ficheros `src\CMain.java` y `src\CProductList.java`, realizamos un *commit* con los cambios:

```
git commit -a -m "Print products completed"
```

El otro desarrollador implementó su característica más rápido y ya hizo el *merge* con la rama `development`, ahora realizaremos un *merge* de nuestra rama con la `development`, previamente descargando del repositorio remoto la operación *merge* que nuestro compañero desarrollador realizó para juntar sus cambios.

```
git checkout development && git pull && git merge list-product
```

Nos produce conflictos de integración ya que hemos tocado el mismo fichero `src\CMain.java`, solucionamos los cambios de la misma forma que el último *merge* con problemas y finalizamos el *merge*:

```
git add src\CMain.java && git commit
```

Subimos nuestros cambios de nuevo al repositorio remoto:

```
git push
```

> Extra: normalmente, en un equipo más grande de desarrolladores, y con un proyecto mayor, cuando se desea implementar una característica o conjunto de características por un equipo, dentro de un proyecto con varios equipos, se realiza una operación *fork* que clona el estado del repositorio en un remoto diferente para cuando la característica se haya implementado (después de crear los *commit* y ramas necesarias en este repositorio clonado), realizar una operación *pull request* que solicite la integración de los cambios del repositorio clonado al repositorio original.
> En un equipo pequeño, estas operaciones suponen un *overhead* no deseado e innecesario ya que se conocen las ramas existentes y su objetivo por lo que no habrá conflictos con ellas a la hora de crear nuevas o avisar a los desarrolladores cuando se estén realizando operaciones del tipo *merge* que impiden avanzar en una rama.

Tercera *release* `ReleaseMJ3.0`

Comprobamos en la rama `development` que la salida es igual a la salida esperada y finalmente, juntamos la rama `development` con la `master` mediante una operación *merge* y anotamos la versión como una nueva *release*:

```
git checkout master && git merge development && git tag -s ReleaseMJ3.0 -m "List clients and products implemented"
```

Subimos la *tag* de la nueva *release* al repositorio remoto:

```
git push --tags
```

Nuevo cliente: Papelería el Lápiz Afilado

Llega un nuevo cliente y requiere de nuevos requisitos, entre los cuales destacan cambiar el nombre de Muebles José por el nombre de su establecimiento y alguna funcionalidad específica.

Esto nos requiere cambiar el modelo de desarrollo y el objetivo de cada una de las ramas que habíamos creado hasta ahora.

Requerimos ahora desarrollar *releases* para clientes diferentes, por lo que para no mezclar estas, partiremos de la *master* dos nuevas ramas, una para desarrollar características para cada cliente por separado.

```
git checkout master && git checkout -b lapiz-afilado
git checkout master && git checkout -b muebles-jose
```

Cuando realicemos cambios para un cliente en específico los realizaremos sobre su

Permitir modificar fecha de factura

Implementamos los cambios y realizamos un *commit* de nuevo con todos los ficheros modificados.

```
git commit -a -m "Added invoice date modification"
```

Imprimir factura según el número de ésta

Desarrollamos los cambios y realizamos un *commit* con éstos

```
git commit -a -m "Added print invoice by number"
```

> Quizás alguna de éstas características, sobretodo la impresión de facturas según su número, podríamos haberla realizado sobre la rama `master` para que ambos clientes pudieran disfrutar de ella aunque sólo Lápiz Afilado la hubiera solicitado proveyendo de extras a los clientes existentes.

Fix para corrección correcta

Realizamos también un *commit* para fijar un error en la salida que se adapte a la salida correcta a corregir

```
git commit -a -m "Fix, no descriptor in print header"
```

Primera *release* para Papelería el Lápiz Afilado `ReleasePLA4.0`

Anotamos el primer *release* en la rama `lapiz-afilado` para su entrega al cliente:

```
git tag -s ReleasePLA4.0 -m "First release for LLA"
```

Error facturas con número replicado

El cliente nuevo detecta un error que permite la inserción y modificación de facturas con el mismo número. Es por ello que se debe arreglar en ambas version ya que el cliente Muebles José también se ve afectado.

Por ello, debemos crear una rama que parta de la `master` para fijar el error, y luego, con el *merge* sobre la `master` realizado de manera que ésta ya tenga los cambios con el error solucionado, hacer un *rebase* de la rama de cada cliente para actualizar sus respectivas versiones.

Fijación del error

Aprovechamos la rama antes creada de `hotfix` que parte de la `master` para arreglar el error allí, eso sí, actualizándola con la última versión de la rama `master`

```
git checkout hotfix && git rebase master
```

A continuación, fijamos el error para que lance una excepción en el caso que se desee modificar o introducir una factura con número ya existente y realizamos un *commit* con

los cambios.

```
git commit -a -m "Fixed duplicated invoices identifier"
```

> Realizamos un *commit* adicional para hacer coincidir el resultado de nuestra ejecución con el resultado esperado ya que hay un cambio inesperado en el resultado esperado no mencionado en el enunciado y sin sentido aparente ya que resta funcionalidad anteriormente implementada

Finalmente, aplicamos una operación *merge* sobre la `master` con la rama presente

```
git checokut master && git merge hotfix
```

Rebase de las versiones

A continuación, vamos a la rama de cada uno de los clientes y hacemos un *rebase* para actualizar ambas versiones de ambos clientes con el error fijado.

```
git checkout muebles-jose && git rebase master
git checkout lapiz-afilado && git rebase master
```

> Hemos de resolver distintos errores de integración, sobretodo en el fichero `src\CMain.java`, de la misma forma que anteriores veces

Releases finales

Finalmente, y con los *merge* finalizados, comprobamos que las salidas sean correctas y lanzamos las etiquetas de las últimas *releases*:

```
git checkout muebles-jose && git tag -s ReleaseMJ3.1 -m "Fixed duplicated invoices identifier"
git checkout lapiz-afilado && git tag -s ReleasePLA4.0 -m "Fixed duplicated invoices identifier"
```

Subimos al repositorio las nuevas etiquetas

```
git push --tags
```

> Comentarios adicionales: subimos manualmente las ramas al repositorio con el comando `git push -u bitbucket <rama>` para cada una de las ramas listadas en `git branch -l`