**INTRODUCCIÓN A LA INGENIERÍA DEL SOFTWARE**

**Practica. Control de Versiones**

**Práctica: uso de git**

Como empleado de la empresa Factoría de Algoritmos Recursivos SA, se le ha asignado el desarrollo de una aplicación que calcule el factorial de un número. En esta empresa se usa git, lo que permite compaginar el desarrollo del software con la corrección de errores del software ya en producción (entre otras muchas otras cosas, por supuesto).

**IMPORTANTE**: durante la realización de esta práctica, debería de guardar en un fichero de texto todos los comandos que vayan siendo ejecutados en la línea de comandos. Le resultará útil al final de la práctica. Aunque no se indica explícitamente, es bueno usar frecuentemente *git status* y *gitk --all* para entender cómo cambia el estado del repositorio según se ejecuta cada comando.

**NOTA**: Durante el transcurso de la práctica, las primeras veces que se debe usar un comando de git, éste aparece especificado entre paréntesis. En ocasiones sucesivas, se espera que sepa qué comando ejecutar a partir de la lectura del enunciado. Por ejemplo, si en el enunciado dice “cambie a la rama **master** (*git checkout master*)” y más tarde se le dice “cambie a la rama **recursivo**”, se espera que sepa que debe de ejecutar *git checkout recursivo*. Por lo tanto, **¡LEA CON ATENCIÓN EL ENUNCIADO!**

**PASOS DE CONFIGURACION**: Desde los laboratorios de la ETSI, debe de configurar git para usar el proxy correspondiente. Para ello, desde una ventana de Git Bash, ejecute el siguiente comando:

git config --global http.proxy http://proxy.lcc.uma.es:3128

Asimismo, una vez creado cada repositorio con git init o git clone, cambie al directorio del repositorio y configure su nombre de usuario y email:

git config user.name nombre\_de\_usuario\_solo\_caracteres\_alfanumericos\_sin\_espacios

git config user.email dirección\_de\_email\_aunque\_sea\_ficticia

Git no permitirá realizar *commits* si esos valores no están configurados. No tienen que ser necesariamente reales; se usan únicamente para firmar los *commits*.

1. Cree un nuevo repositorio en su cuenta de GitHub denominado umafactorial, y clónelo en su ordenador (git clone https://github.com/nombre\_de\_usuario/umafactorial.git).
2. En la rama **master**, cree el fichero factorial.java en su repositorio con el siguiente contenido:

**import** java.util.Scanner;

**public** **class** factorial {

**public** **static** **void** main(String []args) {

Scanner scanInput = **new** Scanner(System.***in***);

System.***out***.print("Introduzca un numero: ");

**long** num = scanInput.nextLong();

**long** fac = 1;

**for** (**long** i=1; i<=num; ++i) {

fac = fac \* i;

}

System.***out***.println("El factorial de "+num+" es "+fac);

scanInput.close();

}

}

1. Añada dicho fichero (git add factorial.java) en un *commit* con el mensaje “implementacion inicial” (git commit –m “implementacion inicial”).
2. El arquitecto del proyecto le pide separar la implementación en distintos métodos aparte, por lo que debe de añadir un nuevo *commit* con el mensaje “refactorizacion”, en el que el factorial se calcula en una función aparte:

**import** java.util.Scanner;

**public** **class** factorial {

**public** **static** **long** fact(**long** num) {

**long** fac = 1;

**for** (**long** i=1; i<=num; ++i) {

fac = fac \* i;

}

**return** fac;

}

**public** **static** **void** main(String []args) {

Scanner scanInput = **new** Scanner(System.***in***);

System.***out***.print("Introduzca un numero: ");

**long** num = scanInput.nextLong();

**long** fac = *fact*(num);

System.***out***.println("El factorial de "+num+" es "+fac);

scanInput.close();

}

}

1. Tras dicha modificación, el software es publicado enviando los cambios en la rama actual (**master**) al repositorio de GitHub (git push origin master). No obstante, algún tiempo después, el CEO de la empresa considera que usar una implementación iterativa daña su imagen corporativa. Para no perturbar el código fuente original mientras se implementa este cambio, deberá de usar una nueva rama. Cree una nueva rama con nombre **recursivo** y cambie el espacio de trabajo a ella (en un solo comando: git checkout –b recursivo).
2. Modifique el fichero de modo que la implementación del método fact() pase a ser la siguiente:

**public** **static** **long** fact(**long** num) {

**if** (num<=0) {

**return** 1;

} **else** {

**return** num\**fact*(num-1);

}

}

1. Añada dicha modificación mediante un *commit* con el mensaje “implementacion recursiva”. El departamento de calidad debe de inspeccionar la nueva implementación, por lo que debe de subir la rama **recursivo** al repositorio de GitHub.
2. Antes de que el departamento de calidad dé el visto bueno a la nueva implementación, un cliente comunica que la aplicación da resultados incorrectos para números negativos, ya que el factorial no está definido en dichos casos. Hay que resolver este problema sin demora, por lo que no podemos esperar a que la implementación en la rama **recursivo** esté lista. Vuelva a la rama **master** (git checkout master).
3. Modifique consecuentemente el método main():

**public** **static** **void** main(String []args) {

Scanner scanInput = **new** Scanner(System.***in***);

System.***out***.print("Introduzca un numero: ");

**long** num = scanInput.nextLong();

**if** (num<0) {

System.***out***.println("El factorial no esta definido"

+ " para numeros negativos");

} **else** {

**long** fac = *fact*(num);

System.***out***.println("El factorial de "+num+" es "+fac);

}

scanInput.close();

}

1. Añada esta modificación mediante un *commit* con el mensaje “correccion numeros negativos”, y envíe los cambios en **master** al repositorio de GitHub. El software revisado es distribuido a los clientes, que respiran aliviados por la desaparición de este problema.
2. El departamento de calidad ya ha evaluado la implementación recursiva, y le pide que haga una optimización para reducir en lo posible el número de llamadas recursivas. Vuelva a la rama **recursivo** y modifique la sentencia condicional del método fact(), cambiando de  **if** (num<=0) a **if** (num<=1) . Añada esta modificación mediante un *commit* con el mensaje “optimizacion”.
3. Una vez el departamento de calidad da el visto bueno, es hora de portar la nueva implementación a la rama que representa el software que se vende a los clientes. Vuelva a la rama **master** y mezcle los cambios de la rama **recursivo** (git merge recursivo). Por suerte, en este caso no hay conflictos que git no sea capaz de resolver, por lo que la mezcla se lleva a cabo sin requerir asistencia humana.
4. El arquitecto del proyecto revisa el código y considera que la entrada y la salida de la aplicación deben estar en métodos separados en vez de en el main(). Primero, cree una nueva rama **refacIO** y cambie el espacio de trabajo a ella. Acto seguido, empiece por refactorizar la entrada en un método aparte, sacando el código pertinente de main() a getNumber():

**public** **static** **long** getNumber() {

Scanner scanInput = **new** Scanner(System.***in***);

System.***out***.print("Introduzca un numero: ");

**long** num = scanInput.nextLong();

scanInput.close();

**return** num;

}

**public** **static** **void** main(String []args) {

**long** num = *getNumber*();

**if** (num<0) {

System.***out***.println("El factorial no esta definido" +

" para numeros negativos");

} **else** {

**long** fac = *fact*(num);

System.***out***.println("El factorial de "+num+" es "+fac);

}

}

1. Añada dicha modificación mediante un *commit* con mensaje “refactorizacion de metodo de entrada”. Acto seguido, refactorice la salida a otro método:

**public** **static** **void** showFac(**long** num) {

**if** (num<0) {

System.***out***.println("El factorial no esta definido" +

" para numeros negativos");

} **else** {

**long** fac = *fact*(num);

System.***out***.println("El factorial de "+num+" es "+fac);

}

}

**public** **static** **void** main(String []args) {

**long** num = *getNumber*();

*showFac*(num);

}

1. Antes de poder hacer añadir los cambios y efectuar el *commit*, llama un cliente quejándose de que no puede calcular el factorial para entradas mayores que 20. No es posible terminar el trabajo de refactorización ahora. Guarde temporalmente en el *stash* los cambios que acaba de hacer al código (git stash) y vuelva a la rama **master**. En la rama máster, cambie el código para usar BigInteger en lugar de **long**:

**import** java.math.BigInteger;

**import** java.util.Scanner;

**public** **class** factorial {

**public** **static** BigInteger fact(BigInteger num) {

**if** (num.compareTo(BigInteger.***ONE***)<=0) {

**return** BigInteger.***ONE***;

} **else** {

**return** num.multiply(*fact*(num.subtract(BigInteger.***ONE***)));

}

}

**public** **static** **void** main(String []args) {

Scanner scanInput = **new** Scanner(System.***in***);

System.***out***.print("Introduzca un numero: ");

BigInteger num = BigInteger.*valueOf*(scanInput.nextLong());

**if** (num.compareTo(BigInteger.***ZERO***)<0) {

System.***out***.println("El factorial no esta definido"

+ " para numeros negativos");

} **else** {

BigInteger fac = *fact*(num);

System.***out***.println("El factorial de "+num+" es "+fac);

}

scanInput.close();

}

}

1. Añada esta modificación con un *commit* con mensaje "cambio de long a BigInteger". Envíe los cambios en **master** al repositorio en GitHub. Acto seguido, puede cambiar a la rama **refacIO**. La rama aparentemente sólo contiene el último *commit*, pero podemos recuperar los cambios que guardamos anteriormente en el *stash* de forma temporal (git stash pop). Una vez recuperados dichos cambios, los añadimos en un nuevo *commit* con mensaje "refactorizacion de metodo de salida".
2. Una vez terminados los cambios en la rama **refacIO**, queremos integrarlos en **master**. Cambiamos a la rama **master** y mezclamos los cambios de **refacIO**. No obstante, el comando de mezclado falla con el siguiente mensaje:

Auto-merging factorial.java

CONFLICT (content): Merge conflict in factorial.java

Automatic merge failed; fix conflicts and then commit the result.

1. Para resolver este problema, debe de abrir el archivo (o los archivos, en casos más complicados que el presente) en el que se indica que hay conflictos. En general, cuando en un archivo hay segmentos que git no puede mezclar de forma automática, indica cada uno de dichos segmentos con marcadores, de la siguiente manera:

<<<<<<< HEAD

… contenido originario de la rama de destino (en la que se está realizando la mezcla) …

=======

… contenido originario de la rama de origen (desde la que se está realizando la mezcla) …

>>>>>>> nombre\_de\_la\_rama\_de\_origen

Según cómo se hayan copiado los fragmentos de código en esta práctica (el sistema de mezclado automático es muy sensible a las discrepancias en los espacios en blanco y los saltos de línea), habrá uno o más segmentos conflictivos. Un posible resultado con dos segmentos conflictivos en el fichero factorial.java sería:

**import** java.math.BigInteger;

**import** java.util.Scanner;

**public** **class** factorial {

**public** **static** BigInteger fact(BigInteger num) {

**if** (num.compareTo(BigInteger.***ONE***)<=0) {

**return** BigInteger.***ONE***;

} **else** {

**return** num.multiply(*fact*(num.subtract(BigInteger.***ONE***)));

}

}

**public** **static** **long** getNumber() {

Scanner scanInput = **new** Scanner(System.in);

System.out.print("Introduzca un numero: ");

<<<<<<< HEAD

BigInteger num = BigInteger.valueOf(scanInput.nextLong());

**if** (num.compareTo(BigInteger.ZERO)<0) {

=======

**long** num = scanInput.nextLong();

scanInput.close();

**return** num;

}

**public** **static** **void** showFac(**long** num) {

**if** (num<0) {

>>>>>>> refacIO

System.out.println("El factorial no esta definido" +

" para numeros negativos");

} **else** {

BigInteger fac = fact(num);

System.out.println("El factorial de "+num+" es "+fac);

}

<<<<<<< HEAD

scanInput.close();

=======

}

**public** **static** **void** main(String []args) {

**long** num = getNumber();

showFac(num);

>>>>>>> refacIO

}

}

Como se puede apreciar, la función fact() ha sido mezclada automáticamente y de forma correcta, pero no así las funciones getNumber, showFac() y main(). Normalmente, resulta suficiente con editar únicamente el contenido de los segmentos marcados como conflictivos, aunque en este caso en particular hay que hacer cambios también fuera del mismo para que el código sea correcto. Lo que se hizo en la rama **refacIO** fue mover algunas líneas de main() a métodos aparte, luego lo que hay que hacer es mover las líneas con sentencias que usan variables de tipo BigInteger, sobre-escribiendo con cada una la línea correspondiente que use variables de tipo **long**, así como cambiar todas las apariciones de **long** a BigInteger, y eliminar líneas redundantes. El resultado será:

**import** java.math.BigInteger;

**import** java.util.Scanner;

**public** **class** factorial {

**public** **static** BigInteger fact(BigInteger num) {

**if** (num.compareTo(BigInteger.***ONE***)<=0) {

**return** BigInteger.***ONE***;

} **else** {

**return** num.multiply(*fact*(num.subtract(BigInteger.***ONE***)));

}

}

**public** **static** BigInteger getNumber() {

Scanner scanInput = **new** Scanner(System.***in***);

System.***out***.print("Introduzca un numero: ");

BigInteger num = BigInteger.*valueOf*(scanInput.nextLong());

scanInput.close();

**return** num;

}

**public** **static** **void** showFac(BigInteger num) {

**if** (num.compareTo(BigInteger.***ZERO***)<0) {

System.***out***.println("El factorial no esta definido" +

" para numeros negativos");

} **else** {

BigInteger fac = *fact*(num);

System.***out***.println("El factorial de "+num+" es "+fac);

}

}

**public** **static** **void** main(String []args) {

BigInteger num = *getNumber*();

*showFac*(num);

}

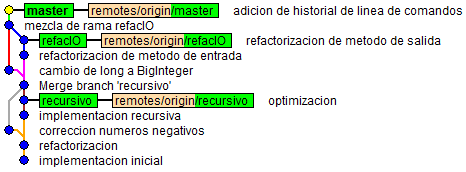
}

1. Tras modificar el fichero, añada los cambios como un *commit* con mensaje "mezcla de rama refacIO". Envíe los cambios en **master** al repositorio de GitHub.
2. El departamento de recursos humanos da la directiva de documentar la operativa para educar a futuros empleados. Cree un nuevo fichero de texto en el repositorio, comandos-git.txt, en el que debe de copiar todos los comandos que ha ejecutado hasta ahora durante la práctica.

**MUY IMPORTANTE**: Antes de cada comando en el fichero comandos-git.txt, introduzca comentarios describiendo exhaustivamente qué hace cada comando y por qué es necesario, para guiar a los futuros empleados, así como cuál fue el número de ítem por el cual fue ejecutado. Por ejemplo, el primer comando (git clone url) fue ejecutado en el ítem 1.

1. Añada dicho fichero al repositorio mediante un nuevo *commit* con el mensaje “adicion de historial de línea de comandos”.
2. El departamento de recursos humanos le conmina a sincronizar todas las ramas de su repositorio local con su repositorio de GitHub (git push --all origin) de modo que los nuevos empleados puedan disponer de su repositorio.

Tras realizar esta práctica, La historia del repositorio resultante debería tener el siguiente aspecto, visto mediante la interfaz gráfica (gitk --all):



Seleccionando cada *commit*, puede ver qué cambios ocurren en cada uno.

**NOTA:** Una vez finalizada esta práctica, habrá visto casi todos los principales aspectos del uso de git a nivel básico, con la importante excepción del comando *git pull*:

git pull nombre\_repositorio nombre\_rama

Dicho comando se usa para recibir nuevos *commits* de repositorios remotos (que habrán sido añadidos por otros desarrolladores), de manera semejante al comando *git push* que ya hemos visto para enviar nuevos *commits* a repositorios remotos:

git push nombre\_repositorio nombre\_rama

El comando *git pull* no entraña dificultades adicionales importantes respecto a *git push*.