1. Buenas Prácticas en el Desarrollo de Proyectos I

Introducción al control de versiones.

Hipólito Guzmán Miranda
Departamento de Ingeniería Electrónica
Universidad de Sevilla
hipolito@gie.esi.us.es

¿Qué es el control de versiones?

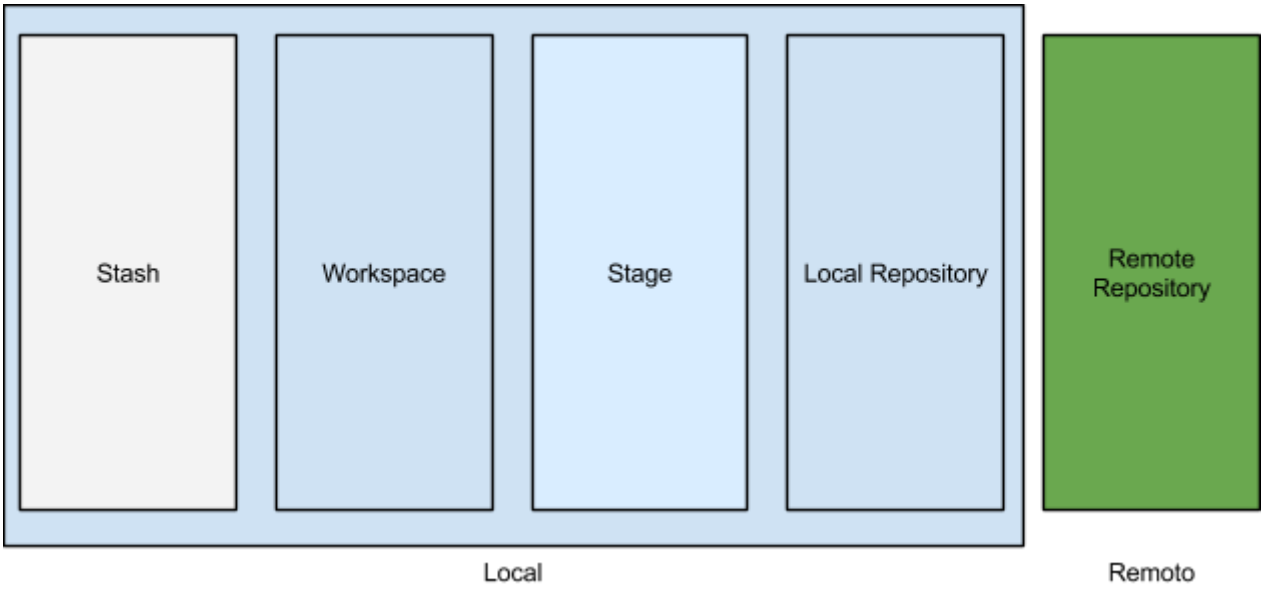
El control de versiones, también llamado control de revisiones o control de código, es la gestión de los cambios que se van realizando sobre ficheros de código, texto, u otro tipo de información.

Un sistema de control de versiones es un software que permite gestionar los cambios que se van realizando sobre un conjunto de ficheros, de forma que se puedan recuperar en cualquier momento versiones válidas del código, etiquetar versiones específicas, y permitir el trabajo simultáneo de diferentes desarrolladores. Los sistemas de control de versiones pueden y deben usarse en proyectos que trabajen con código software (C, C++, Java, etc...), lenguajes de descripción de hardware (VHDL, Verilog, …), documentos (texto plano, Latex, etc...), … Se puede poner cualquier tipo de fichero bajo control de versiones, aunque los ficheros de texto son más sencillos de manejar. Normalmente no se incluyen binarios que pueden generarse a partir de ficheros fuente que estén en el repositorio.

Existen multitud de herramientas diferentes de control de versiones. En esta práctica vamos a usar `git`, pero también existen otras herramientas como `svn` o `mercurial`.

Arquitectura de git:

La arquitectura de git define varias áreas de trabajo. Es importante conocerlas para entender bien los mensajes que nos pueda dar la herramienta:



Repositorio remoto: es una copia completa del historial de versiones del proyecto. Contiene información suficiente para reconstruir cualquier versión del mismo.

Repositorio local: es una copia completa del repositorio remoto. Al igual que el repositorio remoto, contiene toda la información del proyecto.

Workspace (espacio de trabajo). Es el directorio en el que trabajaremos. Se ve como un directorio más de nuestra máquina.

Stage (o index): es una caché de los ficheros que se van a añadir o cambiar en el repositorio local. En el repositorio local no se copian los cambios que haya en el workspace directamente, sino que primero se añaden a stage y luego de stage se copian en el repositorio local. Esto será importante para entender el funcionamiento del comando `git add`.

Stash: es un área temporal donde se pueden esconder temporalmente cambios mientras se trabaja en otra cosa.

A continuación, veremos unos comandos básicos para manejarnos con git.

Accediendo a la máquina de prácticas:

Entra con tu usuario en la máquina de prácticas:

```
ssh usuario@woden.us.es
```

Si aún tienes el password por defecto, cámbialo inmediatamente:

```
passwd
```

Configurando git:

La configuración de git puede editarse en tres ficheros diferentes:

Sistema: `/etc/gitconfig`
Usuario: `~/.gitconfig`
Repositorio: `<path_al_repo>/.git/config`

La configuración específica de repositorio tiene prioridad sobre la de usuario, que a su vez tiene prioridad sobre la de sistema.

Editamos `~/.gitconfig` , que es la configuración a nivel de usuario. En la máquina de prácticas tenemos disponibles los editores `vi`, `vim`, `nano` y `emacs`. Si no tienes experiencia utilizando vim, se recomienda usar nano o emacs.

Un fichero de configuración de ejemplo sería:

```
[user]
        name = Hipolito Guzman
        email = hipolito@gie.esi.us.es
[core]
        editor = nano
[ui]
        color = auto
[push]
        default = matching
```

Para comprobar la configuración que está viendo git, puedes hacer:

```
git config --list
```

Git help:

En esta práctica no va a dar tiempo a que veamos todo git en profundidad. Algunos comandos pueden recibir más argumentos de los que veremos aquí. En caso de duda, puedes utilizar "git help" para obtener más información sobre git o sobre algún comando concreto:

```
git help [comando]
```

Git básico:

Lo primero que haremos será clonar un repositorio remoto. Al clonarlo, crearemos el repositorio local, sobre el que trabajaremos, y el workspace (directorio de trabajo).

Si ya estamos en la misma máquina en la que se encuentra el repositorio:

```
git clone /var/git/nombrerepo.git
```

Si estamos en una máquina diferente debemos acceder a través de ssh (secure shell):

```
git clone usuario@woden.us.es:/var/git/nombrerepo.git
```

Vamos a clonar el repositorio `pseNN`, donde `NN` es el número del grupo de proyecto que tengamos asignado {01, 02, 03, …}, y sobre este repositorio realizaremos la práctica.

Los comandos básicos para trabajar con el repositorio local son:

`git add <fichero>` (añade el fichero al control de versiones: realiza una copia del fichero en el stage)

`git rm <fichero>` (elimina un fichero del workspace y del stage)

`git mv <fichero> <destino>` (mueve un fichero en el workspace y en el stage)

`git status` (muestra el estado actual del repositorio, indicando los ficheros modificados y si existen diferencias entre el repositorio local y el remoto)

`git commit` (añade los cambios que estén en el stage al repositorio local)

Es muy recomendable que el mensaje de commit siga el siguiente formato, para que el log resultante sea entendible:

La primera línea debe tener, como máximo, 50 caracteres, y debe ser un resumen conciso de los cambios. Si vamos a añadir una descripción más extensa, la segunda línea debe dejarse en blanco. A partir de la tercera línea, se puede opcionalmente realizar una descripción más concreta de los cambios realizados. Estas líneas deben tener 72 caracteres como máximo.

`git log` (muestra el historial de commits)

`git log <fichero>` (muestra el historial de commits para un fichero específico)

`git log --oneline --graph` (un ejemplo de argumentos que podemos pasar a git log)

`git diff [fichero]` (muestra los cambios no añadidos al stage, es decir, las diferencias entre working directory y stage. Si no se especifica ningún fichero, muestra todas las diferencias)

Recuperando versiones anteriores:

Antes de hacer un commit, si hemos añadido un cambio que no queremos que se copie al repositorio local, podemos hacer lo que se conoce como un “unstage”, es decir, quitar del stage los cambios añadidos:

```
git reset HEAD <file>
```

También podemos recuperar la versión del repositorio local de un fichero si hacemos:

`git checkout -- <file>` (recupera la versión actual)

Si no queremos la última versión, sino una anterior, podemos hacer:

`git checkout <checksum> <file>` (recupera la versión indicada por el checksum)

Además, podemos pasar los checksum de los commits a git diff si queremos comparar entre commits específicos. Se puede ver la sintaxis concreta haciendo `git help diff`

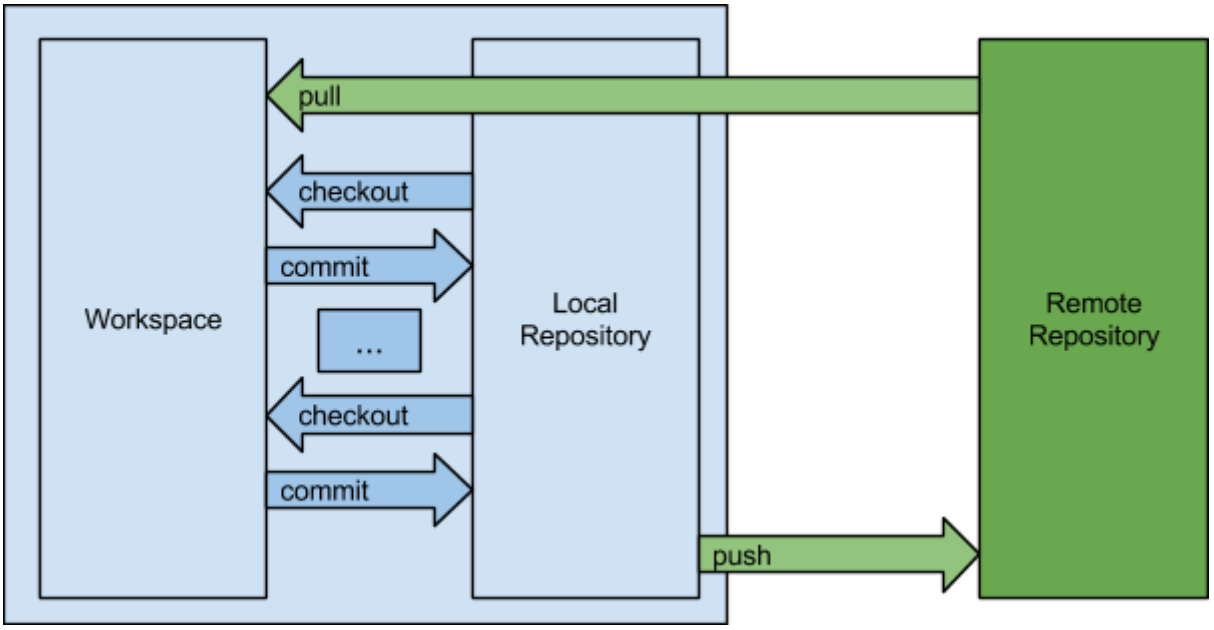
Trabajando con el repositorio remoto:

Para comunicar el repositorio local con el remoto utilizaremos los comandos pull y push:

`git pull` (descargar cambios del repositorio remoto)

`git push` (empujar tus commits al repositorio remoto)

Un esquema típico de una sesión de trabajo con un repositorio remoto sería el siguiente:



De esta forma, en primer lugar se descargan los cambios más recientes del repositorio remoto (pull). Sobre estos cambios se trabaja, y localmente se van realizando commits (y checkouts cuando son necesarios). Finalmente se hace push y se suben los cambios al repositorio remoto. Por supuesto, se puede hacer push y pull más veces si es necesario, por ejemplo, si necesitamos descargar cambios que acaba de hacer otro miembro del equipo.

Etiquetando:

Una etiqueta refiere a una versión específica del código. Es interesante utilizarlas para no tener que referirnos a una versión concreta con un hash como "a719fdaed776006a2fefe80267c445514d92a6bd"

Existen etiquetas ligeras (lightweight, por defecto) y anotadas (annotated). Estas últimas pueden llevar un mensaje.

`git tag` (muestra etiquetas)

`git tag -a <nombretag> [checksum]` (crea una etiqueta anotada)

Las etiquetas no se envían por defecto al repositorio remoto. Si queremos enviar la etiqueta al repositorio remoto tenemos que 'pushearla' explícitamente:

`git push --tags` ('pushea' todas las etiquetas)

`git push origin [tagname]` ('pushea' una etiqueta específica)

Resolución de conflictos:

Cuando varias personas intentan modificar la misma parte del mismo fichero (supongamos por ejemplo, los usuarios A y B), el segundo desarrollador que quiera incluir sus cambios (B) recibirá un mensaje de error de este estilo al hacer `git pull`:

```
CONFLICT (content):
Merge conflict in README Automatic merge failed; fix conflicts and then commit the
result.
```

En este caso el mensaje de error nos está indicando que hay un conflicto en el archivo README. Si abrimos el fichero con un editor de texto:

```
This is the first line
This is the second line
<<<<<<< HEAD
This is B's third line
=======
This is A's third line
>>>>>>> 58828d47b756aaed335d0118d3b09a608b05bf5e
```

Lo que le está mostrando al usuario B el conflicto entre la versión del usuario A (commit 58828d47...) y la versión del usuario B (HEAD, el commit en el que está el usuario).

Para arreglar el conflicto el usuario tiene que decidir manualmente con qué código final se queda para esa parte del fichero y eliminar las líneas de control “<<<...”, “===...”, “>>>...”, tras lo cual debe añadir el fichero con `git add` y hacer `commit` normalmente.

Por ejemplo, el usuario podría modificar el fichero con conflicto para dejarlo de la siguiente manera:

```
This is the first line
This is the second line
This is A and B's third line
```

Tras lo cual el conflicto estaría resuelto. No se añadirá el fichero al repositorio hasta que no se incluya el fichero en el mismo utilizando `git add` y `git commit`.

Ejercicio:

Se realizará un programa entre los miembros del grupo de trabajo, de forma que cada miembro trabaje en su ordenador y suba sus cambios al repositorio. Se propone el siguiente enunciado y estructura de ficheros dentro del repositorio `pseNN`:

`src/fib.h`

Debe incluir la declaración y definición de una función llamada “fib”, que recibe un entero, n, y devuelve un entero cuyo valor debe ser el elemento n-ésimo de la sucesión de fibonacci, tomando `fib(0) = 0` y `fib(1) = 1`.

`src/sort.h`

Debe incluir la declaración y definición de una función llamada “sort”, que recibe el puntero a un array de 4 enteros y los ordena sobre el mismo array. No devuelve nada (void).

`src/print.h`

Debe incluir la declaración y definición de una función llamada “printline”, que recibe dos números enteros e imprime una línea con ambos números, con el siguiente formato:

`i = <argumento1>  <tabulador> fib(i) = <argumento2>`

Dicha función devuelve void.

`src/main.c`:

El programa debe recibir 4 argumentos, además del nombre de programa (argumento 0). El programa debe avisar por pantalla, y terminar, si recibe un número de argumentos inesperado. El programa interpretará cada uno de estos argumentos como si fueran un número entero, pero (por simplicidad) no realizará ningún chequeo sobre los argumentos.

El programa debe ordenar de menor a mayor los argumentos (utilizando la función definida en `sort.h`), calcular el valor `fib(i)` (utilizando la función definida en `fib.h`) de cada uno de éstos argumentos, e imprimir cada pareja `i, fib(i)` (utilizando la función definida en `print.h`).

`Makefile`:

Se debe realizar un fichero Makefile para posibilitar la compilación del código en un único paso (esto también es una buena práctica para el desarrollo). Se recuerda que el comando para compilar ficheros c es “`gcc -g -Wall <ficheros.c> -o <nombresalida.out>`”.

Opcionalmente, es buena práctica también incluir un target “clean” en el Makefile que elimine los ficheros `.o` generados por el compilador `gcc`.

Trabajo no presencial:

1) Busca un core descrito en hardware (VHDL o Verilog) que sea de código libre (puedes empezar a buscar en opencores.org) que pienses que te podría servir para una posible implementación del proyecto de la asignatura. Justifica la decisión y añade el código al repositorio.

2) Branching y merging: (Ramificando y fusionando)
A veces es necesario hacer cambios en una parte del código, sin afectar inmediatamente al resto del proyecto (por ejemplo, al añadir una funcionalidad nueva). Para esto, las herramientas de control de versiones suelen tener funcionalidades de branching y merging. Lee sobre branching y merging en git (puedes empezar a buscar información por las referencias de la práctica) y estudia cómo es el proceso y qué comandos se utilizan.

3) Git blame, git bisect:
Lee sobre los comandos de git 'git blame' y 'git bisect' y explica qué hacen y cómo se utilizan. ¿Cuándo podrían ser útiles dichos comandos en el contexto del desarrollo de proyectos?

El trabajo no presencial se evaluará durante las sesiones de seguimiento.

Agradecimientos:

A Luis Sanz por introducir el control de versiones en nuestro grupo de trabajo y por su inestimable ayuda para preparar esta práctica.

Referencias:

Scott Chacon, "Pro Git", http://git-scm.com/book
Git reference: http://git-scm.com/docs