Sistema de control de versiones

Qué es control de versiones?

- Gestión del desarrollo de cada elemento de un proyecto a lo largo del tiempo
- Proporciona
    - Mecanismos de almacenaje de cada elemento que deba gestionarse
        - archivos de código
        - imágenes
        - Documentación
    - Posibilidad de añadir, modificar, mover, borrar
    - Historial de las acciones realizadas con cada elemento pudiendo volver a un estado anterior
    - Otros: generación de informes de cambios, informes de estado, marcado con nombre identificativo, etc
- Se trabaja sobre un repositorio, donde se almacena la información de todo el desarrollo
    - útil para trabajar individualmente o en grupo
    - Alojado en un servidor local o remoto
    - Permite desarrollos colaborativos, incluso concurrentemente
    - Todo buen equipo profesional de desarrollo de software lo utiliza

Tipos de VCS

- Centralizados
    - VCS
        - El repositorio se encuentra e una localización única
        - Software esta en mi ordenador pero el repo en otro lado
        - Es necesario tener acceso a la ubicación del repositorio para poder trabajar
            - esto implica generalmente acceso a la red (Local o internet)
        - Ejemplos
            - sccs, RCS, CVS
- Distribuidos
    - DVCS
        - No hay un repositorio único sino múltiples copias (clones)
        - Para trabajar, un usuario debe obtener una copia local del repositorio, pero no necesita acceso a la red más que para publicar sus cambios

        - Ejemplo
            - git(C), Mercurial, Bazaar (Python), fossil (C)

Ciclos básico de trabajo

- Crea copia local
    - Checkout
    - Se puede especificar una revisión o fecha particular
- Actualizar la copia de trabajo
    - Update
    - Permite recuperar las últimas modificaciones del repositorio e integrarlas en la copia de trabajo
- Realizar cambios
    - add, delete copy, move
- Examinar cambios
    - status, diff, revert
- Fusionar cambios
    - merge, resolved
- Enviar cambios
    - commit
        - dar por bueno
        - validar
    - Requiere un mensaje "log" que detalla las modificaciones realizadas
- Volver al punto 1
    - update

Vocabulario básico

- Copia de trabajo
    - Working copy
    - Copia local de los ficheros de un repositorio (de una revisión específica)
- Check-out
    - co
    - crea una copia de trabajo local desde el repositorio
    - Se puede especificar una revisión; si no se hace se toma la última
- Check-in o commit (ci)
    - Integra los cambios hechos a una copia local en el repositorio
- Import
    - copia un árbol de directorios local (que no es ese momento una copia de trabajo) en el repositorio por primera vez

- Actualizar (update)
  - Integra los cambios que han sido hechos en el repositorio (por ejemplo por otras personas) en la copia de trabajo local
- Conflicto
  - Ocurre cuando se realizan dos cambios al mismo documento, y el sistema es incapaz de reconciliar los mismos
- Resolver
  - Intervención del usuario para atender un conflicto entre diferentes cambios al mismo documento

Modelos para resolver problemas/Conflictos

- Candados
  - no se puede trabajar al mismo tiempo
  - candado puede quedar bloqueado para siempre
- Merge
  - hay que tener la ultima version subida en local para poder subir al repo

Cliente vs servidor

- Qué hace el apache
  - Proteger, cuidar

Consejos y buenas prácticas

- No entregar código que no funciona o inacabado
  - Si hay que hacerlo, crear una rama y mezclarla cuando funcione
- Añadir siempre un mensaje en cada commit explicando qué se entrega y por qué
- Ejecutar update con mucha frecuencia y siempre al principio de una sesión de desarrollo
- Ejecutar commit con frecuencia
  - las entregas muy grandes y espaciadas aumentan la probabilidad de conflictos
- Ante la duda, resolver los conflictos con el resto de los miembros del equipo
- No versionar ficheros autogenerados por herramientas
- Usar un repositorio por proyecto

Control de versiones con Git

Git History

- Came out of linux development community
- Linus Torvalds, 2005
- Initial goals
    - Speed
    - Support for non-linear development (thousands of parallel branches)
    - Fully distributed
    - Able to handle large project projects like Linux efficiently

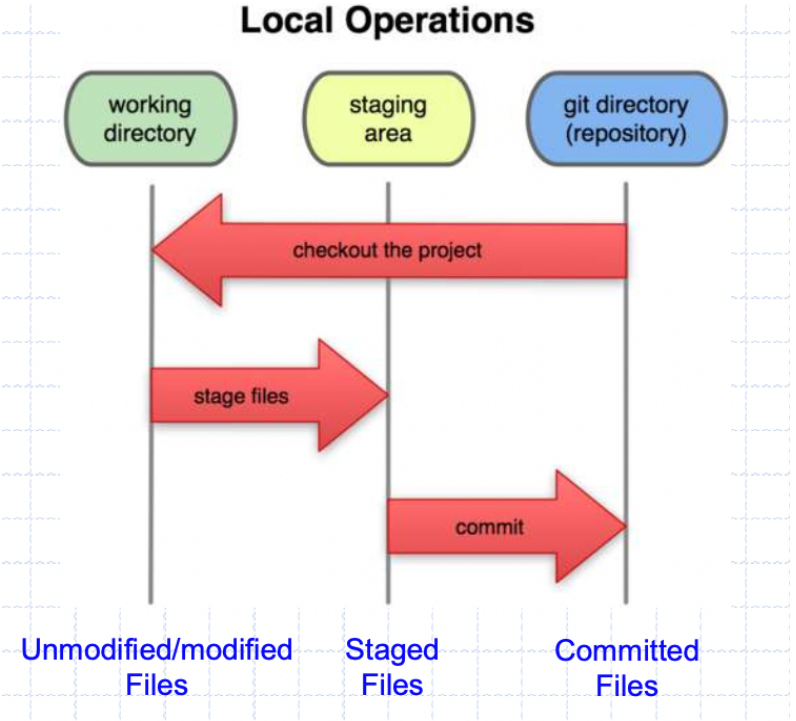
SVN vs Git

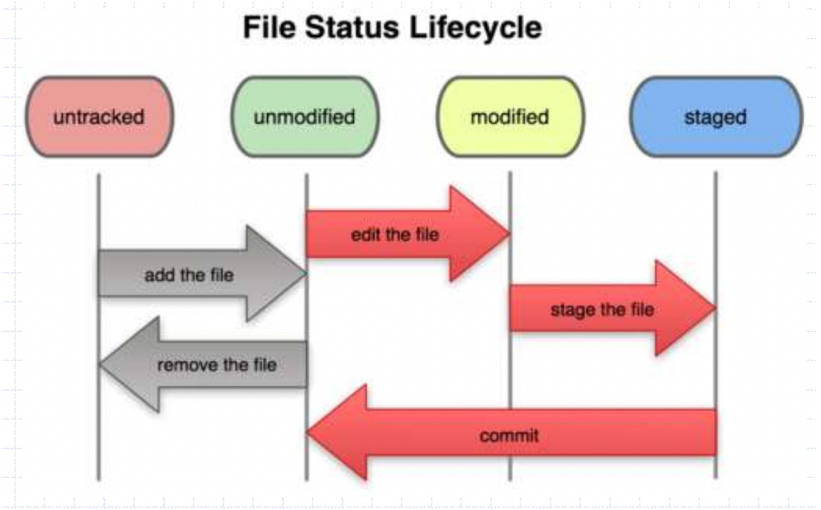
- SVN/CVS
    - Central repository approach - the main repository is the only "true" source, only the main repository has the complete file history
    - Users check out local copies of the current version
- Git/Mercurial
    - Distributed repository approach - every checkout of the repository is a full fledged repository, complete with history
    - Greater redundancy and speed
    - Branching and merging repositories is more heavily used as a result
    - Git takes snapshots

Git uses checksums

- In subversion each modification to the central repo incremented the version number of the overall repo
    - How will this numbering scheme work when each user has their own copy of the repo, and commits changes to their local copy of the repo before pushing to the central server
- Instead, Git genereates a unique SHA-1 hash (40 character string of hex digits), for every commit
    - Refer to commits by this ID rather than a version number
    - We usually only see the first 7 characters (enough)

A local git project has three areas





Basic workflow

- Basic Git Work
  - Modify files in your working directory
  - Stage files, adding snapshots of them to your staging area
  - Do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your git directory
- Notes
  - If a particular version of a file is in the git directory, its considered committed
  - If its modified but has been added to the staging area, it is staged

    - If it was changed since it was checked out but has no been staged, its is modified.

So what is github?

- Is a site for online storage of git repositories
    - Many open source projects use it, such as the Linux Kernel
    - You can get free space for open source projects or you can pay for private projects
    - PaaS el sw es gratuito entonces dan un servicio
    - Es una empresa y ofrece un servicio de alojamiento de repos

Get ready to use git

1. Set the name and email for Git to use when you commit:

```
$ git config --global user.name "Bugs Bunny"
$ git config --global user.email bugs@gmail.com
```

- You can call `git config -list` to verify these are set.
  - These will be set globally for all Git projects you work with.
- You can also set variables on a project-only basis by not using the `--global` flag.
- You can also set the editor that is used for writing commit messages:
  `$ git config --global core.editor vim`
  (it is `vi` by default)

2. Two common scenarios: (only do one of these)

a) Clone an already existing repo to your current directory:

```
$ git clone <url> [local dir name]
```

This will create a directory named *local dir name,* containing a working copy of the files from the repo, and a `.git` directory (used to hold the staging area and your actual repo)

b) Create a Git repo in your current directory:

```
$ git init
```

This will create a `.git` directory in your current directory.

Then you can commit files in that directory into the repo:

```
$ git add file1.java
$ git commit -m "Initial project version"
```

command	description
`git clone url [dir]`	copy a git repository so you can add to it
`git add files`	adds file contents to the staging area
`git commit`	records a snapshot of the staging area
`git status`	view the status of your files in the working directory and staging area
`git diff`	shows diff of what is staged and what is modified but unstaged
`git help [command]`	get help info about a particular command
`git pull`	fetch from a remote repo and try to merge into the current branch
`git push`	push your new branches and data to a remote repository
others: `init, reset, branch, checkout, merge, log, tag`	

- The first time we ask a file to be tracked, *and* every time before we commit a file we must add it to the staging area:

```
$ git add README.txt hello.java
```

This takes a snapshot of these files at this point in time and adds it to the staging area.

- To move staged changes into the repo we commit:

```
$ git commit -m "Fixing bug #22"
```

To unstage a change on a file before you have committed it:

```
$ git reset HEAD -- filename
```

To unmodify a modified file:

```
$ git checkout -- filename
```

These commands are just acting on your local version of repo.

» To view the **status** of your files in the working directory and staging area:

```
$ git status
```

or

```
$ git status -s
```

(`-s` shows a short one line version similar to svn)

» To see what is modified but unstaged:

```
$ git diff
```

» To see staged changes:

```
$ git diff --cached
```

» To see a log of all changes in your local repo:

```
$ git log
```

or

```
$ git log --oneline
```

(to show a shorter version)

```
1677b2d Edited first line of README
258efa7 Added line to README
0e52da7 Initial commit
```

`git log -5` (show only the 5 most recent updates)

» Notes:

- changes will be listed by **commitID** (SHA-1 hash)
- changes made to the remote repo before the last time you cloned/pulled from it will also be included here

- Good practice:
  1. Add and Commit your changes to your local repo
  2. Pull from remote repo to get most recent changes (fix conflicts if necessary, add and commit them to your local repo)
  3. Push your changes to the remote repo
- To fetch the most recent updates from the remote repo into your local repo, and put them into your working directory:

```
$ git pull origin master
```

- To push your changes from your local repo to the remote repo:

```
$ git push origin master
```

Notes: `origin` = an alias for the URL you cloned from

`master` = remote branch you are pulling from/pushing to,

(the local branch you are pulling to/pushing from is your current branch)

- To create a branch called experimental:

```
$ git branch experimental
```

- To list all branches: (* shows which one you are currently on)

```
$ git branch
```

- To switch to the experimental branch:

```
$ git checkout experimental
```

- Later on, changes between the two branches differ, to merge changes from experimental into the master:

```
$ git checkout master
$ git merge experimental
```

- Notes
  - `git log --graph` can be useful for showing branches.
  - These branches are in your local repo!