Tema 10 - Sistemas de Control de Versiones de última generación (DCA)

HISTORIAL DE REVISIONES

NÚMERO	FECHA	MODIFICACIONES	NOMBRE

Índice

Logo DLSI

Tema 10 - Sistemas de Control de Versiones de última generación
Curso 2018-2019

1. ¿Qué es un Sistema de Control de Versiones (SCV)?

- Se trata de un software que nos permite conocer información de las modificaciones que vamos haciendo a nuestro software.
- En los `scv` de hoy en día esta información se aplica como un todo al conjunto de archivos de nuestro producto que hemos ido modificando de alguna manera.
- Con estas modificaciones podemos:
  - Mantener la ultima versión de los archivos que queramos.
  - Volver a alguna versión anterior. . . *no necesariamente la última*.
  - Aislarlas para poder proporcionarlas a otro desarrollador como un *parche*.
- Pero si nos fijamos detenidamente, todo esto puede hacerse manualmente. . .

2. ¿En qué consiste el control de versiones?

¿Qué nos aportan realmente?

- La gestión automática de los cambios que se realizan sobre uno o varios ficheros de un proyecto.
- Restaurar cada uno de los ficheros de un proyecto a un estado de los anteriores por los que ha ido pasando (no solo al inmediatamente anterior).
- Permitir la colaboración de diversos programadores en el desarrollo de un proyecto.
- Un histórico de las acciones llevadas a cabo, cúando y por quién. . . cosa muy útil cuando un proyecto es desarrollado por varias personas de manera simultanea.

3. Conceptos generales de los SCV (I)

Repositorio
Es la *copia maestra* donde se guardan todas las versiones de los archivos de un proyecto.

Copia de trabajo
La copia de los ficheros del proyecto que podemos modificar. Cada desarrollador tiene la suya propia.

Check Out / Clone
La acción empleada para obtener una copia de trabajo desde el repositorio. En los `scv` distribuídos -como Git- esta operación se conoce como *clonar* el repositorio por que, además de la copia de trabajo, proporciona a cada programador su copia local del repositorio a partir de la *copia maestra* del mismo.

Check In / Commit
La acción empleada para llevar los cambios hechos en la copia de trabajo a la copia local del repositorio -*CheckIn*-. Esto crea una nueva *revisión* de los archivos modificados. Cada *commit* debe ir acompañado de un *Log Message* el cual es un comentario -*Una cadena de texto que explica el commit*- que añadimos a una revisión cuando hacemos el *commit* oportuno.

4. Conceptos generales de los SCV (II)

Push
La accion que traslada los contenidos de la copia local del repositorio de un programador a la copia maestra del mismo.

Update/Pull/Fetch+Merge/Rebase
Acción empleada para actualizar nuestra copia local del repositorio a partir de la copia maestra del mismo, además de actualizar la copia de trabajo con el contenido actual del repositorio local.

Conflicto
Situación que surge cuando dos desarrolladores hacen un *commit* con cambios en la *misma región del mismo fichero*. El `scv` lo detecta, pero es el programador el que debe corregirlo.

Etiquetar
Poner un nombre común a todos los ficheros bajo control de versiones de un proyecto en un instante determinado.

Rama
Representa una línea de evolución de nuestro software. Podemos tener todas las que queramos, pero normalmente una es especial: *trunk*, *master*, etc...

5. Tipos de SCV.

Por la forma de almacenar los contenidos

- **Centralizados**. Hay un único repositorio compartido por todos los desarrolladores. Sólo un responsable o grupo de responsables pueden llevar a cabo tareas administrativas en este repositorio. Es el caso de CVS y Subversion.
- **Distribuidos**: Cada usuario tiene su propio repositorio. Los repositorios de todos los desarrolladores del proyecto pueden intercambiar información entre ellos. Es habitual disponer de un repositorio *central* que sirve como sincronización entre todos los repositorios de los desarrolladores. Es el caso de git, mercurial, bazaar, monotone, etc...

Por la forma de modificar los contenidos de la copia local

- **Colaborativos**: Varios desarrolladores pueden estar modificando el mismo fichero simultáneamente. Puden aparecer conflictos por modificar varios desarrolladores la misma parte del archivo. Es el caso de git o mercurial.
- **Exclusivos**: Sólo un desarrollador puede estar modificando un archivo a la vez, el resto sólo pueden *leerlo*. Una vez se han pasado los cambios al repositorio entonces ya lo puede modificar otro desarrollador, pero nuevamente sólo uno. Es el caso de RCS o de SourceSafe.

6. Centralizados vs. Distribuidos en 90sg

7. ¿Qué opciones tenemos disponibles?

Hoy en día disponemos de muchos y diversos *scv*

- bazaar que es el sucesor de arch.
- *CVS*
- darcs
- git
- mercurial
- monotone
- subversion
- plasticscm, breve introducción en la wikipedia
- perforce, breve introducción en la wikipedia
- bitkeeper

8. ¿Qué podemos hacer con un SCV?

Operaciones basicas

- Crear un repositorio
- Clonar un repositorio
- Actualizar un repositorio local con cambios remotos
- Crear un *commit*
- Actualizar un repositorio remoto con cambios locales
- Crear una rama local y hacerla más tarde remota.
- Importar cambios de una rama en otra.

9. Tipos de ramas

Ramas de largo recorrido

- Por lo general indican distintas versiones estables de nuestro software (*v1.0*, *v2.0*, etc...)
- Es el caso de ramas como *master* o *trunk*.
- Algunos proyectos disponen de ramas que complementan a las anteriores: *next*, *proposed*, *pu* (`proposed updates`).

Ramas puntuales

- Son ramas que se crean para trabajar en ellas en un solo aspecto. En inglés se las suele llamar *Topic Branches*.
- Puede representar la corrección de un fallo, el añadido de una característica nueva o incluso probar algo de forma experimental.
- Algunos desarrolladores emplean un prefijo en su nombre para identificar para qué se crearon:
  - Tip: *Una idea* (`tip-speedincrease`, `tip/memsizereduction`, etc...)
  - Wip: *Work in Progress* (`wip-bugdetection`, etc...)
  - Hotfix: *Arreglo de un fallo* (`hotfix/segfault`)
  - Iss: *Un asunto* (**issue**) identificado -fallo, nueva característica, etc... - (`iss32`, `iss32v2`, etc..)

10. Formas de integrar una rama en otra (I)

Mezcla

Supongamos esta situación:

```
master:   A--B--C

topic:    A--B--C
                 \
                  X--Y--Z          ❶

El tiempo pasa: A--B--C--D--E--F   ❷
                          \
                           X--Y--Z

mezcla: A--B--C--D--E--F--M        ❸
                \         /
                 +X--Y--Z+
```

❶ Rama tópico

❷ Tres nuevos commits en master

❸ Un nuevo commit *M* representa la mezcla

11. Formas de integrar una rama en otra (II)

Rebasamiento

- No todos los *scv* disponen de esta posibilidad.
- Aplicado al ejemplo anterior... obtendríamos esta nueva situación:

```
master:   A--B--C

topic:    A--B--C
                 \
                  X--Y--Z          ❶

El tiempo pasa: A--B--C--D--E--F   ❷
                          \
                           X--Y--Z

mezcla: A--B--C--D--E--F           ❸
                        \
                         X'--Y'--Z'
```

❶ Misma rama tópico

❷ Mismos tres nuevos commits en master

❸ La rama tópico se *trasplanta* de su antiguo commit *origen* (**C**) al nuevo (**F**).

Lo que produce una historia más lineal:

```
A--B--C--D--E--F--X'--Y'--Z'
```

- Echa un vistazo a este vídeo que explica de forma sencilla la diferencia entre `rebase` y `merge`:

12. SCV's con los que trabajaremos

- Nos vamos a centrar en tres de ellos. Todos serán distribuídos, ya que con uno distribuído podemos simular el uso de uno centralizado.
- Veremos una breve introducción a cada uno... incluso alguna característica especial que posean.
- Veremos como llevan a cabo las operaciones básicas que hemos descrito anteriormente.

Los tres scv's que emplearemos son:

1. *git*
2. *bazaar*
3. *mercurial*

13. Git (I)

- Los desarrolladores del núcleo `linux` emplean BitKeeper hasta 2005.
- BitKeeper es un scv distribuído. Git también lo es, al igual que `Darcs`, `Mercurial` o `Bazaar` y `Monotone`.
- Linus comienza el desarrollo de `git` el **3 de abril de 2005**, lo anuncia el dia 6 de abril.
- Git se *auto-hospeda* el **7 de abril de 2005**.
- Consta de herramientas de bajo de nivel (*plumbing*) y de alto nivel (*porcelain*). Estas últimas son las que se emplean habitualmente e internamente llaman a las de bajo nivel.
- Cada nuevo *commit* creado en git es una `foto` de nuestro proyecto en ese instante... identificada por un número... su clave SHA-1.
- El algoritmo SHA-1 ya no es considerado seguro y Git ya está implementando el soporte de SHA-256.

14. Git (II)

- Toda la metainformación de *git* se guarda en un directorio que reside en la carpeta raíz de nuestro proyecto y que se llama *.git*:

```
.git
|-- branches
|-- COMMIT_EDITMSG
|-- config
|-- description
|-- FETCH_HEAD
|-- gitk.cache
|-- HEAD
|-- hooks
|-- index
|-- info
|-- logs
|-- objects
|-- ORIG_HEAD
+-- refs
```

15. Git (III)

Uso

- Al igual que ocurre con todas las herramientas de este tipo, existe una órden (`git`) que admite diferentes subórdenes.
- El primer uso que podemos hacer de ella es para saber la versión de `git` que tenemos instalada.

```
git --version
```

- O para iniciar el repositorio local de un proyecto:

```
git init proyecto
```

- funcionamiento habitual...

```
git add .
git status
git commit -m "Primer commit." -m "Descripcion detallada."
git commit -a
```

16. Git (IV)

Configuración local

- Se lleva a cabo en un archivo local al proyecto. Se trata de un archivo de texto.
- Por tanto cada proyecto puede tener su configuración local

```
.git/config
```

Configuración global

- Se lleva a cabo en un archivo particular para cada usuario del s.o. Se trata de un archivo de texto con el mismo formato que en el caso anterior.
- Por tanto cada usuario puede tener su configuración global para todos sus proyectos

```
~/.gitconfig
```

Ejemplos de configuración

```
git config user.name "nombre apellidos"
git config user.email "usuario@email.com"
...
git config --global user.name "nombre apellidos"
git config --global user.email "usuario@email.com"
```

17. Git (V)

Ramas

- Es una de las características más valoradas en git. La facilidad para el trabajo con ramas.
- Además es *barato* trabajar con ramas... a diferencia de otros `scv`

```
git branch [-a] [-r]
git show-branch
...
git checkout [-b] [rama-de-partida]
git log [-p]
```

Estado

```
git status
git log
...
git show
git diff
```

18. Git (VI)

Descartar cambios

```
git reset --hard
git checkout ruta-archivos o rama
```

Repositorios remotos

```
git remote add nombre protocolo
git remote add origin maquina:ruta/hasta/repo
...
[ssh] [http] [git] [git+ssh]
git clone maquina:ruta/hasta/repo
```

Operaciones con repositorios remotos

```
git pull [origin] [rama]
git push [repo] [rama]
git checkout -b rama origin/rama-remota
git fetch
git merge
git pull = git fetch + git merge
git rebase otra-rama
```

19. Git (VII)

Stash

- **Imagina**... estás modificando tu copia de trabajo... montones de archivos modificados (sin hacer un commit aún) desde el último `pull`.

- En ese preciso instante, tu compañero *-al que tanto aprecias-* te dice que la actualices desde el repositorio original porque acaba de subir unos cambios... *importantisimos*... Ah!... y `HAZLO YA`!!
- Ok... lo que tu digas:

```
git stash
git pull
git stash pop
```

20. Git (VIII)

Bisect

- ¿Bi-qué?... búsqueda **B-I-N-A-R-I-A** entre los commits.

```
git bisect start
git bisect bad               # la  version actual esta mal
git bisect good  commit/tag  # una version que sabemos que funciona
```

- Después de ir alternando entre commits *buenos* y *malos* estaremos *parados* en el último commit entre los dos originales que introdujo el cambio erróneo. Para volver al estado original del que hemos partido:

```
git bisect reset
```

- ... *There's one more thing!*

```
git bisect run mi-guion argumentos
```

"`mi-guion`" es un guión de *shell* que devuelve `0`, `1..124`, `126`, `127`

21. Git (IX)

Cherry-Picking

- Al hacer un `merge` o un `rebase`... aplicamos varios `commits` simultáneamente...
- ¿Y si queremos solo uno?..

```
git cherry-pick 7a2ed53
git cherry-pick master
```

Shortlog

- Resume la salida de `git log` en un formato apropiado para anuncios de nuevas versiones...
- Pero también es apropiado para extraer información sobre el trabajo de los desarrolladores. Prueba esto:

```
git shortlog -s -n --all --no-merges
```

22. Git (X)

Aliases

- Podemos crear alias para órdenes+opciones que usemos habitualmente...

```
git config --global alias.co checkout
git config --global alias.ci commit
git config --global alias.st status
git config --global alias.br branch
git config --global alias.hist 'log --pretty=format:"%h %ad | %s%d [%an]" -- ↩
    graph --date=short'
git config --global alias.type 'cat-file -t'
git config --global alias.dump 'cat-file -p'
```

- Y lo que esto hace realmente es añadir a `~./gitconfig` estas líneas:

```
[alias]
co = checkout
ci = commit
st = status
br = branch
hist = log --pretty=format:\"%h %ad | %s%d [%an]\" --graph --date=short
```

23. Git (XI)

sugerencia
Si tu `shell` lo permite... aún puedes abreviar más... , ten en cuenta que esto no específico de `git`.

```
# ~/.profile , ~/.bashrc
# Consulta la documentacion de tu shell.

alias gs='git status '
alias ga='git add '
alias gb='git branch '
alias gc='git commit'
alias gd='git diff'
alias go='git checkout '
alias gk='gitk --all&'
alias gx='gitx --all'

alias got='git '
alias get='git '
```

Un ejemplo
go <branch>

24. Git (XII)

Detached HEAD

- No solo podemos cambiar de rama... sino volver a un punto concreto en el pasado.
- Esto produce una *rama-virtual* cuya cabeza es el numero-*hash* que hemos indicado:

```
$ git checkout 415e7c2
Note: checking out '415e7c2'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 415e7c2... First Commit
$ cat message.txt
Hello, World
```

25. Git (XIII)

Etiquetas / Tags

- Los número que genera `SHA1` para cada *commit*, no proporcionan mucha información a las personas.
- Nos puede interesar más asociar un texto significativo a un *hash*, algo que represente una información para nosotros, p.e.: `v1.0`.
- Eso se hace con la orden `tag` de git.

```
git tag v1.0
```

- Si no indicamos nada... etiquetamos el último commit de la rama actual.
- A partir de este momento podemos emplear esa **etiqueta** en lugar del *hash* asociado al commit que la tiene, p.e:

```
git checkout v1.0
...
git checkout v1.0~3
```

- La órden `git tag` sin parámetros lista las etiquetas que tenemos puestas.

26. Git (XIV)

Bare repos

- Es el término que se emplea en git para referirse al repositorio local de un proyecto... al directorio `.git`.

- No tienen copia de trabajo, sólo *metadatos*.
- Algunas opciones (como `clone`) permiten trabajar con ellos:

```
$ git clone --bare hello hello.git
Cloning into bare repository hello.git...
done.
$ ls hello.git
HEAD
config
description
hooks
info
objects
packed-refs
refs
```

- Habitualmente *-y para identificarlos fácilmente-* estos repositorios se suelen llamar como el proyecto que representan y con la extensión `.git`, p.e.: `hola-mundo.git`.

27. Git (XV)

Hooks

- Los `hooks` son guiones que se ejecutan automáticamente antes o después de determinadas acciones como `commit`, `push`, etc…
- Por ejemplo, en el `hook` pre-commit podríamos comprobar si el mensaje tiene errores ortográficos.
- Cada vez que creamos un repositorio git, disponemos del directorio `.git/hooks` donde ya hay ejemplos de estos guiones:

```
applypatch-msg.sample  post-update.sample         pre-commit.sample
pre-push.sample        update.sample              commit-msg.sample
pre-applypatch.sample  prepare-commit-msg.sample  pre-rebase.sample
```

- Puedes tomar cualquiera de ellos como punto de partida, eso sí:
  1. Quítales el sufijo `.sample`
  2. Dales permiso de ejecución si no lo tienen
- Para tener más información de cada uno de ellos ejecuta:

```
git help hooks
```

28. Git (XVI)

Reflog

- Es la manera en la cual git guarda información de cuándo se actualizan las "*cabezas*" de las ramas.

```
git reflog
git reflog show
```

- Para referirse a los commits emplea la notación:

```
rama@{referencia}
```

```
master@{5}                    ❶
tip/speedincrease@{17}        ❷
bugfix/badcss@{one.month.ago} ❸
```

❶ Dónde apuntaba *master* hace 5 movimientos.
❷ Dónde apuntaba *tip/speedincrease* hace 17 movimientos.
❸ Dónde apuntaba *bugfix/badcss* hace un mes.

Se puede usar, por ejemplo, para recuperar commits perdidos.

Clean

- Elimina de la copia de trabajo los archivos que no están bajo control de versiones. **¡Lleva cuidado al usarla!**

```
git clean
```

Fsck

- Dado que git se puede ver como un sistema de ficheros indexado por contenido... también tiene su herramienta *fsck* o *chkdsk* correspondiente:

```
git fsck
```

29. Git (XVII)

Herramientas gráficas

- gitk
- git gui
- git view
- gitg
- giggle
- gource
- tig
- Y aquí tienes más.

30. Git. Vídeos relacionados

31. Mercurial (I)

- web del proyecto.
- Su nombre le viene del término hidrargirio
- Es un `scv` distribuído. Es *software libre*. Disponible en OS-X, Windows y GNU/Linux.
- Surge aproximadamente a la vez que git, las órdenes que entiende se llaman prácticamente de la misma forma. Escrito en python y partes en `C`.
- Dado que funciona de manera similar a `git` los flujos de trabajo que tenemos disponibles son similares a los que ya conocemos de `git`.
- Toda la información relativa al control de versiones de un proyecto se guarda en el directorio raíz del proyecto, concretamente en el subdirectorio "`.hg`".
- La configuración general para cada usuario se guarda en `~./hgrc` y la configuración local para cada proyecto se guarda en `<repo>/.hg/hgrc`.

32. Mercurial (II)

Ejemplo 1

```
    hg clone http://github.com/repo/hola-mundo
    cd hola-mundo
    (editar ficheros)
    hg add (nuevos ficheros)
    hg commit -m 'Cambios'
    hg push
```

Ejemplo 2

```
    hg init (directorio del proyecto)
    cd (directorio del proyecto)
    (Anyadimos archivos..)
    hg add
    hg commit -m "Primer commit"
```

33. Mercurial (III)

- Admite el añadido de *extensiones*
- Podemos ver las extensiones disponibles con:

```
hg help extensions
```

- Pedir información de una de ellas con:

```
hg help nombre-extension
```

- Las extensiones se deben habilitar, p.e. en `.hg/hgrc`:

```
[extensions]
foo =
# Podemos especificar la ruta concreta
[extensions]
myfeature = ~/.hgext/myfeature.py
```

34. Mercurial (IV)

- También se pueden deshabilitar de una en una:

```
[extensions]
# disabling extension bar residing in /path/to/extension/bar.py
bar = !/path/to/extension/bar.py
# ditto, but no path was supplied for extension baz
baz = !
```

- En la instalación de `mercurial` vienen estas extensiones por defecto, pero también disponemos de otras hechas por terceros. Por cierto...¿a qué te suena la extensión shelve?

35. Mercurial (V)

- Para dar de alta un nuevo repositorio en un proyecto, edita su `<proyecto>/.hg/hgrc` y añade una nueva entrada en `paths` así:

```
[paths]
default = http://www.hgrepos.net/r1        ❶
default-push = ssh://dca@hgrepos.net/r1    ❷
...
alias1 = URL1
alias2 = URL2
```

❶ Es especial, se usa como valor por defecto cuando no especificamos la `URL`.

❷ Idem pero cuando hacemos un `push`.

  - Los conflictos se marcan igual que en otros scv:

```
<<<<<<< /tmp/conflict/crab.cpp
void function() {
=======
int function() {
>>>>>>> /tmp/crab.cpp
```

36. Mercurial (VI)

- Si usas *subversion*, quizás te sean útiles hgsubversion y también estos guiones.
- Estas otras aplicaciones también pueden serte útiles al emplear *mercurial*:
  1. hgview, tiene una variante en modo texto llamada `hgview-curses` (algo menos completa).
  2. tortoisehg
- Dispones también del libro de mercurial y de una guía.
- Si usas `Eclipse` te puede interesar este plugin. Otros editores/IDEs también disponen de plugins para trabajar con *mercurial*.

37. Mercurial vs. Git

De los creadores de Git is MacGyver and Mercurial is James Bond

. . . presentamos: Git is Wesley Snipes, Mercurial is Denzel Washington and Subversion is . . .

Git vs. Mercurial

- En Git los *objetos* con los que trabaja son: `blob`, `tree` y `commit`.
- Mercurial usa los mismos conceptos pero los llama: `file`, `manifest` y `changeset`.
- Ambos `scv`'s usan una clave `SHA-1` para identificar estos tres tipos de objetos. Mercurial llama a esta clave nodeid.
- Ambos *ven* la historia del proyecto como un Grafo Acíclico Dirigido.
- Mercurial distingue entre dos tipos de ramas: *ramas con nombre* y *clones de repositorios*.
- Mercurial además de la clave SHA-1 asociada a un changeset genera un número de revision local.
- Mercurial no necesita añadir los cambios de los archivos al *índice* (`stage`) como git.

38. Mercurial. Vídeos relacionados

39. Bazaar (I) . *It just works*

- web del proyecto.
- En ocasiones se le llama Bazaar-NG (New Generation) ya que es una reimplementación de GNU Arch
- El nombre le viene de un ensayo publicado por Eric S. Raimond: The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.
- En la propia web del proyecto se nos dan 10 razones para usar `bazaar`.
- Como mercurial, se puede ampliar su funcionalidad con plugin's.
- Toda la información relativa al control de versiones de un proyecto se guarda en el directorio raíz del proyecto, concretamente en el subdirectorio "`.bzr`".

40. Bazaar (II)

Funcionamiento

- Creamos e iniciamos un repositorio:

```
bzr init repo-bzr
```

- Trabajamos en él:

```
cd repo-bzr
edit file1 file2 ...
bzr add file1 file2 ...
bzr status
bzr ignore *~
bzr alias commit="commit --strict"
```

41. Bazaar (III)

- Creamos un commit y vemos el `log`:

```
bzr commit -m "Anyadido hola.txt" hola.txt
bzr log
```

```
bzr log
------------------------------------------------------------
revno: 1
committer: Fernando Perez <fperez@ua.es>
branch nick: repo-bzr
timestamp: Mon 2013-12-16 20:59:22 +0100
message:
  Anyadido hola
```

- Probamos a actualizar nuestra copia de trabajo

```
bzr update
```

42. Bazaar (IV)

- Las ramas *viven* en su propio directorio.

```
bzr branch . rama2
bzr branch rama2 display
cd rama2; edit hola.txt
bzr commit -m "decimos adios"
```

```
Committing to: /home/acorbi/Descargas/repo-bzr/rama2/
modificado hola.txt
Committed revision 2.
```

```
cd ../display
bzr merge ../rama2
```

```
M  hola.txt
Todos los cambios se aplicaron correctamente.
```

```
bzr status
modified:
   hola.txt
pending merge tips: (use -v to see all merge revisions)
   Fernando Perez 2013-12-16 decimos adios
```

43. Bazaar (V)

Algunas características especiales

- Hosting recomendado: https://launchpad.net/ o también http://savannah.gnu.org/
- Cliente con interfaz gráfica de usuario: bazaar explorer
- Unity3D es un motor de juegos en 3D que permite configurar bazaar como herramienta de control de versiones para sus proyectos.

Configuración

- En el directorio `$HOME/.bazaar`, en tres archivos:
  - bazaar.conf
  - locations.conf
  - authentication.conf
- Obtenemos información de la configuración con:

```
bzr config
```

- Modificamos la configuración con:

```
bzr config opt=value
bzr config --remove opt
```

44. Bazaar. Vídeos relacionados

45. Git vs. Mercurial vs. Bazaar

- A lo largo del tema no hemos tratado de presentar un *ganador*...
- ...Sólo ver tres *implementaciones* diferentes de una misma tecnología: *sistemas de control de versiones distribuídos*
- Cada uno de vosotros --posiblemente-- tenga un *ganador* claro... incluso es factible que penseis en uno distinto en función de las necesidades del proyecto en el que lo vayáis a emplear.
- Al terminar este tema *sí* que deberíais tener una serie de criterios para ayudaros en esa decisión.
- Si aún así no fuera suficiente, quizás esta web pueda ayudarte un poco más.

46. Prácticas individuales

46.1. Git

- Crea (si no tienes ya) una cuenta en github o bitbucket. Una vez hecho da de alta un proyecto/repositorio y configura git localmente para que puedas hacer push/pull a él.
- Créate algunos alias locales y globales en git.
- Provoca un fallo en el software en un commit de manera que después de crear varios commits más puedas encontrarlo con `git bisect`.
- Haz uso de algún `hook` de git.
- Trata de migrar un repositorio *svn* a *git*. Repasa la órden `svn` de git. Echa un vistazo a la documentación.

46.2. Mercurial

- Crea o clona un proyecto con mercurial.
- Genera cambios y crea los commits correspondientes.
- Crea ramas y comparte commits entre ellas (merge).
- Mercurial también dispone de la orden `bisect`. Úsala en algún ejemplo.

```
hg bisect --help
```

46.3. Bazaar

- Bazaar no dispone directamente de la orden `bisect`, pero se puede instalar mediante un plugin. Úsala en algún ejemplo.
- Crea un repositorio, dótalo de contenido, haz modificaciones, genera commits, crea ramas, etc...
- Clona algún repositorio de launchpad. Modifica archivos y/o añade nuevos. Genera los commits locales correspondientes.
- Crea y prueba algunos *alias*, echa un vistazo a http://doc.bazaar.canonical.com/bzr-0.11/using_aliases.htm. También puedes obtener ayuda con:

```
bzr alias --help
```

46.4. Darcs

- Trata de crear un repositorio con *darcs*. Una vez hecho, úsalo para guardar cambios en tu proyecto.

ENTREGA:

- La práctica se entregará en pracdlsi en las fechas allí indicadas.

47. Prácticas en grupo

- Explicadnos qué es SVK.
- Explicadnos qué es Tailor.
- Explicadnos qué son los submódulos en git. Presentadnos algún ejemplo de uso.
- Explicadnos qué es git-annex. Poned algún ejemplo de uso.
- Bazaar dispone de *bound branches*... tratad de explicarnos qué son.

48. Aclaraciones

EN NINGÚN CASO ESTAS TRANSPARENCIAS SON LA BIBLIOGRAFÍA DE LA ASIGNATURA.

- Debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.