Taller de introducción a GIT

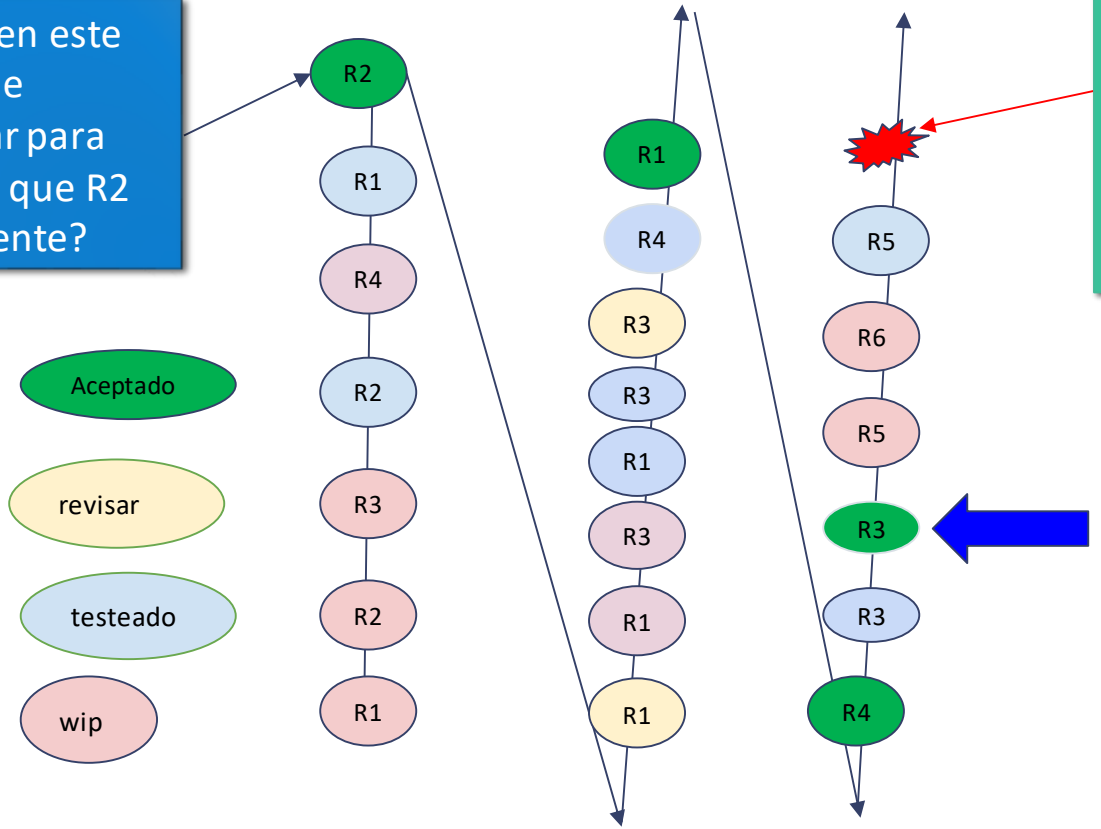
SCM

Control de versiones

- Veamos las siguientes situaciones
  - Se detectó un bug, de altísima prioridad en la versión que actualmente está en producción, y debe ser corregido en menos de 1 hora.
  - Para la siguiente versión del software, en un sprint de 10 días, estaba previsto entregar 5 funcionalidades, las primeras 2 se terminaron en 2 días, las siguientes 2 funcionalidades en los siguientes 3 días y la última funcionalidad se dejó por la mitad dado que se tuvo que corregir problemas de altísima prioridad en la versión del sistema que estaba en producción.
- ¿Cómo organizamos el código fuente de nuestro sistema para tener perfectamente identificado que parte de cada archivo corresponde con que versión de software?

Una situación … “poco comun”…

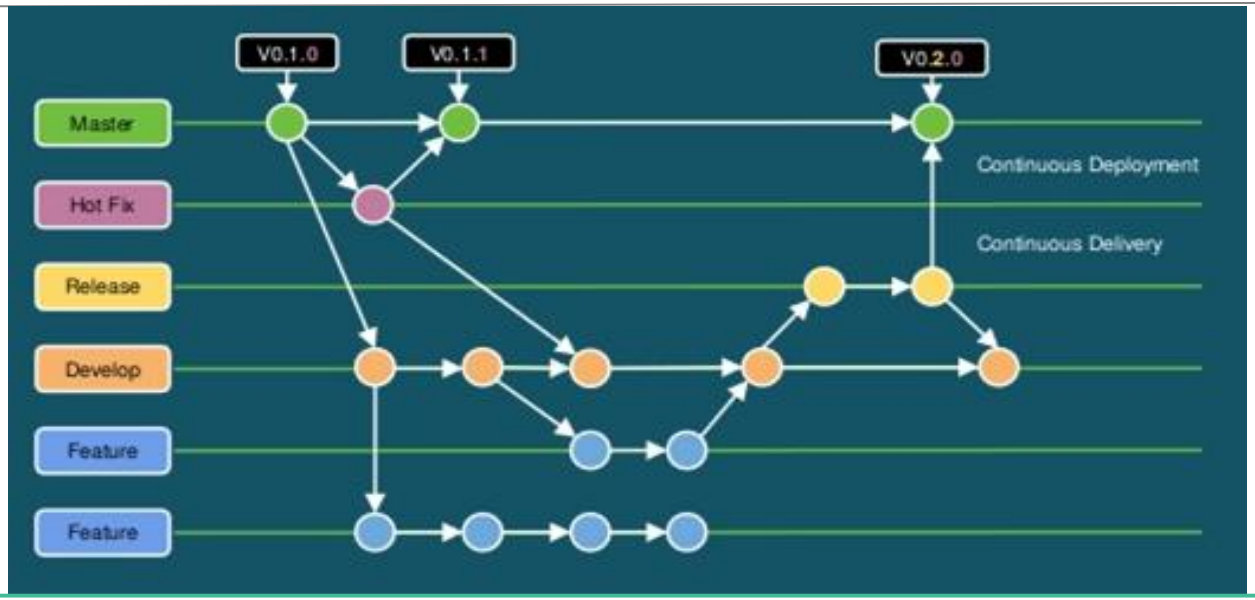
Si “R2”está listo en este momento porque debemos esperar para entregar el valor que R2 representa al cliente?



Aparece un bug en producción, que hacemos? Dejamos R5 y R6 por la mitad?

Versión 1.0.0

… a una situación *(por suerte)* cada vez más común



Con Git se facilita el manejo simultáneo de múltiples versiones del software lo que permite:

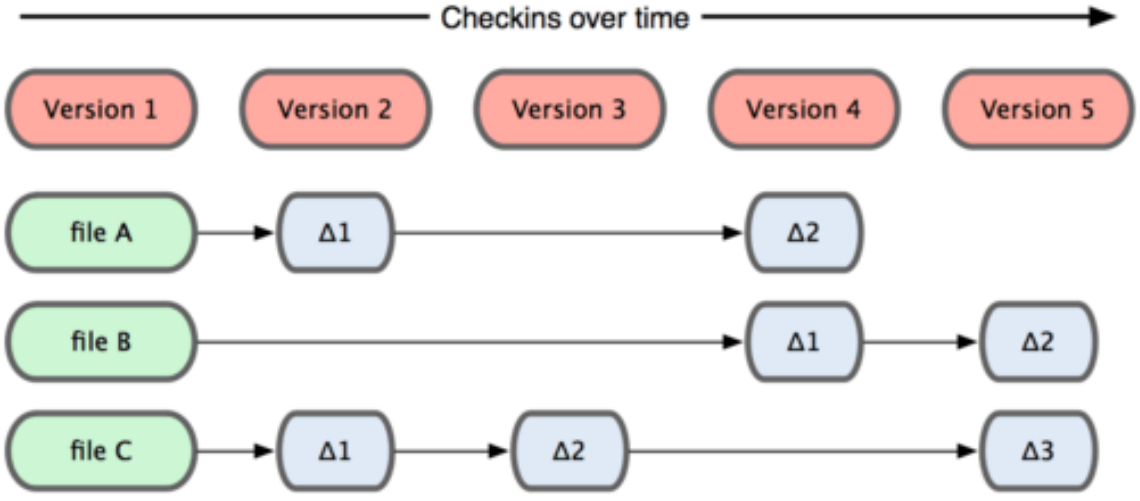
- Trazar requerimientos
- Dividir las piezas de trabajo en niveles de granularidad adecuados.
- Tener distintas configuraciones para cada stage.

Control de versiones

- El control de versiones es un sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que se pueda recuperar versiones específicas más adelante.
- Los primeros sistemas de versiones más populares, se basan en un directorio central (servidor de VCS) donde se guardan las versiones de un proyecto y los clientes sincronizan su copia contra ese servidor central.
- Existe un único servidor que contiene todos los archivos versionados, y varios clientes que descargan los archivos desde ese lugar central.

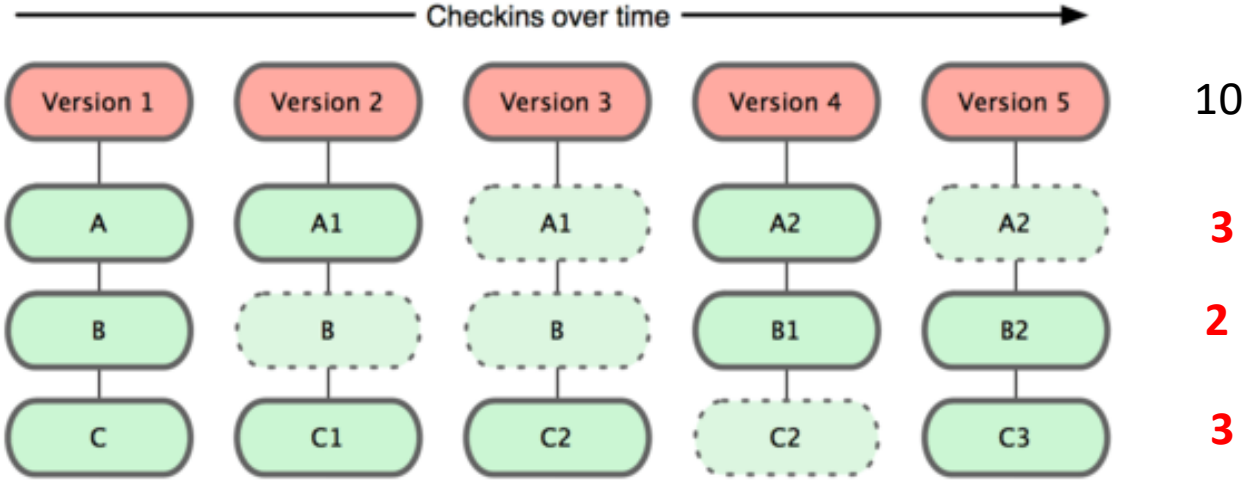
Funcionamiento de SCM tradicionales.

- Los sistemas de gestión de versiones generalmente guardan para cada versión el diferencial de cambios que hay.
- La versión final es la suma del archivo original y las modificaciones que se le realizaron



Funcionamiento de Git

- Git no guarda diferencias sino que ante cada versión guarda nuevamente todos los archivos.
- Pero si un archivo no se ha modificado, Git almacena un enlace al archivo anterior idéntico para no duplicar espacio.



10

3

2

3

Características de Git

- El repositorio completo está en una copia local.
  - Dentro del directorio .git
- Permite desarrollar gestionando versiones, offline.
- Cuando alcanzamos una versión estable, podemos solicitar sincronizar nuestra copia con la del servidor remoto.
- Las operaciones en GIT al ser locales, son muy rápidas.
  - Todo lo que realiza es operaciones de filesystem (no realiza comparaciones).
- Cada commit tiene una verificación de integridad por lo que es casi imposible modificar el contenido sin que git se de cuenta.

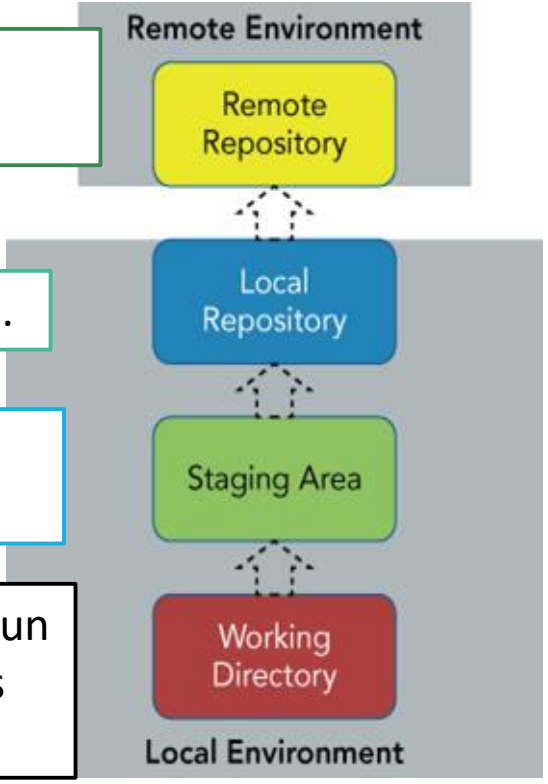
Áreas de trabajo

Repositorio remoto recibe copias del repositorio local para mantenerse sincronizado.

Repositorio local contiene versiones confirmadas de un archivo.

Área de staging: posee archivos marcados para ser gestionados por git, pero todavía no son versionados.

Directorio de trabajo: es la raíz de un repositorio git y es donde un archivo es creado, editado, borrado. Archivos se crean aquí antes de ser gestionados por Git.



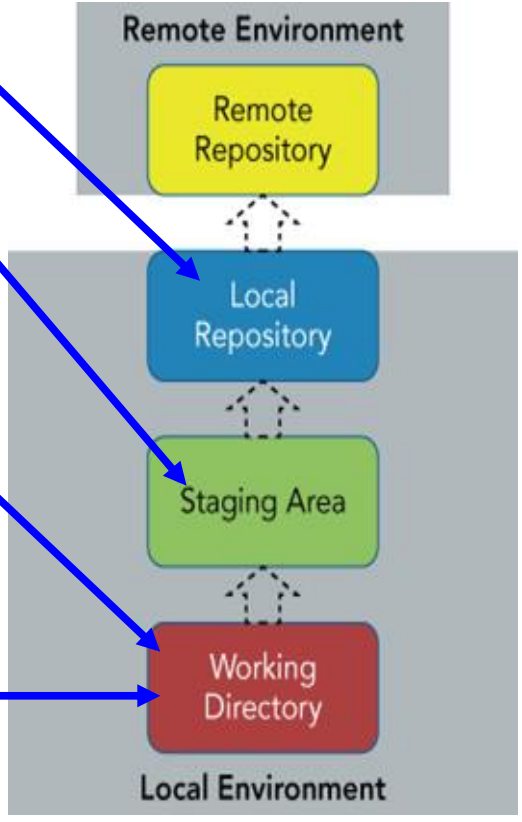
Estados de un archivo

Confirmado (committed):los datos están guardados de manera segura en la base de datos del repositorio local.

Preparado (staged): marcamos un archivo local para que esté en el área de staging y se versiona en la próxima confirmación

Modificado (modified): el archivo local está bajo seguimiento de git (existe una versión en el repositorio) pero está siendo modificado, generando una nueva versión que no ha sido agregada al staging

Sin seguimiento: el archivo ha sido creado pero todavía no se ha agregado al staging ni al repositorio para que git tenga seguimiento del mismo



Tarea 1 - Iniciar repositorio local

- En una Ventana de commando crear un directorio de trabajo
  - `$ mkdir taller-git01`
- Ingresar a dicho directorio
  - `$ cd taller-git01`
- Ejecutar el comando
  - `$ git init`
- Verificar el estado con el comando **$ git status**
  - *Nos indica en que rama estamos y que todavía no hemos "comiteado" nada*
- Listar los archivos y directorios (dir /a)
  - *Notar que existe un directorio oculto, ".git" el cual contiene toda la configuración y base de datos interna de git.*
  - *Borrar dicho directorio implica dejar de gestionar los archivos actuales con git*

El área de staging Staging

- Es donde se preparan los archivos a confirmar para conformar una versión
  - Se la denomina en versiones anteriores de git como “index” o “cache”
- Es donde se agregamos temporalmente de manera individual nuevos archivos o las nuevas versiones de los archivos que todavía no están listas para formar parte de una nueva versión.
- El flujo normal es pasar archivos del directorio de trabajo a al área de staging con el comando “add”.
  - $ git add <nombre_de_archivo>
  - $ git add . *//→agrega todos los archivos del directorio*

Tarea 2 - Agregar dos archivos

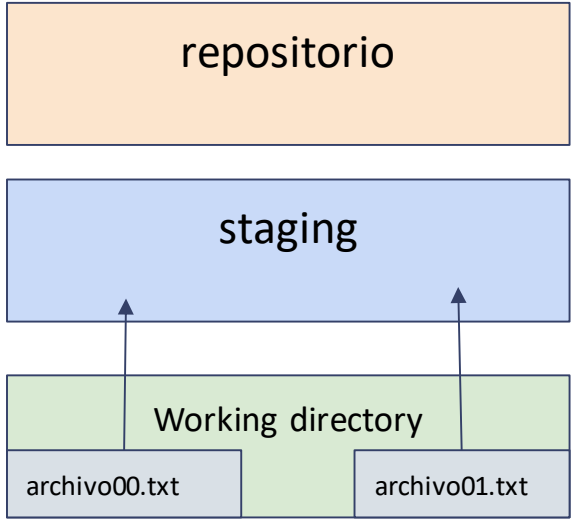
- Crear los archivos “archivo00.txt” y “archivo01.txt”

Sin seguimiento

linea 01
linea 02
linea 03

linea 11
linea 12
linea 13
linea 14

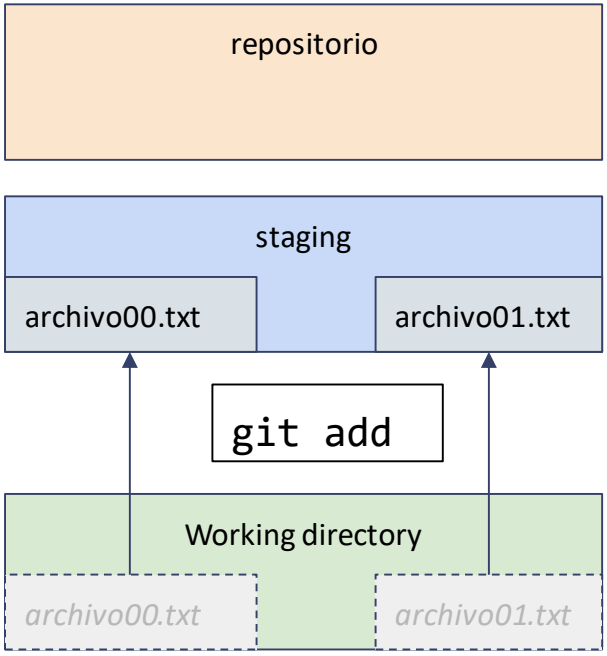
- Luego ejecutar el comando “`git status`”
  - Indica que todavía no hemos realizado un commit
  - Git ya reconoce 2 archivos, como “untracked”



Tarea 3: Agregar los archivos a staging

- Ejecutar `git add`
  - “add” envia cualquier todos los archivos modificados en la copia local, al staging, como paso previo a formar parte de una nueva versión en el repositorio.
- *Envía tanto, **archivos nuevos** (que no existían previamente en el repositorio), como versiones nuevas de archivos versionados.*
- *Luego ejecutar* **`git status`**

```
On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   archivo00.txt
        new file:   archivo01.txt
```



Repositorio local

- El repositorio local guarda las versiones de los archivos que son confirmados.
  - Se almacena en una carpeta separada (.git).
- Es de uso exclusivo del usuario local.
  - **NO hay necesidad de conectividad ni uso de red**.
- Luego podemos sincronizar, sujeto a determinadas reglas, todo el contenido a un repositorio remoto que es accedido por otros usuarios y que es donde otros usuarios también envían sus cambios locales.
- Pasar de Staging a Repositorio local:
  - **$git commit** → confirmamos que enviamos un conjunto de nuevas versiones de archivos al repositorio local para dejar registro de la versión.
  - Solo envía **el contenido del área de staging**, no la copia local.

Si intentamos realizar un primer commit …

…tendremos un error, porque entre otras cosas, git necesita determiner el nombre de usuario y correo electrónico de quien realiza el commit. Debemos configurarlo en una variable de configuración.

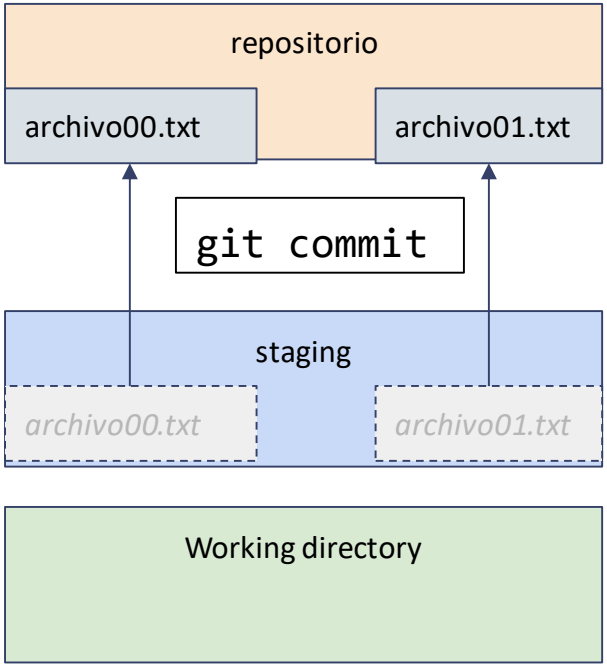
```
>git commit -m "primer commit"

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got 'marti@DESKTOP-8EOROBA.(none)')
```

Configuración

- Cada commit tiene asociado un usuario y un email, por lo tanto tenemos que configurarlo.
- Lo podemos configurar en 3 niveles
  - Por proyecto (--local)
  - Por usuario de SO (--global)
  - Para todos los usuarios en el sistema (--system)
- Ejecutar
  - `git config --local user.name martin`
  - `git config --local user.email mdomingu@gmail.com`
  - `git config -l // listar todas las propiedades.`

Tarea 3: crear el primer commit

- Ejecutar “`git commit -m "<commit message>`“
  - *Confirma que el conjunto de archivos que está en staging debe registrarse en el repositorio local como una nueva versión y los hace permanentes.*
  - *Permite diversas variantes, la más común es controlar cuales archivos del staging pasar al repositorio, pasando como argumento dicha lista de archivos.*



Ejemplo

```
$git  commit -m "<commit message>" clase1.java readme.md
```

Resultado de un commit

Luego del commit Git nos da información de dicha acción

```
>git commit -m "primer commit"
[master (root-commit) dae698f] primer commit
 2 files changed, 7 insertions(+)
 create mode 100644 archivo00.txt
 create mode 100644 archivo01.txt
```

`master` es el branch donde se agregaron los nuevos archivos. Es el branch por defecto de todo proyecto git

"**dae698f**" son los primeros 7 caracteres del SHA1 que representa el presente commit. A continuación aparece el mensaje con que se configuró el commit

Cantidad de archivos afectados por el commit y cantidad de cambios dentro de cada uno de los archivos. La cantidad de cambios dentro de los archivos se categoriza en líneas insertadas y borradas antes y después del commit

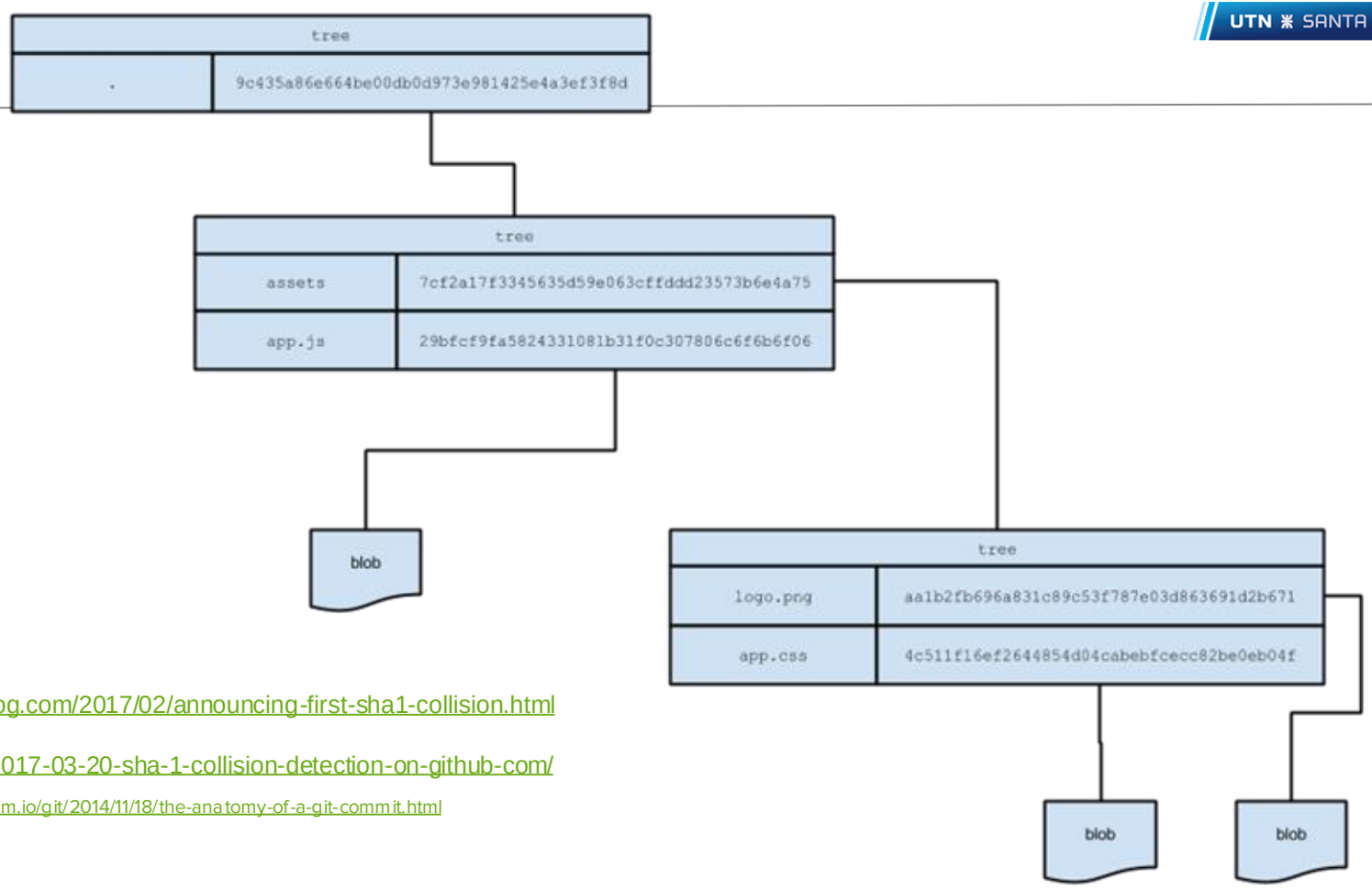
FInalmente tenemos el listado de archivos afectados por el commit con información del archivo (mode)

El valor SHA-1

- Cada objeto y cada “snapshot” resultante de cada cambio confirmado con “commit” se identifica univocamente mediante **un hash valor SHA-1**.
- Ese SHA-1 se forma como
  - sha1(

```
    commit message  => "...."
    commiter        => <user><mail@gmail.com>
    commit date     => Sat Nov 8 11:13:49 2014 +0100
    author          => <nombre><mail@gmail.com>
    author date     => Sat Nov 8 11:13:49 2014 +0100
    tree            => 9c435a86e664be00db0d973e981425e4a3ef3f8d
    parents         => [0d973e9c4353ef3f8ddb98a86e664be001425e4a]
)
```

Ejemplo



https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html

https://blog.github.com/2017-03-20-sha-1-collision-detection-on-github-com/

https://blog.thoughtram.io/git/2014/11/18/the-anatomy-of-a-git-commit.html

Mode de un archivo

- Los códigos de estado de un archivo se interpretan de l siguiente manera
  - 4-bit para el tipo de objeto (1000 [regular file], 1010 [symbolic link], and 1110 [gitlink])
  - 3-bit no usado
  - 9-bit permiso de unix (0755 y 0644 son para archivos comunes. Symbolic links y gitlinks toman valor 0)
- De está combinacion surge:
  - 040000: Directory
  - 100644: Regular non-executable file
  - 100755: Regular executable file
  - 120000: Symbolic link
  - 160000: Gitlink. (avanzado → referencia un commit en otro repositorio)

Tarea 4: Arreglar un commit

- Supongamos que equivocamos el mensaje del commit y queremos corregirlo.
  - Si no tenemos cambios no podemos agregar un nuevo commit.
  - Y si tenemos cambios el mensaje no puede ser la corrección del anterior
- El comando commit tiene el modificador --amend que permite corregir el commit sin crear un nuevo set de cambios.

```
>git commit --amend -m "MI primer commit"
    [master 596aa53] MI primer commit
    Date: Sun May 27 22:19:01 2018 -0300
    2 files changed, 7 insertions(+)
    create mode 100644 archivo00.txt
    create mode 100644 archivo01.txt
```

Tarea 5: modificar archivos

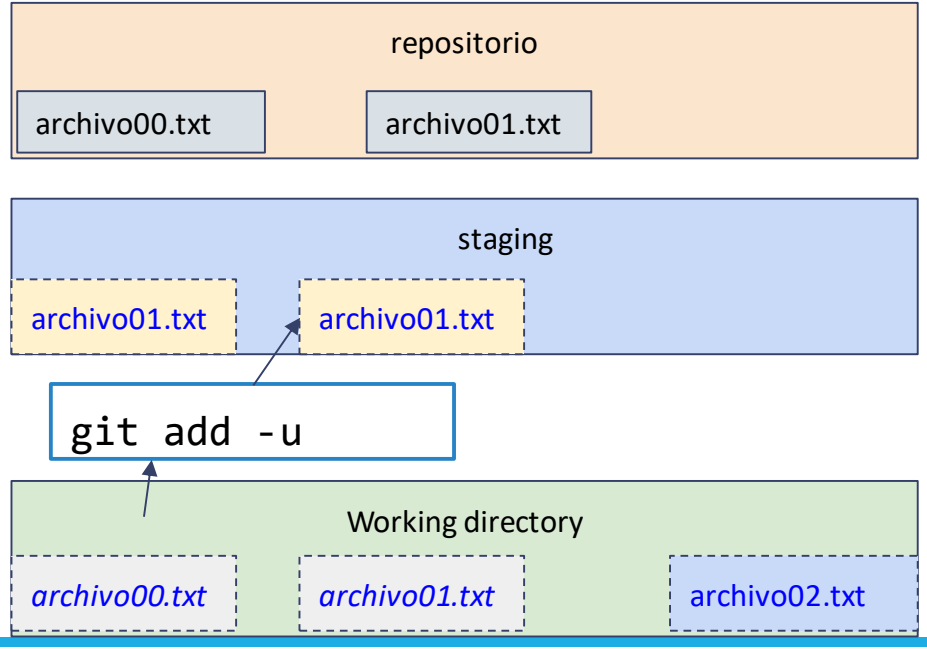
- Modificar en el directorio de trabajos archivos que ya versionados
  - En archivo00.txt
    - agregar al final el texto “linea06”
  - En archivo01.txt
    - borrar “linea 14”
    - modificar “linea 13” por “linea **”
  - Crear el archivo02.txt con el contenido:
    - linea20
    - linea 22
    - linea 24
- ¿Cual es el estado de cada uno de los archivos en este momento?

Estado de cada archivo

- archivo00.txt:
  - hay una versión en el repositorio local.
  - Una nueva versión en el directorio de trabajo. **Estado → modificado.**
- archivo01.txt:
  - hay una versión en el repositorio local.
  - Una nueva versión en el directorio de trabajo. **Estado → modificado**.
- archivo02.txt
  - NO hay una versión en el repositorio local.
  - Una nueva versión en el directorio de trabajo. Estado → **sin seguimiento**.

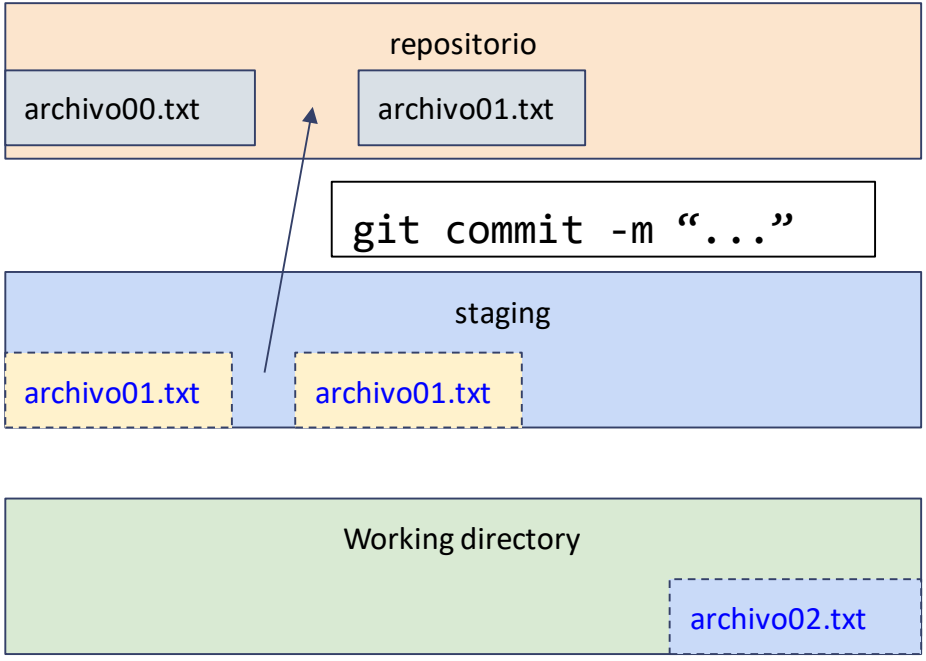
Tarea 6: Agregar a staging solo archivos “modificados”

- El comando de git “add” posee la opción “-u” que sólo añade al staging aquellos archivos que ya están siendo controlados por git.
- Ejecutar : **`git add -u`**
  - agrega solo archivo00.txt
  - archivo01.txt al staging)



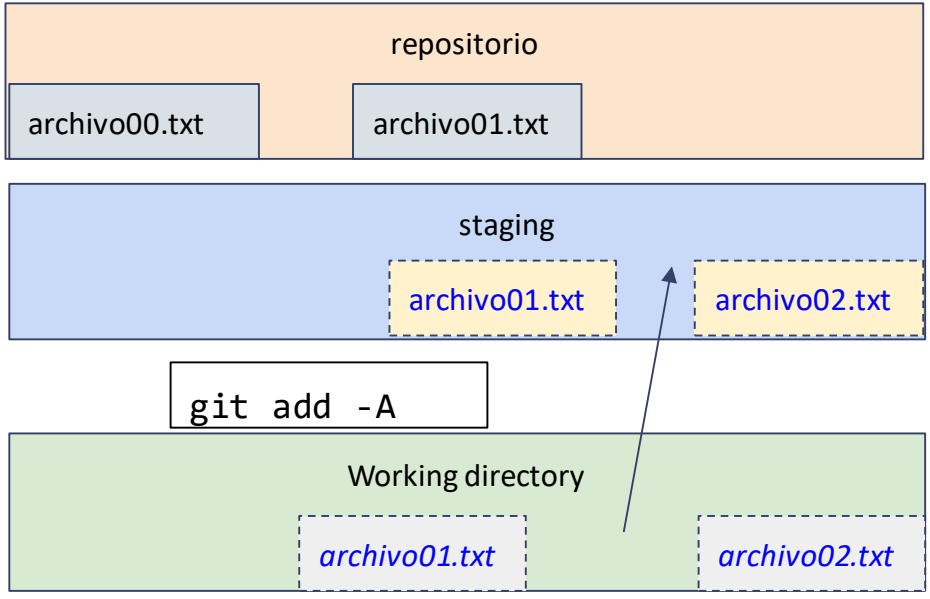
Tarea 7: Crear una nueva versión

- Realizar un commit, con mensaje “Mi segundo commit”.
  - **`git commit -m "Mi segundo commit“`**
- Analizar el estado con “git status”



Tarea 8: modificar “archivo01.txt”

- Abrir “archivo01.txt” y cambiar “linea **” por “linea 14”
- Agregar todas las modificaciones del directorio de trabajo al staging.
  - *Para esto usar el parámetro de git add –A que agrega todos los archivos, nuevos, modificados o sin seguimiento*
- Ejecutar **`git add -A`**



Tarea 9: modificar archivo01 y ver estado.

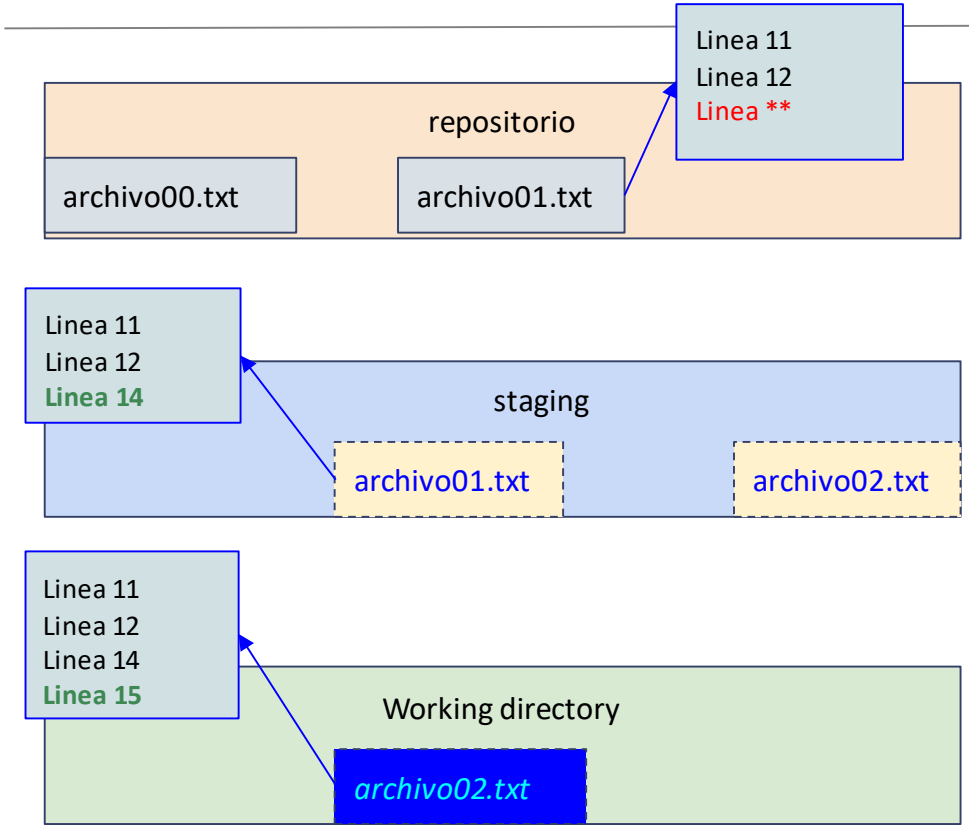
- Ahora modificar nuevamente “archivo01.txt” y agregar al final “linea 15” y analizar el status del proyecto
- `git status`

```
En la rama master
Cambios a ser confirmados:
  (usa "git reset HEAD <archivo>..." para sacar del área de stage)
   modificado:     archivo01.txt
   nuevo archivo:  archivo02.txt
Cambios no rastreados para el commit:
  (usa "git add <archivo>..." para actualizar lo que será confirmado)
  (usa "git checkout -- <archivo>..." para descartar los cambios en el directorio de
trabajo)
   modificado:     archivo01.txt
```

Notar que **archivo1** está listado en los 2 grupos!!

- Tenemos una versión en el repositorio
- Una que agregamos a staging
- Una que estamos modificacando en el directorio local

Analizar estado de archivos



“archivo01.txt” tiene 3 versiones distintas según el lugar en que se encuentra.

Es necesario muchas veces conocer en que estado está cada archivo y las diferencias entre estas versiones de manera sencilla.

- Para ello git nos brinda dos comando

```
$git status
$git diff
```

Tarea 10: Ejecutar “git status -s”

- Con este comando se puede visualizar el estado de los archivos de manera resumida.
  - `$ git status –s`
  - `MM archivo01.txt`
  - `A  archivo02.txt`

Status	Column 1 Code	Column 2 Code
Empty working directory	blank	blank
File staged and unmodified	A	blank
File staged and modified	M	M
Untracked file	?	?

- *La primer columna indica el estado en staging*
- *La segunda columna el estado en el directorio local*

Tarea 10: visualizar diferencias entre “working directory” vs “staging

- Ejecutar el comando “git diff”. Por defecto muestra las diferencias entre los archivos que están en el directorio de trabajo y staging

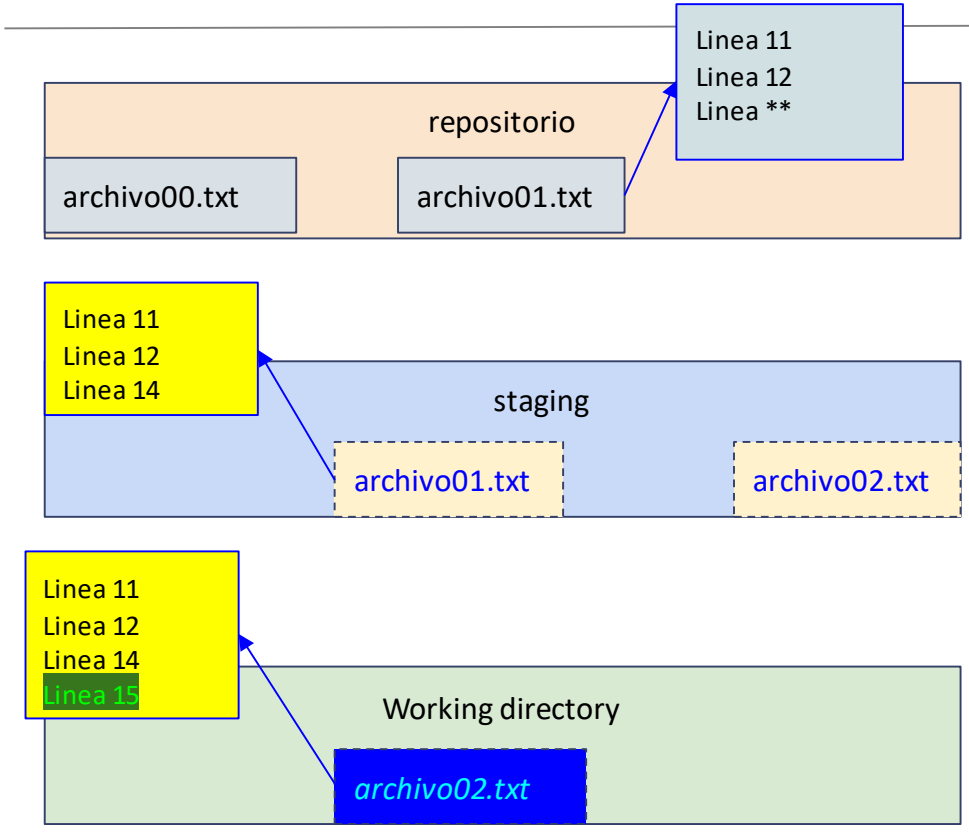
```
git diff
  diff --git a/archivo01.txt b/archivo01.txt
  index 8840772..742bfb9 100644
  --- a/archivo01.txt
  +++ b/archivo01.txt
  @@ -1,3 +1,4 @@
   linea 11
   linea 12
   linea 14
  +linea 15
```

a/ y b/ indican fuente (directorio de trabajo) y destino (staging) de la comparación

Sha1 antes y después del cambio

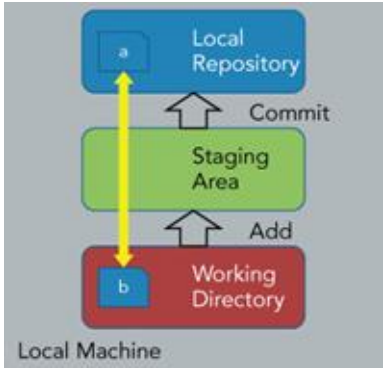
El - corresponde a una linea que está en “a” y no en “b”
El - corresponde a una linea que está en “b” y no en “a”

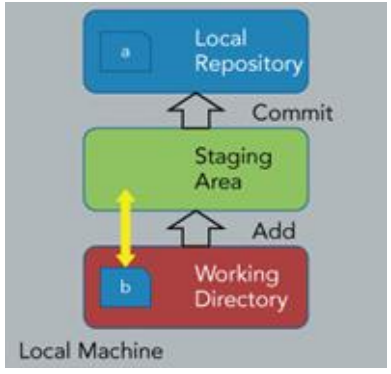
Resultado visual de la comparación



Comparando diferencias.

- El comando diff busca las diferencias entre archivos partiendo del directorio de trabajo hacía arriba.
  - Si existe una versión en staging siempre encontrará dicha versión.
  - Si no existe en staging pero si en el repositorio comparará con esta



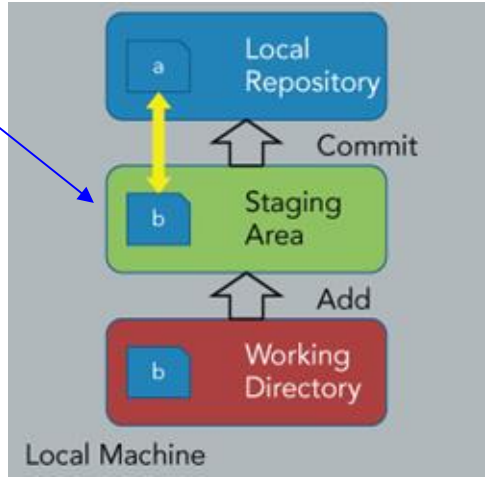


- Posee diferentes formas de uso
- Comparar WD con un SHA1
  - `$git diff [options] [<commit>]`
- Comparar dos SHA1
  - `$git diff [options] <commit> <commit>`

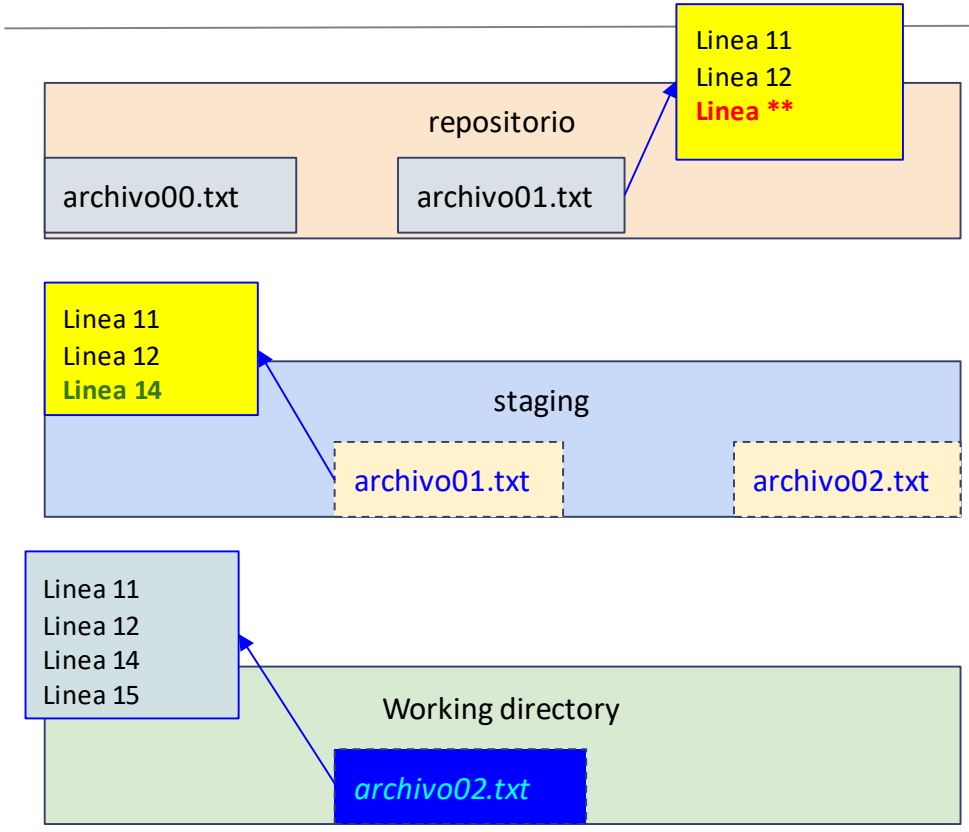
Tarea 11: Comparar “staging” con “repositorio”

Ejecutar git difff con el parámetro “--staged” o “--cached” → Esto hace que la comparación arranque desde staging

```
>git diff --staged
diff --git a/archivo01.txt b/archivo01.txt
index e264843..8840772 100644
--- a/archivo01.txt
+++ b/archivo01.txt
@@ -1,3 +1,3 @@
 linea 11
 linea 12
-linea **
+linea 14
diff --git a/archivo02.txt b/archivo02.txt
new file mode 100644
index 0000000..d0ce14d
--- /dev/null
+++ b/archivo02.txt
@@ -0,0 +1,3 @@
+linea 20
+linea 22
+linea 24
```



Diferencias entre repositorio y staging



¿Como comparamos el “working directory” con el repositorio?

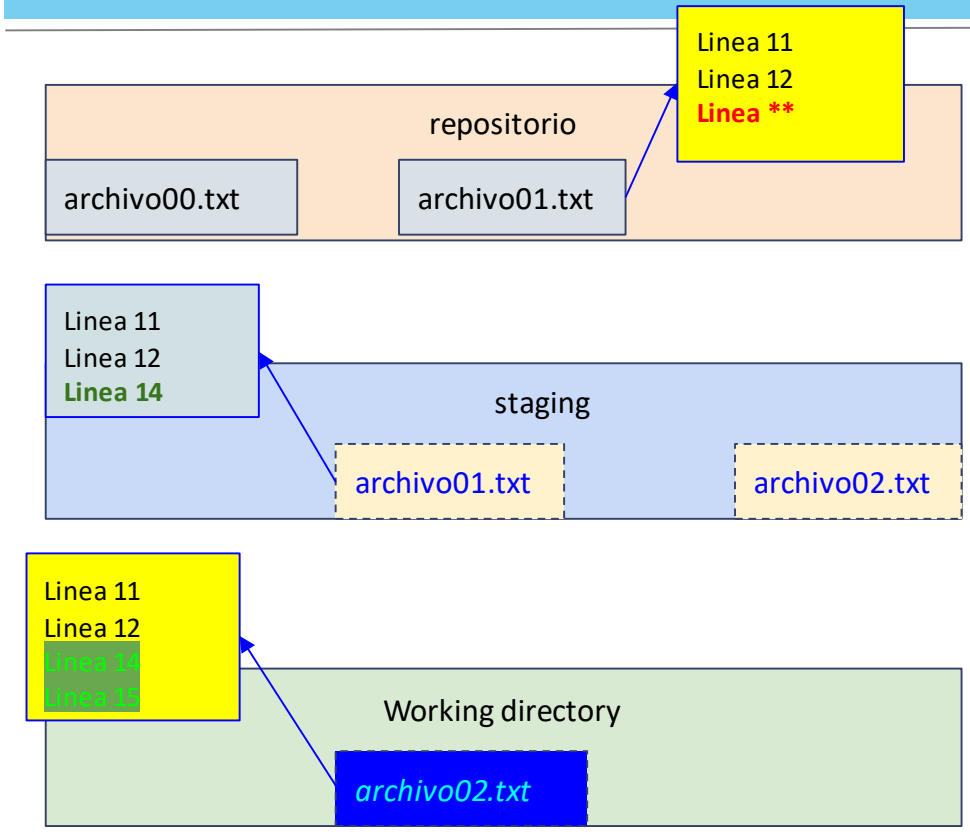
Usando la forma

```
$git diff [options] [<commit>]
```

Y reemplazando **[<commit>]** por el SHA1 del último “commit”.

Afortunadamente no necesitamos anotarlo, git siempre guarda un puntero al SHA1 del último commit (**de la rama actual)** en una variable local denominada “HEAD”

Tarea 12: Comparar “working directory” con “repositorio”



Con “$git diff HEAD” nos mostrará que para ir de la versión del repositorio a la versión del directorio de trabajo, debemos borrar una linea (Linea**) y agregar 2 lineas (Linea14 y Linea15)

Otras variantes de diff

- Listar solo los archivos con diferencias sin detalle
  - `git diff --name-only`
- Listar solo archivos y estado
  - `git diff --name-status`
- Listar archivos y palabra diferente
  - `git diff HEAD --word-diff`

Tarea 13: crear 2 versiones nuevas

- Realizar un commit de los archivos en staging
- Luego agregar los archivos del directorio de trabajo al staging.
- Luego realizar un nuevo commit.
- Revisar el status y verificar que el mensaje es
  - *En la rama master*
  - *nada para hacer commit, el árbol de trabajo está limpio*

Git log

- Las diferencias nos muestran el estado entre los archivos que están en las 3 áreas de trabajo en que se divide “git”.
- Pero un archivo además de estar en estas 3 áreas de trabajo, puede estar guardado en un repositorio “git” en distintas versiones.
- ¿Como podemos ver las diferencias entre un archivo del directorio de trabajo, del stage, o incluso del repositorio, con otro archivo **de otra versión** del repositorio?
- Por el identificador “sha1” asociado a cada **versión**.
- **¿Como lo podemos obtener? A través del comando “git log”**

Git log

- Además de ver las diferencias en ocasiones necesitamos revisar cómo evolucionó una versión a lo largo del tiempo y decidir si volver a una versión anterior o cancelar cambios.
- En git el comando “log” nos permite tener un registro de los cambios de los archivos a lo largo del tiempo.
  - La sintáxis es: git log [<options>] [<revision range>] [[--] <path>…]
- Sin opciones por defecto muestra: SHA1, mail de quien hizo el commit, el mensaje y los archivos involucrados, en orden cronológico.
- Con la opción “-p” (de patch) muestra además todos los cambios realizados a un archivo para llegar de la primer version a la actual

Tarea 14: visualizar historial del repositorio

- Ejecutar los siguientes comandos
  - Git log
  - Git log -2 → muestra los últimos 2 commits
  - Git log -2 --stat → muestra los últimos 2 commits y los cambios asociados.
  - Git log --oneline → muestra todo el historial pero en una linea por commit.
- Luego de ejecutar el comando “git log --oneline” tome el “sha1” del primer commit (el último elemento de la pila) y del penultimo commit y compararlos con git diff
- Modificar archivo00.txt agregando “***” al final de “linea 02”
- Comparar ahora el penúltimo commit con la versión del directorio de trabajo y con staged.
- Realizar un commit

Git “blame”

- La versión actual de un archivo, puede deberse a que luego de numerosos commits, hemos llegado al código que actualmente está ejecutando.
- En ocasiones, cuando sucede un problema, y debemos corregir el código, muchas veces necesitamos saber quién es el culpable del código (en realidad conocer quien puso esa línea y por que).
- El comando log, nos dice los cambios que tuvo un commit, pero no nos indica nada acerca de una archivo .
- Con el comando “blame” podemos ver como fue la historia de construcción de un archivo en particular.

Git Blame

- Se ejecuta pasando como argumento un nombre de archivo y luego diversos parámetros
  - Limitar la cantidad de líneas en el análisis con el parametro -L, por ejemplo limitar de las lineas 3 a la 6
  - git blame archivo2.txt -L3,5
    7ca71ebd (mart-dominguez 2018-01-25 16:19:06 -0300 3) linea99
    7ca71ebd (mart-dominguez 2018-01-25 16:19:06 -0300 4) linea100
    8f9646e7 (mart-dominguez 2018-01-25 16:17:59 -0300 5) linea5
  - Mostrar los cambios realizados en un archivo en un rango de fechas
  - $ git blame --after=2016-03-28 build.gradle
  - $ git blame --since=2.weeks build.gradle
  - $ git blame --before="8 weeks ago" build.gradle

Tarea 15 : ejecutar git blame sobre cada archivo

- `git blame archivo00.txt`
- `^ec6cc88 (mart-dominguez 2018-05-28 08:56:04 -0300 1) linea 01`
- `5e4f884c (mart-dominguez 2018-05-28 12:38:46 -0300 2) linea 02***`
- `^ec6cc88 (mart-dominguez 2018-05-28 08:56:04 -0300 3) linea 03`
- `7cf2ab31 (mart-dominguez 2018-05-28 09:41:50 -0300 4) linea 06`
- *Que significa ^ ? Indica el SHA1 del primer commit de creación de este archivo en el repositorio.*

Tarea 16: agregar a staging cambios no deseados

- Modificar “archivo00.txt” agregando al final “linea 99”.
- Agregar dicho archivo al staging.
- Volver a modificar el archivo y agregar “linea 101” al final
  - Problema 1, hemos hecho cambios en el directorio de trabajo, pero los queremos deshacer y queremos volver a tomar como base lo que tenemos en el staging.
  - Problema 2: queremos deshacer los cambios locales y del staging, y reemplazarlos con la última versión del repositorio.
- Verificar las diferencias entre las 3 áreas
  - `git diff`
  - `git diff --staging`
  - `git diff HEAD`

Tarea 19: Resolver problema 1

repositorio

Linea 01
Linea 02***
Linea 03
Linea 06

staging

Linea 01
Linea 02***
Linea 03
Linea 06
Linea 99

Working directory

Linea 01
Linea 02***
Linea 03
Linea 06
Linea 99
Linea 101

Checkout: copia los archivos del "staging" al directorio de trabajo

```
git checkout -- archivo01
```

Verificar las diferencias
Git diff → no hay
Git diff HEAD y git diff --staged dan la misma diferencia.

repositorio

Linea 01
Linea 02***
Linea 03
Linea 06

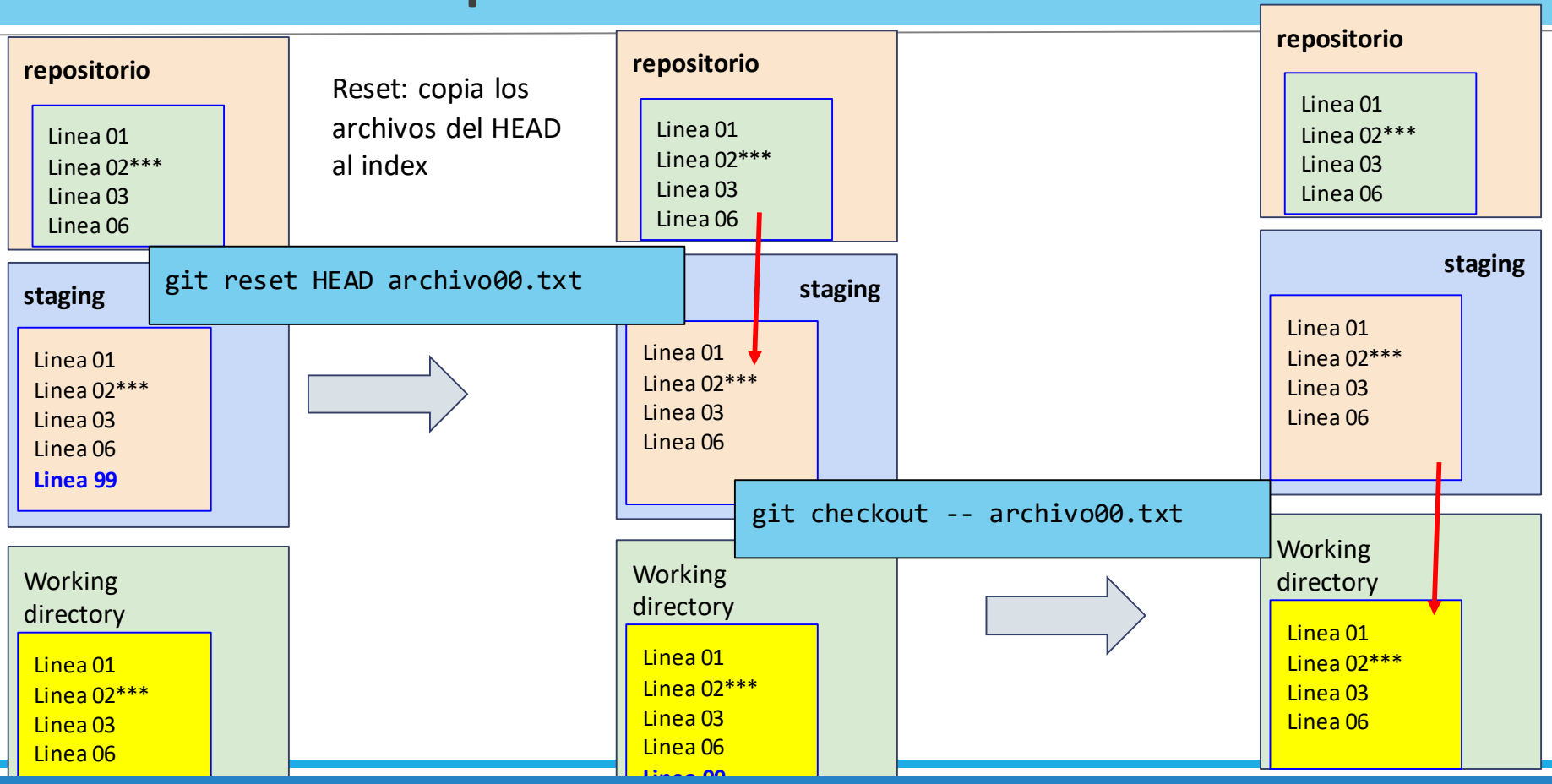
staging

Linea 01
Linea 02***
Linea 03
Linea 06
Linea 99

Working directory

Linea 01
Linea 02***
Linea 03
Linea 06
Linea 99

Tarea 20: Resolver problema 2



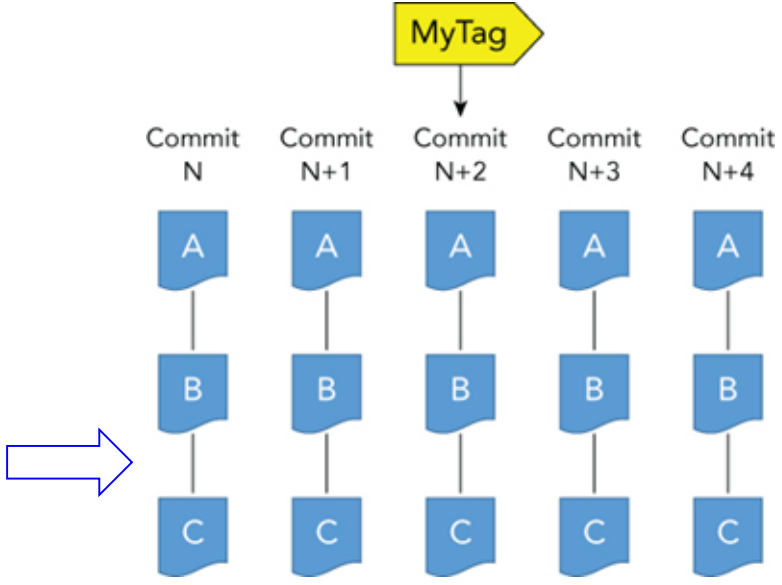
Tags

- Cada commit en “git” referencia un estado del conjunto de archivos de un proyecto.
- Cada “commit” representa un incremento en las funcionalidades del sistema.
- Si es posible desplegarlo en producción, dado que cumple con todas las condiciones requeridas, deberíamos poder identificarlo fácilmente.
- En git esto se hace con un **“SHA1” pero no es muy práctico** para el humano.
- Para simplificar la asociación entre “snapshots” y “versiones” podemos crear tags, que permiten **asociar nombres simbólicos a un commit**.
- Para crear un tag simplemente se usa
  - `$ git tag NOMBRE_TAG SHA1_A_ASOCIAR`
    - Si no pasamos nombre del SHA1 por defecto tomará el HEAD
      - `$ git tag NOMBRE_TAG`

Tags

- Podemos realizar un tag de cualquier versión en la historia.
- No tiene costo solo es simplemente asociar un nombre a un SHA1
- Podemos listar los tags con
- Git tag -l
- Ver las referencias entre SHA y TAG
- Git show-ref --tag

Un punto importante es que el tag corresponde a un snapshot de un repositorio **no a una versión en particular de un único archivo en particular.**



Tarea 21: trabajar con tags

- Listar los commits en una linea (git log --oneline)
- Crear un tag para los últimos 3 commits con nombre
  - Version_A - Version_B - Version_C respectivamente
- Analizar las diferencias entre Version_A y Version_B
- Analizar las diferencias entre Version_A y Version_C
- Analizar las diferencias entre Version_B y Version_C

Sincronizar con un repositorio remoto

- El repositorio local guarda las versiones de los archivos sobre los que realizamos commit.
- Es de uso exclusivo del usuario local. Puede realizar todas las modificaciones que desee hasta que se alcance el resultado buscado.
- Pero la esencia de “git” es que podamos sincronizar nuestros repositorios locales con un repositorio remoto.
- Hasta ahora todo lo que hemos visto de git funciona en el repositorio local, pero esto no es de utilidad. Necesitamos git para poder compartir código en ubicaciones remotas.
- En git un repositorio remoto es solo un repositorio Git con un protocolo de acceso.

Sincronizar repositorios remotos

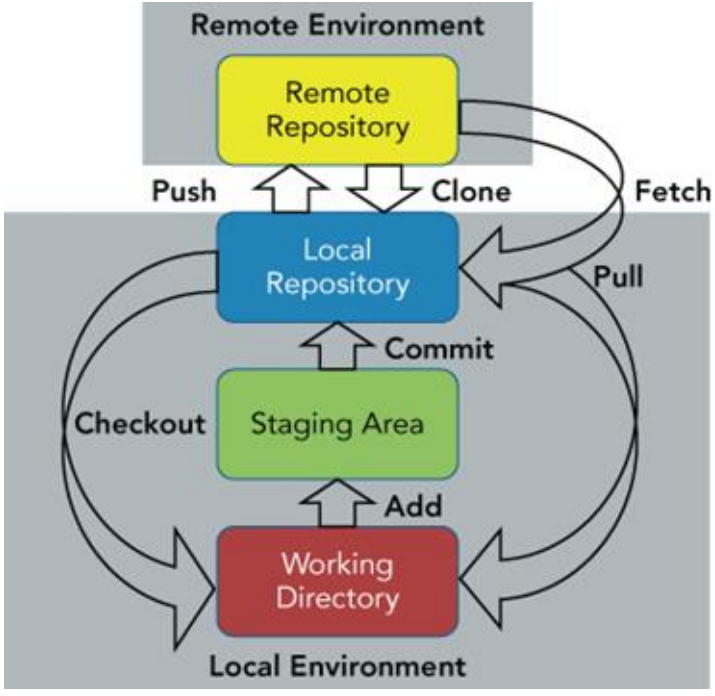
- Un repositorio remoto no realiza modificaciones al contenido como la resolución de conflictos (como si se hace automáticamente cuando es posible mediante el merge en el repositorio local)
- Se usa para sincronizar los cambios hacia y desde los repositorios locales de usuarios individuales.
- Si hay conflictos que necesitan solución en el momento en que el contenido se transfiere al control remoto, se debe descargar el contenido de la última versión del repositorio remoto al staging local, resolverse allí y luego sincronizarse con el control remoto.

Sincronizar

- Git permite referenciar los repositorios remotos con un nombre, que por defecto se denomina “origin”
- EL comando “remote” es el que permite gestionar los repositorios remotos
- Los para agregar, o quitar una URL con un nombre ...
  - **`git remote add`** `[-t <branch>] [-m <master>] [-f] [--[no-]tags]` **`<name> <url>`**
  - `git remote rename <old> <new>`
  - `git remote remove <name>`
- Para listar los repositorios
  - **`$ git remote -v`**

Sincronizar

- Para enviar cambios a un repositorio remoto, simplemente usamos el comando “push”.
- Recibe 2 argumentos: una referencia a un **repositorio remoto** y una referencia a la **rama remota** donde enviar los cambios de la rama de trabajo actual.
- El caso más común
  - `$ git push origin master`
- Si queremos “pushear” un branch especifico
  - `$ git push origin ramax:ramax`



Tarea 22: rama master local al servidor

- Crear un repositorio en github
- Marcar que lo inicialice con un “README.md”
- Agregar el repositorio remoto al repositorio local
  - Git remote add origin <http://github/<user>/ <proyecto>.git
- Verificar que todos los cambios estén “comiteados” y no haya nada pendiente de confirmar en el stage.
- Realizar un “pull” del servidor remoto para traernos los cambios remotos (el archivo README.md que se creo en el servidor)
  - git pull origin master **--allow-unrelated-histories**
  - *// usar --allow-unrelated-histories porque por defecto git no permite combinar dos repositorios que tengan commits iniciados, dado que su historia no es común.*

Tarea 23: Enviar tags locales al servidor

- Realizar un push al servidor remoto
  - Git push origin master
- Verificar el contenido del servidor remoto.
- Enviar ahora todos los tags locales al servidor remoto
  - git push --tags
- Verificar el contenido en el servidor remoto.
- Notar que como los valores SHA1 dependen de los archivos y no de los repositorios los mismos se mantienen.