CONTROL DE VERSIONES

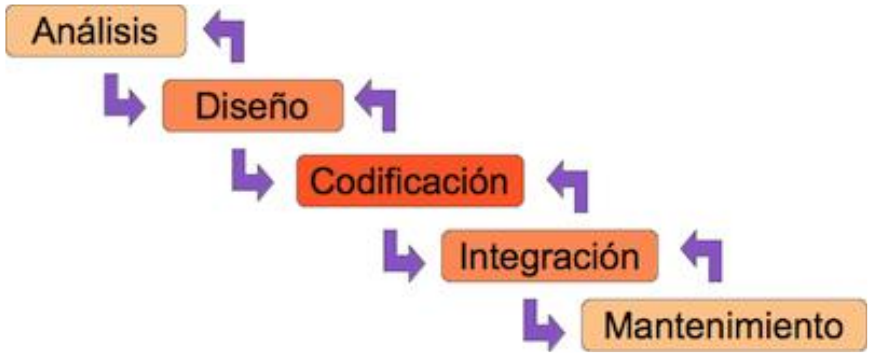
Índice

1. Sistemas de control de versiones
2. Funcionamiento básico de Git

Sistemas de control de versiones

Introducción al Control de Versiones

- Un Sistema de Control de Versiones (*VCS, Version Control System*) es una herramienta CASE, utilizada en la fase de codificación, integración y mantenimiento
- No necesariamente incluida en los entornos de desarrollo



- Las dos funciones principales de los VCS :
  - el almacenamiento de un histórico de la evolución del desarrollo
  - el control de conflictos cuando haya más de una persona trabajando sobre el mismo fichero

Introducción al Control de Versiones

- El funcionamiento de un VCS de manera general y resumida se podría describir como:
  1. Se almacenan los ficheros del código fuente en un servidor
  2. Los ficheros de ese servidor no se modifican directamente
  3. Cada desarrollador tiene su propia copia del código
  4. Si un desarrollador quiere modificar un fichero informa que va a realizar dicha modificación
  5. Cuando la modificación está realizada, actualiza el fichero del servidor
  6. Cuando dos usuarios modifican el mismo fichero, el sistema resuelve los conflictos automáticamente línea por línea
     si un programador modifica las líneas superiores del fichero y otro las inferiores, el sistema automáticamente realizará la mezcla
  7. En el caso de que ambos programadores modifiquen las mismas líneas, el sistema avisa para que el último desarrollador resuelva el conflicto de manera manual

¿Cuándo usar un VCS?

- La respuesta es sencilla:
  - un desarrollador debería siempre utilizar un VCS para controlar su código

- Seguro que como desarrolladores hemos tenido alguna vez la necesidad de realizar alguna de estas cosas:
  - Una vez modificado el código, detectar un error y querer volver atrás
  - Intentar recuperar código perdido y sólo disponer de una copia demasiado antigua
  - Al trabajar en equipo, estar al día de la última versión del desarrollo
  - Mantener varias versiones de un mismo producto
  - Ver las diferencias entre dos versiones del código
  - Probar como afecta a la aplicación la solución de un determinado error
  - Probar código que ha realizado otro programador
  - Comprobar qué, cuándo y dónde se realizó determinado cambio
  - Experimentar con nuevas características sin interferir con el código de producción

- En todos estos casos el uso de un VCS nos ayudará a ser más eficientes en nuestro trabajo

Git

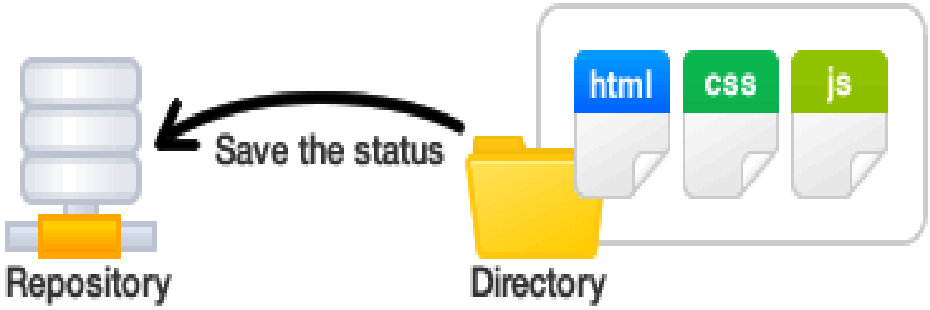
- Existen en el mercado multitud de VCS
  - Concurrent Versions System (CVS)
  - Subversion (svn)
  - Visual SourceSafe y Visual Studio Team Foundation Server
  - BitKeeper
  - Mercurial
- Pero sin duda el referente libre actual esta siendo Git:
  - Existen muchos interfaces gráficos para el manejo de Git. Además cada vez más IDE's van incorporando soporte para él, ya sea de manera nativa o mediante plugins
  - Cada vez hay más desarrolladores que trabajan con Git. Eso amplifica el efecto "viral"
  - Es distribuido. Los proyectos cada vez son más grandes y más modularizados. La distribución permite que varios equipos trabajen con sus propias ramas y revisiones sin interferir en el trabajo diario del resto de grupos.
  - Incorpora sistemas de importación/exportación a otros VCS, lo que hace que las migraciones no sean demasiado traumáticas
  - Existe mucha documentación en Internet (mucha de ella gratuita)
  - Su escalabilidad es excelente. Permite trabajar igual de bien con proyectos pequeños como con grandes
  - Es libre

Conceptos

Git

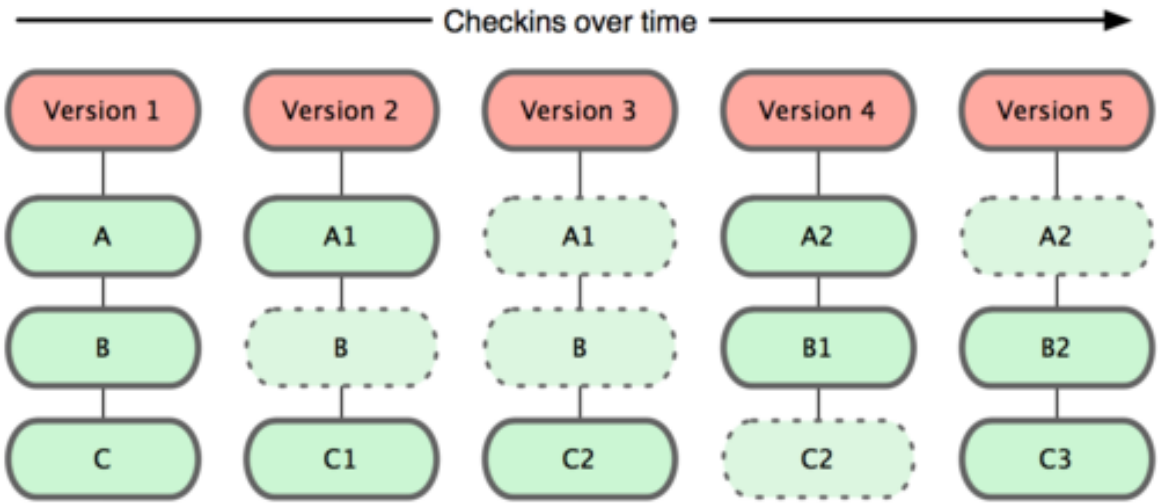
Repositorio

- **Repositorio o repo** [en: *repository*]:
  - Es la base de datos donde se almacenan el historial, las diferentes versiones, las diferentes ramas, etc... de un proyecto



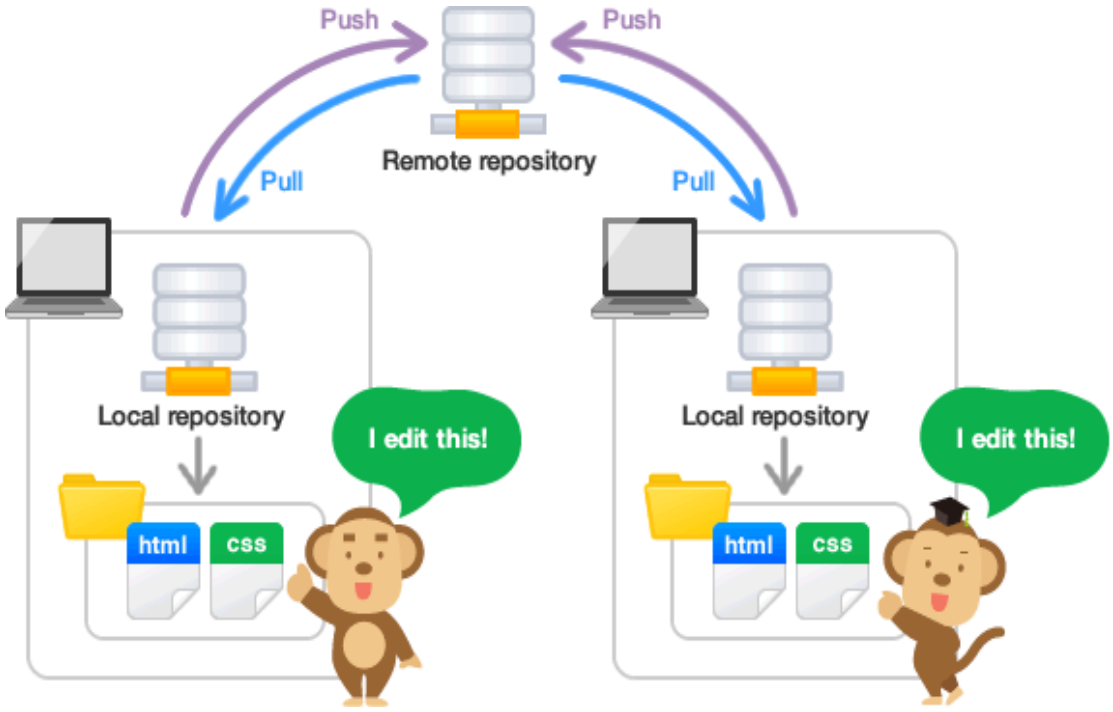
Repositorio

- Git modela sus datos como un conjunto de instantáneas de un mini sistema de archivos:
  - Cada vez que confirmas un cambio o guardas el estado de tu proyecto, básicamente hace una foto del aspecto de todos tus archivos en ese momento y guarda una referencia a esa instantánea
  - Si los archivos no se han modificado no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado



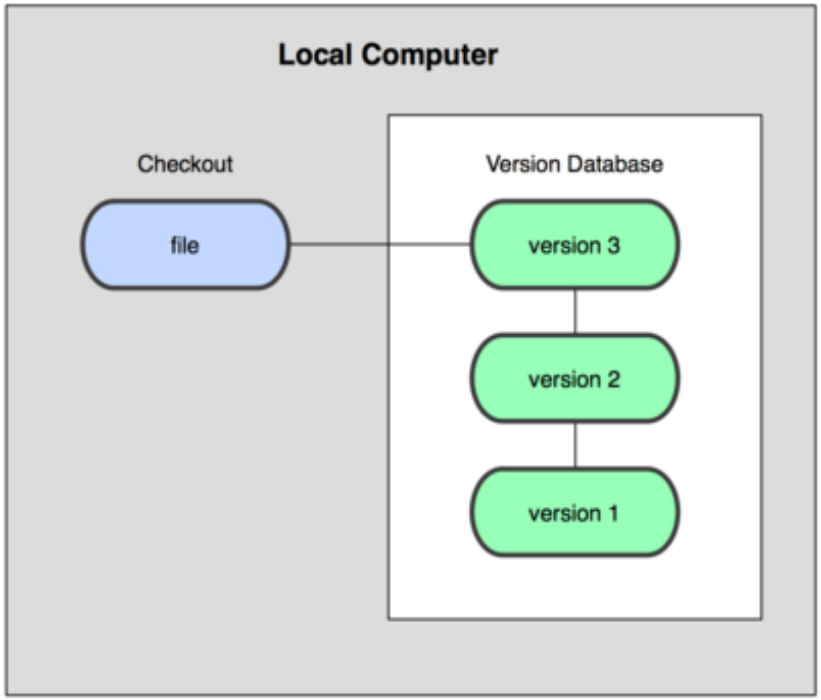
Repositorios locales y remotos

- **Remoto:** reside en el servidor y es compartido por todos los miembros del equipo
- **Local:** Individual para cada usuario, reside en su máquina



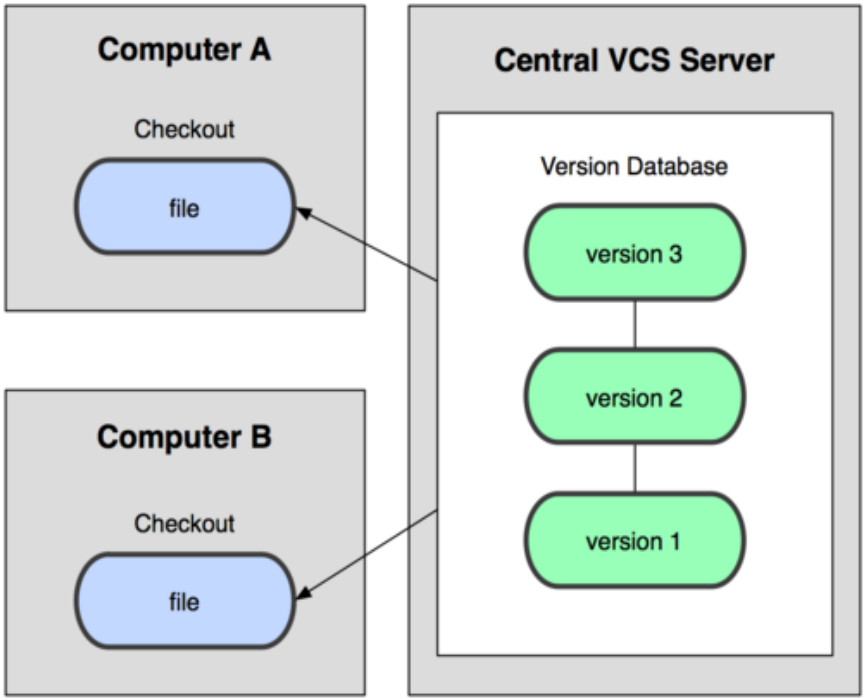
Git: ¿local, centralizado o distribuido?

- **Sistemas de control de versiones local**
  - Podemos instalar **GIT local**
  - El repositorio local lleva un registro de todos los cambios realizados sobre los archivos de nuestro proyecto



Git: ¿local, centralizado o distribuido?

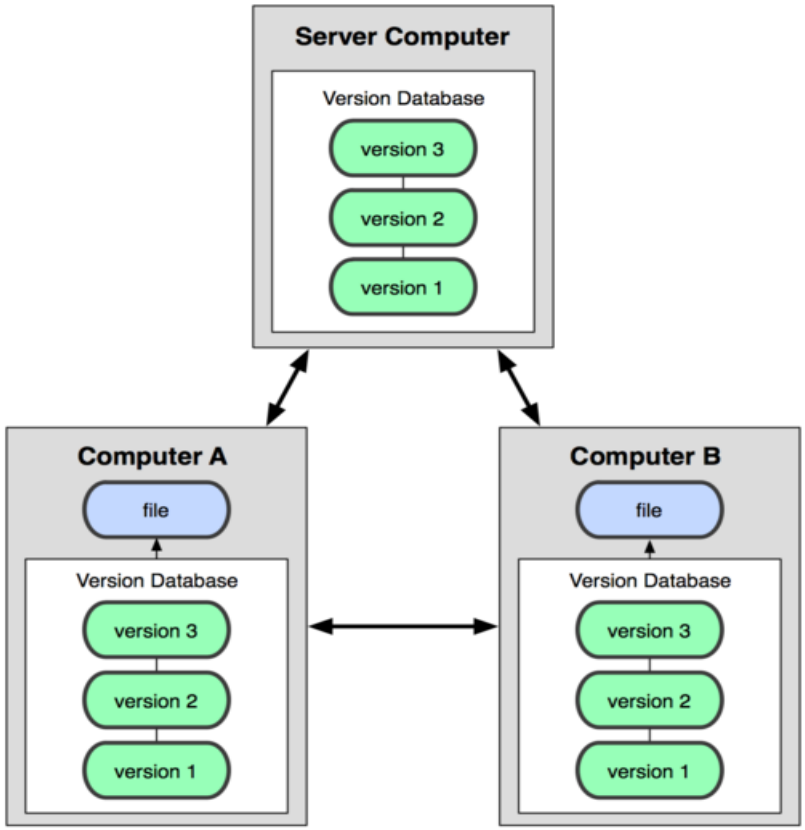
- **Sistemas de control de versiones centralizado**
  - Necesitamos tener un **repositorio remoto común** donde todos tengan acceso de
    - envío *(push)* y
    - recepción *(pull)*



- Ej: GitHub, GitLab

Git: ¿local, centralizado o distribuido?

- Git es un sistema de control de versiones distribuido (*Distributed Version Control Systems* o DVCSs):
  - En un DVCS **los clientes** no sólo descargan la última instantánea de los archivos sino que **replican completamente el repositorio**
  - Si un servidor muere, y estos sistemas estaban colaborando a través de él, cualquiera de los repositorios de los clientes puede copiarse en el servidor para restaurarlo.



Funcionamiento básico de Git

Libro oficial de Git en Español:

https://git-scm.com/book/es/v2/

Guía sencilla:

http://rogerdudler.github.io/git-guide/index.es.html

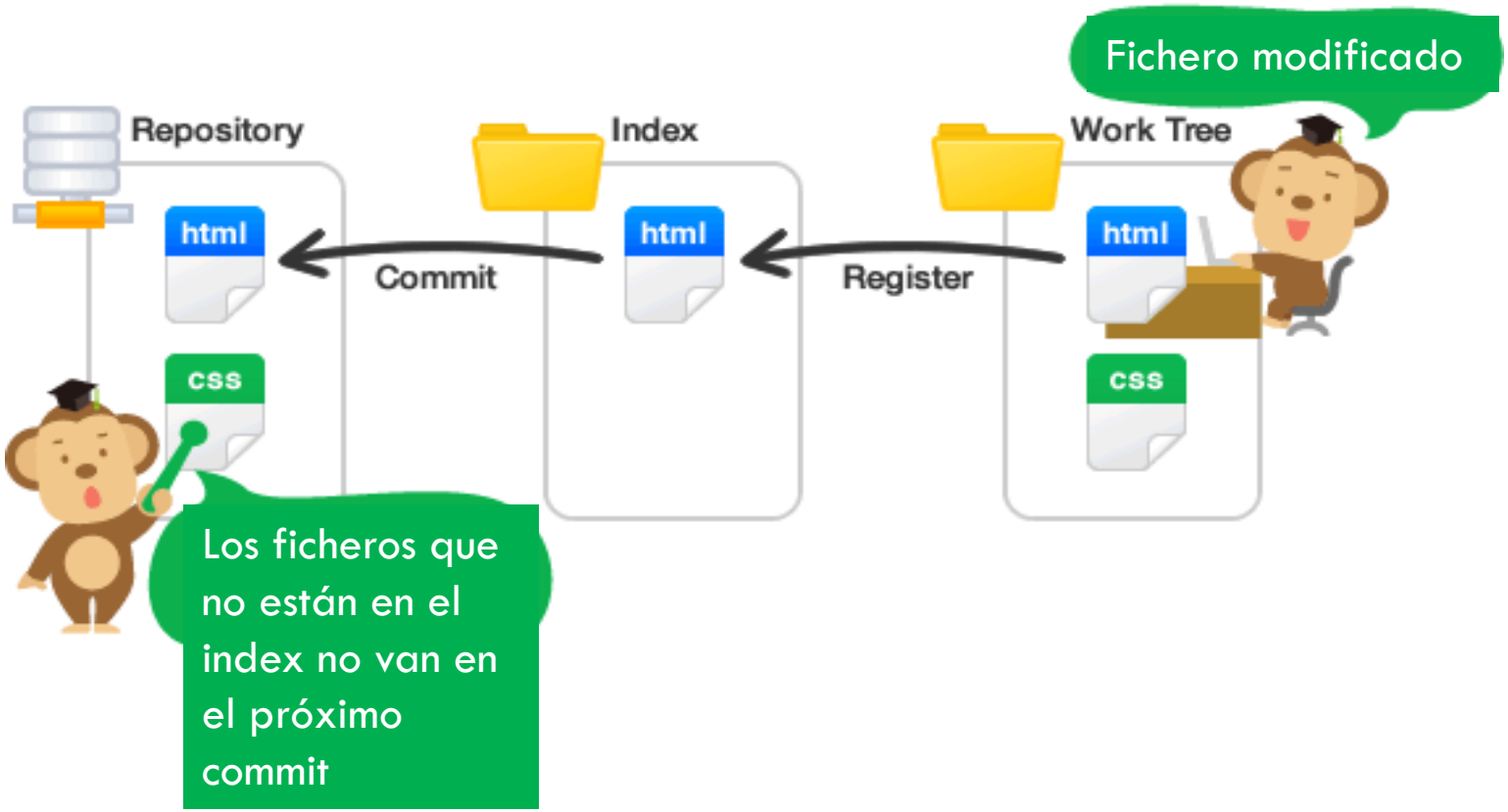
Crear un repositorio

- Hay 2 modos de obtener un repositorio local:
  - **Crear un repositorio** en un directorio existente:
    - Situado en el directorio que queremos seguir, ejecuta

      > git init
  - **Clonar un repositorio** existente en otro servidor:
    - Ejecutando la orden > git clone [url]
      - Por ejemplo.... >git clone https://github.com/libgit2/libgit2

Áreas de trabajo en un proyecto Git

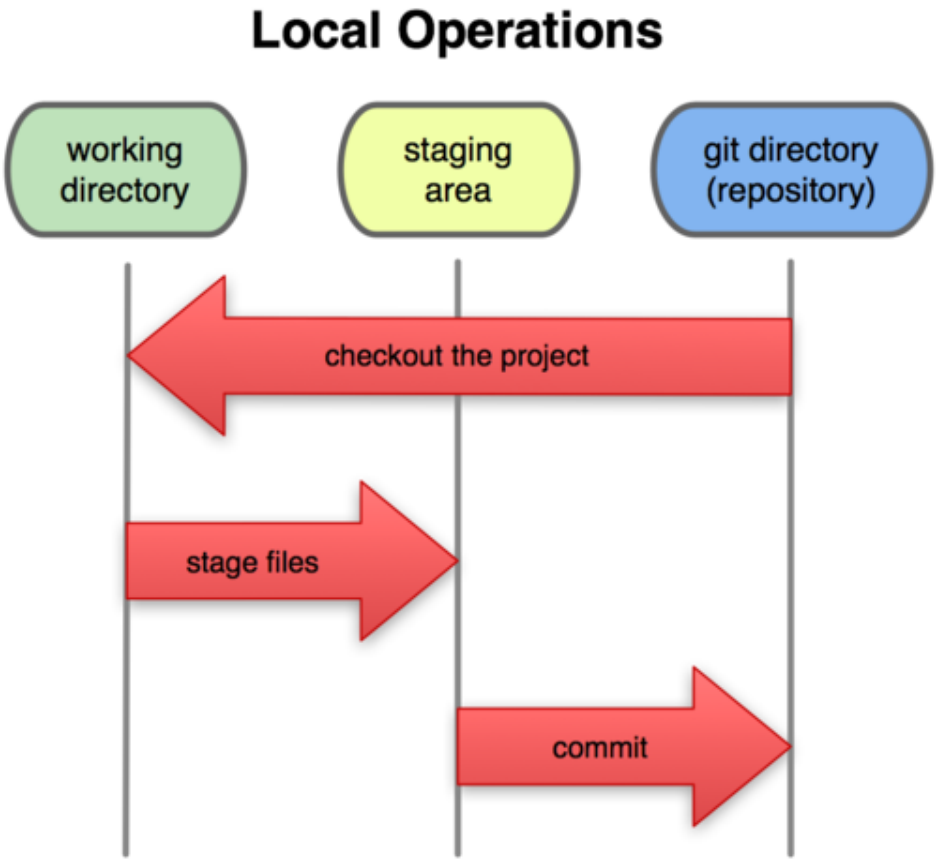
- En un proyecto Git se contemplan tres secciones o áreas de trabajo:



Áreas de trabajo en un proyecto Git

- **el directorio de trabajo** (Working Directory): ficheros sobre los que trabajamos
- **el área de preparación** (Staging Area): también llamada **Index:** donde se preparan los archivos que han sido modificados y se quieren incluir en el siguiente commit
- **el directorio de Git** (Git directory o Repository): donde se almacenan las instantáneas del estado del área de preparación en el momento de hacer un commit y toda la información necesaria para llevar el seguimiento de las versiones del proyecto

Áreas de trabajo en un proyecto Git

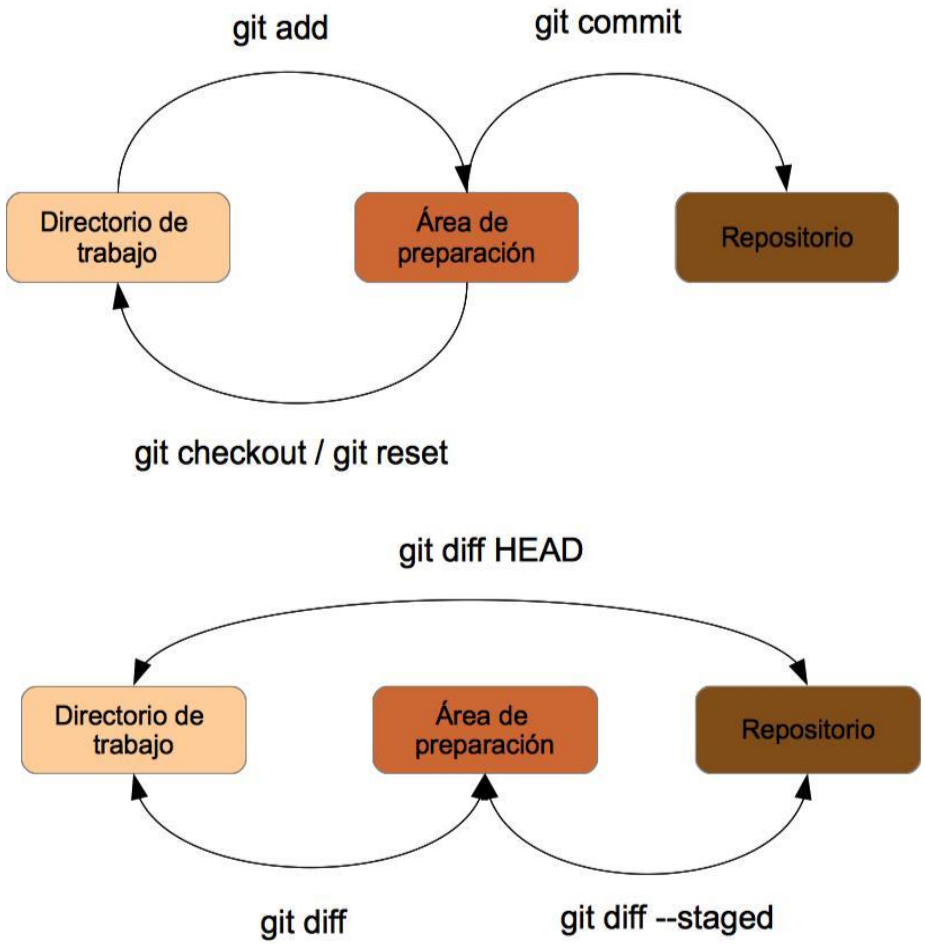


Áreas de trabajo en un proyecto Git

- Nuestra tarea habitualmente será:

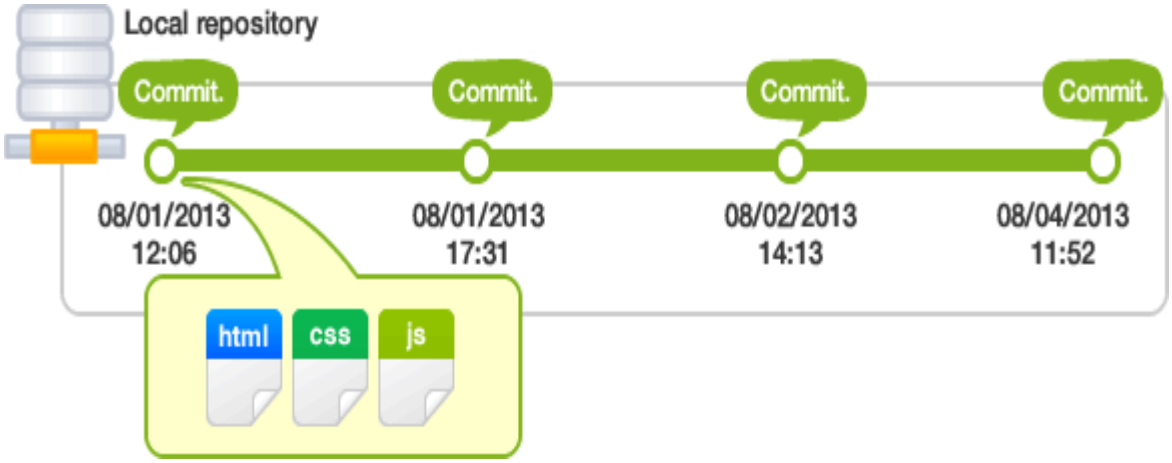
Áreas de trabajo. Algunos comandos GIT

- Con diferentes órdenes podrás:
  - Observar las diferencias entre las distintas áreas
  - Deshacer cambios
  - Volver a un commit anterior
  - Etc...



Commit

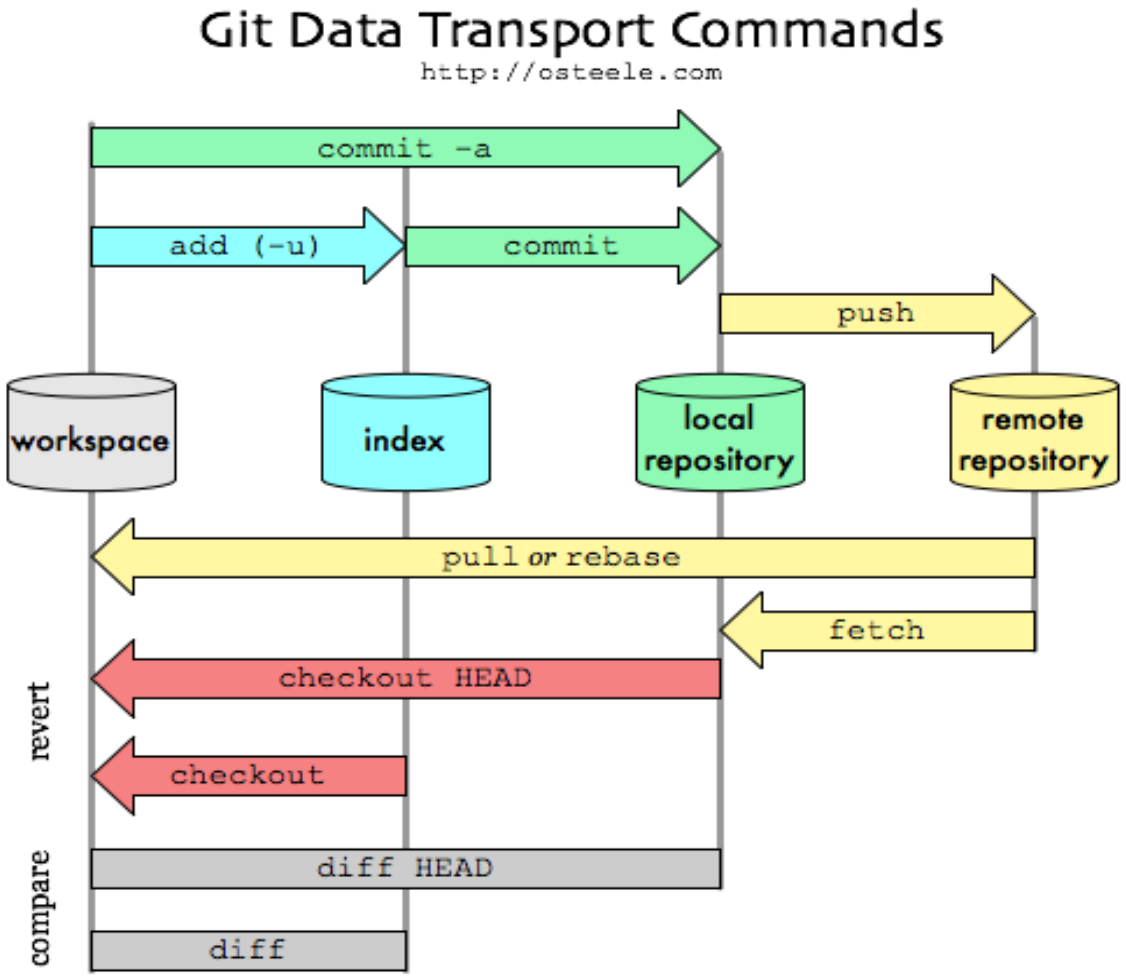
- Recoge los cambios que han sufrido los archivos en el área de preparación y guarda una INSTANTÁNEA de su estado actual en el repositorio



Commit

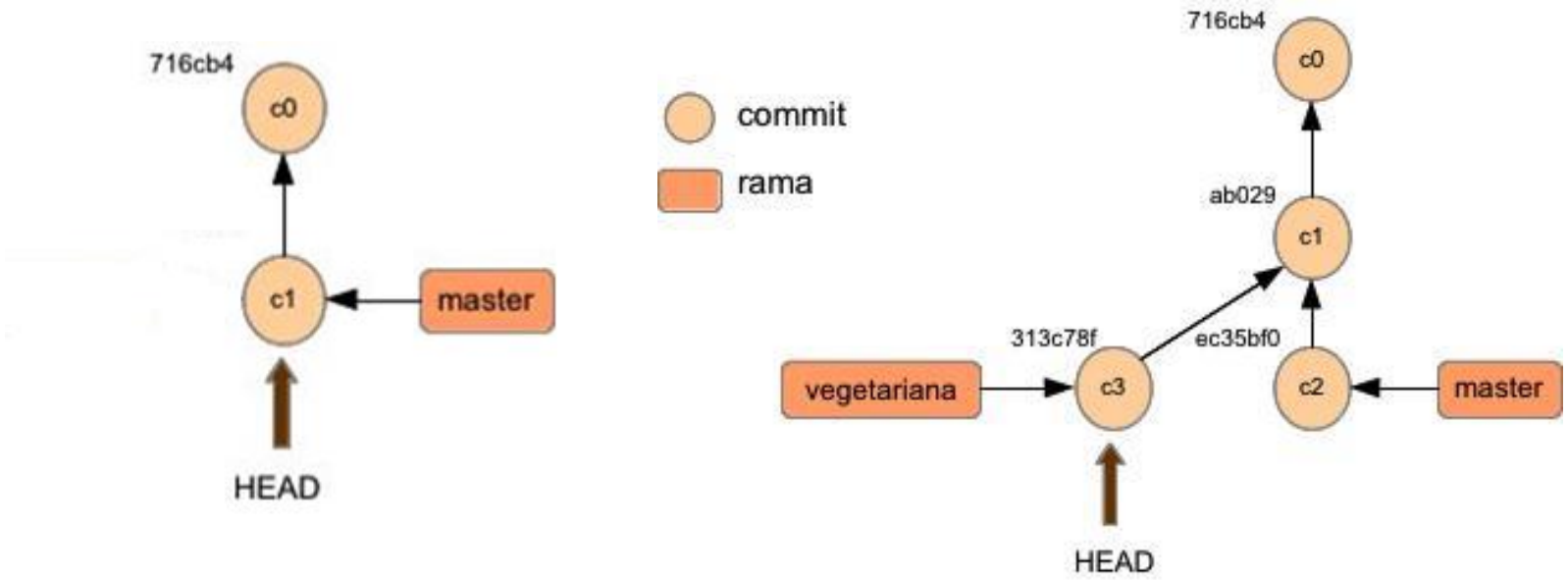
- Un commit se identifica de manera unívoca con 40 caracteres obtenidos mediante checksum de su contenido
- Será muy útil separar diferentes tipos de cambios como corrección de errores, nuevas funcionalidades, mejoras…en diferentes commits y etiquetar estos adecuadamente (al hacer un commit se requiere añadirle un mensaje que lo describa)
  - Esto ayudará en un futuro a entender cuándo, por qué y cómo se hicieron esos cambios
  - Para ello el mensaje debe ser lo más descriptivo y sencillo posible

Movimiento de datos y sus comandos



HEAD

- HEAD es una referencia simbólica que apunta al *commit* en el que estamos trabajando
  - Por defecto al último commit de la rama MASTER



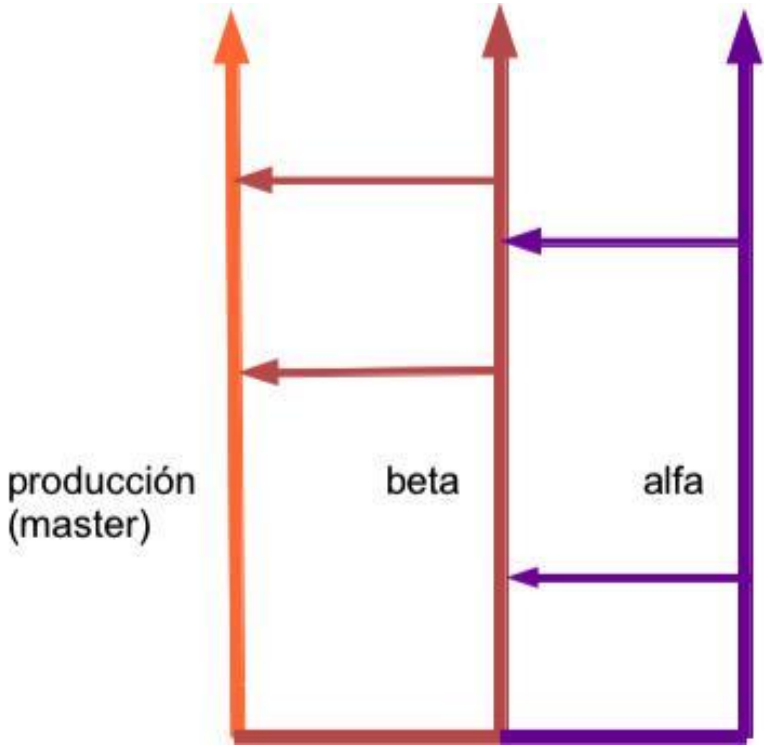
Ramas o Branches

- Una rama es un mecanismo que integran algunos CVS para permitir aislar de la línea de desarrollo principal cambios que se quieren realizar

- Esa línea de desarrollo principal ya es en sí una rama y por lo tanto todo proyecto tiene una
  - En Git a esa rama principal se le llama MASTER

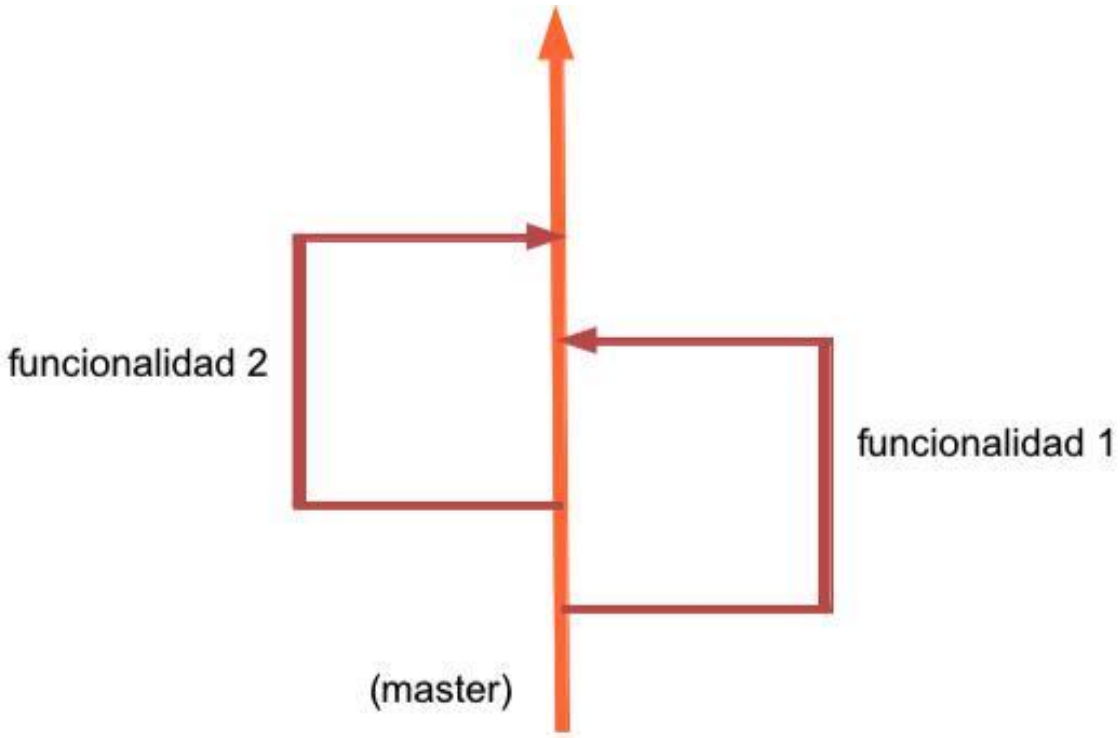
Ramas

- Ramas por estabilidad



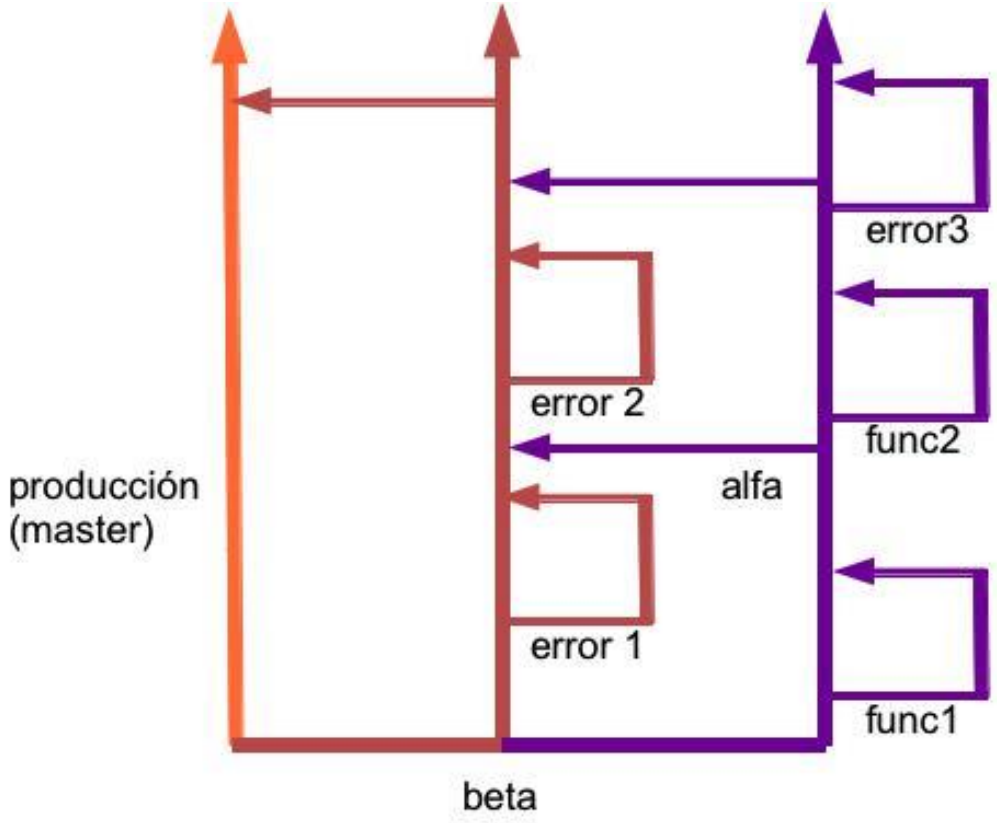
Ramas

- Ramas por nuevas funcionalidades

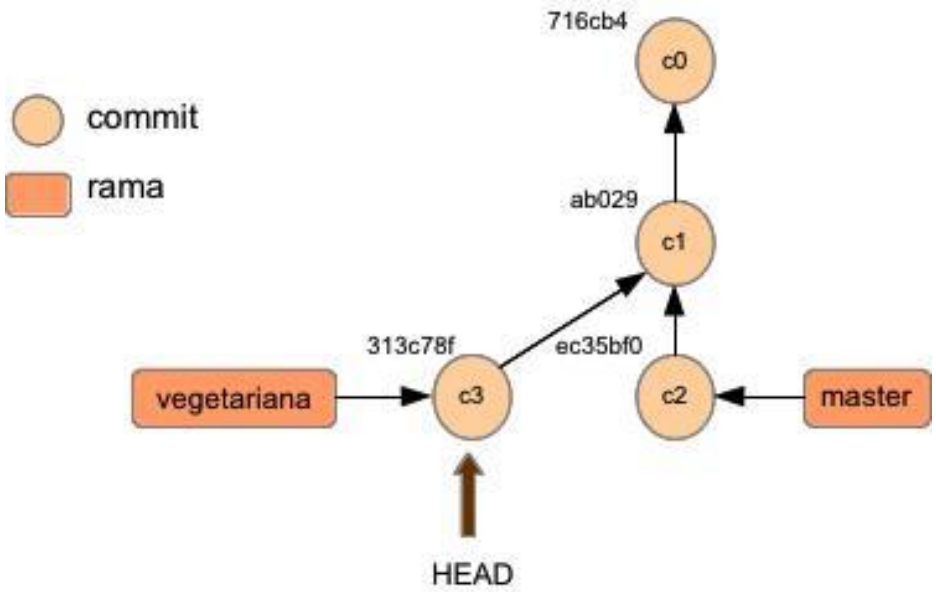


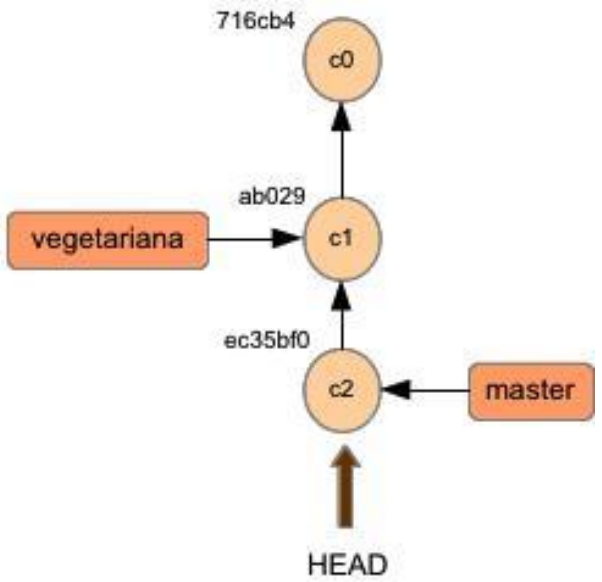
Ramas

- Todas las ramas juntas



Ramas


commit
rama
716cb4
c0
ab029
c1
313c78f
ec35bf0
vegetariana
c3
c2
master
HEAD


716cb4
c0
ab029
vegetariana
c1
ec35bf0
c2
master
HEAD

Ramas. Algunos comandos GIT

- ***Crear ramas***
  - git branch <nombre_rama>
- ***Ver ramas disponibles***
  - git branch
- ***Cambiar de rama***
  - git checkout <nombre_rama>
- ***Fusionar una rama***
  - Primero nos posicionamos en la rama sobre la que se va a realizar la fusión
  - git merge <nombre_rama_a_fusionar>
- ***Eliminar una rama***
  - git branch -d <nombre_rama>

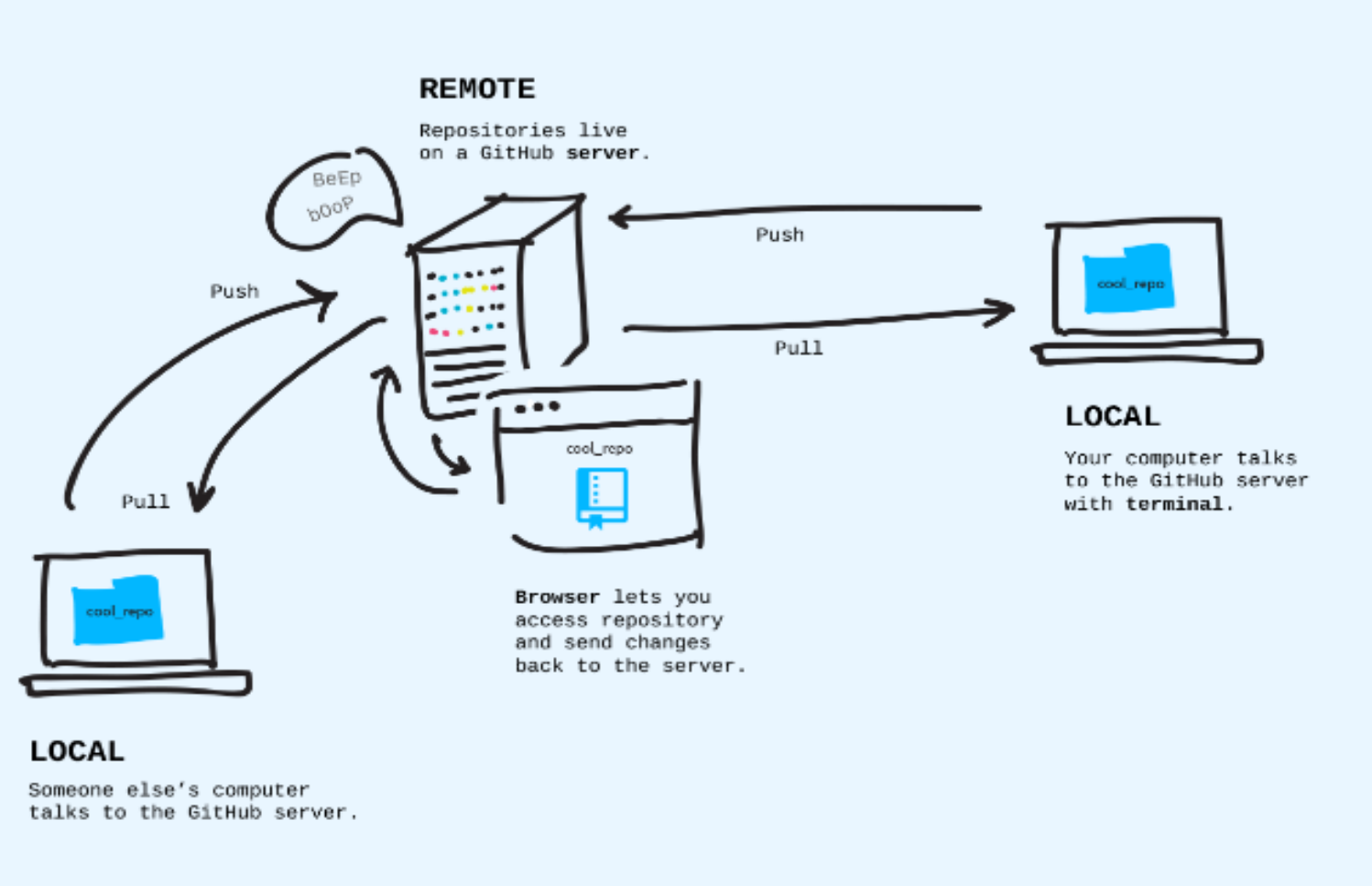
Sobre la fusión de ramas

- El resultado al fusionar las ramas va a depender de la situación previa.
- Existen 3 posibilidades de realizar ese mezclado:
  - fast-forward (avance rápido)
  - merge (fusión)
  - rebase (reorganización)
- Es interesante entender internamente la fusión...
  - Lo planteamos a final del tema como ampliación

Repositorios Remotos

- Los repositorios remotos son versiones de tu proyecto que están hospedadas en Internet en cualquier otra red
  - Puedes tener varios
  - En cada uno tendrás diferentes permisos (solo lectura, lectura y escritura,...)
- Para poder colaborar en cualquier proyecto Git necesitas:
  - Enviar y traer datos de ellos cada vez que necesites compartir tu trabajo
  - Gestionar repositorios remotos: añadir un repositorio, eliminar los remotos que ya no son válidos, gestionar varias ramas remotas, etc

Remotos


REMOTE
Repositories live on a GitHub server.
BeEp
bOoP
Push
Push
Pull
Pull
cool_repo
cool_repo
cool_repo
LOCAL
Your computer talks to the GitHub server with terminal.
Browser lets you access repository and send changes back to the server.
LOCAL
Someone else's computer talks to the GitHub server.

Remotos. Algunos comandos GIT

- **Clonar un repositorio**
  - git clone <URL_REPOSITORIO>
- **Traer información del remoto**

  Este comando NO realiza la fusión en la rama local

  Si se desean incorporar los cambios habría que realizar un git merge

  El nombre del remoto por defecto es ORIGIN
  - git fetch [NOMBRE_REMOTO]
- **Traer y fusionar cambios del remoto**

  La orden Pull conlleva un fetch y un merge
  - git pull [NOMBRE_REMOTO] [NOMBRE_RAMA]
- **Enviar cambios al remoto**
  - git push [NOMBRE_REMOTO] [NOMBRE_RAMA]
- **Enviar los cambios de una rama al remoto y crear una rama remota asociada**
  - git push -u NOMBRE_REMOTO NOMBRE_RAMA

Remotos. GitHub

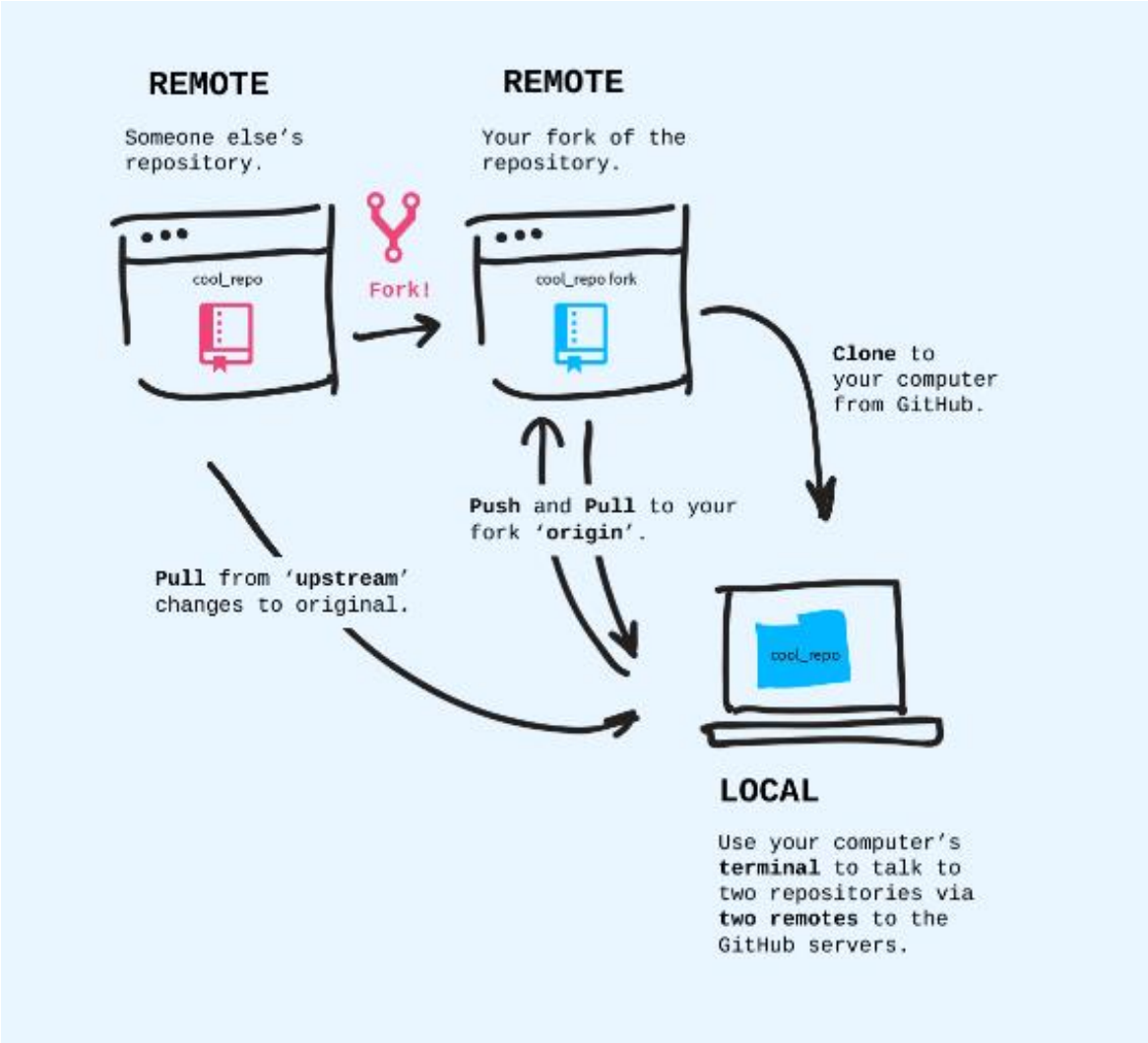
- GitHub es el mayor proveedor de alojamiento de repositorios Git
  - Muchos proyectos de código abierto lo utilizan para hospedar su Git, realizar su seguimiento de fallos, hacer revisiones de código y otras cosas
- Para crearte una cuenta, sigue las instrucciones:

  https://git-scm.com/book/es/v2/GitHub-Creaci%C3%B3n-y-configuraci%C3%B3n-de-la-cuenta

Remotos. Fork y Pull Request

Bifurcar un proyecto (o hacer un fork"):

Para participar en un proyecto existente en el que no tengamos permisos de escritura



Remotos. Fork y Pull Request

1. En la página del proyecto que queremos bifurcar pulsamos sobre el botón "Fork" del lado superior derecho de la página:

   - Se crea una copia completa del repositorio en nuestra cuenta
   - Este remoto recibe el nombre UPSTREAM por defecto

2. A continuación modificamos esa copia del proyecto:
   - Clonamos nuestro fork en nuestro equipo
   - Creamos una rama (que sea descriptiva)
   - Realizamos nuestros cambios
   - Comprobamos los cambios
   - Realizamos un commit de los cambios en la rama
   - Enviamos nuestra nueva rama de vuelta a nuestro fork

Remotos. Fork y Pull Request

3. Si miramos nuestra bifurcación en GitHub, aparece un aviso de creación de la rama y nos da la oportunidad de hacer una solicitud de integración (Pull Request) con el proyecto original:

- permite crear un título y descripción para darle al propietario original una buena argumentación
  - debemos realizar cierto esfuerzo en hacer una buena descripción para que el autor sepa realmente qué estamos aportando y lo valore adecuadamente
- El propietario puede revisar el cambio sugerido e incorporarlo (merge) al proyecto o bien rechazarlo o comentarlo

Instalación de Git

Instalación Git

- Git está planteado como un sistema multiplataforma
- Existen paquetes distribuibles para los principales sistemas operativos (Linux, MacOS y Windows) así como para Solaris
- **En estos apuntes** se van a presentar las principales propiedades en **entorno Windows**:
  - Simplemente descargarlo e instalarlo como cualquier otra aplicación:

    http://git-scm.com/download/win

Ejemplo de uso

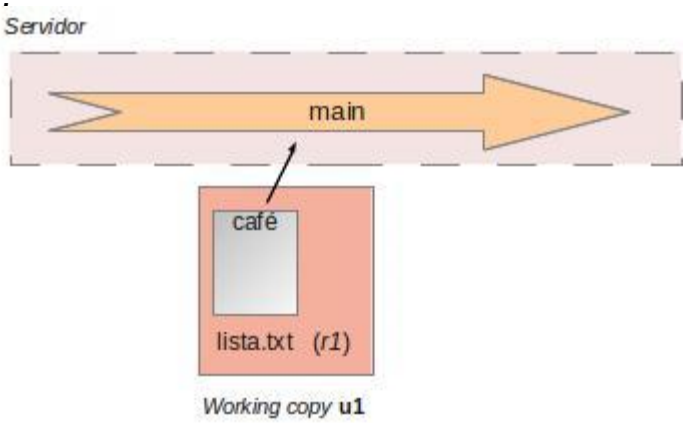
Para ampliar...

Ejemplo de uso

- Para entender un poco mejor la terminología anterior vamos a trabajar con un escenario ficticio sobre el que realizaremos algunas operaciones:
  - Tenemos un sistema con 2 usuarios: **u1, u2**
  - Ambos usuarios quieren llevar el control de la lista de la compra semanal (simulado con un fichero llamado *lista.txt*) incorporando ítems a la misma..

Ejemplo de uso

- **init**
- Se inicia el repositorio
- **add**
  - Lo primero que hay que hacer es añadir *lista.txt* al control de versiones mediante una comando de *Ad*



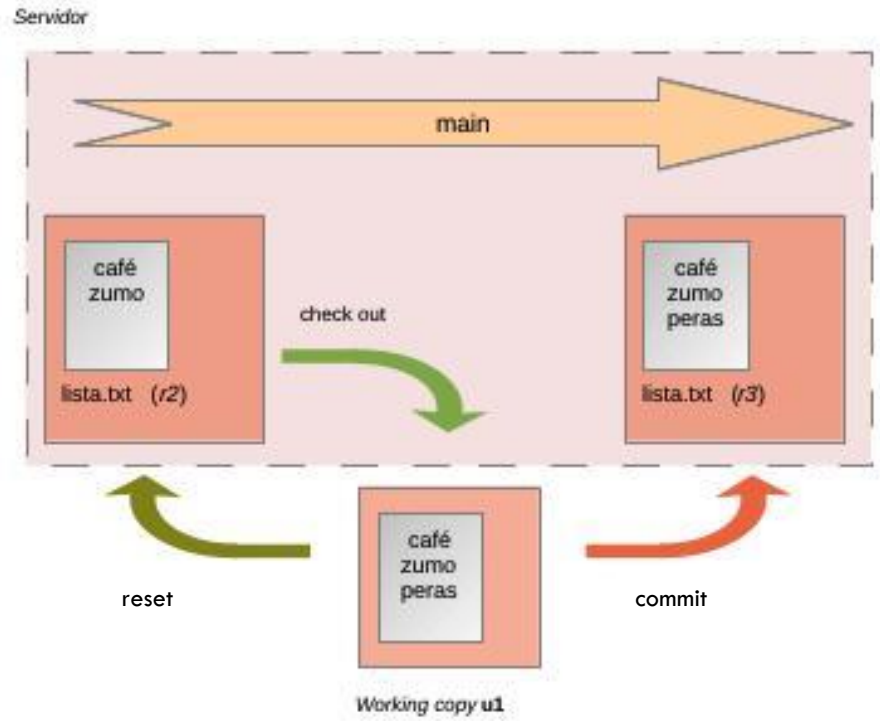
  - En este caso podemos ver como el usuario **u1**, desde su directorio de trabajo, incluye el fichero *lista.txt* a la línea principal (*main*) del servidor donde se encuentra el sistema de control
  - El fichero *lista.txt* ya tiene un ítem (*café*) pero podría estar vacío
  - Automáticamente el VCS le asigna un número de revisión (r1)

Ejemplo de uso

- **checkout , commit (o check in) y reset**
  - **u1** va modificando el fichero a lo largo de la semana incluyendo nuevos ítems a la lista
    - Para ello, **u1** realiza primero un ***checkout***. Esta operación en realidad son dos operaciones:
      - se actualiza el fichero *lista.txt* del directorio de trabajo a la última versión existente en el servidor (r1)
      - se activa la posibilidad de editarlo
    - Una vez **u1** tiene a su disposición la copia, la edita y añade un ítem (*peras*) y, una vez añadido, lo actualiza en el servidor mediante un ***commit*** creando una nueva revisión (r3)
    - Es posible que **u1** una vez modificado el fichero, se de cuenta de un error y quiera descartar los cambios y volver a la última versión del servidor para lo cual realizaría una acción ***reset***

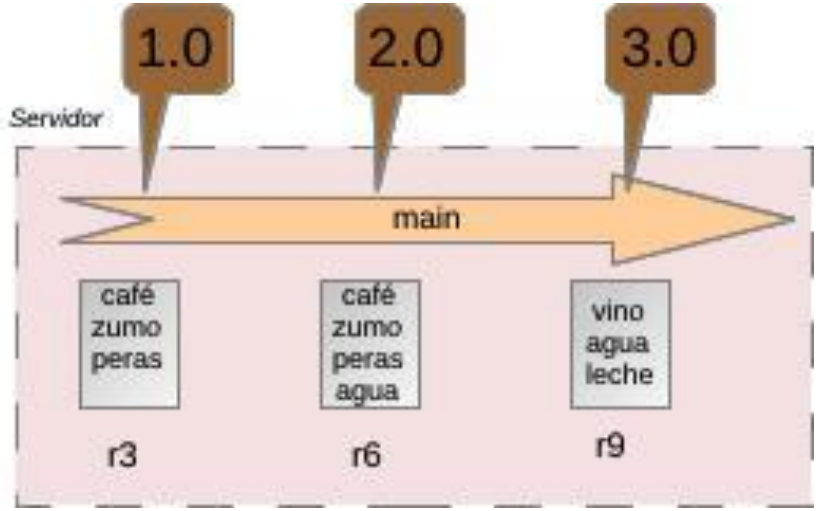
Ejemplo de uso

- **checkout, commit y reset**



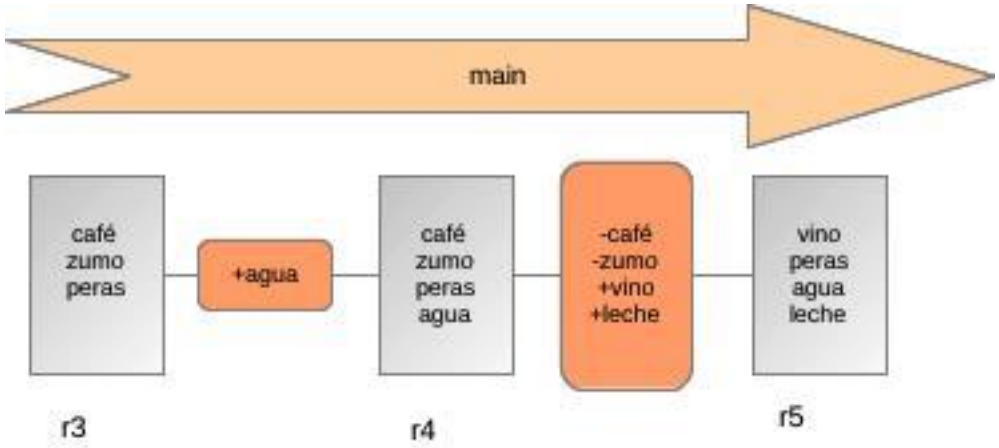
Ejemplo de uso

- **tags**
  - Puede ser interesante al final de cada semana, antes de empezar con la lista de la semana siguiente, etiquetar cada versión del fichero para saber que fue lo que se compró en una determinada fecha



Ejemplo de uso

- **diff**
  - Uno de los usuarios tiene interés en ver en qué se ha modificado la lista entre dos versiones. Para ello realizará una acción *dif*



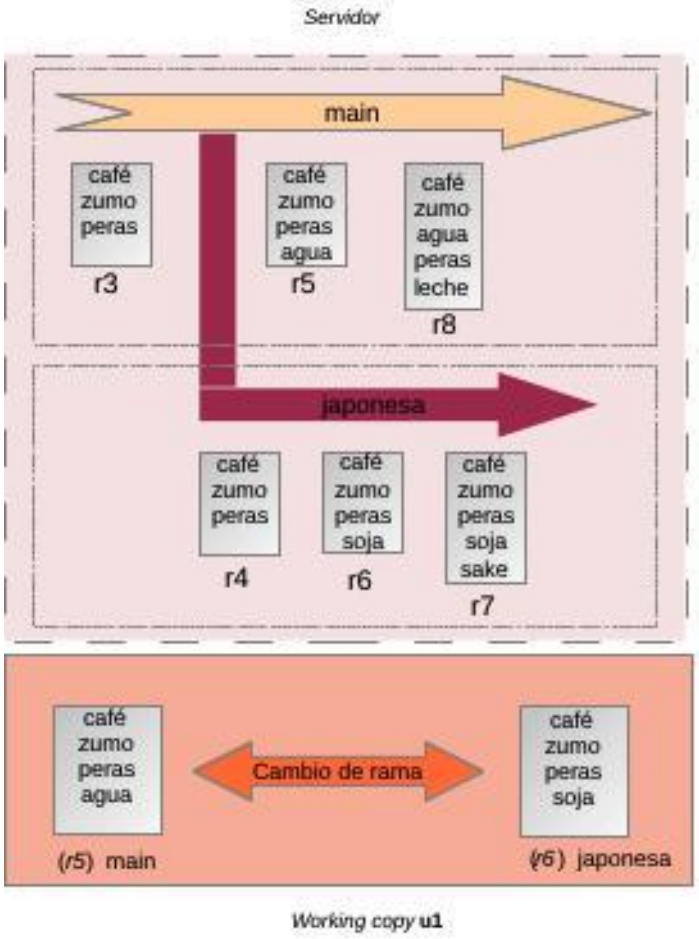
  - Para obtener las diferencias, el VCS se hace preguntas como *¿qué debo modificar en una versión para llegar a otra?*
  - Si nos fajamos en el ejemplo, para llegar a r5 a partir de r4, habría que añadir (+) *vino* y *leche* y eliminar (-) *café* y *zumo*

Ejemplo de uso

- **branch**
  - **u1** decide que quiere experimentar cocinando comida japonesa. Para ello abre una rama llamada *japonesa*, donde irá incluyendo ítems relacionados con ese tipo de comida, hasta que una semana decida ponerse a ello
  - Al abrir una rama *lista.txt* puede evolucionar por dos caminos distintos *main* y *japonesa*:
    - El punto de partida de *japonesa* (r4) es una copia de la revisión existente en *main* en el momento de abrir la rama
    - A partir de ese momento **u1** puede elegir sobre que rama trabajar
      - Si desea incluir algo en la rama *japonesa*, **u1** tiene que ir a esa rama, hacer un *chekout*, modificar y hacer un *commit* sobre la rama
      - Posteriormente para seguir añadiendo cosas en la lista del día a día debe volver a la rama *main*

Ejemplo de uso

- **branch**

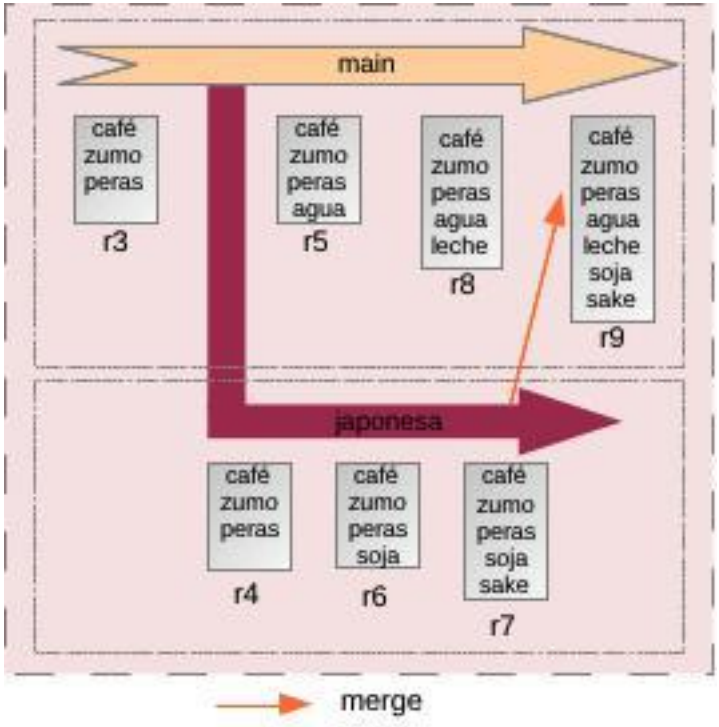


Ejemplo de uso

- **merge**
  - **u1** por fin se decide y la semana que viene está dispuesto a cocinar comida japonesa, por lo que ha de mezclar (*merge*) la lista de la rama *japonesa* con la rama principal que es de donde se obtiene la lista con la que se irá al mercado
  - Cuando el usuario mezcla una rama con otra, el VCS calcula las diferencias entre el principio de la rama a mezclar (o desde la última versión del fichero que fue mezclada) y la versión final y aplica dichas diferencias a la versión actual de la rama destino
  - En el ejemplo se calcula la diferencia entre r7 y r4, es decir añadir (+) *soja* y *sake*

Ejemplo de uso

- **merge**



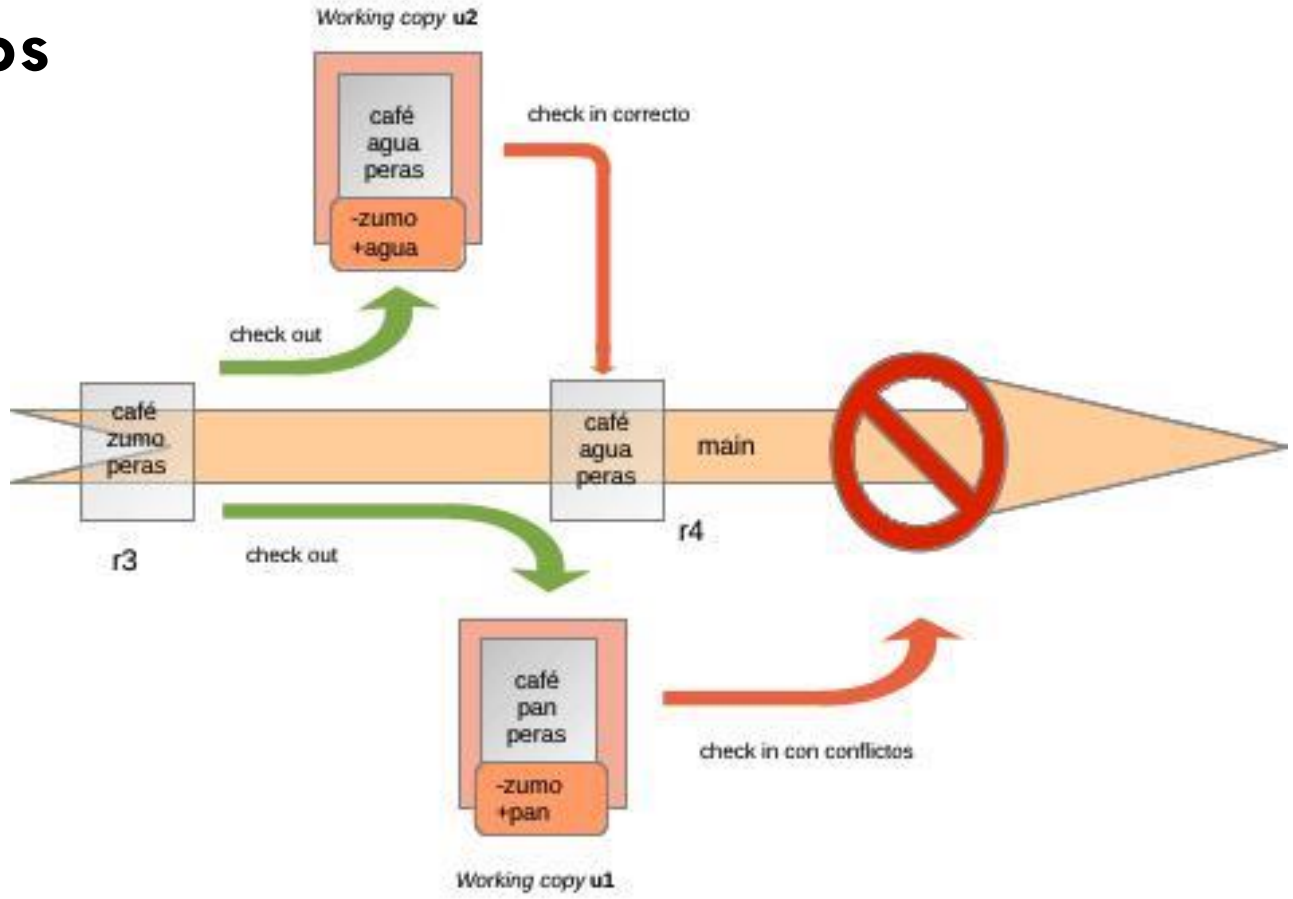
- El mezclado no sólo implica la existencia de dos ramas
- En el caso de que dos usuarios estén editando el mismo fichero, cuando ambos realizan el *checkin*, el VCS realiza automáticamente el *merge*

Ejemplo de uso

- **Conflictos**
- En la gran mayoría de casos el propio VCS mezcla las versiones sin mayor problema, pero puede darse casos en que esto no sea así:
  - Tanto **u1** como **u2** deciden que quieren cambiar la lista y para ello hacen un *check out* de manera más o menos simultánea
  - **u2** no quiere zumo y en cambio quiere agua, mientras que **u1** que tampoco quiere zumo, quiere pan en su lugar. **u2** realiza el cambio y el *checkin* antes que **u1**
  - Cuando **u1** va a realizar el *chekin*, su copia local tiene que mezclarse con la r4 pero el sistema detecta que desde que se hizo el *checkout* (r3) otro usuario ha cambiado las mismas líneas (en este caso la nº2)
  - El VCS no tiene información suficiente para decidir con cual quedarse (¿*agua* o *pan*?) y genera un conflicto
  - En este caso solicita al usuario **u2** que resuelva el conflicto, es decir, o que manualmente decida cual es la línea que deberá permanecer o que incluya las dos líneas
- En condiciones normales la aparición de conflictos no es (o al menos no debería ser) muy habitual
- Con un poco de organización, distribución ordenada del trabajo y haciendo *checkin* de manera más o menos habitual, su aparición disminuye mucho
- Aun así, si se va a editar gran parte del fichero y durante mucho tiempo, puede ser recomendable que el usuario lo bloqueé para evitar la edición por parte de otros

Ejemplo de uso

- **Conflictos**

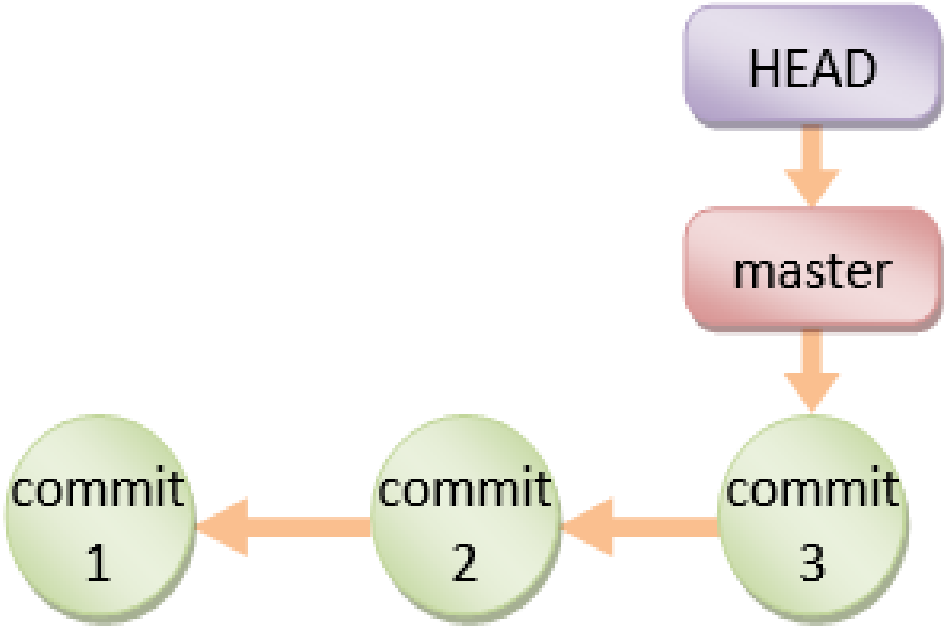


Fusión de Ramas

Para ampliar...

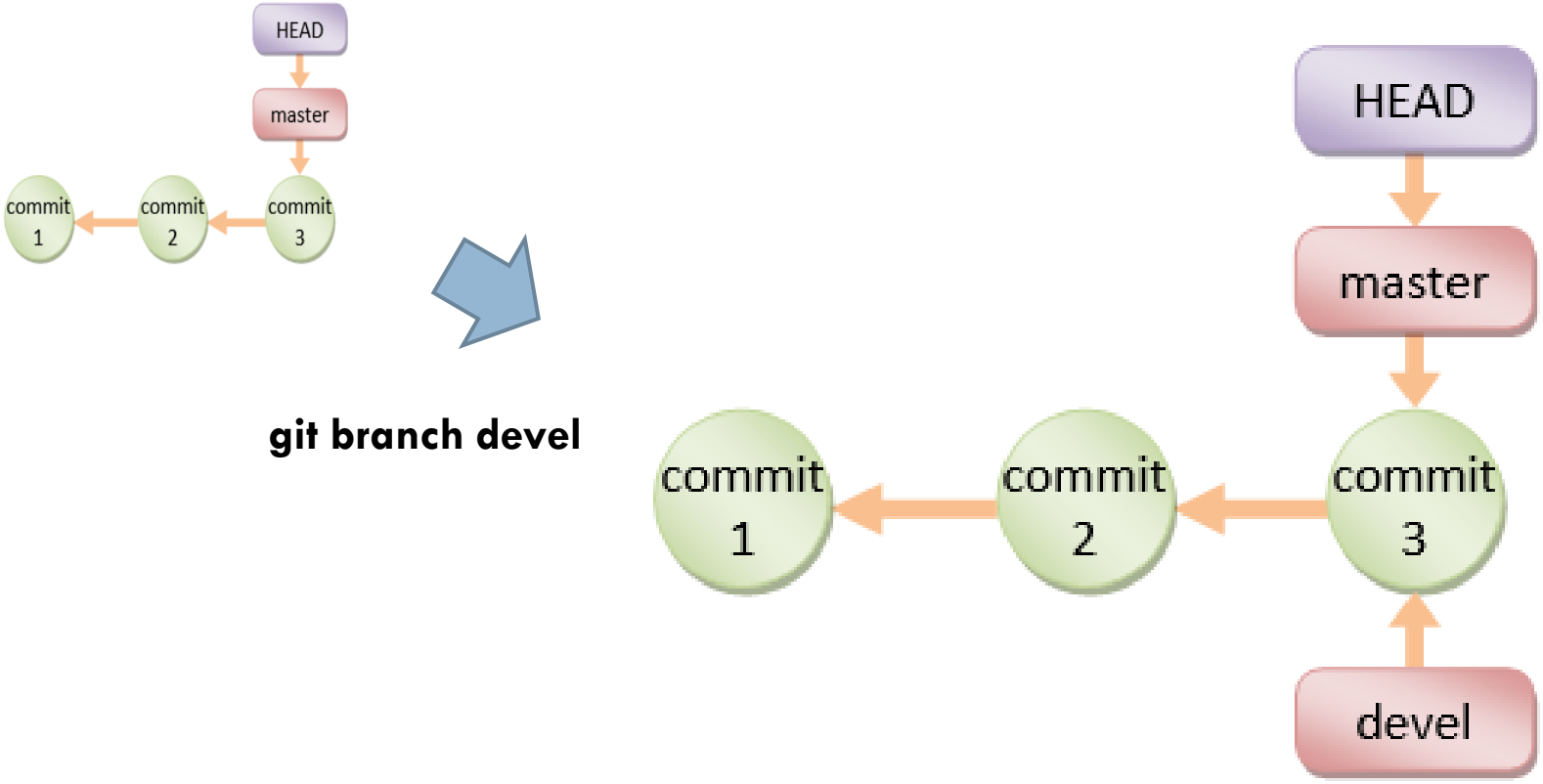
Rama principal (master)

- Inicialmente tenemos sólo la rama principal



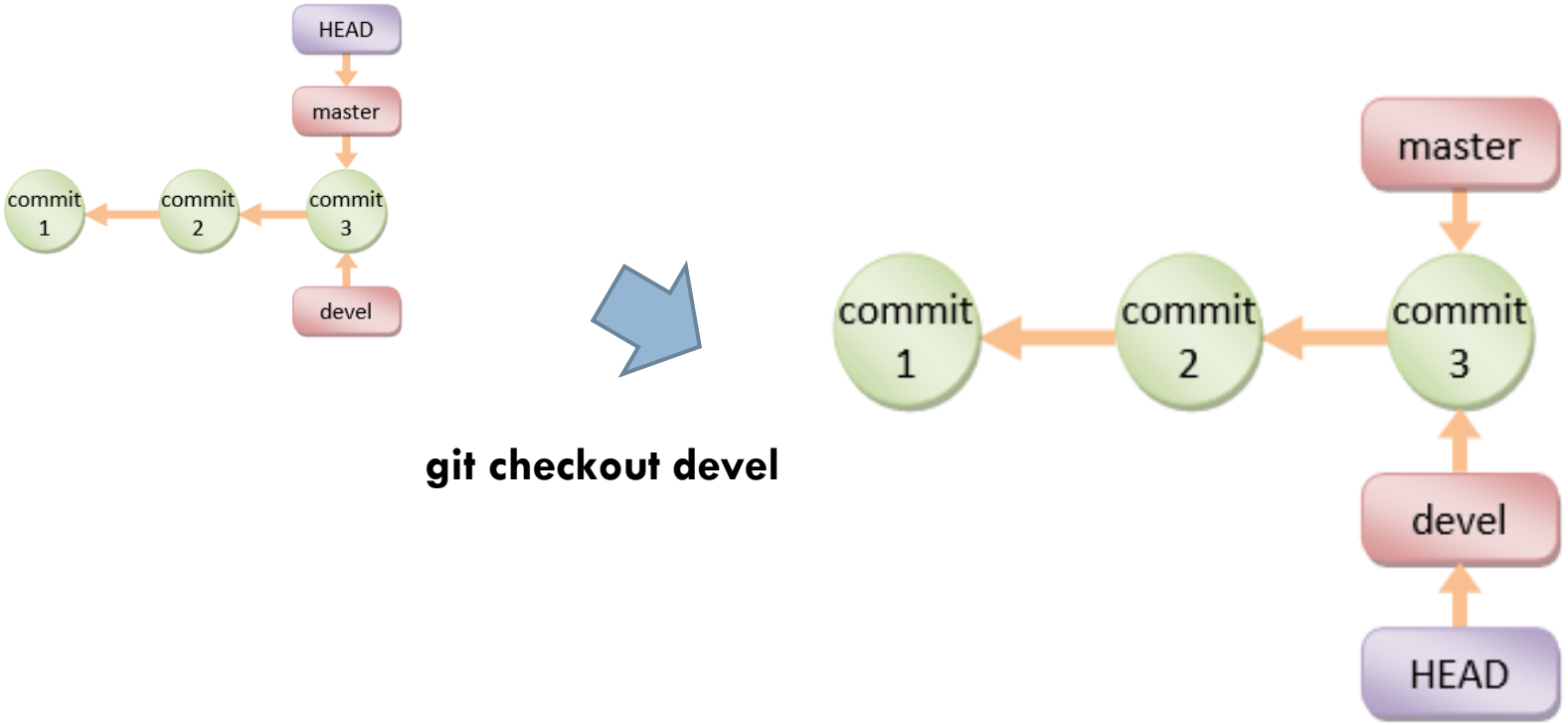
Crear una rama

- **$ git branch <nombre rama>**

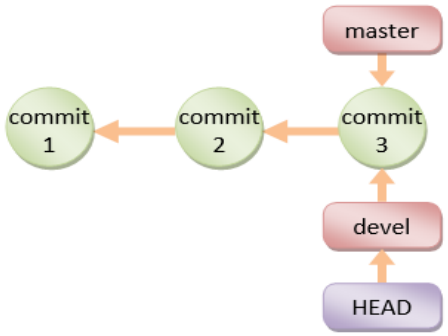


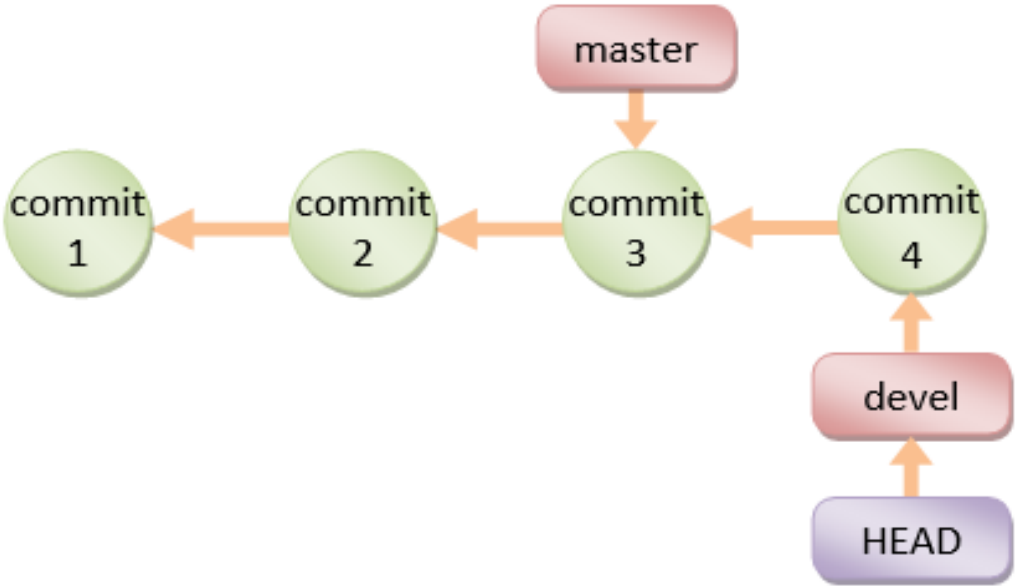
Cambiar a otra rama

- **git checkout <branch-name>**



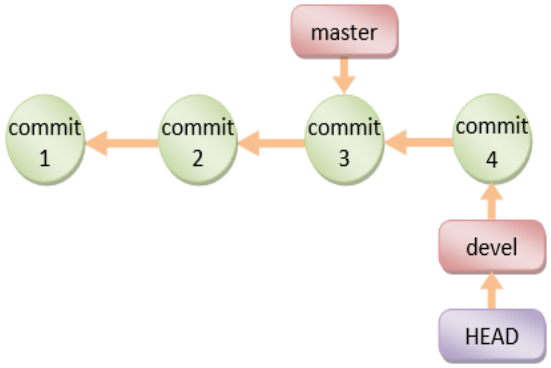
Commit en la nueva rama

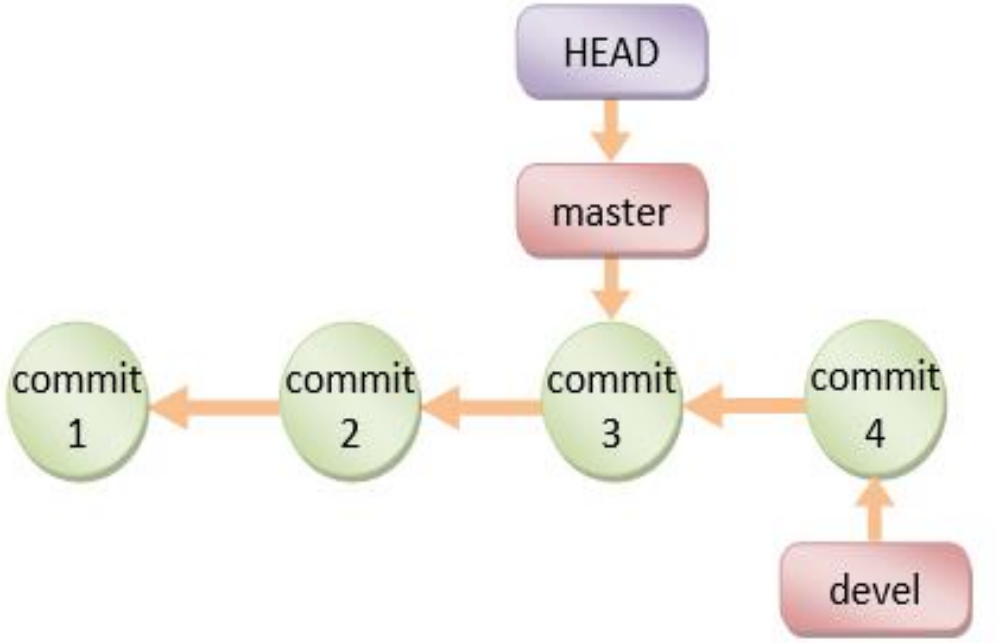




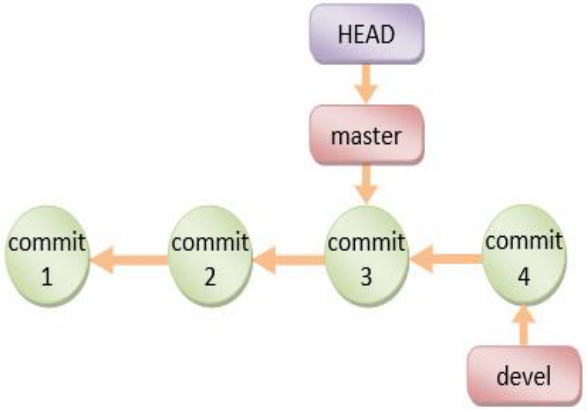
Volvemos a la rama principal

- git checkout master

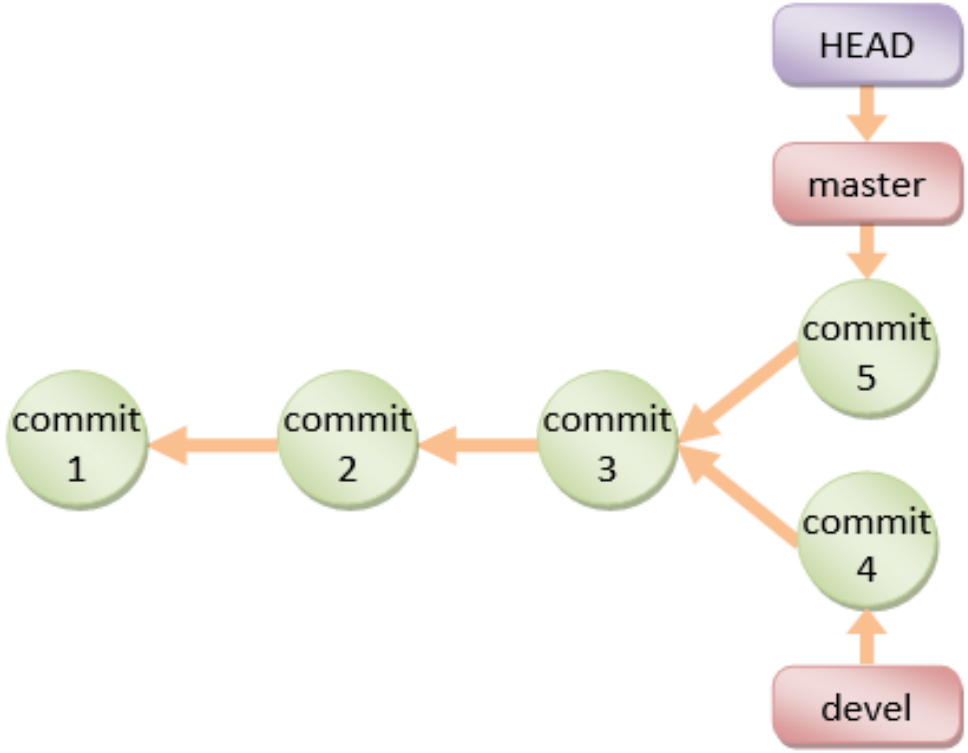




Commit en master



Ahora las ramas divergen

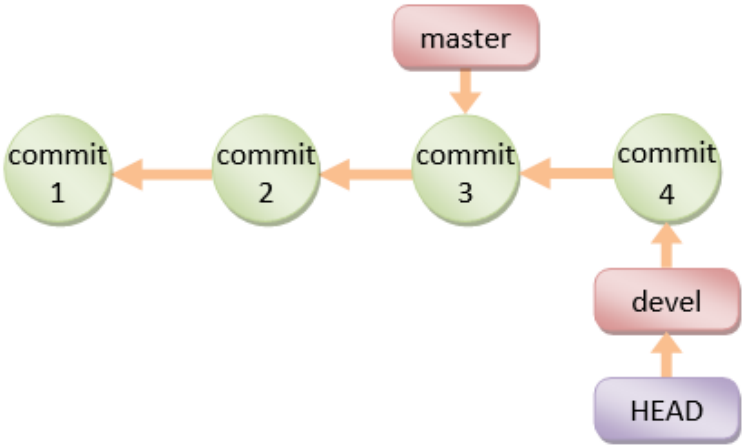


Merging

Existen 3 posibilidades de realizar la fusión:

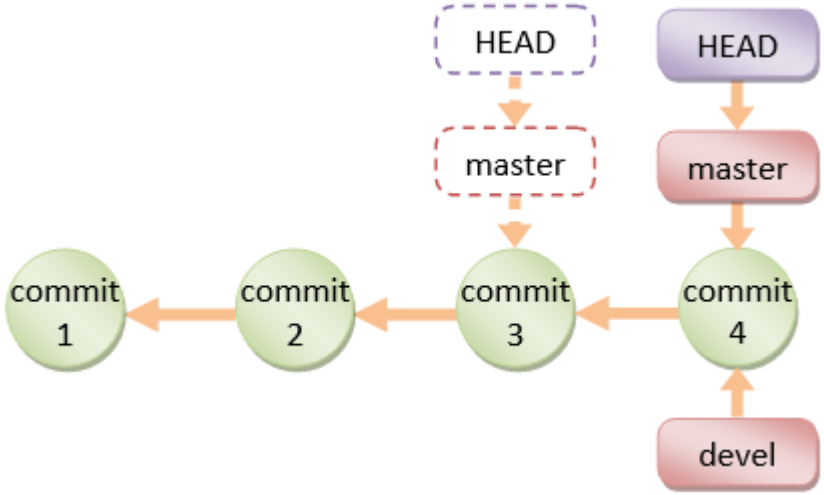
- **Fast-Forward** (avance rápido): queremos mezclar 2 ramas donde un desciende del último commit de la otra
- **Merging**: Queremos mezclar 2 ramas divergentes
- **Rebase**: Se reorganizan las ramas para mantener un histórico lineal

Fast-Forward



Sólo se mueve la cabeza de la rama que queremos actualizar al último commit. No se crea un nuevo commit

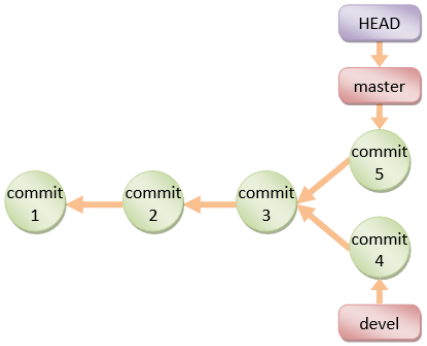
```
git checkout master
git merge devel
```

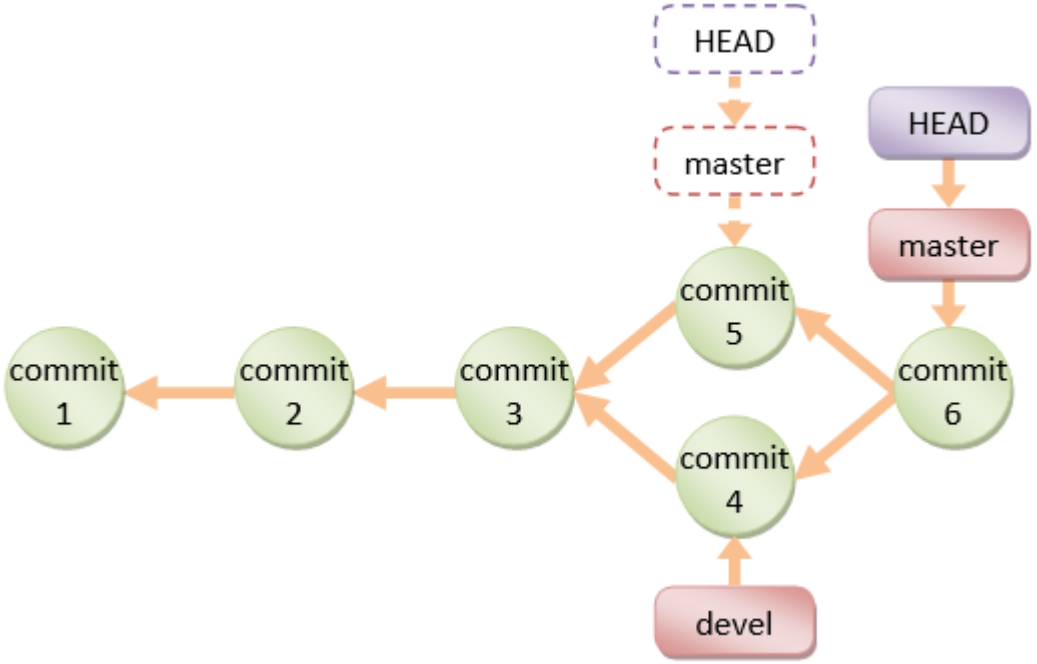


Mezcal habitual: Merge

- Si las 2 ramas divergen

En la rama que queremos actualizar se crea un nuevo commit que reúne las diferencias de ambas

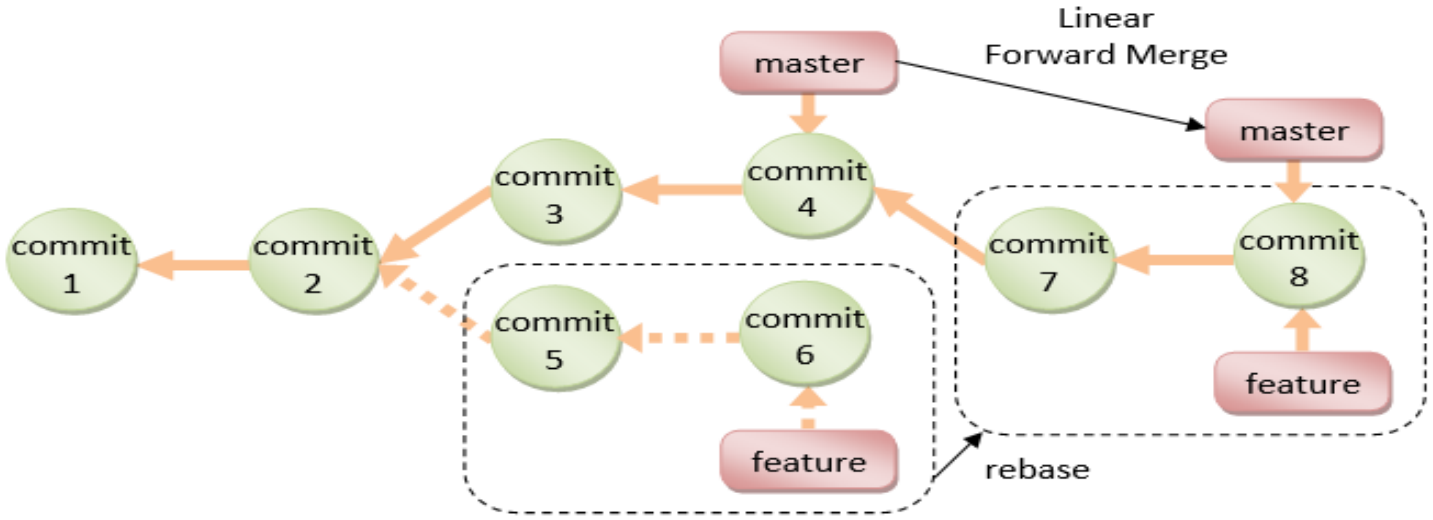





```
git checkout master
git merge devel
```

Rebasing

- El propósito es mantener un histórico lineal
- La rama a fusionar se reestructura con sus propios cambios y con los cambios de los commits de la rama principal (desde que divergieron) y queda como descendiente de la principal. Así queda preparada para un fast-forward si interesa



Checkout y Detached HEAD

- Si después del checkout commit 2 hacemos otro commit (5), no podemos fusionar con la rama master porque el commit 5 no está en ninguna rama que podamos nombrar
- Resumiendo: Se puede usar un git checkout <commit-name> para inspeccionar un commit, pero SIEMPRE SE DEBE TRABAJAR EN UNA RAMA, NO en una detached HEAD