CIMATECH S.A.

Metodología de trabajo con GIT

Introducción al trabajo en equipo distribuido con control de versionado mediante GIT

Jorge Morando

11/04/2016

El presente documento explica los conceptos básicos de la herramienta de versionado GIT introduciendo al equipo de trabajo a la metodología de trabajo distribuido.

Versionado

Versión	Detalle	Autor	Fecha
1.0-draft	Versión inicial	Jorge Morando	11/04/2016
1.0-draft	Revisión Formal	Rodolfo Ruiz	12/04/2016
1.1-draft	Resumen rutina diaria & correcciones menores	Jorge Morando	13/04/2016

Tabla de contenido

Introducción

Entonces, ¿qué es GIT en pocas palabras? Es muy importante asimilar esta sección, porque si se entiende lo que es GIT y los fundamentos de cómo funciona, será mucho más fácil usar la herramienta de manera eficaz.
Se intentará marcar las diferencias centrales que GIT posee con respecto a otras herramientas de versionado, ya que el paradigma para el que GIT fue creado se considera más moderno con respecto al de los paradigmas de versionado instaurado desde comienzos del desarrollo tecnológico de la informática.
Se recomienda que a medida que nos vayamos introduciendo en Git, se intente olvidar todo lo que se pueda saber sobre otros VCSs (Sistemas de control de versión – sigla en Inglés), como Subversion y Perforce; hacerlo nos ayudará a evitar confusiones sutiles a la hora de utilizar la herramienta.
GIT almacena y modela la información de forma muy diferente a esos otros sistemas, a pesar de que su interfaz sea bastante similar, comprender esas diferencias evitará confusiones a la hora de usarlo.

Instantáneas, no diferencias

La principal diferencia entre GIT y cualquier otro VCS (Subversion y compañía incluidos) es cómo GIT modela sus datos. Conceptualmente, la mayoría de los demás sistemas almacenan la información como una lista de cambios en los archivos. Estos sistemas (CVS, Subversion, Perforce, Bazaar, etc.) modelan la información que almacenan como un conjunto de archivos y las modificaciones hechas sobre cada uno de ellos a lo largo del tiempo, como ilustra la Figura 1-4.

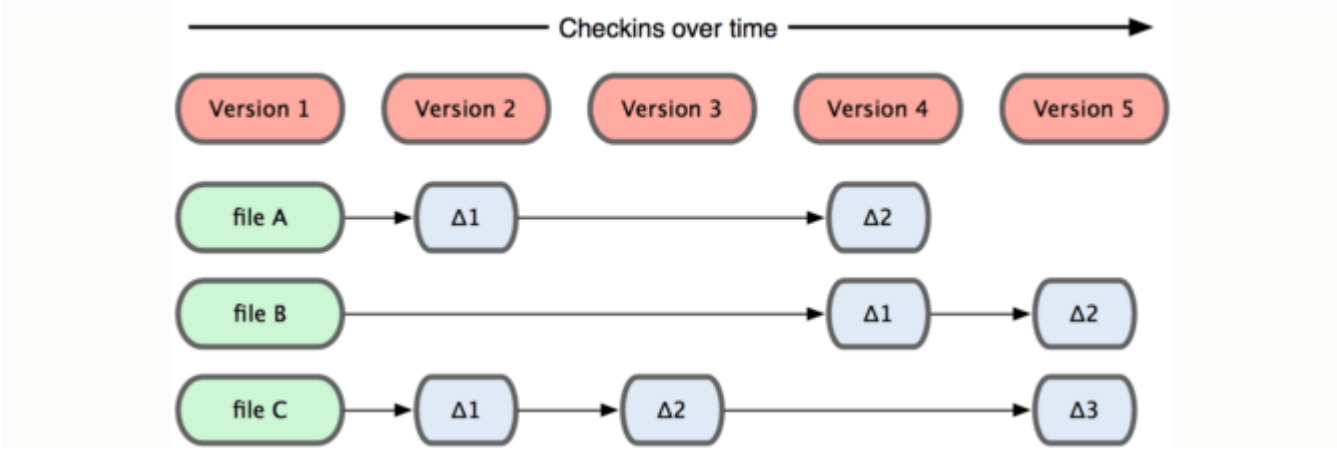


Figura 1-1. Otros sistemas tienden a almacenar los datos como cambios de cada archivo respecto a una versión base.

GIT no modela ni almacena sus datos de este modo. En cambio, GIT modela sus datos más como un conjunto de instantáneas de un mini sistema de archivos. Cada vez que confirmas un cambio, o guardas el estado de tu proyecto en el repositorio, éste básicamente hace una foto del aspecto de todos tus archivos en ese momento, y

guarda una referencia a esa instantánea. Para ser eficiente, si los archivos no se han modificado, GIT no almacena el archivo de nuevo, sólo un enlace al archivo anterior idéntico que ya tiene almacenado.

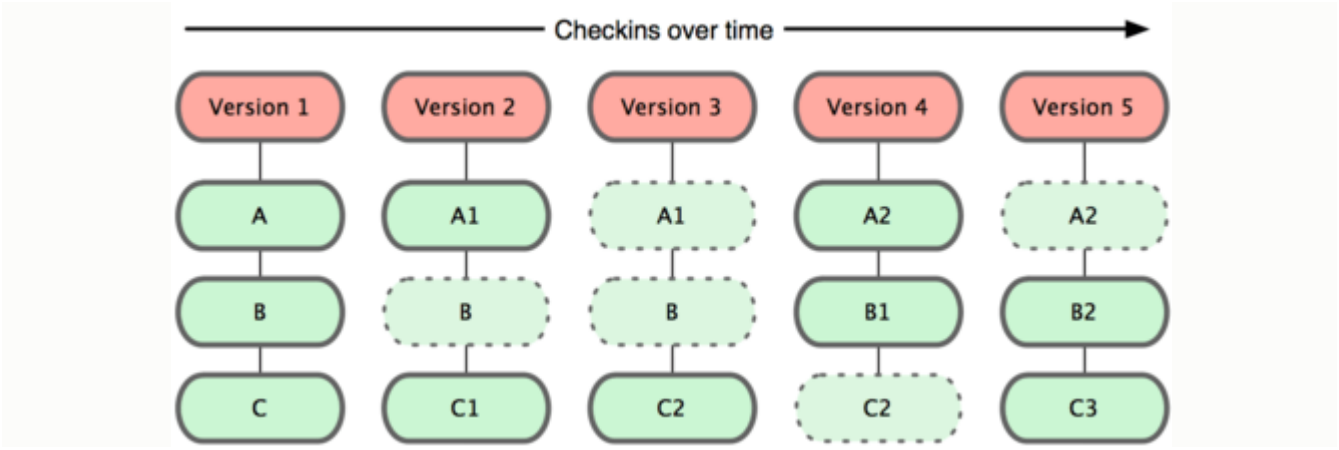


Figura 1-2. GIT almacena la información como instantáneas del proyecto a lo largo del tiempo.

Esta es una distinción importante entre GIT y prácticamente todos los demás VCSs. Hace que GIT reconsidere casi todos los aspectos del control de versiones que muchos de los demás sistemas copiaron de la generación anterior. Esto hace que GIT se parezca más a un mini sistema de archivos con algunas herramientas tremendamente potentes construidas sobre él, que a un VCS.
Exploraremos algunos de los beneficios que obtienes al modelar tus datos de esta manera cuando veamos ramificaciones (branching) en GIT.

Cambios locales

La mayoría de las operaciones en GIT sólo necesitan archivos y recursos locales para operar. Por lo general no se necesita información de ningún otro ordenador de tu red. Si estamos acostumbrados a un CVCS (Sistema de control de versiones centralizado – sigla en Inglés) donde la mayoría de las operaciones tienen esa sobrecarga del retardo de la red. Como se posee toda la historia del proyecto ahí mismo, en nuestro disco local, la mayoría de las operaciones parecen prácticamente inmediatas.

Por ejemplo, para navegar por la historia del proyecto, GIT no se necesita salir al servidor para obtener la historia y visualizarla, simplemente la lee directamente de la base de datos local. Esto significa que se ve la historia del proyecto casi al instante. Si se quiere ver los cambios introducidos en un archivo entre la versión actual y la de hace un mes, GIT puede buscar el archivo hace un mes y hacer un cálculo de diferencias localmente, en lugar de tener que pedirle a un servidor remoto que lo haga, u obtener una versión antigua desde la red y hacerlo de manera local.

Esto también significa que hay muy poco que no se pueda hacer si se está desconectado o sin VPN. Se puede trabajar en cualquier momento, en cualquier lugar.

Metodología de Trabajo

En ésta sección se establecerán normas de organización del trabajo de desarrollo aprovechando capacidades centrales de GIT.

Ramas o líneas de desarrollo

Uno de los conceptos más útiles para los usuarios es el trabajo con ramas, se pueden utilizar diversas configuraciones de ramas. Por ejemplo, podemos tener una rama estable (master), rama desarrollo (develop) y ramas puntuales (desarrolladores independientes y/o líneas de desarrollo como features, bugfixing, etc)

- Tendremos por defecto una rama principal (**master**) que se utilizará para almacenar código estable **(rama release)** y no se trabajara directamente en esta rama
- Para desarrollar se creara una **rama de desarrollo** (**develop**) donde se trabajara habitualmente, cuando el código de esta rama se considere estable, se fusionara con la rama estable. No será posible subir cambios/funcionalidades que no se encuentren finalizadas (**rama estable**).
- Para tareas más puntuales como arreglar errores o probar funcionalidades o llevar el desarrollo rutinario, se crearán ramas de "alto impacto" (**ramas inestables**), si el código de esta rama se considera correcto se fusionará con la rama de desarrollo, sino se puede borrar la rama, "freezar" el desarrollo de dichas funcionalidades/errores o directamente mantener la línea hasta que se cumplan las condiciones para fusionar con la rama de desarrollo.

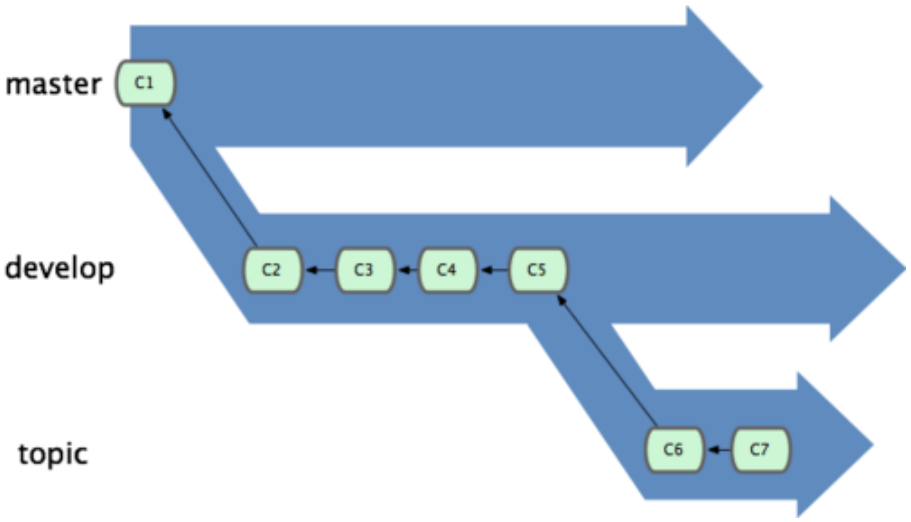


Figura 3-1. GIT Evolución de ramas en comparación a la rama estable.

Con esta configuración de ramas, siempre tendremos código estable mientras seguimos desarrollando en otras ramas, con la posibilidad agregada de mantener el código actualizado en el repositorio remoto sin importar la estabilidad del desarrollo en curso

Filosofía de commit

La principal tarea que se realiza en GIT son los "commit", estos se almacenan en el historial y es conveniente que estén lo mas "limpio" posible (una funcionalidad por commit), GIT no impone ninguna restricción de cuando hacer commit, es el usuario el que decide cuando realizarlo. No conviene realizar commit cada vez que se escribe una línea de código porque resultará en la visualización del historial lleno de cambios que no aportan nada. Una serie de normas que se pueden utilizar son las siguientes:

- Realizar commit que engloben partes completas, como una funcionalidad nueva, arreglo de un error o añadir una nueva función.
- No introducir errores con los commit. Si he añadido una nueva función al código, probar antes que funciona correctamente (Exceptuando el caso de que la rama sea de desarrollo personal, en la que el desarrollador sabe el estado de la misma y nadie más que él accederá a su contenido).
- **Todos los commit deben tener un mensaje claro y breve del contenido de los cambios.**

Escenarios de trabajo en Git

En este apartado se verán diferentes configuraciones de GIT y los tipo de escenarios de trabajos se pueden aplicar, viendo las ventajas y desventajas de cada uno. Un concepto importante a tener en cuenta es que todos los escenarios extienden al más básico, esto se explicará a continuación.

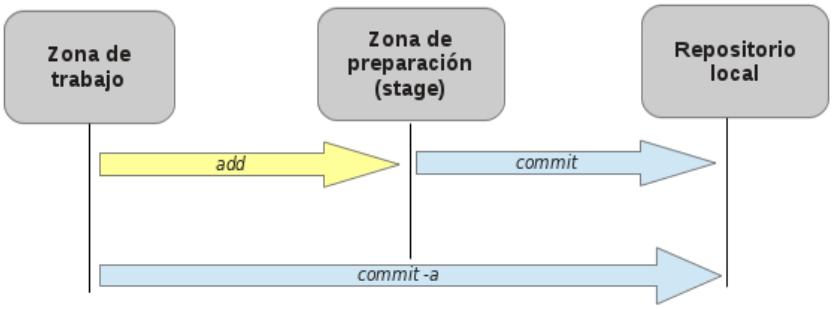
Escenario con repositorio local

El primer escenario es el típico para un usuario que empieza con GIT, tiene un ordenador donde esta desarrolla software y crea un repositorio para ver probar GIT.

Creamos un repositorio vacio y empezamos a desarrollar un proyecto , o creamos el repositorio en el directorio donde almacenamos el código de algún proyecto.

Un repositorio local de GIT está compuesto de tres zonas:

- **Zona de trabajo**: contiene el código del programa, es donde programamos con nuestro editor de código.
- **Repositorio**: esta zona es donde se almacenan los cambios en el historial de GIT.
- **Zona de staging o escenario**: esta zona intermedia almacena los archivos pero todavía nos se encuentran en el historial local de GIT. En esta zona podemos preparar los commit con los cambios más nos interesan.



Ventajas:

- Fácil de crear y utilizar.
- Rápido, todas las operaciones son locales.

Desventajas:

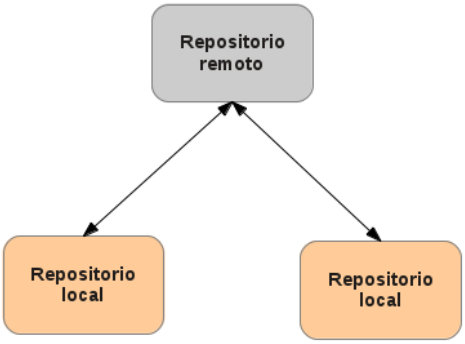
- Disponibilidad, el código esta en un solo repositorio.
- No se puede compartir código.

Esta configuración es ideal para un solo programador, si de forma puntual otro programador va colaborar, hay diversas formas para compartir el código como ramas remotas, bundle o parches. **Pero si estamos trabajando en un equipo que estará proporcionando contribuciones permanente o se pretende que el código esté accesible a más desarrolladores es aconsejable expandir esta configuración con el uso de un repositorio remoto.**

Escenario centralizado (repositorio remoto)

Este escenario es útil para pequeños grupos de desarrolladores, requiere que un recurso cumpla la funcionalidad extra de mantener el repositorio remoto o en su defecto que ambos desarrolladores tengan capacidad de actualizar el repositorio donde se compartirá el código de manera responsable. Para aquellos equipos de desarrollo que vengan de un sistema de control centralizado (SubVersión o CVS) estarán familiarizados con esta configuración.

Cada desarrollador trabajará en su repositorio local y subirá los cambios al repositorio remoto para compartirlos con el resto de desarrolladores.



Cuando se utiliza un repositorio remoto, hay que tener en cuenta que antes de que un desarrollador suba cambios al repositorio remoto debe comprobar si tiene actualizado su repositorio local. Si un desarrollador A sube código al repositorio remoto y después un desarrollador B sube código al repositorio, GIT mostrara un mensaje indicando que el desarrollador B no tiene los cambios introducidos por el desarrollador A. El desarrollador B deberá descargarse los cambios introducidos para tener en su código los cambios del desarrollador A antes de subir sus cambios.

Ventajas:

- Fácil de compartir código, si un desarrollador desea obtener el código solo deberá clonar el repositorio remoto.
- Disponibilidad, tenemos el código en varios repositorios.

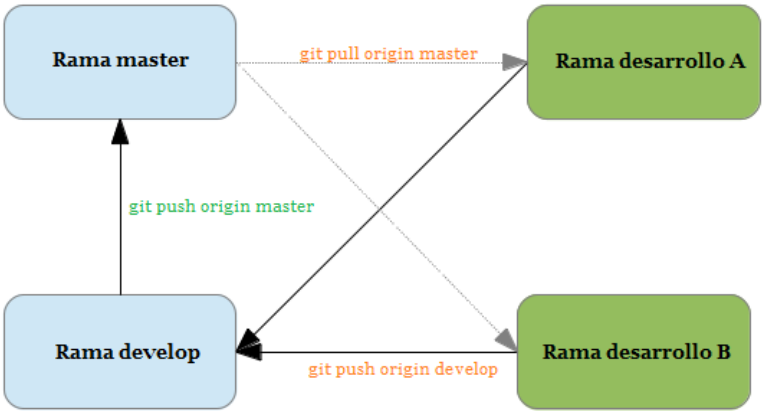
Desventajas:

- No hay control del código que se sube.
- Sincronización de los repositorios.

Escenario integrador (ramas)

Cuando el número de desarrolladores aumenta, el tráfico de código entre los repositorios también lo hace, entonces es conveniente tener un poco de organización. En este escenario aparece la figura del gestor de integración (integrador) que se encarga de recibir los cambios introducido por los desarrolladores, realiza pruebas para comprobar el código de los desarrolladores, si todo esta correcto lo integra con su rama master y lo sube al repositorio remoto, desde donde los desarrolladores obtendrán nuevas versiones del código. El integrador es el único que debería tener acceso de escritura al repositorio remoto o a la rama integradora, el resto de desarrolladores solo debería tener acceso de lectura al repositorio remoto o rama integradora.

Este escenario ya se vuelve un poco más complejo, ya que hablamos de ramas o repositorios integradores. Los repositorios integradores son logrados a través de los “forks” pero no estaremos profundizando sobre este concepto ya que en nuestro caso no aplica, solo estaremos explicando el escenario de la rama integradora (develop o rama de desarrollo).



Ventajas:

- Control de código.
- Seguridad. Todo el código es comprobado por el gestor integrador y los desarrolladores no tienen acceso a escritura al repositorio remoto.

Desventajas:

- Complejidad para el integrador.
- Publicación a la rama master restringida, se debe aplicar alguna capa de acceso para que los desarrolladores no tengan acceso de escritura o en su defecto un alineamiento que impida que los usuarios suban cambios a la rama master.

Rutina de trabajo

A continuación se explicará la rutina de trabajo con git, reuniendo todos los conceptos previamente descritos, aplicados al proyecto OCU.

Se deberán asumir las siguientes características

1. El repositorio se encuentra remotamente localizado en http://www.cima-it.com/git/ocu-repo.git
2. El directorio de trabajo local será "C:\proyectos\" o "/home/juanperez/proyectos" y será referenciado por la etiqueta "**<project-dir>**"

Consideraciones previas para el desarrollador

- Asegurarse de tener configuradas correctamente sus credenciales en GIT. Para esto ejecutará los siguientes comandos en una terminal de Linux o una consola GIT-BASH de Windows:

```
$ git config --global user.name "Juan Perez"
$ git config --global user.email juan.perez@cima-it.com
```

- Localizarnos en el directorio de trabajo **<project-dir>**
- Clonar el repositorio para poder trabajar localmente, para lograr esto deberá ejecutar el comando.

```
$ git clone http://www.cima-it.com/git/ocu-repo.git
```

GIT buscará en la URL especificada descargando todo el contenido, creando directorio "ocu-repo" y finalmente inicializando el repositorio local de trabajo.

- Crear su rama de desarrollo a partir del código en la rama develop

```
$ git checkout develop
$ git checkout –b juanperez
```

El primer comando actualiza todos los archivos con los cambios estables de desarrollo ubicados en la rama develop (esta rama nunca contendrá cambios inestables).
El segundo comando crea una rama independiente de desarrollo inestable específica para el desarrollador a partir de los cambios de la rama develop.

A) Rutina de desarrollo

A1. Cambiar archivos en el repositorio

A2. Incorporar esos cambios en el sistema de control de cambios con el siguiente comando:

```
$ git add [path_archivo_cambiado_1 [path_archivo_cambiado_2]]
```

Cada nombre debe estar separado por un espacio, si estamos en Windows y el path del archivo contiene un espacio, será necesario escapar cada uno de ellos para que el sistema identifique el espacio separador del espacio en el path del archivo. Esto hará que los cambios estén en zona de staging para commitear (También puede utilizarse "git add *" pero esto puede incluir cambios que no se quieren incorporar.)

A3. Commit de los cambios preparados en zona de staging

```
$ git commit –m "Mensaje descriptivo de funcionalidad terminada"
```

Esto ingresa los cambios en nuestro repositorio local. (También se puede utilizar el flag "-a" que reemplaza el comando git add, agregando todos los cambios antes de realizar el commit y así evitando agregar uno por uno los archivos)

A4. Subir los cambios que contiene la rama de desarrollo inestable en el repositorio local al repositorio remoto

```
$ git push origin juanperez
```

Cuando se llega al punto en el que se ha concluido totalmente una funcionalidad y pasa todas las pruebas unitarias y de desarrollo, se deben incorporar esos cambios a la rama de desarrollo estable “develop”. Es responsabilidad de cada desarrollador no subir cambios inestables a dicha rama.

B) Rutina de sincronización de ramas de desarrollo (inestable/estable)

C1. Incorporar los cambios (que pueda tener la rama estable) en nuestra rama inestable de desarrollo

```
$ git pull origin develop
```

Este comando descargará todos los cambios de la rama develop en nuestra rama (si se ha cambiado de rama anteriormente deberemos ubicarnos en nuestra rama ejecutando “git checkout <rama>”) haciendo un merge (Si se encontraron conflictos se deberá continuar con el paso C1 en el flujo “C” de Resolución de Conflictos)

C2. Ubicarnos en la rama local de desarrollo estable “develop”

```
$ git checkout develop
```

C3. Incorporar todos nuestros cambios en la rama local de desarrollo estable “develop”

```
$ git merge juanperez
```

C4. Sincronizar la rama local de desarrollo con la rama remota.

```
$ git push origin develop
```

C5. Ubicarnos nuevamente en la rama local de desarrollo inestable

```
$ git checkout juanperez
```

C) Rutina de resolución de conflictos

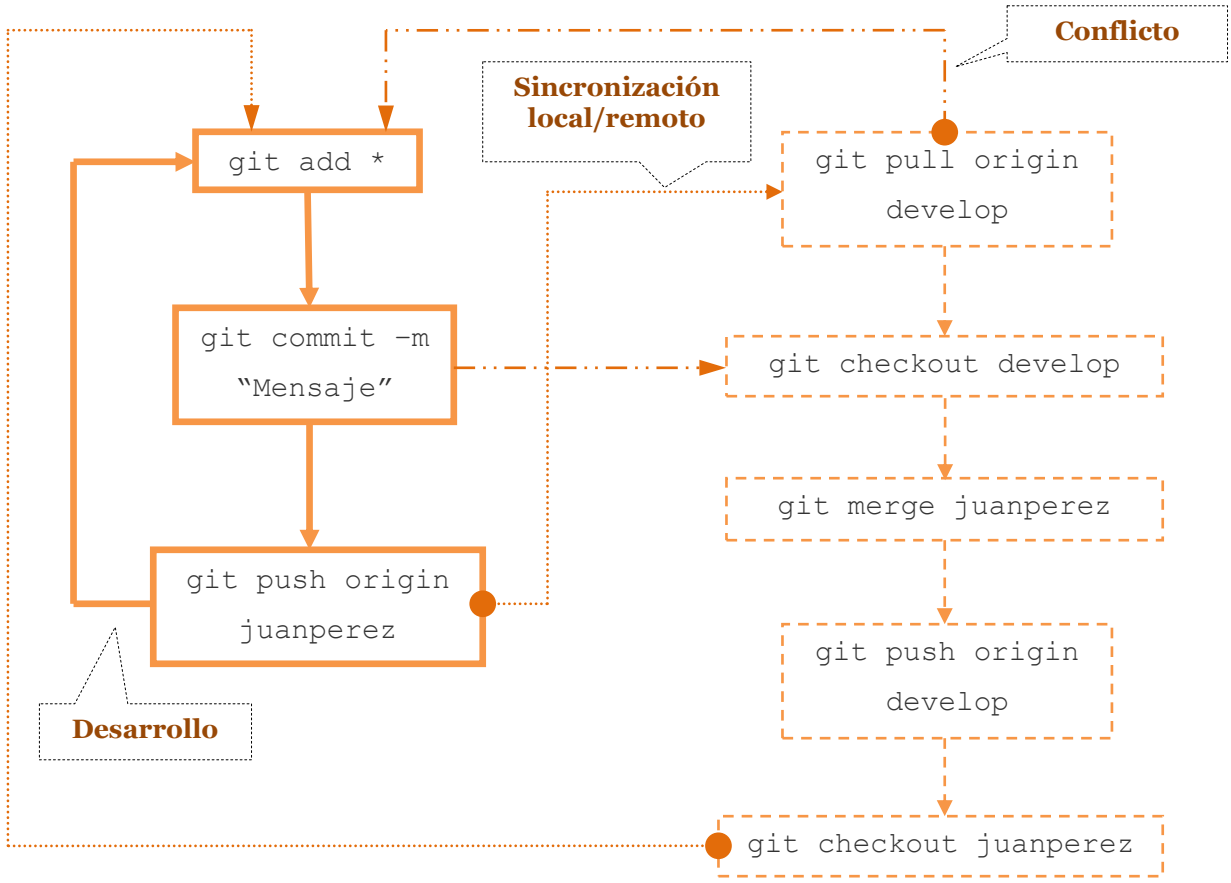
Cuando ejecutamos el paso B1, puede ocurrir que vengan cambios considerados estables que modifican los cambios incorporados localmente, esta situación se denomina “conflicto” y debe resolverse eligiendo qué código es el correcto y deberá subirse de la misma forma en la que se subiría cualquier otro cambio.

C1. Solucionar el conflicto modificando el archivo en cuestión (el nombre del mismo se informa luego de hacer “pull” en caso de encontrarse algún conflicto que no se pueda resolver automáticamente por git)

C2. Agregar los cambios a la etapa de staging ejecutando el flujo determinado en la rutina de desarrollo

C3. Continuar con el paso "C2" de la rutina de sincronización de ramas

Resumen global de Rutinas



Guía básica de funciones y/o comandos

A continuación se detallan los comandos (o funciones a través de programas clientes de GIT) para llevar a cabo las capacidades rutinarias de desarrollo

Descargar repositorio

```
$ git clone /path/to/repository
```

Este comando nos permitirá descargarnos un repositorio ya inicializado desde la su localización y creará una copia prístina del mismo para que podamos trabajar normalmente.
La dirección puede ser local (dentro del mismo filesystem en el que vamos a trabajar) o remota (vía ssh o http).

Gestión de cambios

```
$ git add <filename>
$ git add .
```

Estos comandos nos permiten agregar archives no versionados al repositorio, también permiten agregar cambios realizados a archivos ya versionados al escenario de commit. (Ver figura escenario repositorio local)

```
$ git commit –m “Mensaje simple”
```

Este comando permite incorporar los cambios en el escenario de commits al control final de versionado. En este caso nuestros cambios estarán a aceptados pero en nuestro repositorio local.

```
$ git push origin
```

Este comando permite sincronizar nuestro repositorio local en el repositorio remoto. Asumiendo que no haya cambios en el repositorio remoto que **no** tengamos, entonces subirá todos los cambios commiteados al mismo. En el caso que haya cambios en el repositorio que **no** tengamos localmente será necesario que sincronicemos el repositorio remoto en nuestro local con el siguiente comando.

```
$ git pull origin
```

Este comando permite sincronizar el repositorio remoto en nuestro repositorio local, si hubiese cambios que otros desarrolladores han subido en sus ramas y nosotros queremos poder ver localmente sus cambios por algún motivo, se bajarán y se fusionarán en sus respectivas ramas.

```
$ git pull origin <rama>
$ git push origin <rama>
```

Estos comandos permitirán sincronizar una rama específica entre su instancia remota y nuestra instancia local.
Si un desarrollador ha completado una funcionalidad hará “push” a la rama develop, en caso contrario hará push a su rama de desarrollo asignada.
Si un desarrollador quiere incorporar cambios que se encuentran en la rama “develop” que otro desarrollador ha subido hará “pull” de develop a su rama de desarrollo asignada.