

```
PPOptional.add_argument('-V', dest='verbose', action='count', default=0, help='increase verbosity level')
PPOptional.add_argument('--version', action='version', version='Version ' + str(constant.CURRENT_VERSION), help='laZagne version')

PWrite = argparse.ArgumentParser(add_help=False, formatter_class=lambda prog: argparse.HelpFormatter(prog, max_help_position=constant.MAX_HELP_POSITION))
PWrite._optionals.title = 'output'
PWrite.add_argument('-W', dest='write', action='store_true', help='write a text file on the current directory')

all_subparser = []
for c in category.keys():
    category[c]['parser'] = argparse.ArgumentParser(add_help=False, formatter_class=lambda prog: argparse.HelpFormatter(prog, max_help_position=constant.MAX_HELP_POSITION))
    category[c]['parser']._optionals.title = category[c]['help']

    category[c]['subparser'] = []
    for module in modules[c].keys():
        m = modules[c][module]
        category[c]['parser'].add_argument(m.options['action'], dest=m.options['dest'], help=m.options['help'])

        if m.suboptions and m.name != 'thunderbird':
            tmp = []
            for sub in m.suboptions:
                tmp_subparser = argparse.ArgumentParser(add_help=False, formatter_class=lambda prog: argparse.HelpFormatter(prog, max_help_position=constant.MAX_HELP_POSITION))
                tmp_subparser._optionals.title = sub['help']
                if 'type' in sub:
                    tmp_subparser.add_argument(sub['action'], dest=sub['dest'], help=sub['help'], type=sub['type'])
                else:
                    tmp_subparser.add_argument(sub['action'], dest=sub['dest'], help=sub['help'])
                tmp.append(tmp_subparser)
            all_subparser.append(tmp_subparser)
        category[c]['subparser'] += tmp

parents = [PPOptional] + all_subparser + [PWrite]
dic = {'all': {'parents': parents, 'help': 'Run all modules', 'func': run_all_modules}}
for c in category.keys():
    parser_tab = [PPOptional, category[c]['parser']]
    if 'subparser' in category[c]:
        if category[c]['subparser']:
            parser_tab += category[c]['subparser']
    parser_tab += [PWrite]
    dic[tmp_name] = {'parents': parser_tab, 'help': 'Run the module ' + c, 'func': dic[tmp_name]['func']}
    dic[tmp_name].set_defaults(func=dic[tmp_name]['func'], auditType=d)

subparsers = argparse.ArgumentParser(add_help=False, formatter_class=lambda prog: argparse.HelpFormatter(prog, max_help_position=constant.MAX_HELP_POSITION))
for d in dic.keys():
    subparsers.add_parser(d, parents=dic[d]['parents'], help=dic[d]['help']).set_defaults(func=dic[d]['func'], auditType=d)
```



# MANEJO DE REPOSITORIOS GIT

## PROGRAMACIÓN III

Abdel G. Martínez L.

# ACERCA DEL CONTROL DE VERSIONES

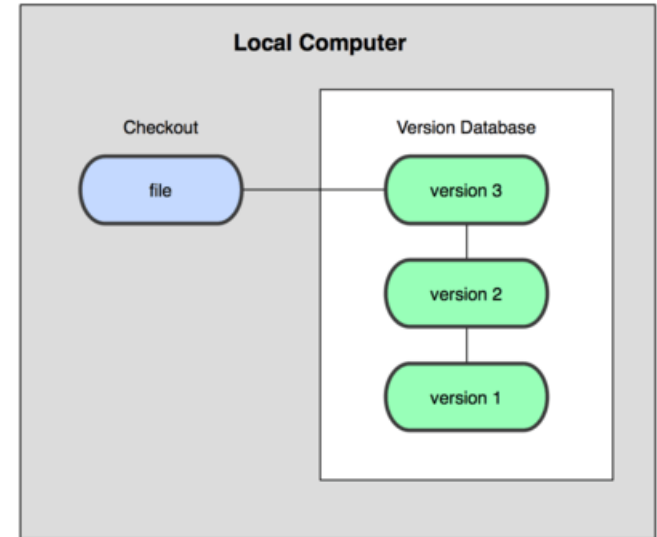
---

- ⦿ Sistema que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo
- ⦿ Es útil para cualquier profesión, no únicamente programadores
- ⦿ Permite revertir archivos específicos a su estado anterior, o bien todo el proyecto entero
- ⦿ Compara cambios a lo largo del tiempo, permitiendo ver quién modificó por última vez un archivo
- ⦿ Su coste de implementación es muy bajo

# CONTROL DE VERSIONES LOCAL

---

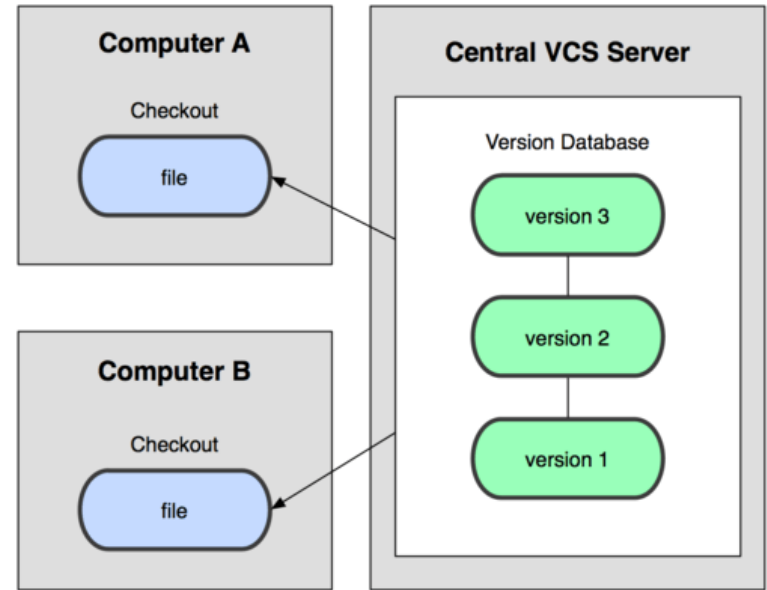
- ◉ El método más primitivo, utilizado por mucha gente, es copiar los archivos a otro directorio
- ◉ Este enfoque es muy común por su simpleza, pero está propenso a múltiples errores: olvidar la ubicación del directorio, sobrescribir archivos no deseados



# CONTROL DE VERSIONES CENTRALIZADO

---

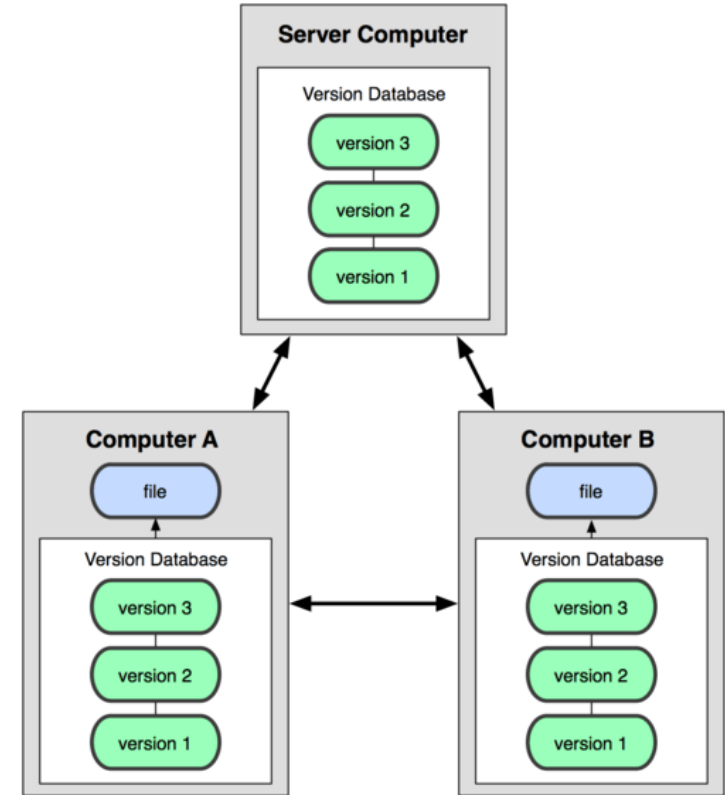
- ◉ Nacen por la necesidad de colaborar en otros sistemas
- ◉ El servidor contiene todos los archivos centralizados y los clientes descargan del servidor
- ◉ Su mayor desventaja es la alta dependencia al servidor
- ◉ Ejemplos: CVS, Subversion.



# CONTROL DE VERSIONES DISTRIBUIDO

---

- ◉ No sólo descargan la última versión de los archivos, replican completamente el repositorio
- ◉ Se elimina la dependencia total al servidor primario
- ◉ Se puede trabajar de forma simultánea en distintos grupos
- ◉ Ejemplos: Git, Mercurial



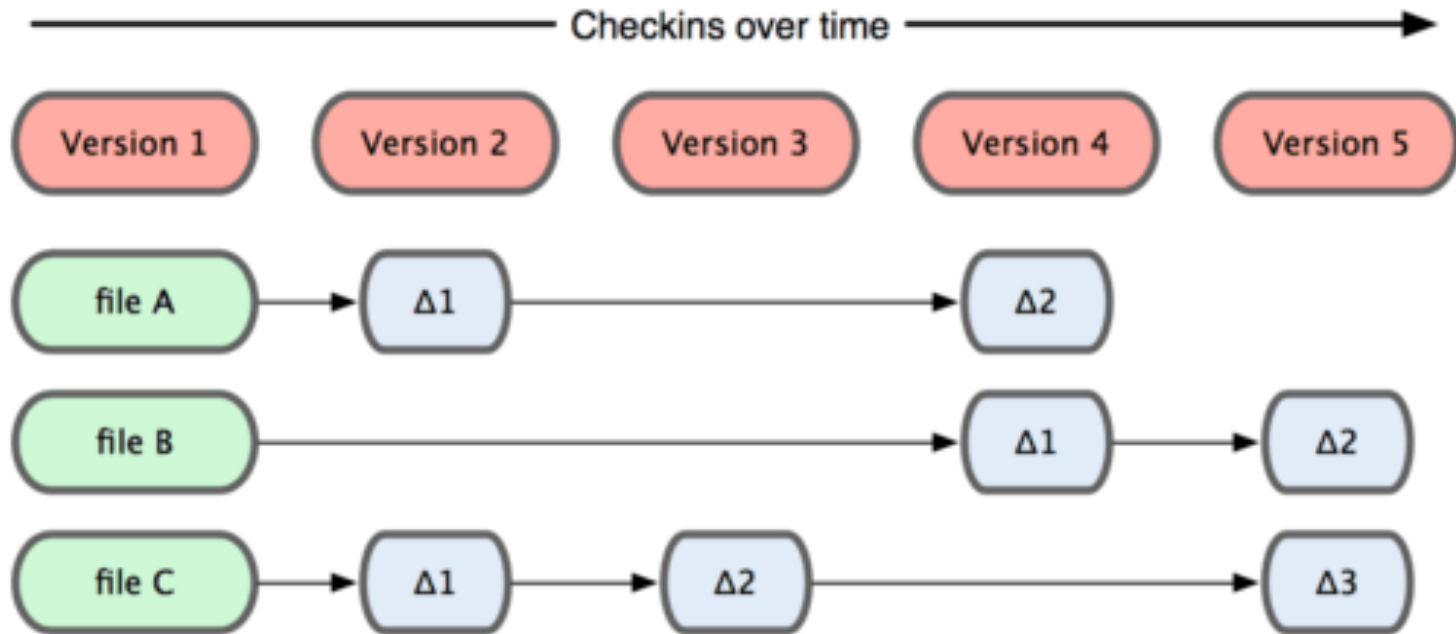
# BREVE HISTORIA DE GIT

---

- ⦿ Comenzó de la mala relación entre la comunidad desarrolladora del núcleo de Linux y la compañía que desarrollaba BitKeeper
- ⦿ Linus Torvalds, impulsó a desarrollar su propia herramienta basada en la experiencia del uso de BitKeeper
- ⦿ Git ha evolucionado y madurado, desde 2005, para ser fácil de usar, ser tremendamente rápido, eficiente con grandes proyectos y tener un sistema de ramificación increíble para el desarrollo no lineal

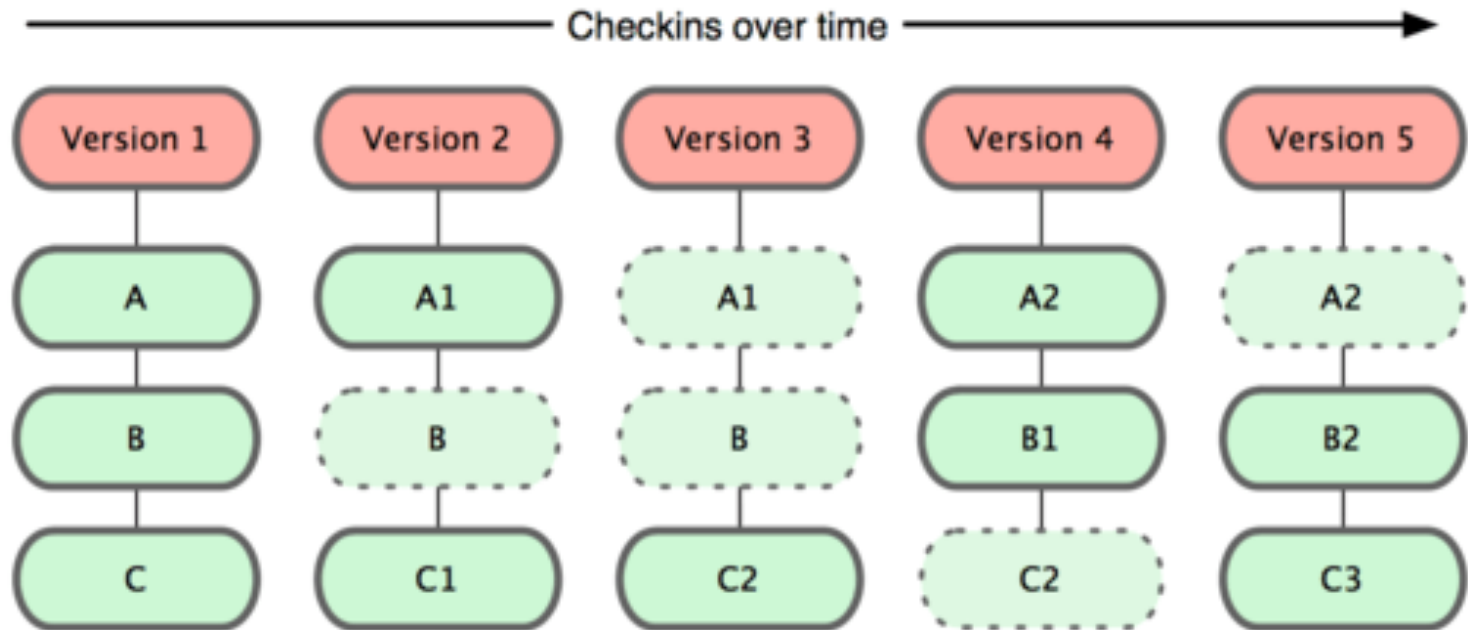
# FUNDAMENTOS DE GIT

---



# FUNDAMENTOS DE GIT

---





# GIT USA OPERACIONES LOCALES

---

- ⦿ La mayoría de las operaciones necesitan archivos y recursos locales
- ⦿ No se necesita información de ningún otro ordenador de la red
- ⦿ Como toda la historia del proyecto es local, parece inmediato
- ⦿ Otros controles de versiones necesitan conexión al servidor

# GIT TIENE INTEGRIDAD

---

- ◉ Verifica todo mediante una suma de comprobación (checksum)
  - ◉ Es imposible cambiar contenidos sin que Git se entere
  - ◉ Esto forma parte de su funcionalidad y su filosofía
  - ◉ El mecanismo utilizado por Git es el hash SHA-1:
    - Cadena de 40 caracteres hexadecimales (0-9 y a-f)
    - Calculado por contenido del archivo o estructura de directorio
- da39a3ee5e6b4b0d3255bfeef95601890afd80709
- ◉ Git guarda todo por el valor hash de su contenido

# GIT ES SEGURO

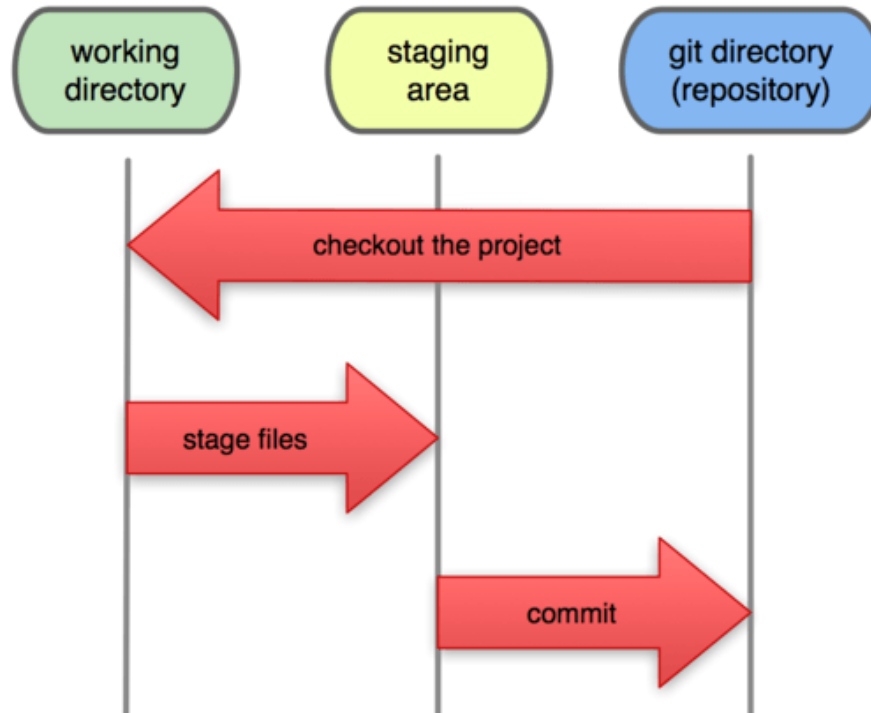
---

- ⦿ Las acciones realizadas sólo añaden información a la base de datos
- ⦿ Es difícil conseguir que el sistema haga algo que no se pueda deshacer, o que borre información

# LOS TRES ESTADOS DE GIT

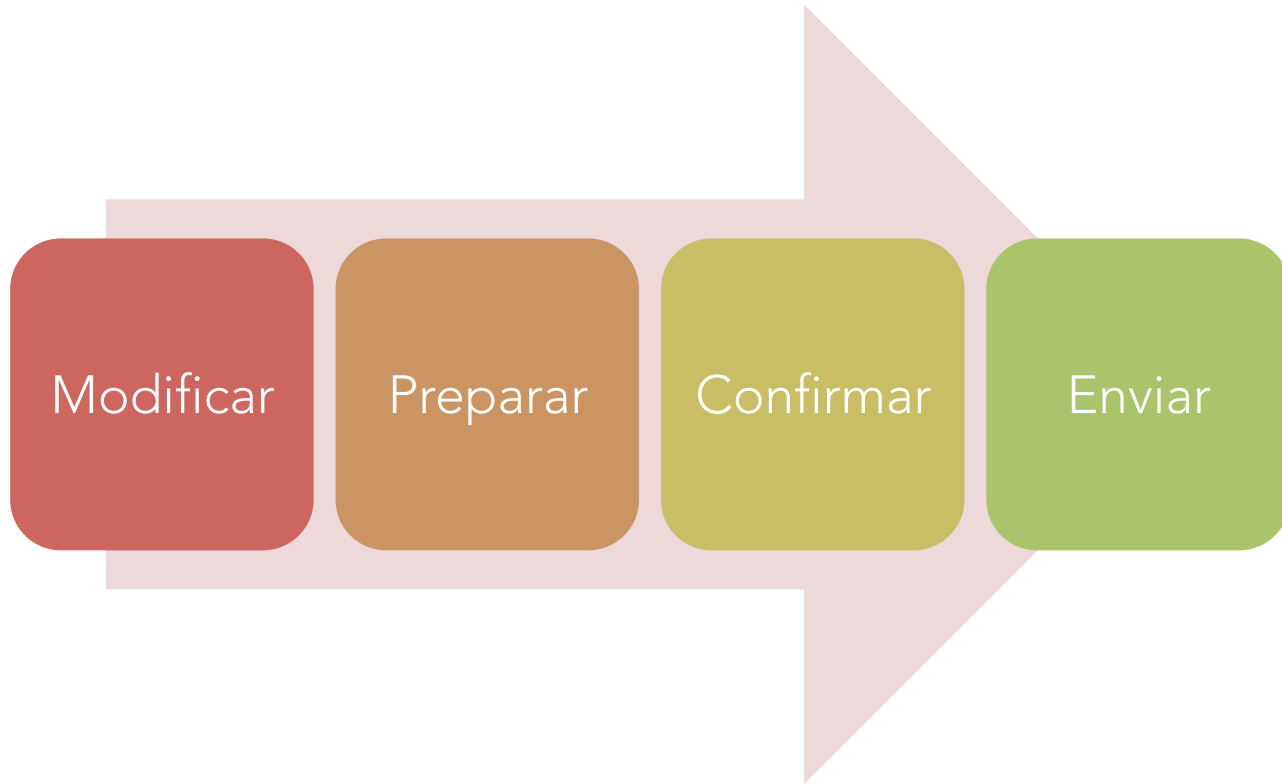
---

## Local Operations



# FLUJO DE TRABAJO DE GIT

---



# INSTALACIÓN

---

## ⦿ *Windows*

- msysGit (<http://msysgit.github.com/>)

## ⦿ *Mac OS X*

- Git Installer (<http://sourceforge.net/projects/git-osx-installer>)
- Usando el comando `brew install git`

## ⦿ *GNU/Linux*

- En Fedora se instala usando `dnf install git-core`
- En Ubuntu se instala usando `apt-get install git`

# COMANDOS BÁSICOS - CREACIÓN

---

- ◉ Clonación de repositorio existente

```
$ git clone https://domain.com/user/repo.git
```

- ◉ Creación de un nuevo repositorio local

```
$ git init
```

# COMANDOS BÁSICOS - CONFIGURACIÓN

---

- ◉ Establecer usuario, correo, editor de texto y diferenciador

```
$ git config --global user.name "Carlos Martínez"
```

```
$ git config --global user.email me@example.com
```

```
$ git config --global core.editor emacs
```

```
$ git config --global merge.tool vimdiff
```

- ◉ Comprobar la configuración actual

```
$ git config --list
```



# COMANDOS BÁSICOS – CAMBIOS LOCALES

---

- ◉ Ver los archivos cambiados en el directorio de trabajo

```
$ git status
```

- ◉ Ver los cambios en los archivos monitorizados

```
$ git diff
```

- ◉ Agregar todos los cambios actuales en el siguiente *commit*

```
$ git add .
```

# COMANDOS BÁSICOS – CAMBIOS LOCALES

---

- ◉ Agregar algunos cambios en <archivo> en el siguiente *commit*  
`$ git add <archivo>`
- ◉ Entregar todos los cambios locales en los archivos monitorizados  
`$ git commit -a`
- ◉ Cambiar el último *commit*  
`$ git commit --amend`

# COMANDOS BÁSICOS – HISTÓRICO

---

- Mostrar todos los *commits*, iniciando con el más nuevo

```
$ git log
```

- Mostrar los cambios en el tiempo de un archivo específico

```
$ git log -p <archivo>
```

- Ver quién cambió qué y cuándo en un archivo específico

```
$ git blame <archivo>
```

# COMANDOS BÁSICOS – RAMAS Y ETIQUETAS

---

- ◉ Lista todas las ramas existentes

```
$ git branch -av
```

- ◉ Cambiar rama en el HEAD

```
$ git checkout <rama>
```

- ◉ Crear una nueva rama basada en el contenido en el HEAD

```
$ git branch <nueva-rama>
```

# COMANDOS BÁSICOS – RAMAS Y ETIQUETAS

---

- ◉ Eliminar una rama local

```
$ git branch -d <rama>
```

- ◉ Clonar una rama remota

```
$ git checkout --track -b <nombre> <rep>/<bra>
```

- ◉ Marcar el commit actual con una etiqueta

```
$ git tag <etiqueta>
```

# COMANDOS BÁSICOS – ACTUALIZAR

---

- ◉ Listar todos los sitios remotos configurados

```
$ git remote -v
```

- ◉ Muestra información de un repositorio remoto

```
$ git remote show <remoto>
```

- ◉ Agrega un nuevo repositorio remoto

```
$ git remote add <nombre-corto> <url>
```

# COMANDOS BÁSICOS – ACTUALIZAR

---

- ◉ Descarga todos los cambios de <remoto>, sin integrar en HEAD

```
$ git pull <remoto> <rama>
```

- ◉ Publica los cambios locales en un repositorio remoto

```
$ git push <remoto> <rama>
```

- ◉ Elimina una rama de un repositorio remoto

```
$ git branch -dr <remoto>/<rama>
```

# COMANDOS BÁSICOS – ACTUALIZAR

---

- ⦿ Publica todas las etiquetas

```
$ git push --tags
```



# COMANDOS BÁSICOS – MEZCLAR

---

- ◉ Mezcla la <rama> en el HEAD actual

```
$ git merge <rama>
```

- ◉ Rebase de HEAD actual en <rama>

```
$ git rebase <rama>
```

- ◉ Abortar un rebase

```
$ git rebase --abort
```

# COMANDOS BÁSICOS – MEZCLAR

---

- ◉ Continúa un rebase luego de resolver conflictos

```
$ git rebase --continue
```

- ◉ Resuelve el conflicto entre archivos

```
$ git mergetool
```

```
$ git add <archivo>
```

```
$ git rm <archivo>
```

# COMANDOS BÁSICOS – DESHACER

---

- ◉ Descartar todos los cambios locales en el directorio de trabajo

```
$ git reset --hard HEAD
```

- ◉ Descartar los cambios locales en un archivo específico

```
$ git checkout HEAD <file>
```

- ◉ Revertir un *commit*

```
$ git revert <commit>
```

# COMANDOS BÁSICOS – DESHACER

---

⦿ Resetear el puntero del HEAD a un *commit* anterior

- *Descartando todos los cambios*

```
$ git reset --hard <commit>
```

- *Preservando los cambios como cambios no enviados*

```
$ git reset <commit>
```

- *Preservando cambios locales sin commit*

```
$ git reset --keep <commit>
```

# COMANDOS BÁSICOS – AYUDA GENERAL

---

- ◉ Obtener ayuda desde la línea de comandos

```
$ git help <comando>
```

# MEJORES PRÁCTICAS

---

- ⦿ Hacer *commit* de cambios relacionados
- ⦿ Hacer *commit* más seguidos
- ⦿ No hacer *commit* de trabajo medio-terminado
- ⦿ Probar el código antes de hacer *commit*
- ⦿ Escribir buen mensaje en los *commit*
- ⦿ Un control de versiones no es sinónimo de sistema de respaldos
- ⦿ Utilizar ramas
- ⦿ Seguir un flujo de trabajo

# OTRAS REFERENCIAS

---

- ◉ **Git – la guía sencilla**

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

- ◉ **Try Git**

<https://try.github.io/>