Evolución y Gestión de la Configuración

Grado en Ingeniería Informática – Ingeniería del Software

Práctica 2

Gestión del código fuente

Cuaderno de ejercicios

Usuario A

Trabajarás emparejado/a con el usuario B.

Curso 2026/2027



Antes de empezar

Este cuaderno tiene **dos partes**, y se hacen en sitios distintos.

> **Parte 1 – en clase, por parejas.** Cinco ejercicios que necesitan a otra persona: compartir un repositorio, provocar un conflicto de verdad y traerse el trabajo del otro. Alrededor de hora y cuarto.
>
> **Parte 2 – en casa, individual.** Ocho ejercicios más cortos que se hacen a solas: deshacer, recuperar, hooks, parches, etiquetas y búsqueda binaria. Alrededor de hora y media.

En la parte por parejas, uno de los dos es el usuario A y el otro el usuario B, y cada uno tiene su propio cuaderno: el que tienes delante es el del **usuario A**. Los ejercicios no son simétricos, así que no vale con mirar por encima del hombro.

Todo se hace sobre **un solo fork** del repositorio `uvlhub_practicas` que creasteis en la Práctica 0: el del usuario A, que es el que los dos clonáis en el ejercicio 1 y sobre el que sigue trabajando cada uno, también en la parte individual. No hace falta que conozcas el código del proyecto: los ficheros que vas a crear y romper son de usar y tirar. Lo que se practica aquí es Git, no la aplicación.

Cómo se lee este cuaderno

> **0.0)** Un paso numerado. El primer número es el ejercicio y el segundo el paso dentro de él. Lo que tienes que escribir en el terminal aparece así:
>
> ```
> git status
> ```
>
> Y los nombres de ficheros, ramas y comandos sueltos, así: `git push`, `feature/practicandogit_a`.

Además te vas a encontrar cuatro tipos de aviso:

¿Por qué esto importa?
Por qué el ejercicio que viene tiene sentido y qué problema real resuelve.

Truco
Un atajo, una variante del comando o algo que te va a ahorrar tiempo.

¡Ojo!
Un error clásico. Si te saltas esto, el ejercicio se te va a torcer.

Zona de peligro
Comandos que destruyen trabajo. Se pueden usar, pero hay que saber lo que se está haciendo.

Y, solo en la primera parte, la barrera de sincronización:

> **ALTO.** Espera a que el usuario B llegue hasta aquí antes de seguir. Si te adelantas, romperás el ejercicio para los dos.

Cuando veas ese cuadro, **párate**. Los ejercicios en pareja están coreografiados: si sigues antes de que tu compañero/a llegue, el estado del repositorio deja de ser el que el enunciado espera y a partir de ahí nada cuadra. Es, con diferencia, la causa número uno de que una pareja se atasque.

Los mensajes de commit

A lo largo del cuaderno los mensajes siguen un formato fijo:

```
tipo: descripcion breve en minuscula

feat: add testing file
fix: fix conflicts
chore: step one
```

Es la convención *Conventional Commits*, muy extendida y la que usa el repositorio del proyecto. Los tipos habituales son `feat` para una funcionalidad nueva, `fix` para un arreglo, `docs`, `test`, `refactor`, `style` y `chore` para el resto. En el ejercicio 8 vas a escribir un hook que lo compruebe automáticamente antes de dejarte confirmar.

> **Los mensajes van en inglés**
> Es la convención del proyecto y se mantiene también aquí.

PARTE 1

Trabajo en pareja

En clase · Alrededor de 1 h 15 min

Ejercicio 1 · en pareja
Emparejamiento y acceso

¿Por qué esto importa?

Durante toda esta parte los dos vais a trabajar sobre **un único repositorio**: el fork del usuario A. Es la situación normal de un equipo, y es la que hace que aparezcan conflictos de verdad en lugar de simulaciones.

1.1) Da acceso a tu compañero/a. En tu repositorio `uvlhub_practicas`, ve a `Settings` → `Collaborators` → `Add people`, escribe su usuario de GitHub e invítale.

1.2) Activa las issues, que las vais a necesitar enseguida: `Settings` → `General` → `Features` → marca `Issues`.

1.3) Clona tu propio repositorio en el equipo del laboratorio.

1.4) Comprueba que el clonado ha usado SSH y no HTTPS.

Truco

¿Clonaste con HTTPS por error? No hace falta volver a empezar:

```
git remote set-url origin git@github.com:<usuario_de_A>/uvlhub_practicas.git
```

ALTO. Espera a que el usuario B llegue hasta aquí antes de seguir. Si te adelantas, romperás el ejercicio para los dos.

Ejercicio 2 · en pareja
Ramas, issues y tu primer commit

¿Por qué esto importa?

Una rama es una línea de desarrollo independiente: una versión paralela del proyecto en la que puedes trabajar sin tocar lo que hay en `main`. Es donde vais a vivir el resto de la práctica.

2.1) Crea una issue titulada `Testing file (A)` con la descripción que quieras, y asigna a tu compañero/a B en `Assignees`. Apunta su número.

> **ALTO.** Espera a que el usuario B llegue hasta aquí antes de seguir. Si te adelantas, romperás el ejercicio para los dos.

2.2) Crea una rama local llamada `feature/practicandogit_a` y sitúate en ella.

2.3) Comprueba en qué rama estás.

2.4) Sube la rama al repositorio remoto dejando puesta la referencia de seguimiento.

> **Truco**
> El flag `-u` (de *upstream*) conecta la rama local con la remota. A partir de ahí, `git push` y `git pull` a secas ya saben con quién hablar. Sin él, tendrías que escribir el remoto y la rama cada vez.

2.5) Crea en la raíz del proyecto un fichero `practicandogit_a.txt` que contenga una sola línea con tu UVUS.

2.6) Comprueba si git está siguiendo ese fichero.

2.7) Añade *ese fichero en concreto* al área de preparación.

> **¡Ojo!**
> Acostúmbrate a añadir ficheros por su nombre en lugar de tirar de `git add .`. El punto se lleva por delante todo lo que haya en el directorio, incluidos ficheros temporales y cambios a medias que no querías subir.

2.8) Vuelve a mirar el estado y confirma que ahora sí está preparado.

2.9) Crea un commit con el mensaje `feat: add testing file`, pero no lo subas todavía.

2.10) Nos hemos dejado algo: el commit debería cerrar la issue que creó tu compañero/a, `Testing file (B)`. Corrige el mensaje del commit que acabas de hacer para que la referencie, sin crear un commit nuevo.

> **Zona de peligro**
> `git commit --amend` reescribe historia. Aquí es inofensivo porque el commit todavía no ha salido de tu máquina. Si ya lo hubieras subido, el remoto y tú tendríais historias distintas y solo podríais reconciliarlas con un `push --force`, que le puede borrar trabajo a quien haya partido de tu commit anterior.

2.11) Ahora sí, sube el commit a tu rama remota.

> **ALTO.** Espera a que el usuario B llegue hasta aquí antes de seguir. Si te adelantas, romperás el ejercicio para los dos.

Ejercicio 3 · en pareja
Fusionar ramas: avance rápido y commit de fusión

¿Por qué esto importa?

Fusionar dos ramas no siempre produce lo mismo. Si la rama de destino no se ha movido desde que saliste de ella, git se limita a adelantar el puntero: es un *avance rápido*, o *fast-forward*, y no crea ningún commit nuevo. Si las dos ramas han avanzado por su cuenta, no queda otra que crear un **commit de fusión** con dos padres.
Los dos vais a fusionar vuestra rama en `main`, uno detrás de otro. Y como el segundo se encontrará un `main` que ya se ha movido, cada uno verá un caso distinto. Esa es la gracia del ejercicio.

3.1) Sitúate en `main` y asegúrate de que está al día.

3.2) Antes de fusionar, mira dónde está cada rama.

3.3) Fusiona tu rama en `main` y fíjate en lo que dice git.

3.4) Sube `main`.

ALTO. Espera a que el usuario B llegue hasta aquí antes de seguir. Si te adelantas, romperás el ejercicio para los dos.

3.5) Mirad juntos el resultado. Los dos deberíais ver exactamente lo mismo.

Truco

`git log --graph --oneline --all` es la forma más rápida de entender en qué estado está un repositorio. Merece la pena aprendérselo de memoria.

¡Ojo!

Que la fusión sea un avance rápido o un commit de fusión no depende de lo que tú pidas, sino de si la rama de destino se ha movido. Con `--no-ff` puedes forzar que siempre se cree el commit de fusión, algo que muchos equipos hacen para que en el historial se vea dónde empezó y acabó cada rama.

Ejercicio 4 · en pareja
Conflictos

¿Por qué esto importa?

Un conflicto aparece cuando dos personas cambian **la misma línea del mismo fichero** en ramas distintas. Git sabe fusionar cambios que no se pisan, pero cuando se pisan no puede adivinar cuál vale, y te pasa la pelota.
Vamos a provocarlo a propósito, que es la única forma de perderle el miedo.

4.1) En `main`, crea un fichero `equipo.txt` con una única línea, y súbelo.

> **ALTO.** Espera a que el usuario B llegue hasta aquí antes de seguir. Si te adelantas, romperás el ejercicio para los dos.

4.2) Vuelve a tu rama y tráete el fichero desde `main`.

4.3) Edita `equipo.txt` y sustituye esa única línea por tu nombre. Los dos vais a cambiar *la misma línea*, cada uno con su nombre.

> **ALTO.** Espera a que el usuario B llegue hasta aquí antes de seguir. Si te adelantas, romperás el ejercicio para los dos.

4.4) Ahora trae la rama de tu compañero/a y fusiónala en la tuya.

4.5) Git no ha podido fusionar solo. Comprueba qué fichero está en conflicto.

Un fichero en conflicto se ve así:

```
<<<<<<< HEAD
Responsable: <tu nombre>
=======
Responsable: <nombre de tu companero>
>>>>>>> origin/feature/practicandogit_b
```

Entre `«««<` y `=======` está tu versión; entre `=======` y `»»»>`, la de la otra rama.

> **Resolver no es elegir**
>
> La tentación es quedarse con una de las dos versiones y borrar la otra. Aquí sería un error: sois dos y los dos sois responsables. La resolución correcta es **quedarse con los dos nombres** y borrar los marcadores.
> Resolver un conflicto es decidir cuál es el estado correcto del fichero, no elegir bando.

4.6) Edita `equipo.txt` para que queden los dos nombres y desaparezcan las siete marcas `<`, `=` y `>`. Después cierra la fusión.

> **Truco**
>
> `git diff` durante un conflicto enseña solo las partes en disputa. Y si te lías del todo, `git merge --abort` deja el repositorio exactamente como estaba antes de intentar la fusión. No es rendirse: es empezar de nuevo con la cabeza fría.

> **ALTO.** Espera a que el usuario B llegue hasta aquí antes de seguir. Si te adelantas, romperás el ejercicio para los dos.

Ejercicio 5 · en pareja
Cherry-picking

¿Por qué esto importa?

Escenario habitual: estás en tu rama, con trabajo a medias, y encuentras un fallo crítico. Lo arreglas, pero tu rama no está lista para fusionarse. Lo único que hay que llevarse a las demás ramas es *ese* commit.
Eso es `cherry-pick`: coger un commit concreto de una rama y aplicarlo en otra, sin arrastrar el resto del historial.

5.1) Asegúrate de estar en tu rama `feature/practicandogit_a`.

5.2) Crea un fichero `fix_bug_a.txt` en la raíz, con el contenido que quieras, y confírmalo con el mensaje `fix: fix important bug (developer A)`. Súbelo.

ALTO. Espera a que el usuario B llegue hasta aquí antes de seguir. Si te adelantas, romperás el ejercicio para los dos.

5.3) Actualiza las referencias remotas y localiza el hash del commit de tu compañero/a, el que dice `fix: fix important bug (developer B)`.

5.4) Trae ese commit, y solo ese, a tu rama.

5.5) Comprueba que ahora tienes el fichero de tu compañero/a y sube los cambios.

Truco

`git cherry-pick` admite varios hashes seguidos, y también rangos con `A..B`. Si aparece un conflicto, se resuelve igual que en una fusión y se sigue con `git cherry-pick --continue`; para tirar la toalla, `git cherry-pick --abort`.

¡Ojo!

El commit que traes tiene un hash **distinto** del original: es un commit nuevo con el mismo contenido. Por eso, si más adelante fusionas de verdad las dos ramas, git puede volver a ver ese cambio y quejarse.

Fin de la parte en pareja. Lo que queda del cuaderno se hace a solas y en casa. Ya no hay barreras de sincronización: a partir de aquí vas a tu ritmo.

PARTE 2

Trabajo individual

En casa Alrededor de 1 h 30 min

Ejercicio 6 · individual
La pila de stash

¿Por qué esto importa?

Estás a medias de algo, sin nada que merezca todavía un commit, y te piden mirar otra cosa con urgencia. No puedes cambiar de rama arrastrando el destrozo, ni quieres tirarlo. El *stash* es el cajón donde se guarda temporalmente lo que tienes sin confirmar.

6.1) Sitúate en tu rama `feature/practicandogit_a`.

6.2) Crea en la raíz un fichero `estoestasinterminar.txt` con lo que quieras.

6.3) Guarda el trabajo en la pila de stash.

¡Ojo!

El `git add` de arriba no sobra. `git stash` solo guarda ficheros que git ya esté siguiendo, y este acaba de nacer. Sin añadirlo antes, el stash lo ignora y te lo dejas en el directorio de trabajo. La alternativa es `git stash -u`, que sí incluye los no rastreados.

6.4) Comprueba que ahora puedes saltar a `main` sin problemas, y vuelve.

6.5) Mira qué hay guardado en la pila.

6.6) Recupera el trabajo sin sacarlo de la pila.

6.7) Ya no lo necesitas guardado. Vacía la entrada de la pila.

6.8) El fichero ha vuelto, y fíjate en que vuelve *preparado*: `git status` lo lista bajo `Changes to be committed`. Era de usar y tirar, así que sácalo del área de preparación y bórralo, para que no se cuele en el siguiente commit.

Truco

`git stash pop` hace `apply` y `drop` de una vez. Usa `apply` cuando quieras aplicar lo mismo en varias ramas, y `pop` en el caso normal.

Ejercicio 7 · individual
Deshacer: revert, reset y el paracaídas

¿Por qué esto importa?

Deshacer en git tiene dos filosofías, y confundirlas cuesta caro:

- `git revert` crea un commit **nuevo** que hace lo contrario del anterior. La historia no se toca, solo crece. Es lo seguro en una rama compartida.
- `git reset` mueve la rama hacia atrás y **borra** los commits que quedan por delante. Reescribe la historia. Es para lo que todavía no ha salido de tu máquina.

Y como red de seguridad está el *reflog*, que se acuerda de todo aunque `git log` ya no lo enseñe.

Zona de peligro

Todo este ejercicio se hace en una rama nueva que **no vas a subir nunca**. Se destruyen commits a propósito, y no conviene hacerlo sobre una rama que comparte otra persona.

7.1) Crea la rama de pruebas `experimento/deshacer`.

7.2) Haz tres commits pequeños, uno detrás de otro.

7.3) Apunta en un papel el hash del **tercer** commit. Te va a hacer falta al final.

Revert: deshacer sin borrar

7.4) Deshaz el segundo commit, el de `paso2.txt`, sin tocar los otros dos. Localízalo primero por su mensaje.

7.5) Comprueba qué ha pasado: en el log y en el disco.

Reset: los tres niveles

7.6) Retrocede un commit dejando los cambios preparados en el *stage*, y mira el estado.

7.7) Retrocede otro commit más, esta vez sacando también los cambios del *stage*.

7.8) Y ahora el definitivo. Primero vuelve a preparar `paso3.txt`, que el `--mixed` anterior dejó sin seguimiento, porque `--hard` solo destruye lo que git rastrea. Después retrocede otro commit borrando también los cambios del directorio de trabajo.

Los tres resets, en una frase

`--soft` mueve la rama. `--mixed` mueve la rama y vacía el *stage*. `--hard` mueve la rama, vacía el *stage* y borra tus ficheros. Los dos primeros nunca pierden trabajo; el tercero sí.

Reflog: el paracaídas

¿Por qué esto importa?

Acabas de borrar commits con `reset --hard` y `git log` ya no los enseña. Parecen perdidos, pero casi nunca lo están: git apunta *todos* los sitios por los que ha pasado `HEAD` en un registro aparte, el *reflog*. Es la respuesta a la pregunta que sale en clase todos los años: "he hecho `reset --hard` y lo he perdido todo".

7.9) Comprueba que `git log` ya no sabe nada de aquellos commits.

7.10) Ahora mira el reflog, que sí se acuerda.

7.11) Recupera el estado que tenías, usando el hash del tercer commit que apuntaste antes. Si lo has perdido, sácalo del reflog.

Truco

El reflog también entiende posiciones relativas: `HEAD@{1}` es donde estaba `HEAD` justo antes del último movimiento, así que `git reset --hard HEAD@{1}` deshace el último reset sin buscar ningún hash. Y funciona igual después de un `rebase` o un `merge` que salió mal.

¡Ojo!

El reflog es **local y temporal**: vive en tu `.git`, no se sube al remoto, y git limpia las entradas viejas al cabo de unas semanas. Si clonas de nuevo el repositorio, empiezas con el reflog vacío. No es una copia de seguridad, es una red bajo el trapecio.

7.12) Vuelve a tu rama de trabajo.

Ejercicio 8 · individual
Hooks: que git te pare a tiempo

¿Por qué esto importa?

Los *hooks* son scripts que git ejecuta solo en determinados momentos: antes de crear un commit, después de recibir un push, al fusionar… Los que se ejecutan en tu máquina se llaman *client-side* (`pre-commit`, `commit-msg`, `post-commit`) y los que corren en el servidor, *server-side* (`pre-receive`, `post-receive`).
Sirven para que un error se detecte en tu equipo y no tres pasos más tarde, cuando ya lo has subido y ha molestado a alguien.

¡Ojo!

Los hooks viven en `.git/hooks/`, y `.git` no se versiona. Es decir: no se suben ni se descargan con el repositorio. Cada persona del equipo tiene que instalárselos, y por eso los proyectos serios los guardan en una carpeta versionada y los enlazan.

Hook 1: validar el mensaje de commit

8.1) Crea el fichero `.git/hooks/commit-msg` con este contenido.

```
#!/bin/bash
# Valida la primera linea del mensaje contra Conventional Commits.
# git pasa como $1 la ruta del fichero temporal con el mensaje.
mensaje=$(head -n 1 "$1")
patron='^(build|chore|ci|docs|feat|fix|perf|refactor|revert|style|test)(\([a-z0-9_-]+\))
    ?!?: .+'

if ! [[ "$mensaje" =~ $patron ]]; then
    echo "ERROR: el mensaje no sigue Conventional Commits."
    echo "  Has escrito: $mensaje"
    echo "  Formato:     tipo(ambito opcional): descripcion"
    echo "  Tipos:       build chore ci docs feat fix perf refactor revert style test"
    exit 1
fi
```

8.2) Dale permisos de ejecución. Sin esto el hook no sirve para nada.

⚠ El error que se lleva media clase por delante

Si el fichero no es ejecutable, git lo ignora: suelta un aviso fácil de pasar por alto, `hint: The '.git/hooks/commit-msg' hook was ignored because it's not set as executable`, y **el commit pasa** como si el hook no existiera. Si tu hook parece no funcionar, lo primero que hay que mirar es `ls -l .git/hooks/`.

8.3) Compruébalo con un mensaje que no cumple.

8.4) Ahora con un mensaje correcto.

Hook 2: impedir que suba lo que no debe

¿Por qué esto importa?

Un clásico: te dejas una marca de trabajo pendiente, o una contraseña de pruebas, dentro de un fichero y lo confirmas sin darte cuenta. Un hook `pre-commit` puede revisar lo que está a punto de entrar y negarse.

8.5) Crea `.git/hooks/pre-commit` con este contenido y hazlo ejecutable.

```
#!/bin/bash
# Rechaza el commit si algun fichero preparado contiene la marca NO-COMMIT.
# --cached mira lo que esta en el area de preparacion, no el disco: es lo que
# de verdad va a entrar en el commit.
if git diff --cached | grep -q "NO-COMMIT"; then
    echo "ERROR: hay una marca NO-COMMIT en los cambios preparados."
    git diff --cached --name-only
    exit 1
fi
```

8.6) Pruébalo: crea un fichero con esa marca e intenta confirmarlo.

8.7) Quita la marca y vuelve a intentarlo.

Truco

Un hook se salta con `git commit --no-verify`. Sirve para emergencias reales; usarlo por costumbre es tirar a la basura la razón de tener el hook.

Ejercicio 9 · individual
Commits quirúrgicos con git add -p

¿Por qué esto importa?

Lo normal es tocar varias cosas antes de acordarse de commitear, y acabar metiéndolo todo junto con `git add .`. El resultado es un commit que hace tres cosas a la vez: imposible de revisar, imposible de revertir por partes.
`git add -p` convierte la preparación en un diálogo: git te enseña los cambios por bloques y tú decides cuáles entran.

Cada bloque se llama *hunk*: un grupo de líneas contiguas que git ha detectado como un cambio. Para cada uno pregunta `Stage this hunk [y,n,q,a,d,s,e,?]?`:

`y`	sí, prepara este bloque
`n`	no, déjalo sin preparar
`s`	divide el bloque en trozos más pequeños
`e`	ábrelo en el editor para elegir línea a línea
`q`	sal sin preparar el resto
`?`	ayuda

9.1) Crea un fichero de configuración y confírmalo, para tener una base.

9.2) Haz **dos cambios sin relación entre sí** en ese fichero: cambia `LOG_LEVEL` a `DEBUG` y `TIMEOUT` a `9000`. El fichero debe quedar así:

```
LOG_LEVEL = DEBUG
PORT = 8080
TIMEOUT = 9000
```

9.3) Solo quieres confirmar, de momento, el cambio de `LOG_LEVEL`. Entra en modo interactivo.

9.4) Git te presenta las dos modificaciones en un único bloque, porque solo las separa una línea y comparten contexto. Pulsa `s` para dividirlo, `y` en el trozo de `LOG_LEVEL` y `n` en el de `TIMEOUT`.

9.5) Comprueba que solo una de las dos líneas está preparada, y confírmala.

9.6) Ahora confirma el cambio que queda.

Cuando git no deja dividir

A veces `s` no puede separar más, porque las dos cosas están en la misma línea. Ahí entra `e`, que abre el bloque en el editor: borra el `+` de las líneas que no quieras preparar y guarda. Cuidado con no tocar las cabeceras del parche (las que empiezan por `@@`) o git contestará `corrupt patch` y se quedará sin hacer nada.

Ejercicio 10 · individual
Parches: cambios que viajan solos

¿Por qué esto importa?

Casi todo se integra fusionando ramas, pero no siempre se puede. Un parche es un fichero de texto que encapsula un conjunto de cambios y se puede mandar por correo, copiar a una máquina sin red o aplicar sobre un repositorio que ni siquiera comparte historia con el tuyo. Así se contribuyó al kernel de Linux durante décadas, y así se sigue haciendo hoy. Hay dos familias:

- `git diff` y `git apply`, para cambios sueltos del directorio de trabajo. Rápido y sin metadatos.
- `git format-patch` y `git am`, para commits completos. Llevan autor, fecha y mensaje, y al aplicarlos se recrea el commit tal cual.

10.1) Asegúrate de que no tienes nada sin confirmar y crea una rama `fix/patch_source` donde preparar el arreglo.

10.2) Crea `cambios_urgentes.txt` con una línea y confírmalo.

10.3) Vuelve a tu rama de trabajo y trae ese commit con una fusión, para que las dos ramas partan del mismo sitio.

Parche de cambios sueltos: diff y apply

10.4) Vuelve a `fix/patch_source` y simula el arreglo, sin confirmarlo.

10.5) Exporta a un fichero `bugfix.patch` la diferencia entre tu directorio de trabajo y el último commit.

10.6) Ábrelo y reconoce sus partes: la cabecera `diff --git`, las líneas de contexto y las que empiezan por `+`.

10.7) Descarta los cambios locales, que ya están a salvo dentro del parche, y vuelve a tu rama.

¡Ojo!

`git restore <fichero>` es la forma moderna de tirar cambios no confirmados. La antigua, `git checkout -- <fichero>`, sigue funcionando; fíjate en que son **dos guiones** seguidos, no uno largo.

10.8) Aplica el parche en tu rama. Antes, mira qué promete tocar.

10.9) Confirma el contenido del fichero, pero **no** el parche: el `.patch` es el medio de transporte, no parte del proyecto.

Truco

`git apply --check parche.patch` prueba si el parche entraría limpio sin tocar nada. Es lo primero que se hace con un parche de origen desconocido.

Parche de un commit entero: format-patch y am

10.10) Vuelve a `fix/patch_source`, crea `archivo_nuevo.txt` y confírmalo.

10.11) Exporta ese último commit a un parche.

10.12) Vuelve a tu rama y aplícalo con `git am`.

Truco

`git format-patch main..mirama` exporta de golpe todos los commits de una rama, numerados. Y si un parche no encaja del todo, `git am --3way` intenta una fusión a tres bandas en lugar de rendirse.

Ejercicio 11 · individual
Rebase interactivo: limpiar antes de enseñar

¿Por qué esto importa?

Tu rama tiene ahora un historial realista: commits pequeños, alguno con mensaje mejorable, algún "arreglo del arreglo". Está bien mientras trabajas, pero es ruido para quien tenga que revisarlo.
`git rebase -i` te deja reescribir esa serie de commits antes de enseñarla: juntarlos, renombrarlos, reordenarlos o tirarlos.

11.1) Crea una rama de pruebas y haz tres commits que pidan a gritos ser uno solo.

11.2) Abre el editor de rebase para los tres últimos commits.

Git abre una lista como esta, del más antiguo al más reciente:

```
pick a1b2c3d feat: add cleanup file
pick e4f5g6h fix: typo
pick i7j8k9l fix: another typo
```

Las órdenes que más se usan son:

`pick`	déjalo como está
`reword`	mismo cambio, otro mensaje
`squash`	fúndelo con el de arriba y combina los mensajes
`fixup`	fúndelo con el de arriba y tira su mensaje
`drop`	elimínalo

11.3) Deja el primero en `pick`, cambia los otros dos a `fixup`, guarda y cierra. Después mira el historial.

11.4) Cambia ahora el mensaje de ese commit usando `reword`.

Zona de peligro

El rebase **reescribe** los commits: todos cambian de hash. Hacerlo sobre una rama que ya has compartido obliga a un `push --force` y le rompe la historia a quien haya partido de ella.

La regla, sencilla: rebase solo sobre lo que todavía es tuyo. Sobre lo compartido, merge.

11.5) Vuelve a tu rama de trabajo.

Ejercicio 12 · individual
Etiquetas: ponerle nombre a una versión

¿Por qué esto importa?

Una etiqueta, o *tag*, es un nombre fijo sobre un commit concreto. A diferencia de una rama, no se mueve: `v1.0.0` apunta para siempre al mismo sitio. Es como se marcan las versiones que se publican, y lo que muchos sistemas de despliegue usan para saber qué hay que publicar.

12.1) Crea una etiqueta anotada, `v0.1.0-a`, sobre el último commit de tu rama. Lleva tu letra porque compartes el remoto con tu compañero/a y una etiqueta no puede existir dos veces en el mismo repositorio.

¡Ojo!

Hay dos clases de etiqueta. La *ligera* (`git tag v0.1.0-a`) es un puntero y poco más. La *anotada* (`-a`) es un objeto completo, con autor, fecha y mensaje, y se puede firmar. Para marcar versiones se usa siempre la anotada.

12.2) Compruébalo: lista las etiquetas y mira qué hay dentro de la tuya.

12.3) Súbela. Ojo, que `git push` a secas no se lleva las etiquetas.

12.4) Pregunta a git dónde estás respecto a la última etiqueta.

Truco

Borrar una etiqueta lleva dos pasos, uno local y otro remoto:

```
git tag -d v0.1.0-a
git push origin --delete v0.1.0-a
```

Ejercicio 13 · individual
git bisect: encontrar la aguja

¿Por qué esto importa?

Algo funcionaba la semana pasada y hoy no. Entre medias hay cuarenta commits. Revisarlos uno a uno es media mañana; `git bisect` lo resuelve en seis pasos, porque hace una búsqueda binaria: prueba el commit de en medio, descarta la mitad del rango y repite.
Y lo mejor: puede hacerlo solo. Le das el comando que decide si un commit está bien o mal, y `git bisect run` recorre el historial sin ti.

13.1) Crea una rama de pruebas y un fichero con un valor correcto.

13.2) Comprueba el criterio que vas a usar y **apunta el hash de este commit**: es tu último punto bueno conocido.

13.3) Ahora genera diez commits. Uno de ellos, elegido al azar, estropea el valor. Pega el bloque entero y no mires demasiado la salida: la gracia está en no saber cuál es.

13.4) Confirma que ahora el criterio falla.

13.5) Arranca la búsqueda: marca dónde estás como malo y el commit que apuntaste como bueno.

13.6) Deja que lo haga solo: dale el criterio como comando.

¡Ojo!

`git bisect run` interpreta el código de salida del comando: 0 significa bueno y cualquier otro (salvo 125) significa malo. Por eso encaja con cualquier cosa que devuelva un código sensato: un `grep`, un script, o la batería de pruebas del proyecto. Un 125 quiere decir "no puedo evaluar este commit", y git lo salta.

13.7) Sal de la búsqueda y vuelve a donde estabas.

Truco

En un proyecto de verdad el comando sería el que ejecuta las pruebas, por ejemplo `git bisect run pytest -q`. Cuanto más rápida sea la comprobación, más barata sale la búsqueda.

Recoger la mesa

Has dejado por el camino unas cuantas ramas de usar y tirar y algún fichero de pruebas. Antes de terminar, déjalo presentable:

```
git checkout feature/practicandogit_a
git branch -D experimento/deshacer experimento/rebase experimento/bisect fix/patch_source
rm -f bugfix.patch 0001-*.patch
```

Truco

Prueba primero con `-d`: se negará con `not fully merged`, porque las cuatro ramas tienen commits que no están en ningún otro sitio (también `fix/patch_source`: `git am` creó un commit nuevo, con otro hash). Es una red de seguridad. Como aquí ese trabajo sobra, `-D` las borra igualmente. Y si te pasas de frenada, ya sabes dónde mirar: `git reflog`.

Lo que te llevas

De la parte en pareja

- ☑ **Flujo completo** — árbol de trabajo, área de preparación, rama local, referencia remota y repositorio remoto, y qué comando mueve qué entre ellos.
- ☑ **Ramas** — crearlas, publicarlas y conectarlas con su rama remota.
- ☑ **Fusiones** — la diferencia entre un avance rápido y un commit de fusión, y de qué depende que salga una u otra.
- ☑ **Conflictos** — leerlos sin miedo y resolverlos decidiendo cuál es el estado correcto del fichero, no eligiendo bando.
- ☑ **Cherry-pick** — llevarte un commit suelto sin arrastrar una rama entera.

De la parte individual

- ☑ **Stash** — aparcar lo que tienes a medias sin ensuciar el historial.
- ☑ **Revert y reset** — las dos formas de deshacer, y cuándo usar cada una: una crece hacia delante, la otra reescribe.
- ☑ **Reflog** — el paracaídas. Después de un `reset --hard` desafortunado, casi nada está realmente perdido.
- ☑ **Hooks** — que git compruebe en tu máquina lo que si no descubrirías tarde.
- ☑ **git add -p** — separar cambios que no tienen nada que ver en commits distintos y revisables.
- ☑ **Parches** — mover trabajo entre repositorios que no se hablan.
- ☑ **Rebase interactivo** — presentar un historial limpio.
- ☑ **Etiquetas** — marcar versiones sobre un commit concreto.
- ☑ **Bisect** — encontrar en minutos el commit que rompió algo.

Y yo, ¿qué puedo hacer en el proyecto?

Nada de esto se queda en el cuaderno. En el proyecto en equipo os toca decidir, y escribirlo en algún sitio donde todos lo lean:

- **Vuestra política de commits.** Qué formato exigís y con qué detalle. Repartid el hook `commit-msg` entre todo el equipo el primer día: es la diferencia entre acordar una convención y cumplirla.
- **Vuestra política de ramas.** Quién crea ramas, cómo se llaman, cuánto viven y si se integran con merge o con rebase.
- **Vuestra política de issues.** Cuándo se abre una, cómo se enlaza con el commit que la cierra, cómo se etiqueta.
- **Vuestra política de versiones.** Cuándo se pone una etiqueta y qué significa.

¡Diseña tu propia gestión del código!

“La vida es como un repositorio de Git: nunca sabes qué conflicto te va a tocar.”
— Forrest Dump