Controllo di versione e Git

Cos’è un Controllo di Versione (VCS)?

Il controllo di versione è un sistema che registra, nel tempo, i cambiamenti ad un file o ad una serie di file, così da poter richiamare una specifica versione in un secondo momento.

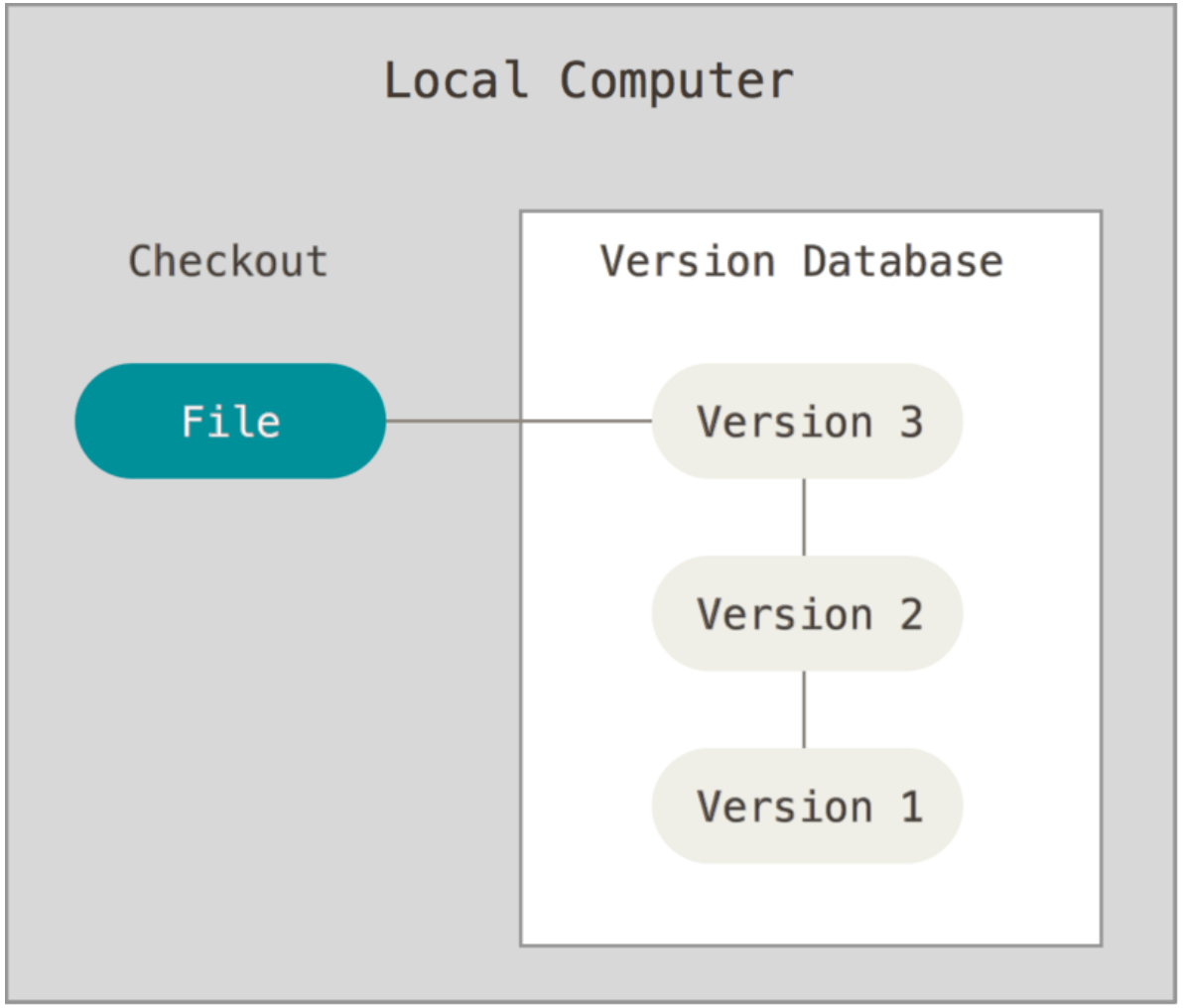
Esistono due entità:

1. Il database contenente tutte le versioni
2. La versione «di lavoro»

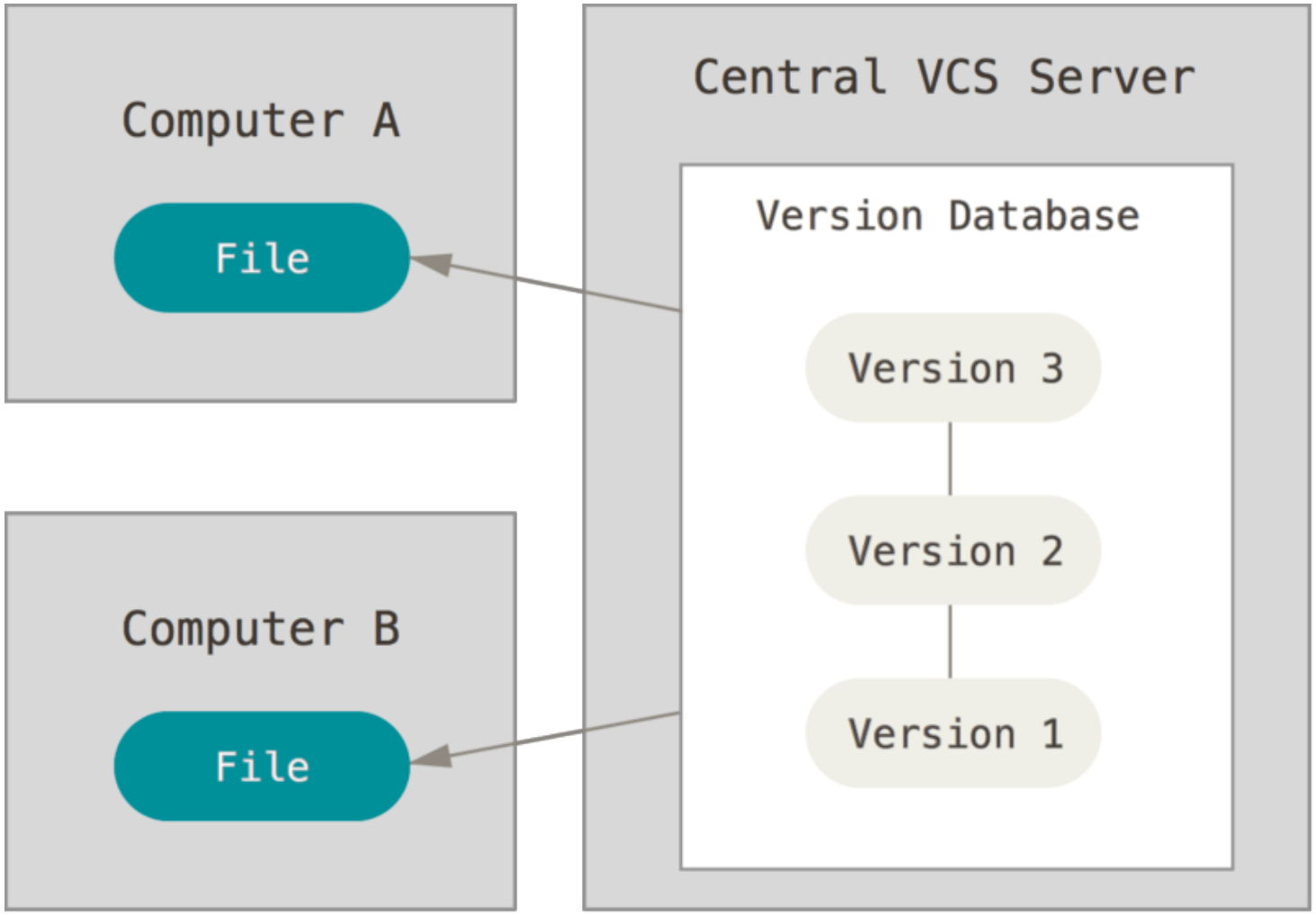
Perché usare un VCS?

Un VCS ti permette di ripristinare i file ad una versione precedente, ripristinare l'intero progetto a uno stato precedente, revisionare le modifiche fatte nel tempo, vedere chi ha cambiato qualcosa che può aver causato un problema, chi ha introdotto un problema e quando, e molto altro ancora. Usare un VCS, in generale, significa anche che se fai un pasticcio o perdi qualche file, puoi facilmente recuperare la situazione. E ottieni tutto questo con poca fatica.

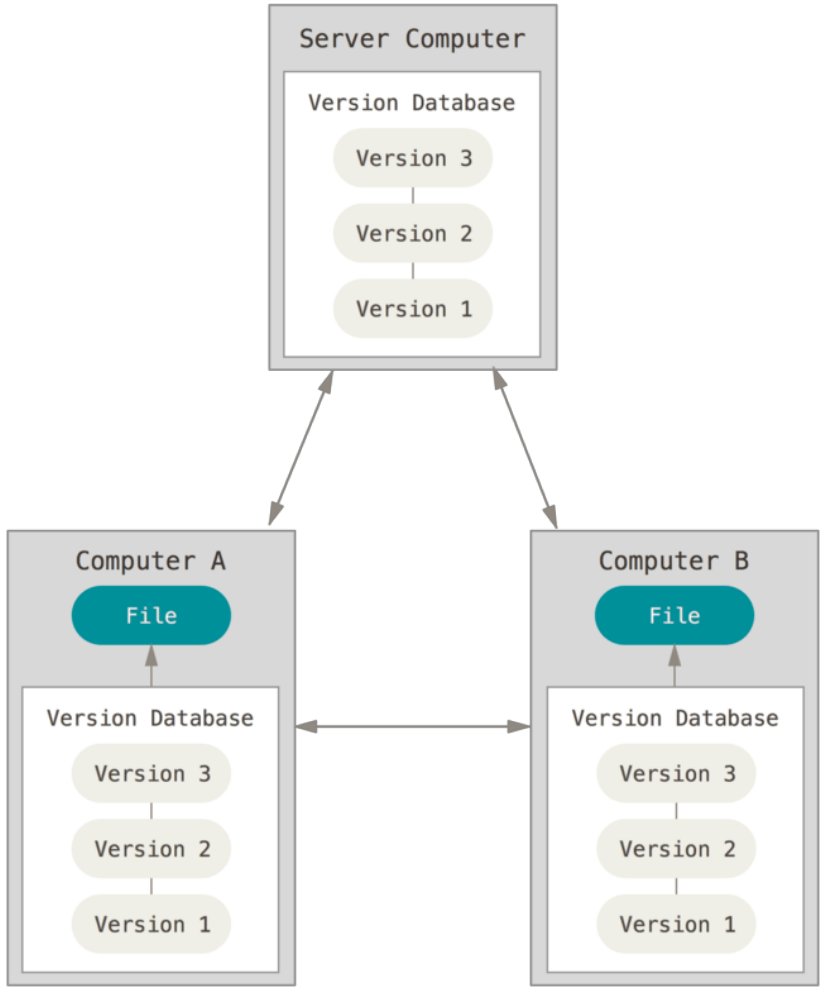
VCS Locale


Local Computer
Checkout
Version Database
File
Version 3
Version 2
Version 1

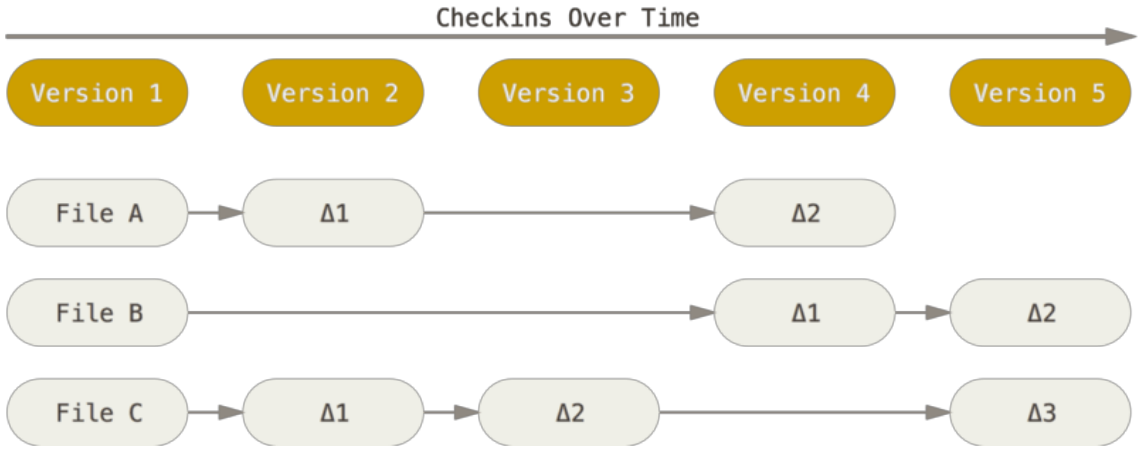
VCS Centralizzato



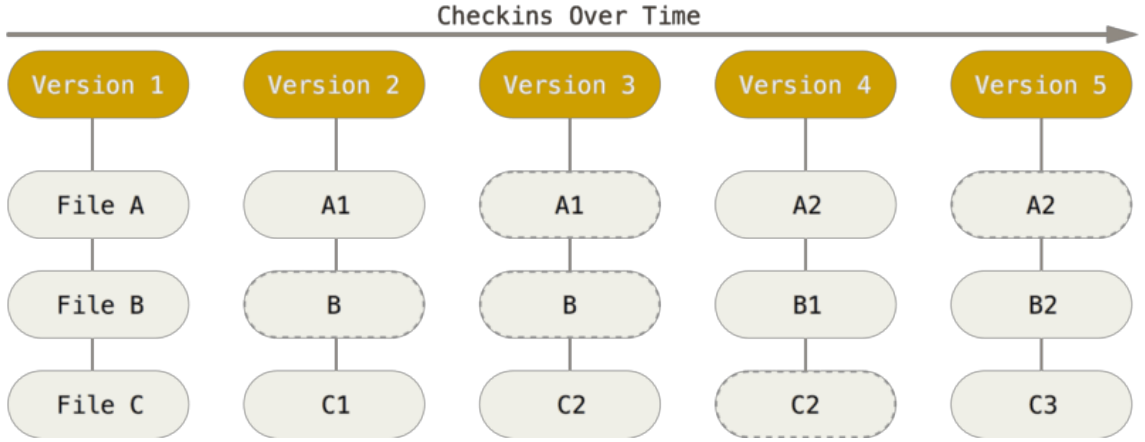
VCS Distribuito



Version database – Differenze vs istantanee



differenze



istantanee

git-scm.com/

Git

Git è un VCS:

- Open source, distribuito, con gestione delle istantanee
- Progettato per essere molto performante
- Quasi tutte le operazioni sono locali e offline
- Gestisce l’integrità dei file tramite checksum (sha-1)
- E’ quasi esclusivamente additivo
- Utilizzabile da riga di comando
- Utilizzabile tramite client grafici (anche integrati in IDE)

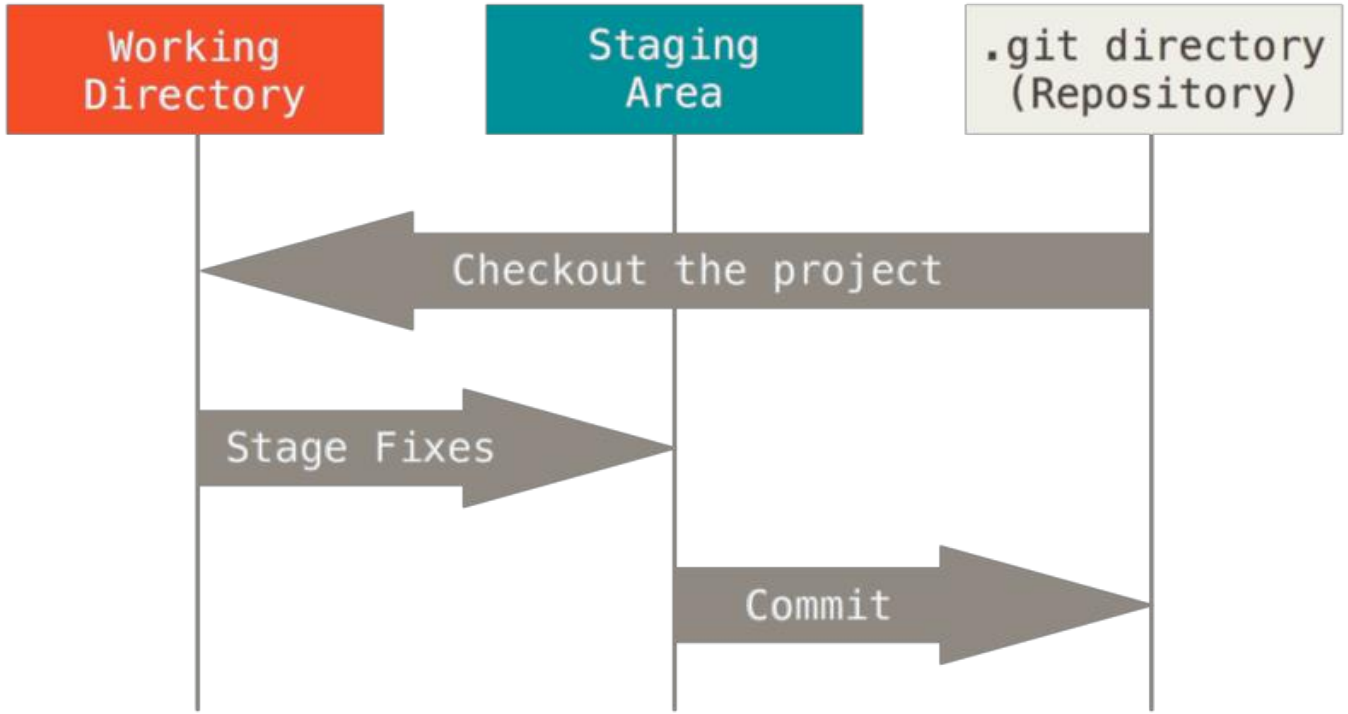
I tre stati dei file(*)

Un file controllato da git può essere in tre stati:

- **Committed**: Il file è archiviato nel database di git
- **Modified**: Il file è stato modificato ma non è stato archiviato nel database
- **Staged (indexed):** Il file è modificato ed è stato selezionato per essere archiviato nel database git al prossimo «commit»

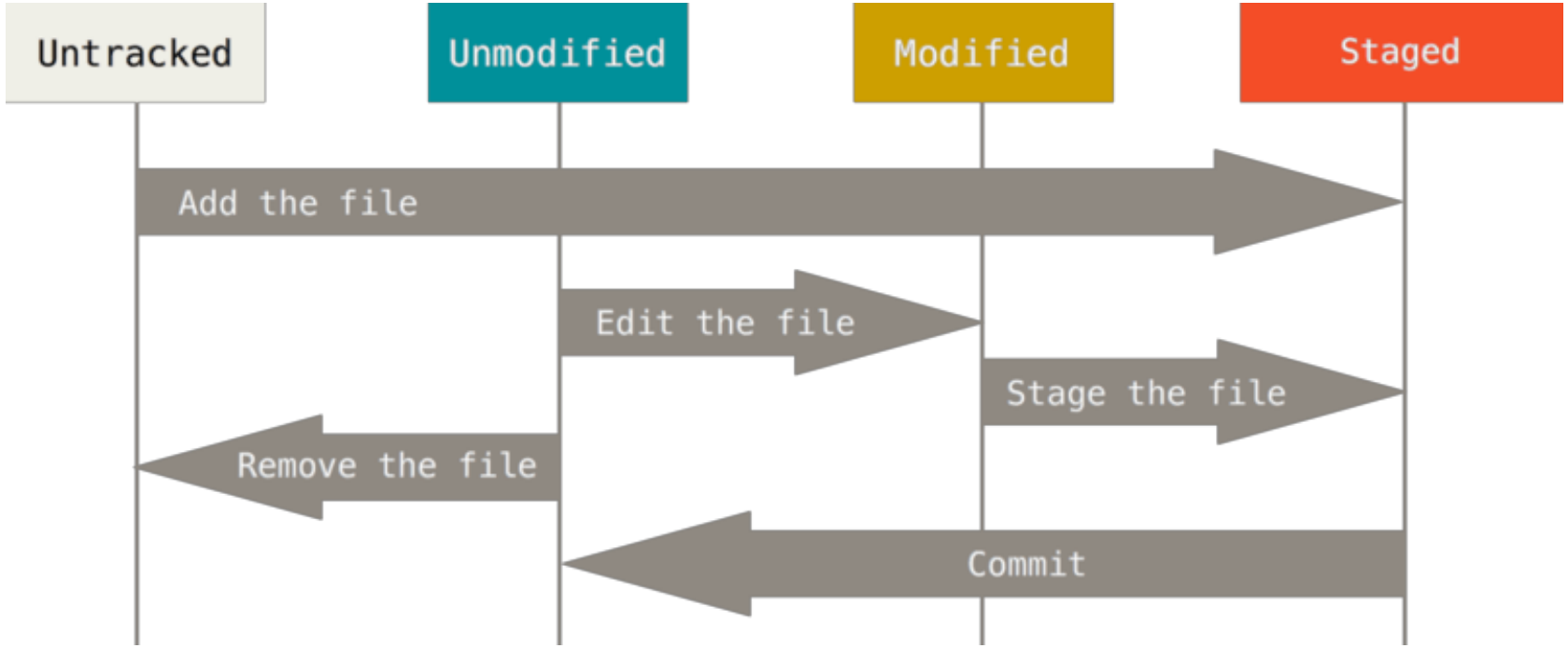
(*) esistono anche gli stati **untracked** e **unmodified.** Li vedremo tra poco

I tre stati dei file



- Repository: il database (cartella) su cui sono archiviate tutte le versioni
- To stage: aggiungere file alla staging area / index
- To commit: archiviare nel repository i file presenti nella staging area
- To checkout: prelevare una specifica versione di un file portandolo nella working area

File untracked e unmodified



- **Untracked:** file che non sono monitorati da git
- **Unmodified:** file monitorati da git che non presentano modifiche rispetto l'ultimo commit.

Installare git

Esistono vari modi di installare git:

- Tramite pacchetti di installazione:
  - http://git-scm.com/download/win
  - http://git-scm.com/download/mac
  - `apt-get install git`
- Tramite compilazione dei sorgenti
  - https://github.com/git/git/releases
  - `git clone git://git.kernel.org/pub/scm/git/git.git`

Comandi di configurazione

Operazione	Comando git
Visualizzazione versione installata	`git --version`
Visualizzazione help	`git help` `git help <verb> (p.e. git help config)`
Visualizzazione configurazione	`git config --list`
Settaggio di un parametro globale	`git config --global user.name "Mario Rossi"` `git config --global user.email "mario.rossi@gmail.com"`

Git bash

In ambiente Windows l’installazione di git mette a disposizione una shell dei comandi tramite la quale è possibile utilizzare comandi Linux.

E’ disponibile tra i programmi installati e nel menù contestuale del tasto destro.

Riga di comando Windows e Linux – Comandi base

Comando	Windows	Linux
Cambiare directory	`cd`	`cd`
Directory corrente	`chdir`	`pwd`
Pulizia finestra	`cls`	`clear`
Copia di file	`copy`	`cp`
Cancellazione di file	`del`	`rm`
Lista file	`dir`	`ls`
Creazione di una nuova directory	`mkdir`	`Mkdir`
Apertura editor	`notepad`	`vim` `nano`
Visualizzazione contenuto file	`type`	`cat`

Comandi di base

Operazione	Comando git
Creazione di un repository vuoto	`git init`
Clonazione di un repository esistente	`git clone <RepositoryUrl>`
Visualizzazione dello stato del repository	`git status`
Aggiunta di un file allo stage	`git add <NomeFile>`

Aggiunta di file multipli allo stage	`git add <NomeFile1> <NomeFile2> …`
	`git add *.cs`
	`git add .`

Rimozione di file dallo stage	`git restore --staged <NomeFile>`
Commit dei file nello stage	`git commit` `git commit -m "commento di commit"`
Commit diretto dei file nella working area	`git commit -a` `git commit -a -m "commento di commit"`

Comandi di base

Operazione	Comando git
Visualizzazione differenze tra working area e ultimo commit	`git diff` `git diff <NomeFile>`
Visualizzazione differenze tra staging area e ultimo commit	`git diff --staged` `git diff --staged <NomeFile>`
Visualizzazione differenze un commit e il precedente	`git show <CommitName>`
Riportare la working directory ad un determinato commit	`git checkout <CommitName>` `git checkout <BranchName>`

Comandi git – annullare comandi

Operazione	Comando git
Sovrascrivere l'ultimo commit	`git commit --amend`
Rimuovere file da staging area	`git restore --staged <nomefile>`
Eliminare modifiche (unmodify file)	`git restore <nomefile>`
"Annullare un commit"	`git revert <CommitName>`

.gitignore

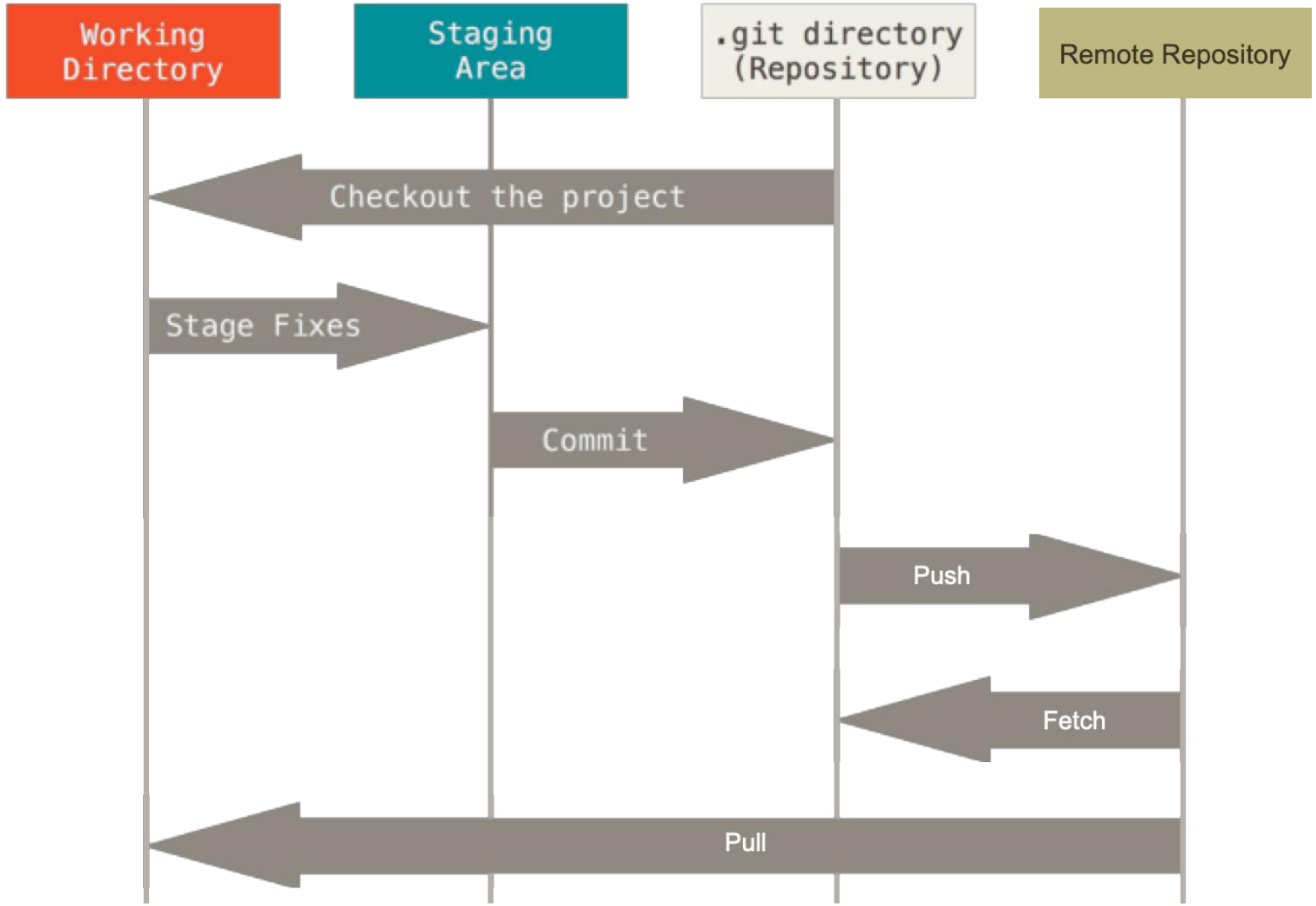
E’ possibile specificare i file da ignorare completamente tramite la creazione e compilazione del file .gitignore. Alcuni esempi di sintassi:

```
# no .txt file, si licence.txt
*.txt
!licence.txt
# ignora il file TODO solo nella root directory
/TODO
# ignora la cartella build
build/
```

Comandi git – commit history

Operazione	Comando git
Comando base	`git log`
Inclusione delle differenze	`git log -p`
Limite di commit mostrati	`git log -<n> (es -2)`
Output riassuntivo	`git log --stat`
Output su una riga	`git log --pretty=oneline`
Output con formato personalizzato	`git log --pretty=format:"%h - %an, %ar : %s"`
Output a grafo	`git log --pretty=oneline --graph`
Limite temporale ai commit mostrati	`git log --since=2.weeks`
Filtro per autore	`git log --author=Davide`
Filtro sul messaggio di commit	`git log --grep=filtro`

Repository remoti



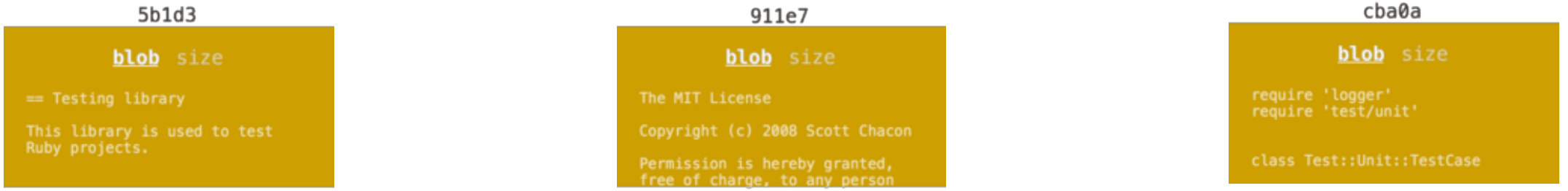
Comandi git – repository remoti

Operazione	Comando git
Elenco repository remoti	`git remote -v`
Aggiungere repository remoto	`git remote add <alias> <remoteUrl>`
Visualizzare info su repository remoto	`git remote show <alias>`
Rinominare un repository remoto	`git remote rename <alias> <nuovoalias>`
Cancellare un repository remoto	`git remote remove <alias>`
Fetch di repository remoto	`git fetch <alias>`
Pull di repository remoto	`git pull <alias> <branch>`
Push su repository remoto	`git push <alias> <branch>`
Clonare un repository remoto	`git clone <remoteUrl>`

Staging - internals

L'operazione di staging di uno o più file comporta le seguenti operazioni:

- Calcolo del checksum (hash SHA-1) di ogni file
- Copia dei file nel repository, nominandoli con il checksum (blob)
- Aggiunta dei checksum alla staging area



Funzioni di Hash

Le funzioni di HASH sono funzioni crittografiche, NON reversibili. Dato l'output di una funzione di hash non è possibile risalire all'input che lo ha generato.

Hanno caratteristiche utili in molti contesti:

- Generano un output di lunghezza fissa indipendentemente dalla lunghezza dell'input
- Una minima modifica dell'input genera una modifica sostanziale dell'output
- La probabilità che due input differenti generino lo stesso output è trascurabile (collisioni)

Funzioni di Hash

Alcune funzioni di hash:

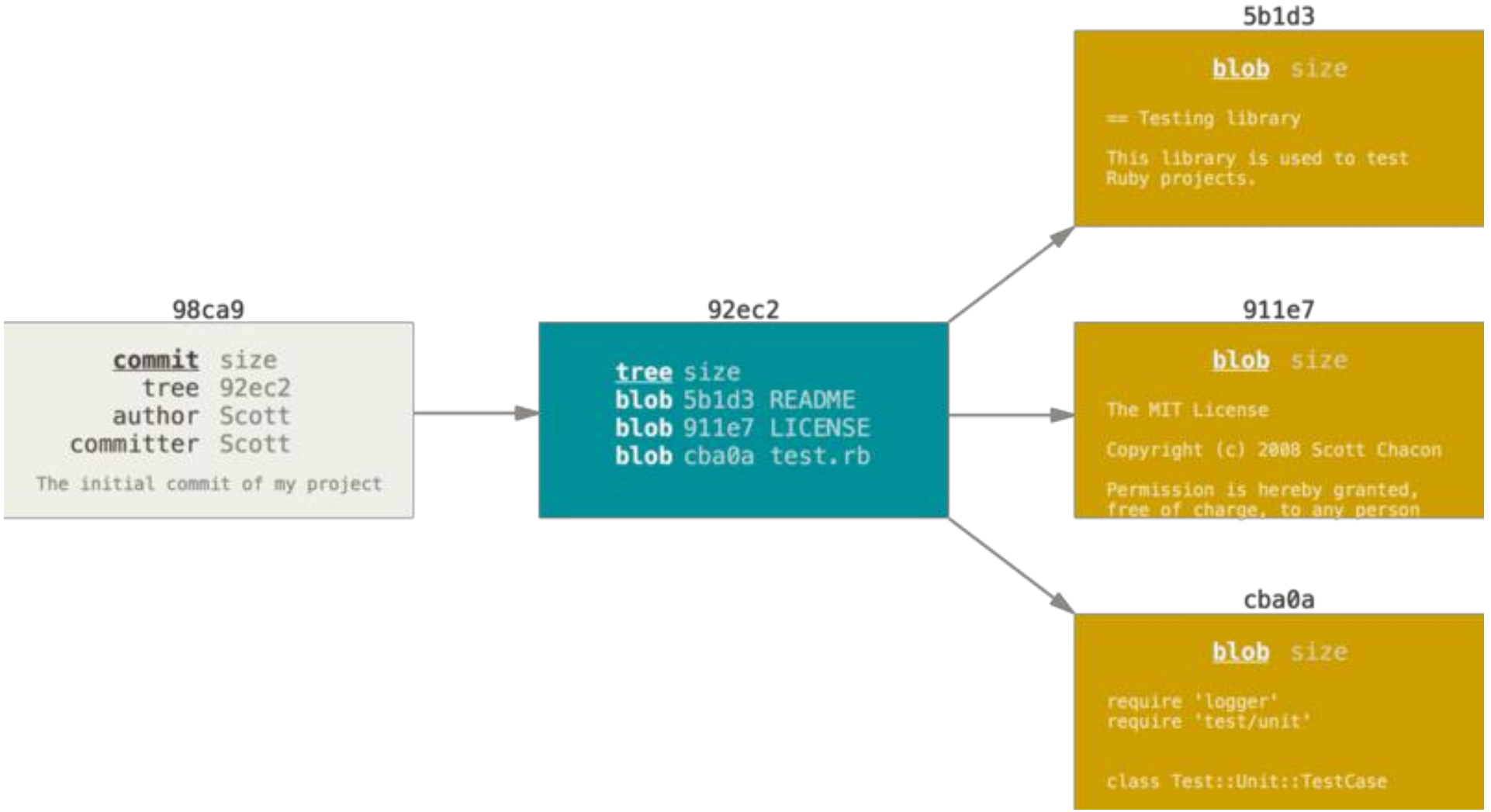
- MD5: Produce output di 128 bit ma la probabilità di collisioni non è trascurabile. Non è più molto utilizzato.
- SHA-1: Creata dall'NSA e produce output di 160 bit. Ha una probabilità di collisioni molto inferiore a MD5.
- SHA-2: Estensione di SHA-1 che produce output di 256 o 512 bit.
- SHA-3: Estensione di SHA-2 con la possibilità di creare output di lunghezza arbitraria.

Commit - internals

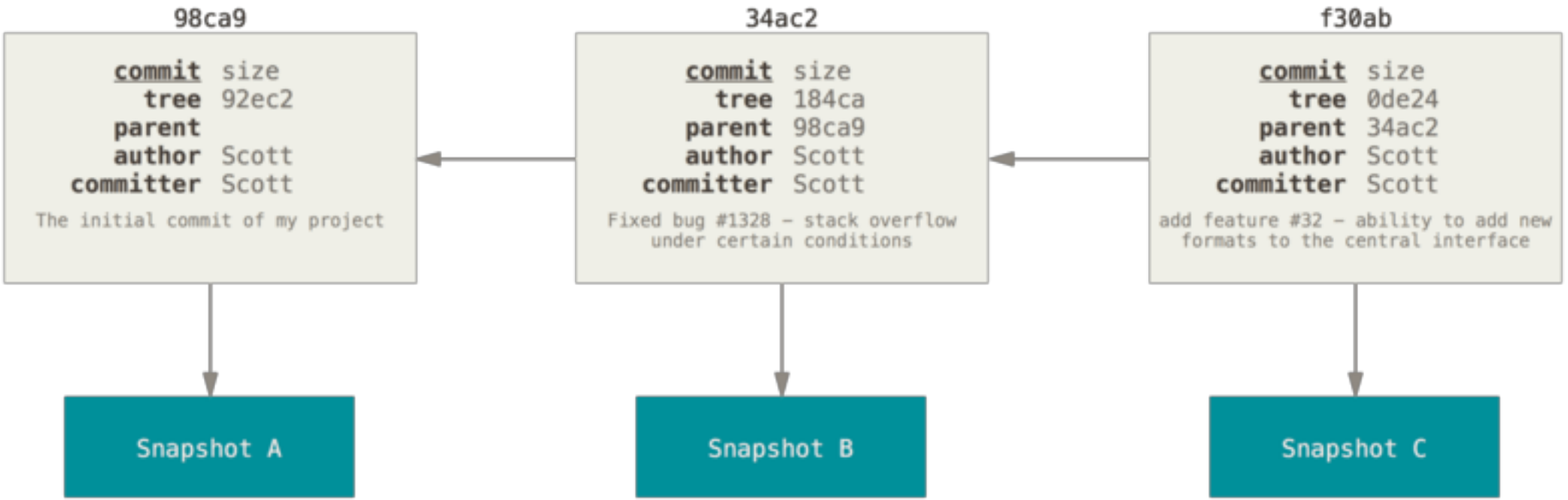
L’operazione di commit esegue le seguenti operazioni:

- Creazione di un tree object contenente i checksum di tutti i file inseriti nello stage
- Calcolo del checksum del tree object, che ne diventa l’identificatore
- Creazione di un commit object contenete:
  - Metadati (autore, commento…)
  - Checksum del tree object
  - Checksum del commit precedente (se esistente)

Commit - internals



Commit - internals



Branches

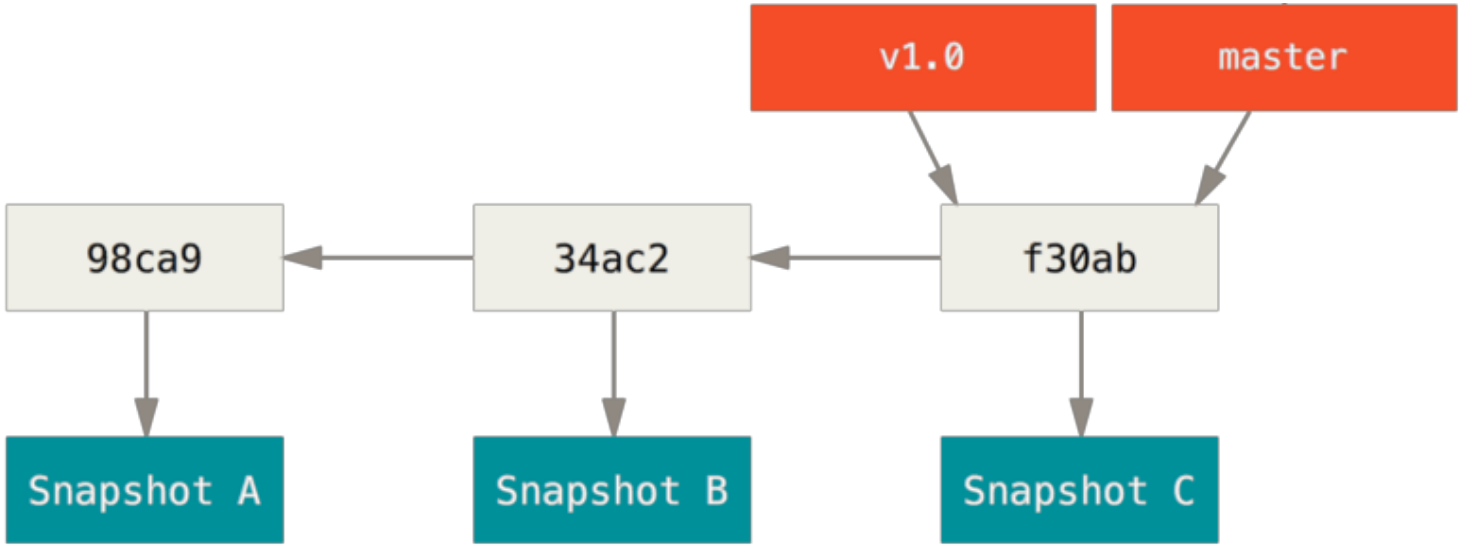
La successione dei commit non deve essere necessariamente lineare. In progetti reali la successione dei commit crea una struttura ad albero partendo dal commit iniziale.

Ogni ramo dell'albero rappresenta un flusso di implementazione ben preciso: ramo principale (master), ramo di sviluppo, ramo per correzione bug n.xx…

Git gestisce i rami tramite il concetto di branch

Branches

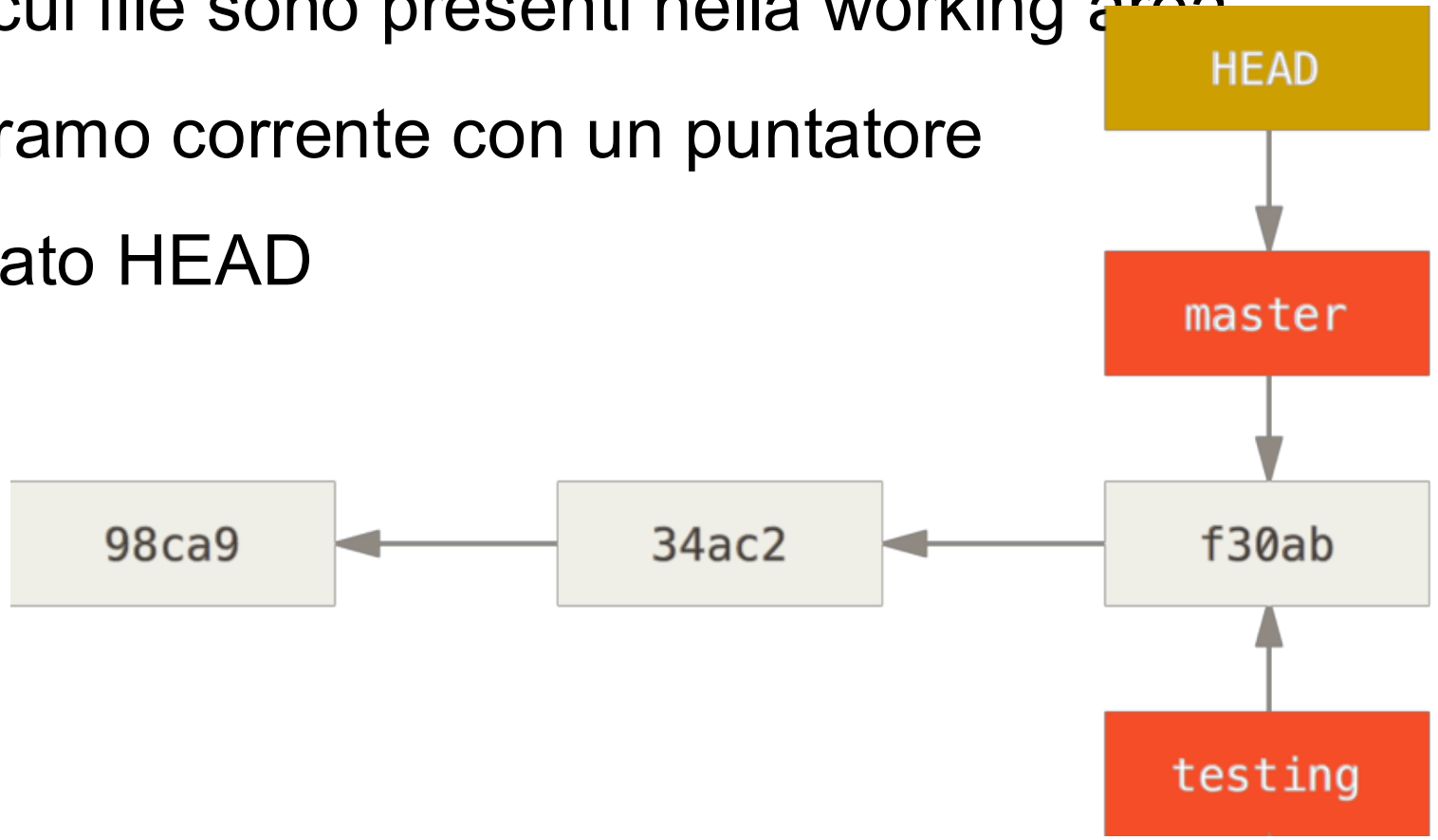
In git un branch non è altro che un puntatore dinamico ad un commit. Il puntatore che identifica il ramo può cioè spostarsi da un commit ad un altro.



Branches - HEAD

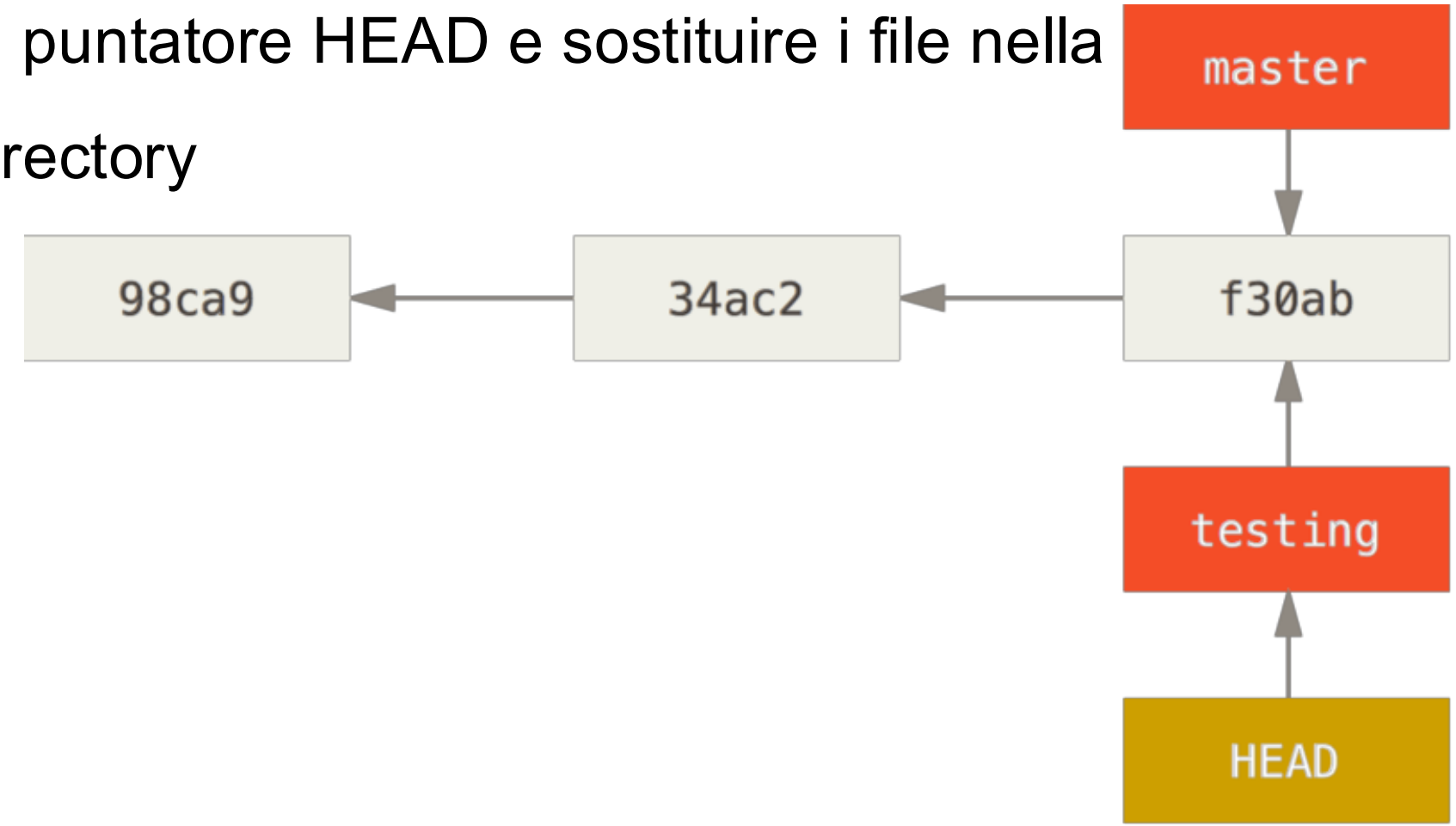
Nonostante possano essere presenti più branch, si opera sempre su un, quello i cui file sono presenti nella working area

Git identifica il ramo corrente con un puntatore speciale chiamato HEAD



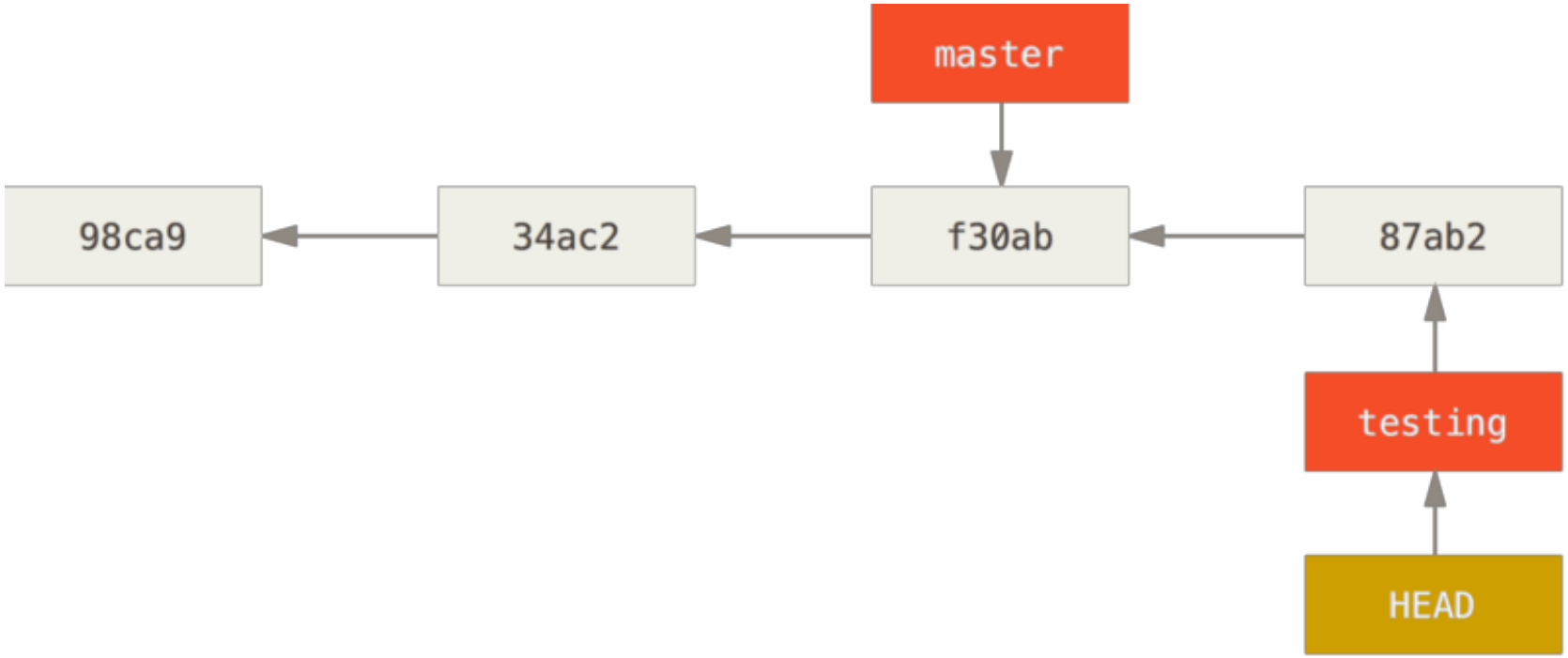
Branches - HEAD

Per passare a lavorare su un branch differente è necessario spostare il puntatore HEAD e sostituire i file nella working directory

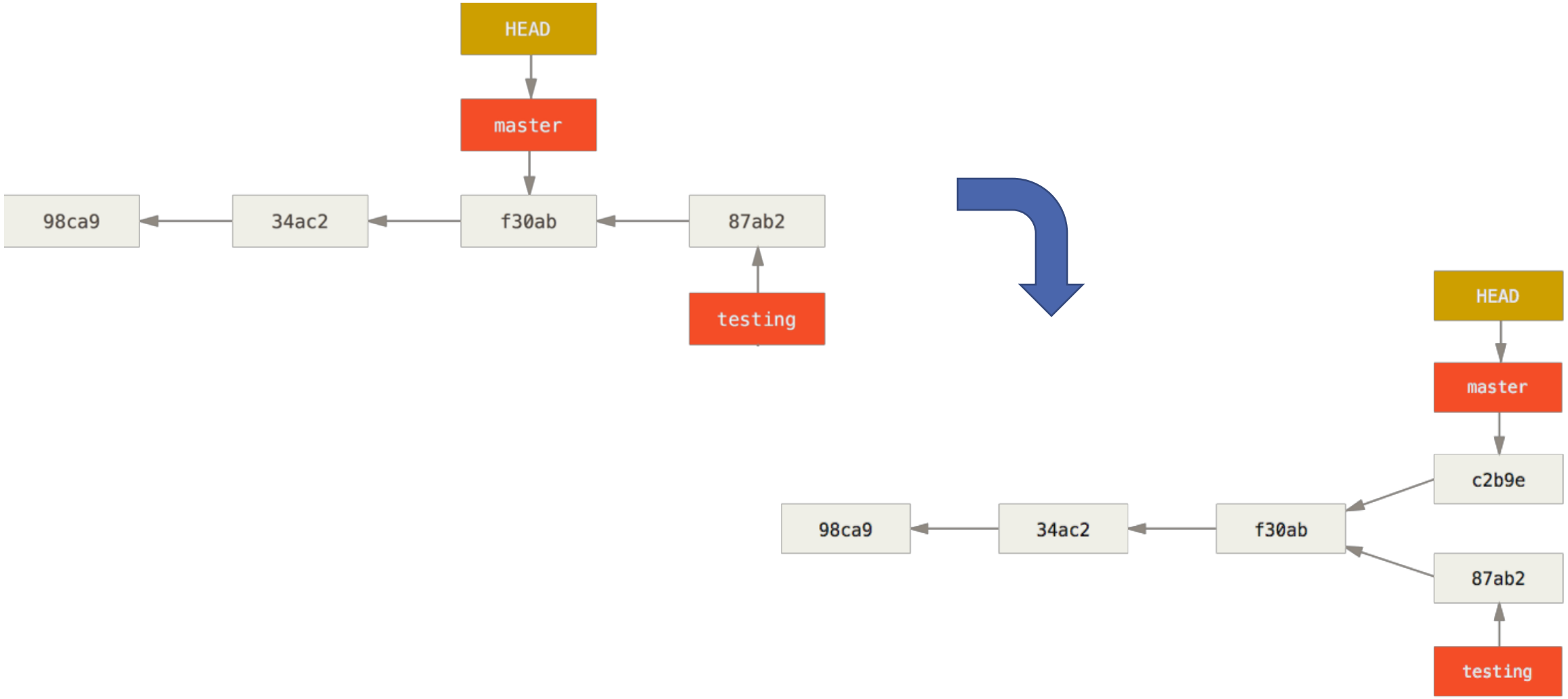


Branches - HEAD

Ad ogni nuovo commit vengono spostati sia il puntatore al branch corrente che il puntatore HEAD



Branches

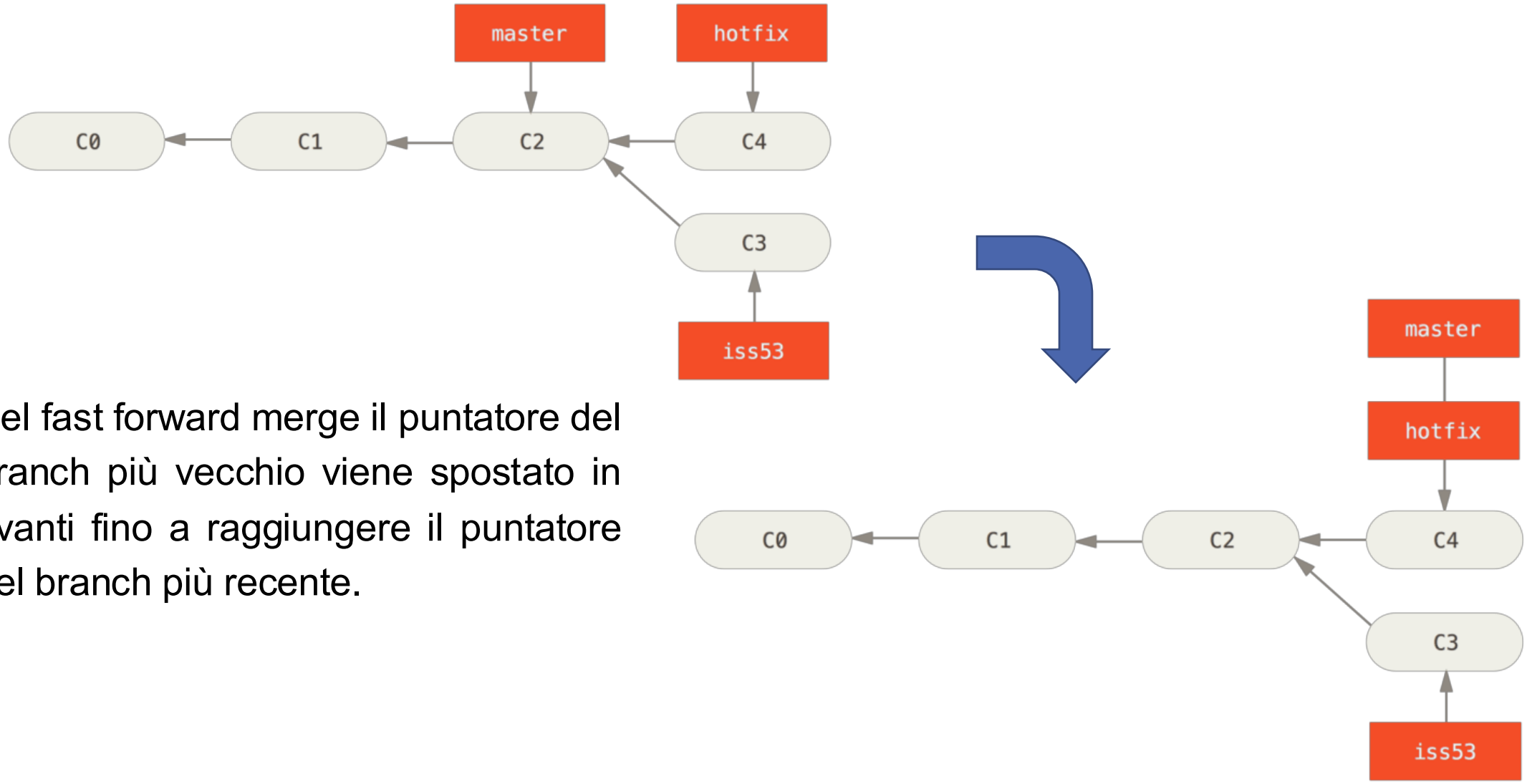

HEAD
master
98ca9
34ac2
f30ab
87ab2
testing
HEAD
master
c2b9e
98ca9
34ac2
f30ab
87ab2
testing

Branches - merge

L’unione di più rami viene detta merge. Git gestisce il merge di due rami alla volta. Può essere di due tipi:

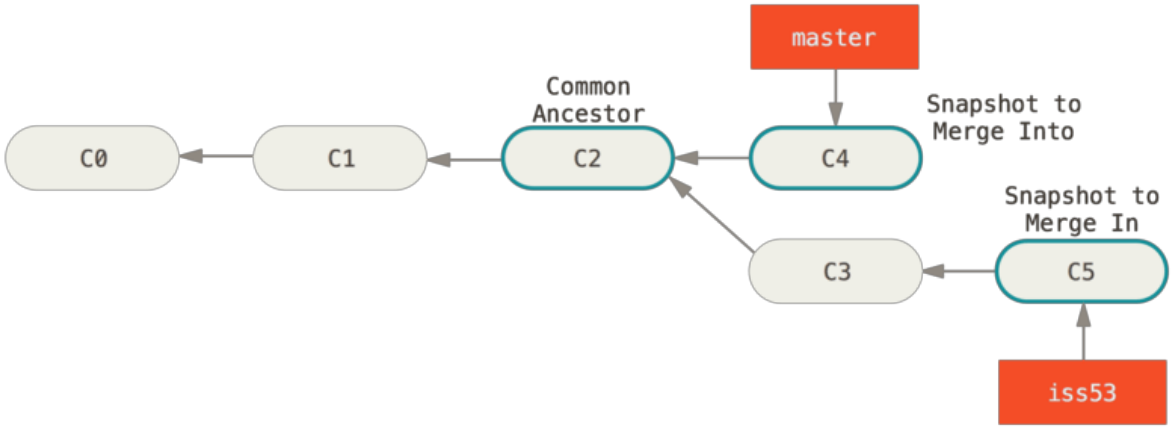
1. Fast forward merge: quando uno dei due branch può essere raggiunto seguendo la storia dei commit dell’altro
2. Three way merge: quando nessuno dei due branch può essere raggiunto seguendo la storia dei commit dell’altro

Branches – Fast forward merge



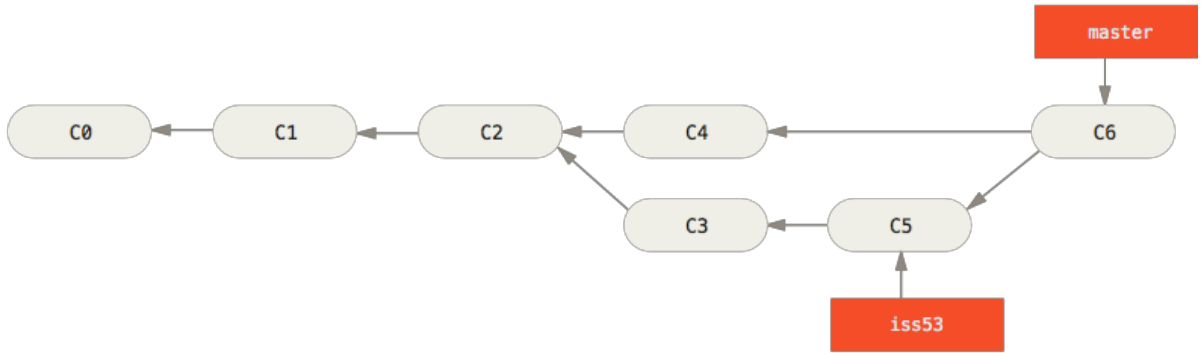
Nel fast forward merge il puntatore del branch più vecchio viene spostato in avanti fino a raggiungere il puntatore del branch più recente.

Branches – Three way merge



Nel tree way merge viene creato un nuovo commit che incorpora tutte le modifiche effettuate nei due branch, rispetto al commit comune ai due.

Possono nascere conflitti se entrambi i branch hanno modificato i file nelle stesse posizioni.



Branches – Three way merge - conflitti

Git non è in grado di completare in autonomia il merge quando ci sono state modifiche alla stessa porzione dello stesso file. Serve un intervento manuale per determinale quale porzione mantenere e quale scartare.

I file contenenti i conflitti vengono modificati da git inserendo entrambe le porzioni di codice ed impedendo il commit.

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
 please contact us at support@github.com
</div>
>>>>>>> iss53:index.html
```

Dopo la risoluzione manuale del conflitto è possibile effettuare il commit.

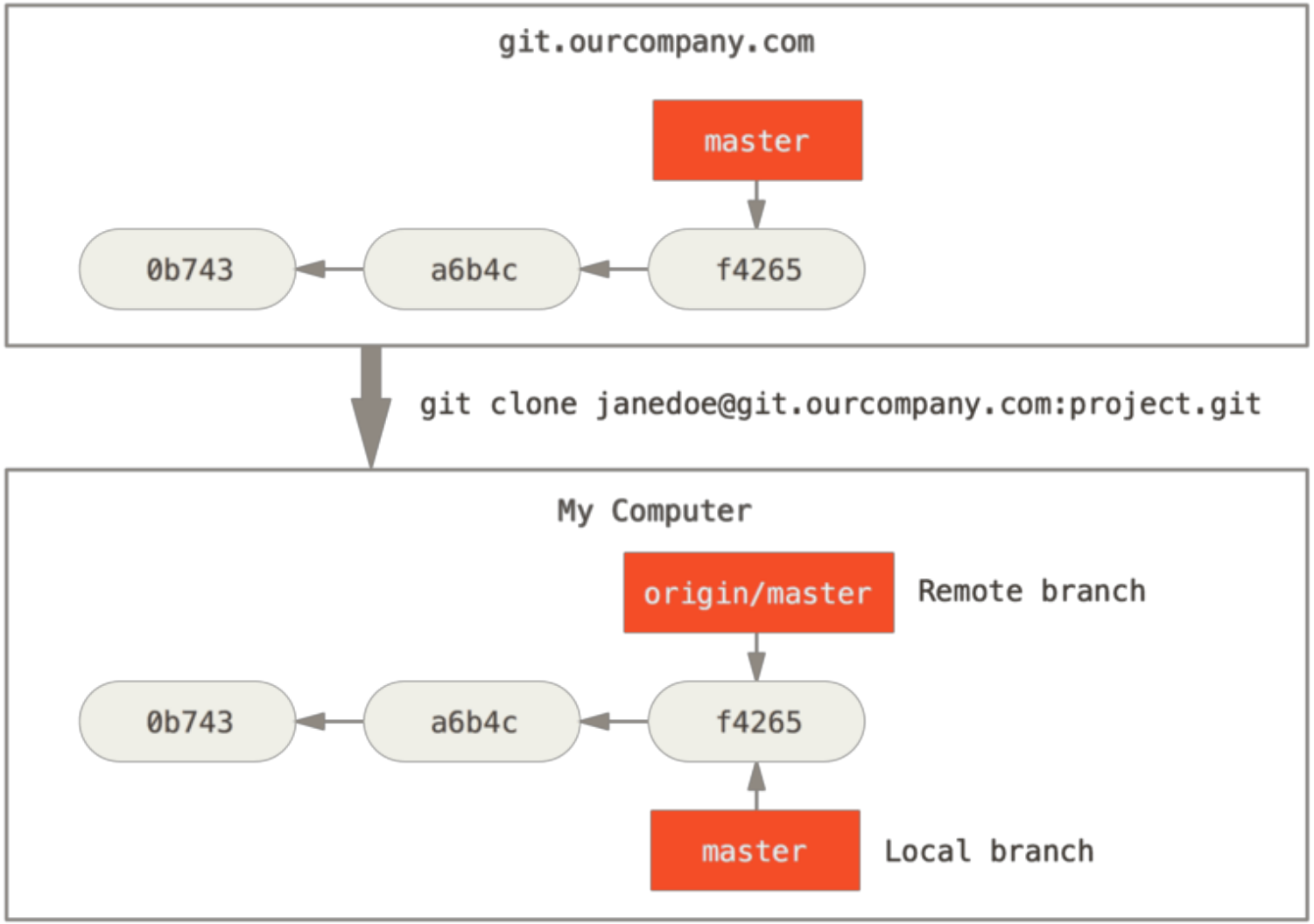
Branch remoti

Anche i branch remoti contengono branch, anche se non tutti i branch locali saranno sincronizzati con i repository remoti.

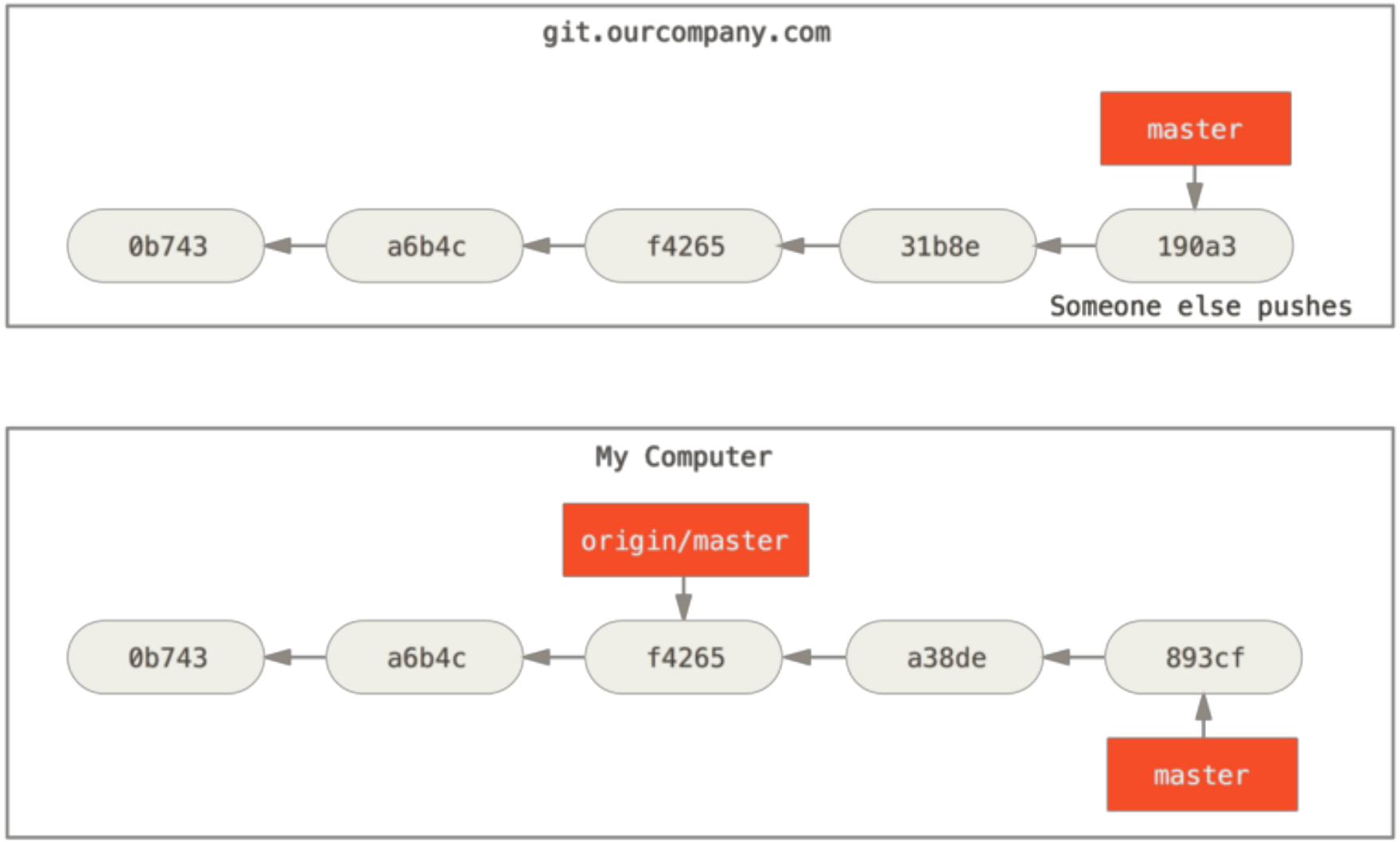
I branch remoti sono identificati come <nomeRemoto>/<branch> e sono in sola lettura. Non avanzano con i commit locali ma solamente con i push remoti.

Non è possibile lavorare direttamente con i branch remoti. Vanno mergiati in un branch locale, oppure va creato un tracking branch.

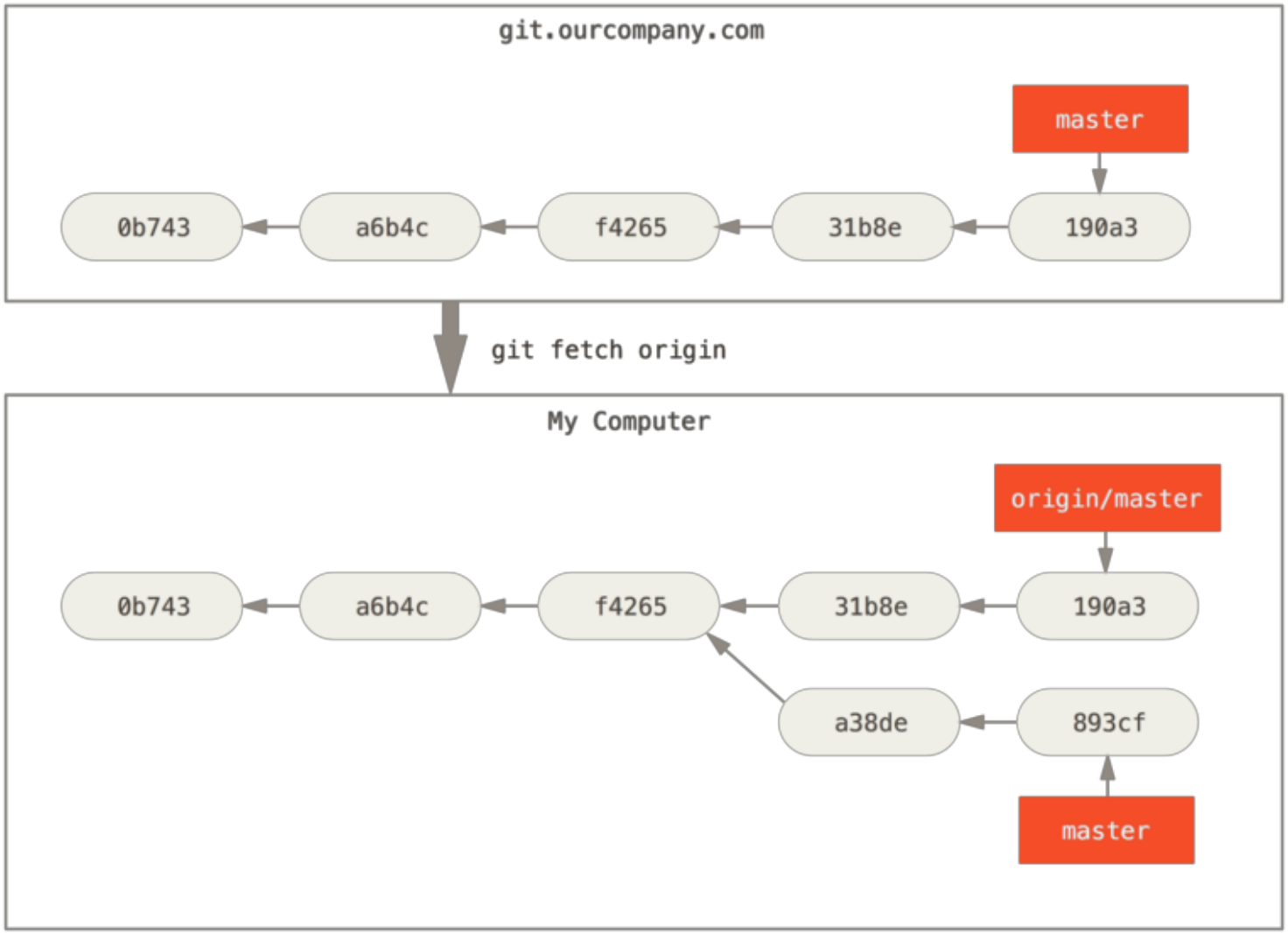
Branch remoti



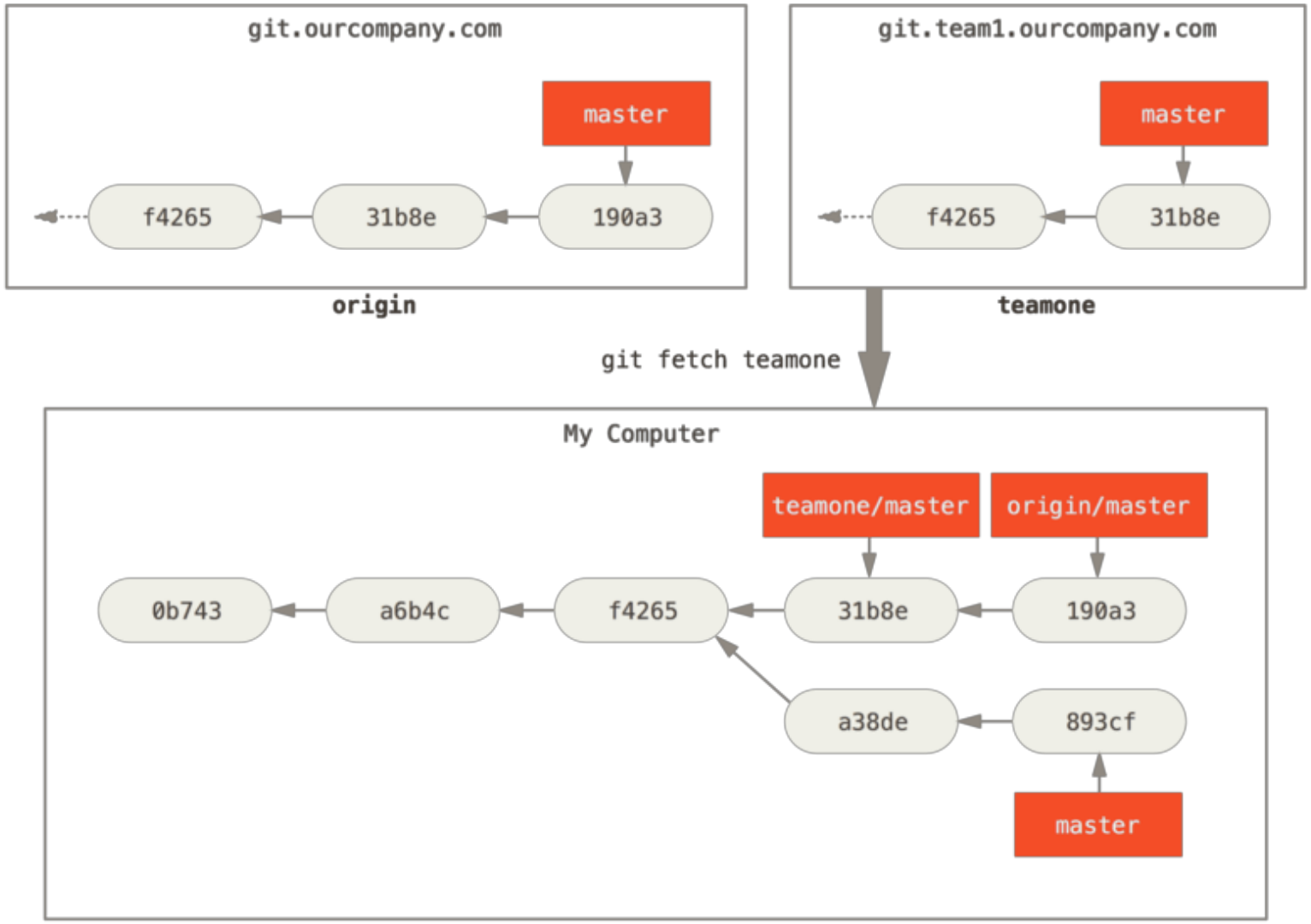
Branch remoti



Branch remoti



Branch remoti



Comandi git – gestione branches

Operazione	Comando git
Elenco brach	`git branch`
Creazione di un nuovo branch posizionato sul commit corrente (senza spostamento di HEAD)	`git branch <nomeBranch>`
Modificare il branch attivo (spostamento di HEAD)	`git checkout <nomeBranch>` `git switch <nomeBranch>`
Creazione nuovo branch e contestuale modifica del branch attivo	`git checkout -b <nomeBranch>` `git switch -c <nomeBranch>`
Merge tra un branch e il branch corrente	`git merge <nomeBranch>`
Cancellare un branch	`git branch -d <nomeBranch>`
Risoluzione dei conflitti con tool grafico esterno	`git mergetool`

Comandi git – gestione branches

Operazione	Comando git
Elenco di tutti i branch	`git branch --all`
Elenco branch con ultimo commit	`git branch -v`
Elenco branch mergiati / non mergiati con il branch corrente	`git branch --merged` `git branch --no-merged`
Eliminazione di un branch non mergiato (perdendone tutti i dati)	`git branch -D <nomeBranch>`
Rinominare un branch	`git branch --move <nomeAttuale> <nomeNuovo>`
Propagare la modifica del nome sul repository remoto	`git push --set-upstream <alias> <nomeNuovo>`
Cancellazione di branch sul repository remoto	`git push <alias> --delete <branch>`
Lettura branch da repository remoto	`git fetch <alias>`

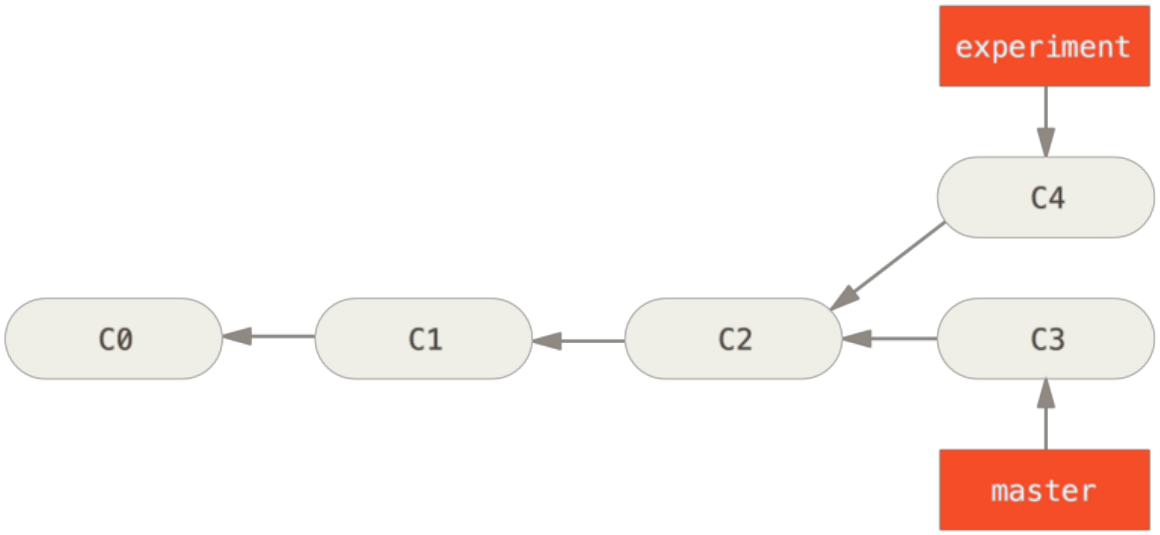
Branches - Rebasing

Il three way merge non è l'unico modo per unire il lavoro fatto su rami divergenti. Una seconda possibilità è data dal rebasing.

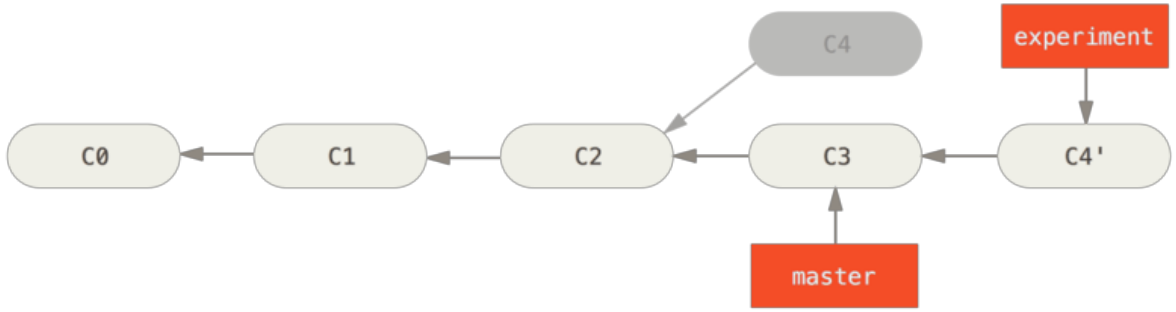
Il rebase, a differenza del merge, elimina alcuni commit già eseguiti generando uno storico molto più snello e lineare.

ATTENZIONE ad utilizzare il rebase quando si lavora su repository remoti.

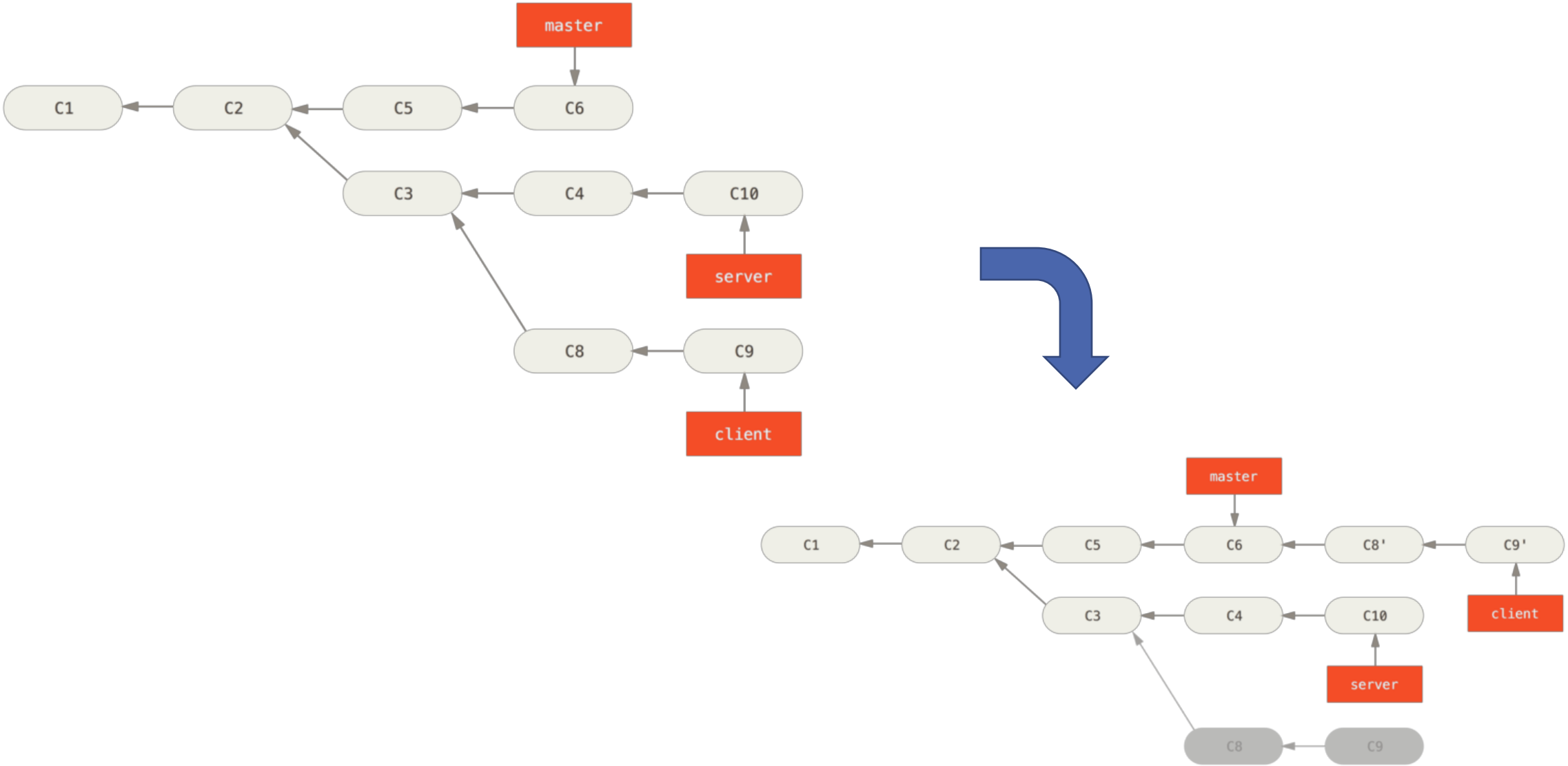
Branches – Rebasing



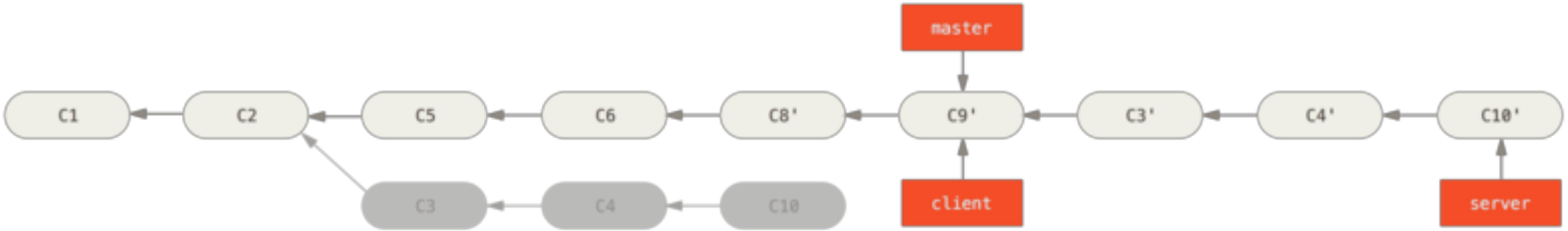
Il rebase crea nuovi commit per incorpora tutte le modifiche effettuate nel secondo branch rispetto al primo. Il secondo branch viene poi completamente cancellato



Branches – Rebasing



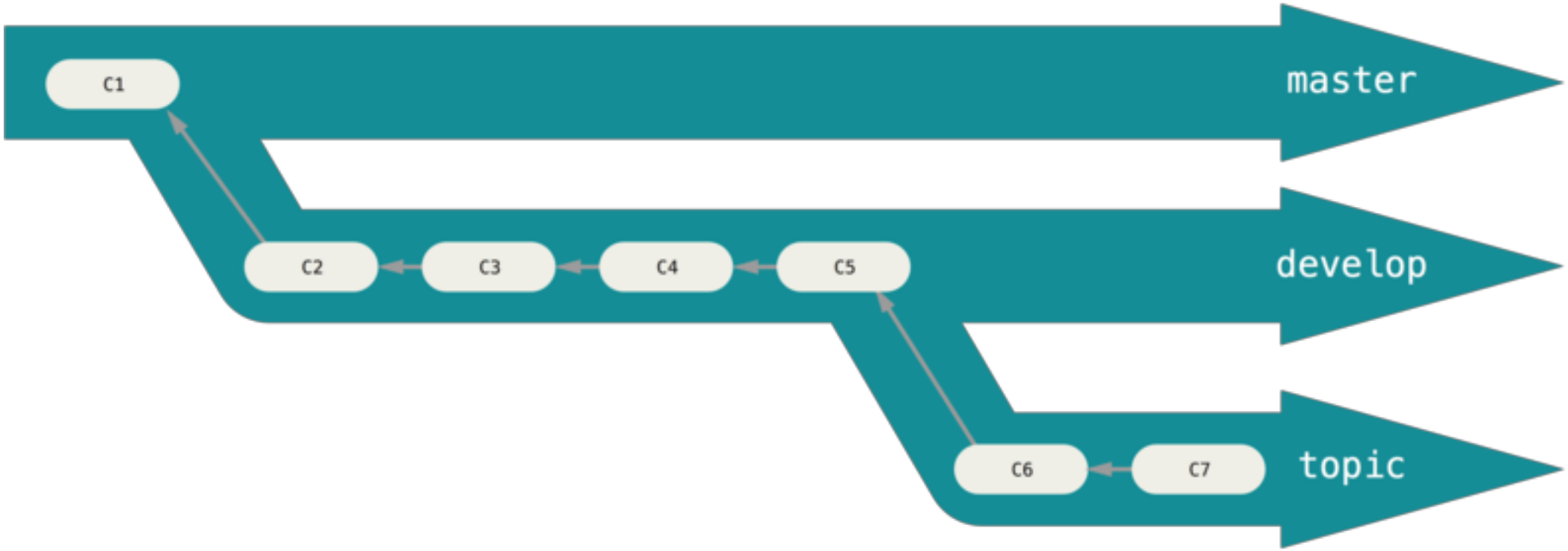
Branches – Rebasing



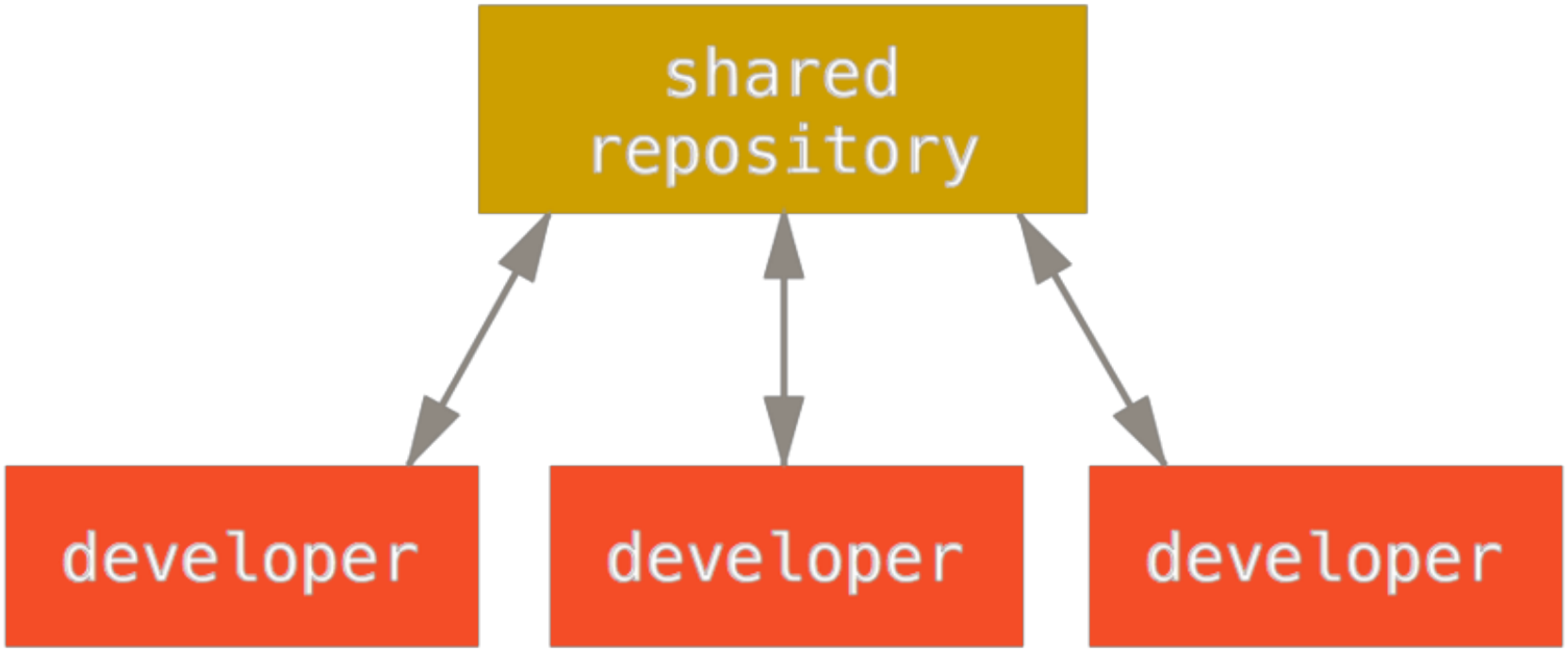
Comandi git – gestione rebase

Operazione	Comando git
Rebase del ramo corrente dentro altro ramo (i commit del ramo corrente sono cancellati)	`git rebase <nomeBranch>`
Rebase di un ramo all'interno di un altro ramo	`git rebase <baseBranch> <topicBranch>`

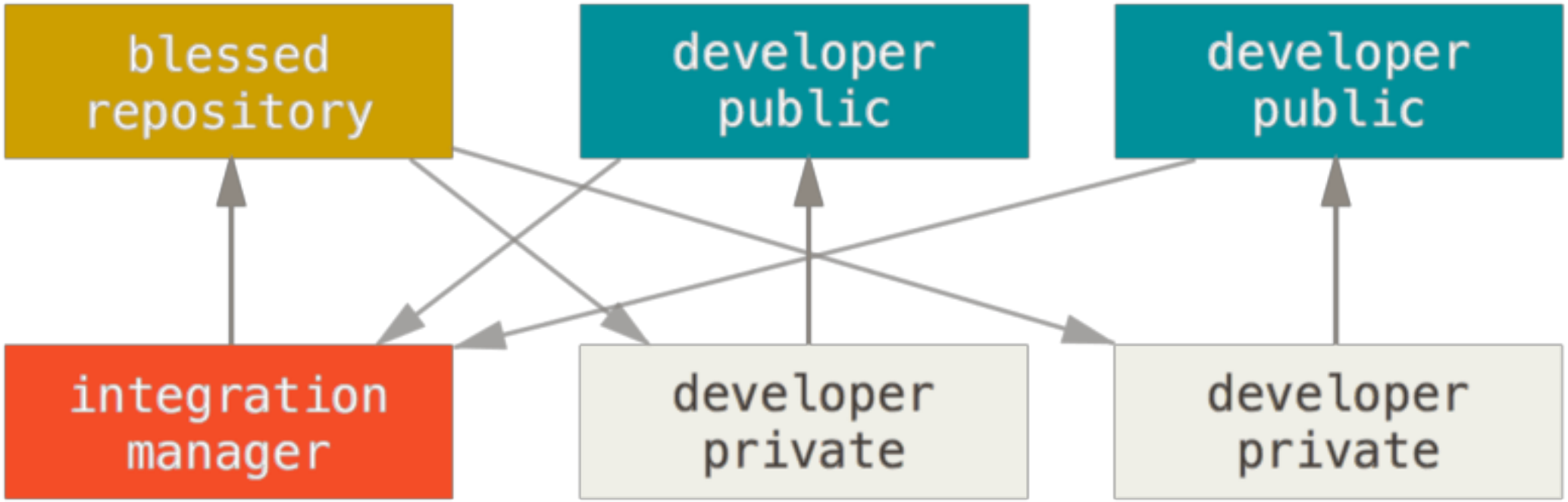
Basic workflows



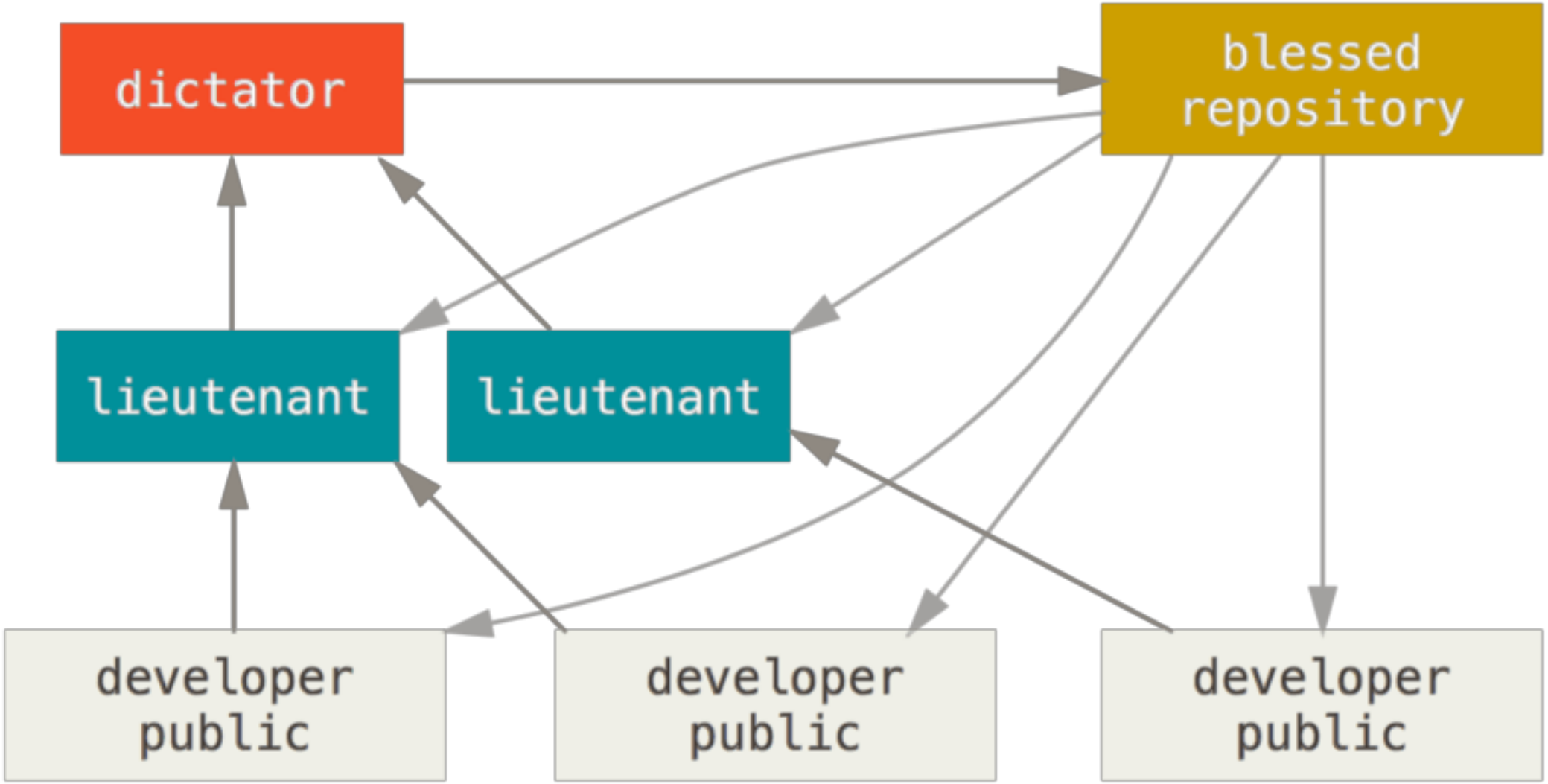
Basic workflows - centralized



Basic workflows – integration manager



Basic workflows – dictator & liutenants



github.com

GitHub

GitHub è la più grande piattaforma per l'hosting di repository git ed ospita milioni di progetti Open Source.

In aggiunta all'hosting offre molteplici features per la gestione del ciclo di sviluppo e per la gestione di progetti di molteplici dimensioni.

In GitHub è possibile costruire molteplici flussi di lavoro, per adattarsi alle esigenze di ogni progetto.

GitHub – Caratteristiche principali

- Progetti pubblici e privati
- Gestione autorizzazioni
- Molteplici flussi di lavoro
- Pull request e Code Review
- Issues
- Automazione e CI/CD
- Gestione sicurezza

GitHub – Issues

La sezione Issues permette di inserire attività da svolgere sul progetto, con la possibilità di associare una categoria (labels) e assegnare uno sviluppatore.

Gli Issues possono essere chiusi quando la relativa attività è stata svolta o si è esaurita.

E’ possibile creare un collegamento tra gli issues e i commit semplicemente inserendo un riferimento all’issue nel commento del commit → #n (fix #n chiude anche l’issue in sede di commit)

GitHub – Pull request

Quando non si hanno le autorizzazioni per modificare un ramo, la soluzione proposta da GitHub consiste in:

- Lavorare su un ramo/repository differente sul quale si hanno le autorizzazioni per lavorare (branching o forking)
- Chiedere all'amministratore del ramo che si vorrebbe modificare di effettuare il merge con le nostre modifiche
- L'amministratore del ramo ha la possibilità di effettuare Code Review delle modifiche proposte

GitHub Desktop

GitHub Desktop

GitHub Desktop è il client ufficiale, di tipo visuale, per gestire repository git locali e remoti.

GitHub Desktop permette di utilizzare anche le funzioni specifiche di GitHub (fork, pull request…)

NB: L'utilizzo di client visuali può agevolare la gestione dei repository ma non deve sostituire la conoscenza dei comandi da riga di comando.