Git

Git

- Git: è un sistema di versioning **distribuito**
- È stato pensato per essere utilizzato da linea di comando.
- Ma, ad oggi, esistono molti ottimi client grafici, sia stand-alone che integrati nelle maggiori piattaforme di sviluppo (Eclipse, IntelliJIdea, NetBeans...)

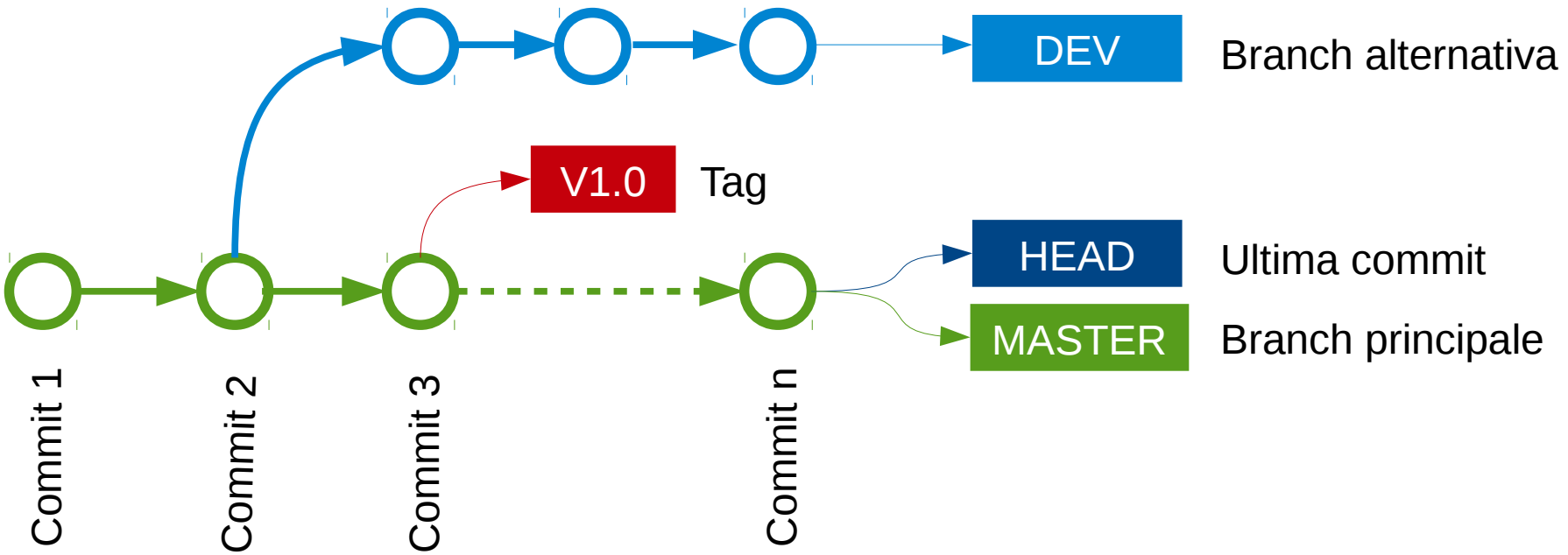
Git

- Git: impostazioni preliminari
- Per spiegare tutte le possibilità di configurazione di Git occorrerebbe più tempo (e pazienza). Le essenziali, per iniziare, sono:
- git config --global user.name "Tuo Nome"
- git config --global user.email "myname@domain.com"

In questo modo Git conosce il nome di chi ha creato una commit

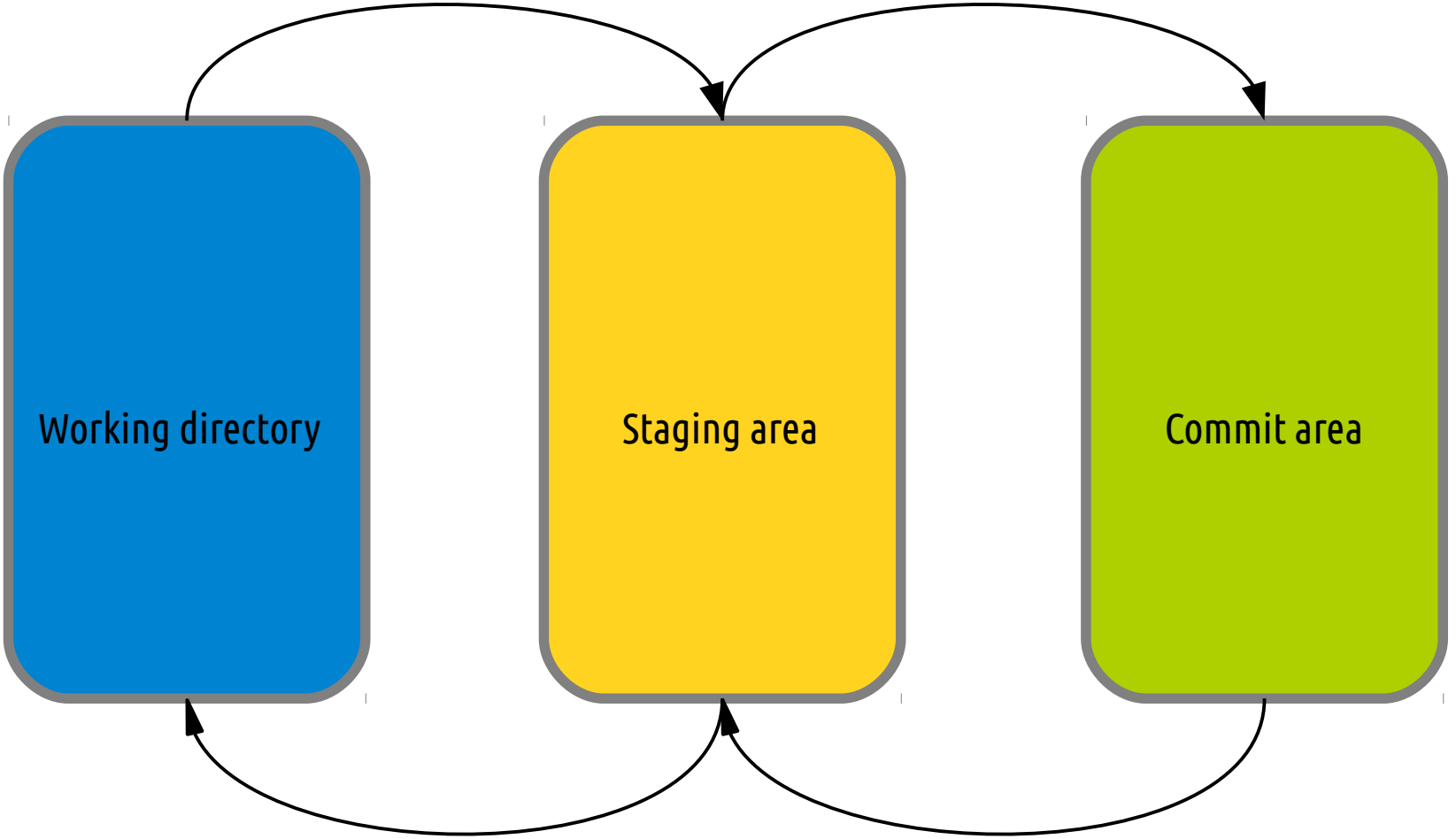
Git

Terminologia



Git

Aree di lavoro

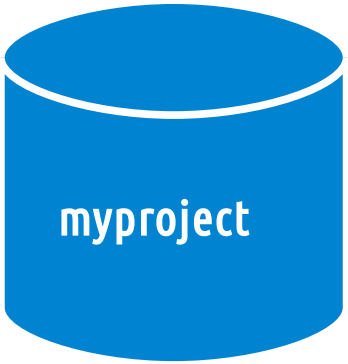


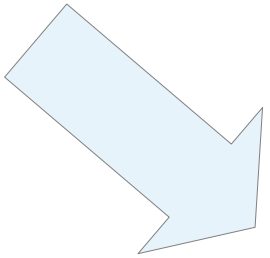
Git

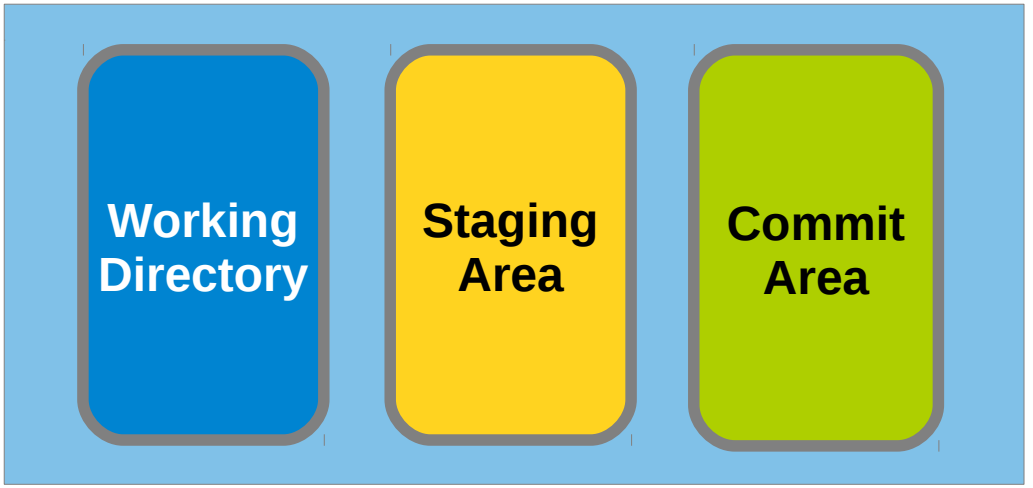
Creazione di un repository

Per creare un repository, basta posizionarsi nella relativa dir ed inizializzarlo

```
$ cd myproject
$ git init
Initialized empty Git repository in ~/myproject/.git/
```







Git

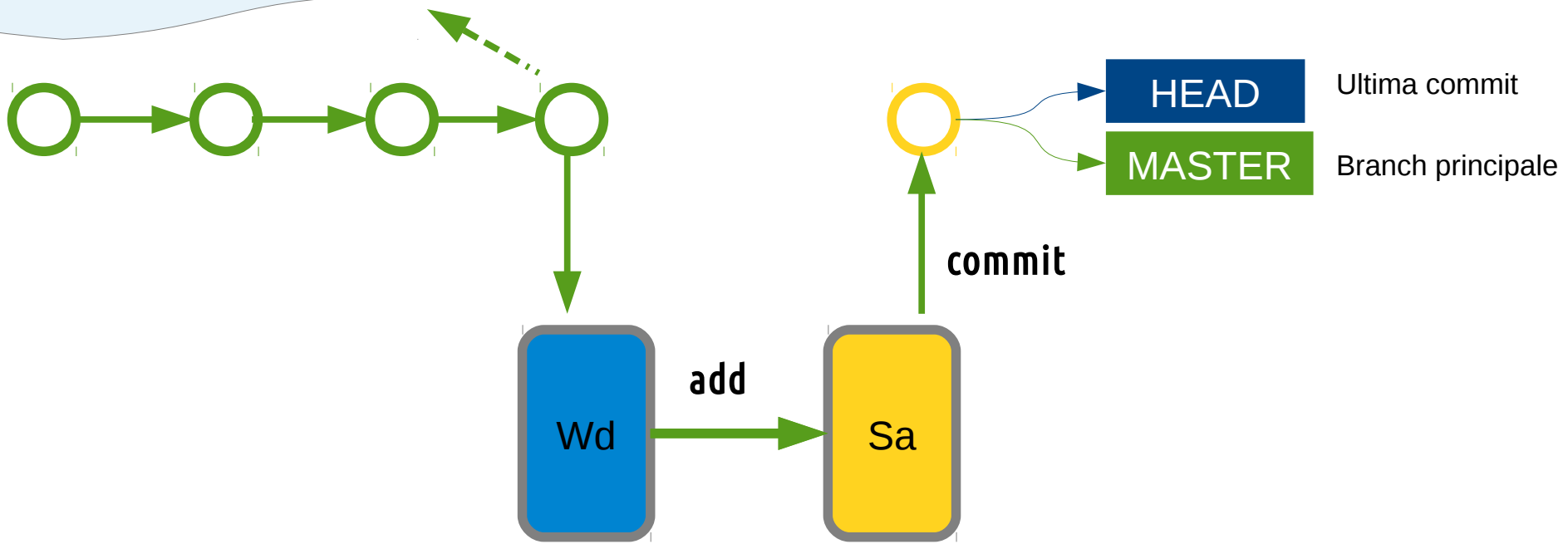
Commit

Modifico un file, lo metto on stage (**add**), e poi nella commit area (**commit**)

```
$ git add hello.txt
$ git commit -m “added role developer”
...
$ git status
# On branch master
nothing to commit, working directory clean
```

hello.txt

```
# “Created by $USER” - role developer
Ciao Mondo
added row to use tagging
```

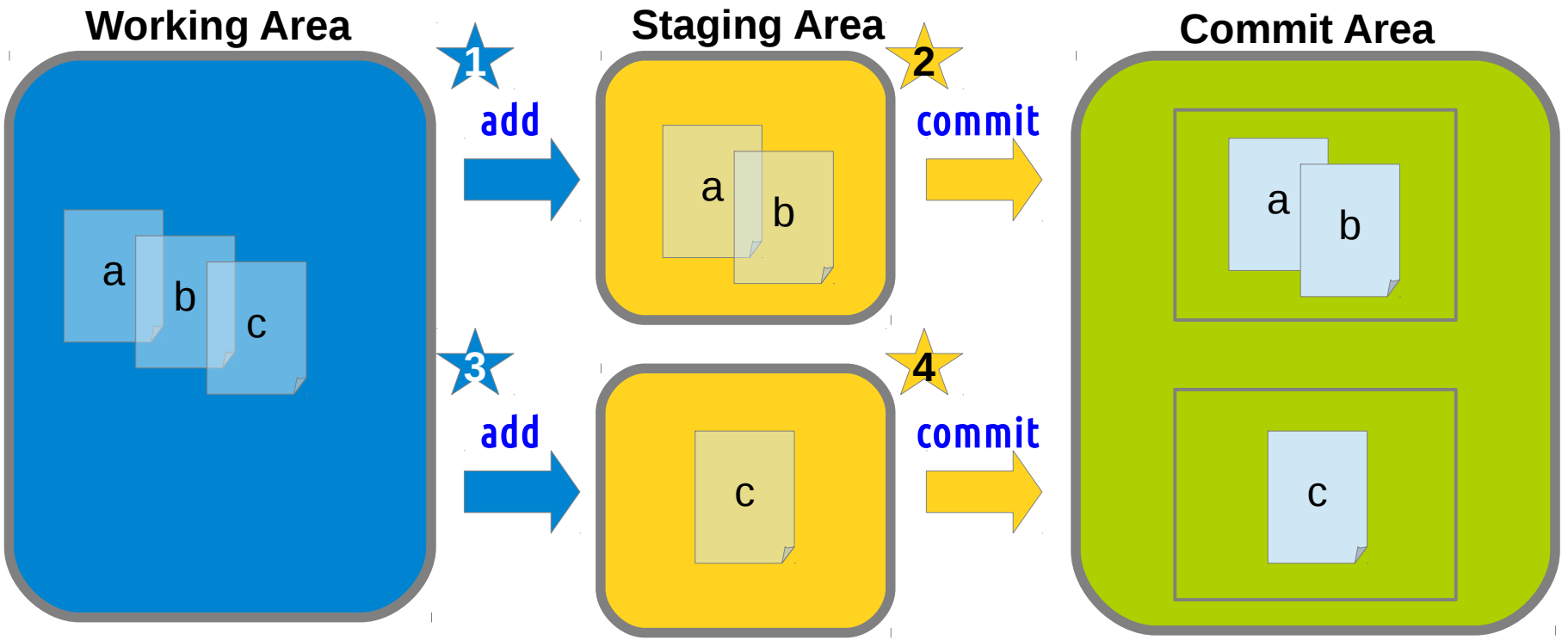


Git

Staging e commit

• Modello Git: Git consente di organizzare le modifiche in modo che riflettano esattamente il lavoro svolto.

Ad esempio: ci sono tre file modificati (a.txt, b.txt e c.txt). Tutte le modifiche saranno messe in scena (on staging), ma **si desidera che i cambiamenti in a.txt e b.txt appartengano ad un singolo commit, e quelli in c.txt ad un altro**.



Git

Commit

Si può creare una commit passando il commento come parametro: git commit -m "Changes for a and b"

Oppure Git consente di configurare un editor di testo agganciato alla fase di commit, che verrà aperto automaticamente per scrivere/modificare il commento.

Per scegliere l'editor preferito, si usa il comando (per vi):

- $ git config –global core.editor vi

Quindi: una commit senza parametro -m attiva un editor che permette di scrivere il commento in modo più agevole

• NB: è sempre possibile lasciare un commento vuoto (-m “”) ma non è una Best Practice...

Git

Cambiamenti e non files

Git registra come entità elementari i singoli cambiamenti , e non i files modificati

- Per esempio, modifichiamo due volte il file hello.txt
  - $ <edit> hello.txt, e aggiungiamo un commento: “Creato da <user>”
  - $ git add hello.txt

- Ora facciamo una seconda modifica allo stesso file
  - $ <edit> hello.txt, e modifichiamo il commento: “Created by $user”
  - $ git status

```
$ git status
...
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   hello.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   hello.txt
```

Git

Cambiamenti e non files

- Ora facciamo una commit
  - $ git commit -m "Added something"
  - $ git status

```
$ git status
...
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#     modified:   hello.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

- Ora mettiamo di nuovo hello.txt on stage, ed eseguiamo una commit
  - $ git add hello.txt
  - $ git commit -m "Modified default comment"
  - $ git status

```
# On branch master
nothing to commit, working directory clean
```

Git

Storia

Si possono visualizzare tutte le commit effettuate nel progetto

- $ git log

```
$ git log
commit bcc814ee0e61b46ee7be6a7c675d3777f3bb219d
Author: Luigi Talamona <luigi@talamona.org>
Date:   Fri Aug 30 16:39:37 2013 +0200

    Modified default comment

commit 385d2d8a4551d1b083d0f2aab0486c9942836953
Author: Luigi Talamona <luigi@talamona.org>
Date:   Fri Aug 30 16:28:21 2013 +0200

    Added a default comment

commit 10e8fc4f85dee3f17a9b9bb2b8826ac4a915a97f
Author: Luigi Talamona <luigi@talamona.org>
Date:   Fri Aug 30 15:30:01 2013 +0200

    First Commit
```

Git

Ritornare sui propri passi

E' facile ritornare sui propri passi, usando il codice hash della commit

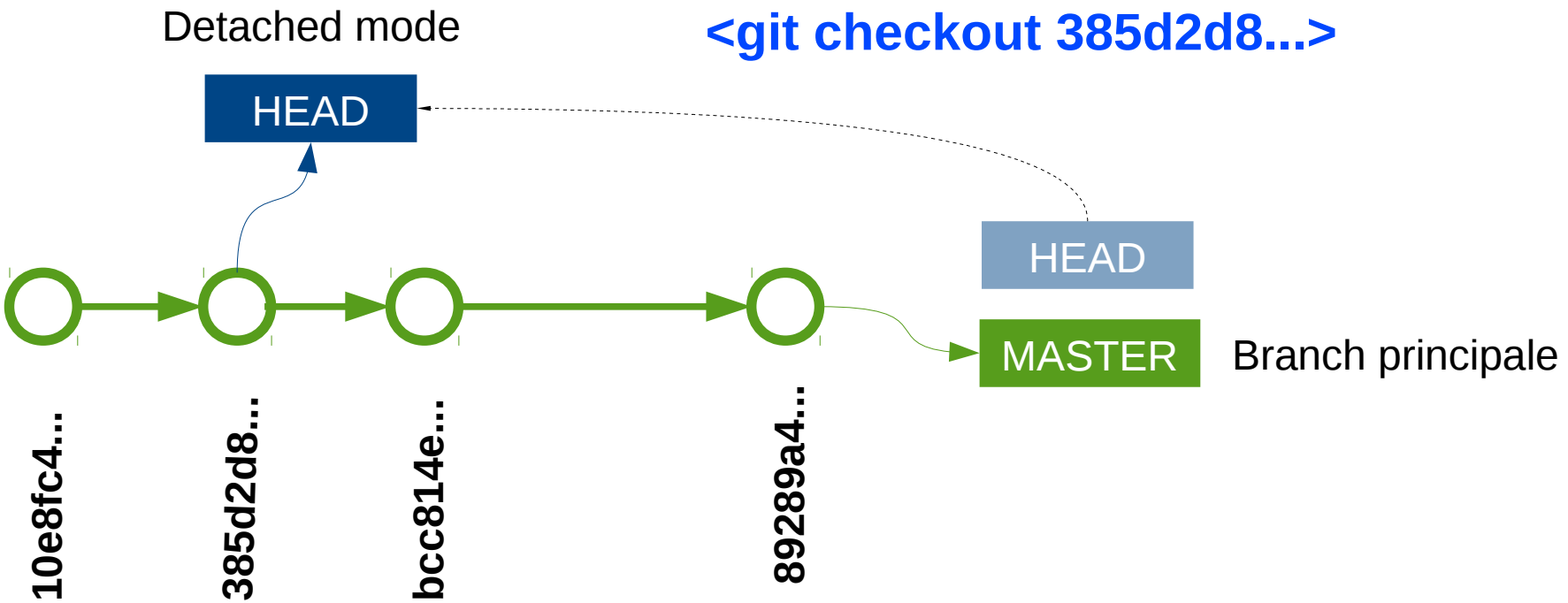
- $ git log, come prima, e scegliamo la prima commit
  10E8fc4...

- $ git **checkout** 10e8fc4

```
Note: checking out '10e8fc4f85dee3f17a9b9bb2b8826ac4a915a97f'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b new_branch_name

HEAD is now at 10e8fc4... First Commit
```

Git

Ritornare sui propri passi

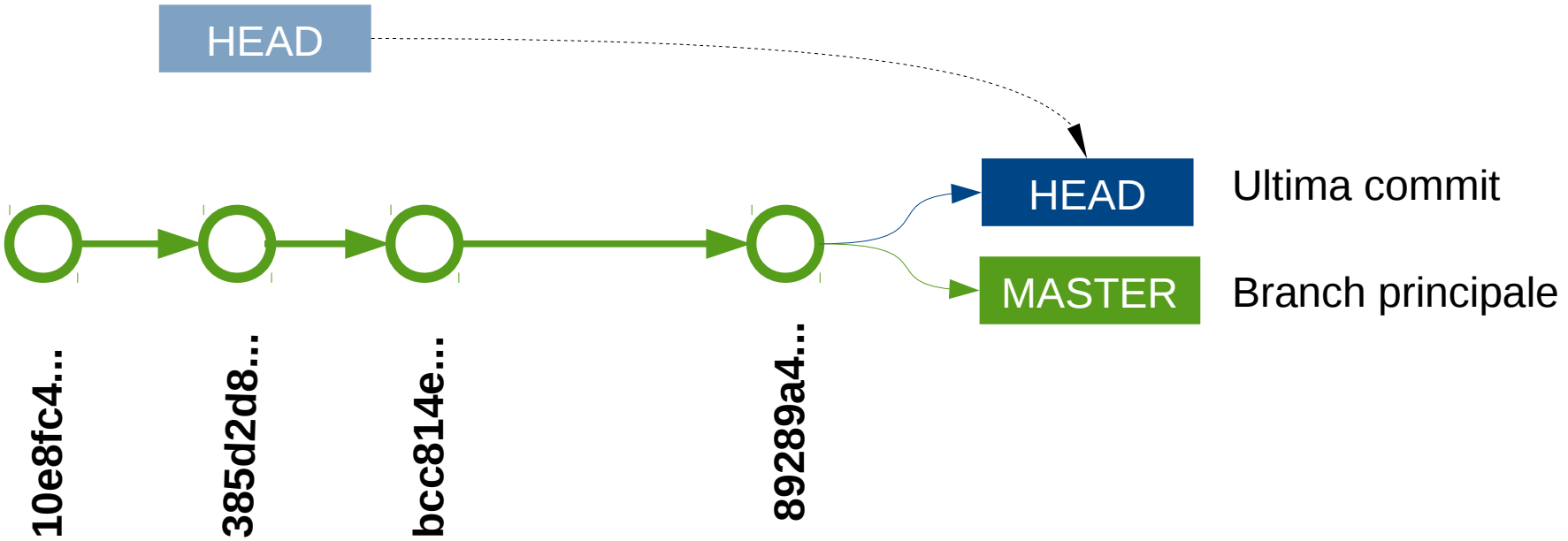


Git

Ritornare sui propri passi

Ora, per tornare allo “stato dell'arte”

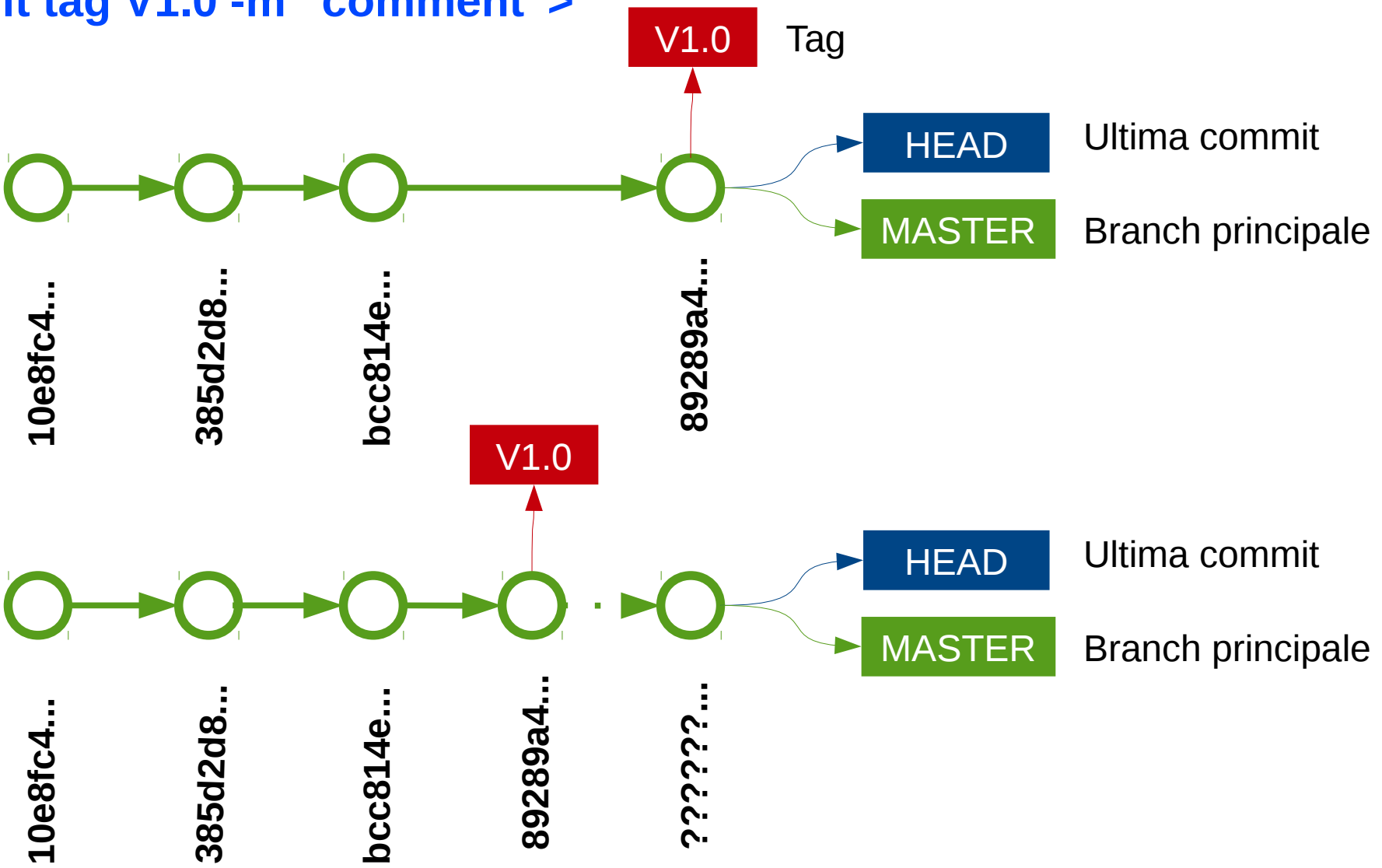
<git checkout master>



Git

Etichettare i passi salienti (Tagging)

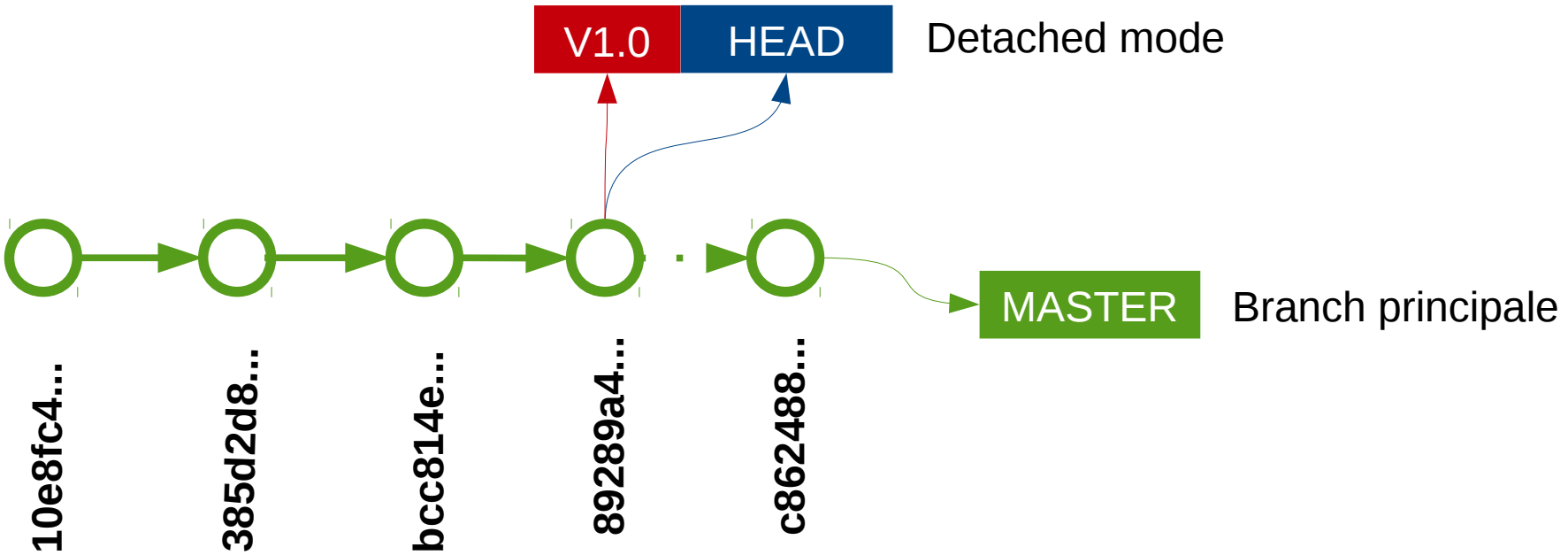
<git tag V1.0 -m “comment”>



Git

Etichettare i passi salienti (Tagging)

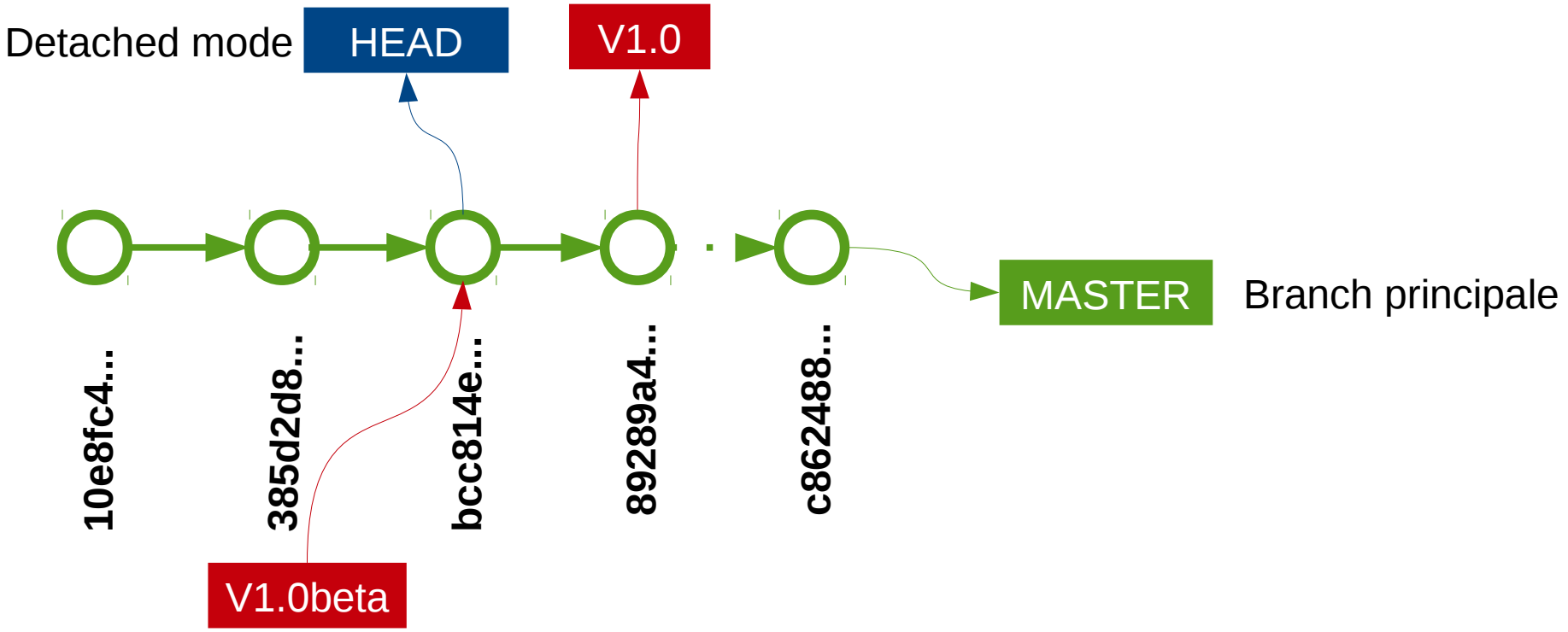
<git checkout V1.0>



Git

Etichettare i passi salienti (Tagging)

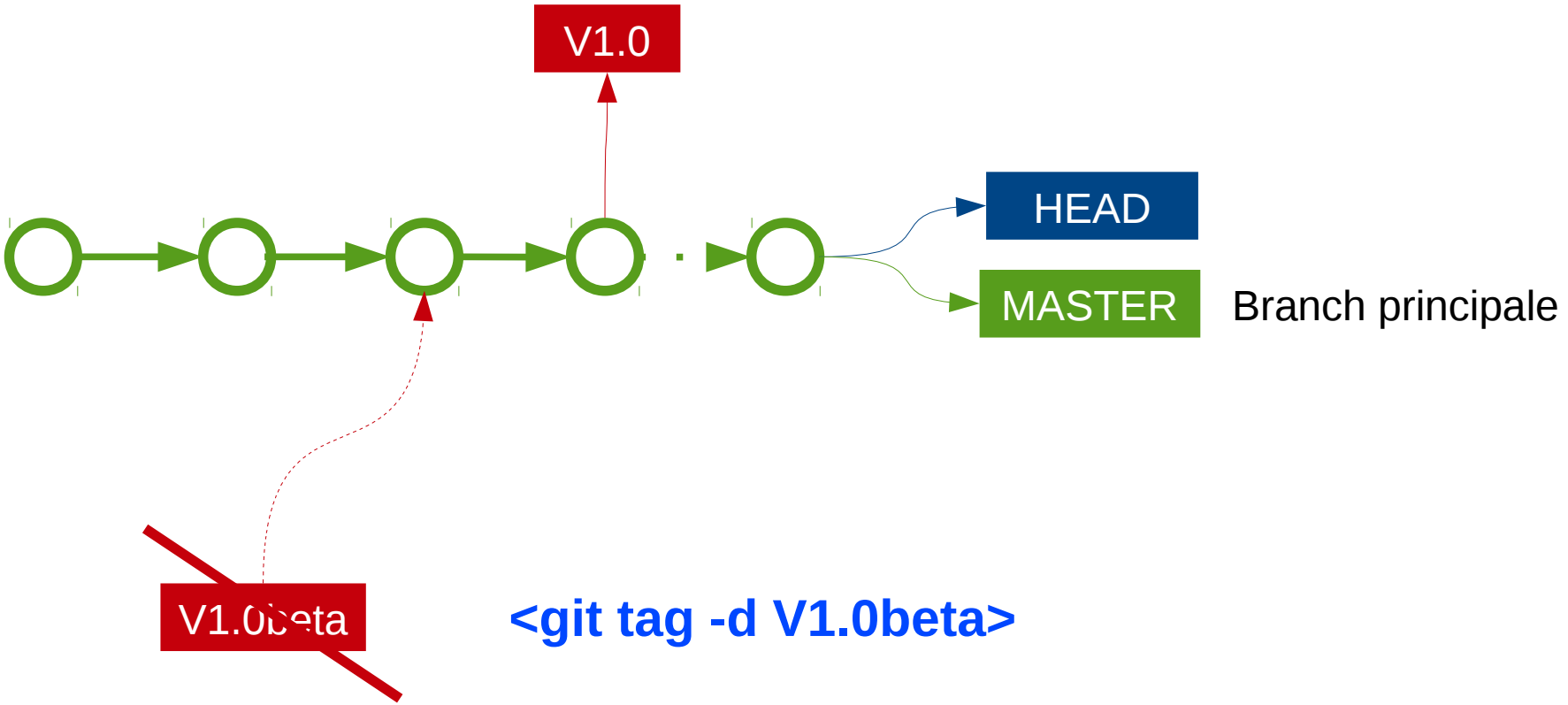
<git checkout V1.0^> (cerca il primo predecessore della commit indicata)



<git tag V1.0beta>

Git

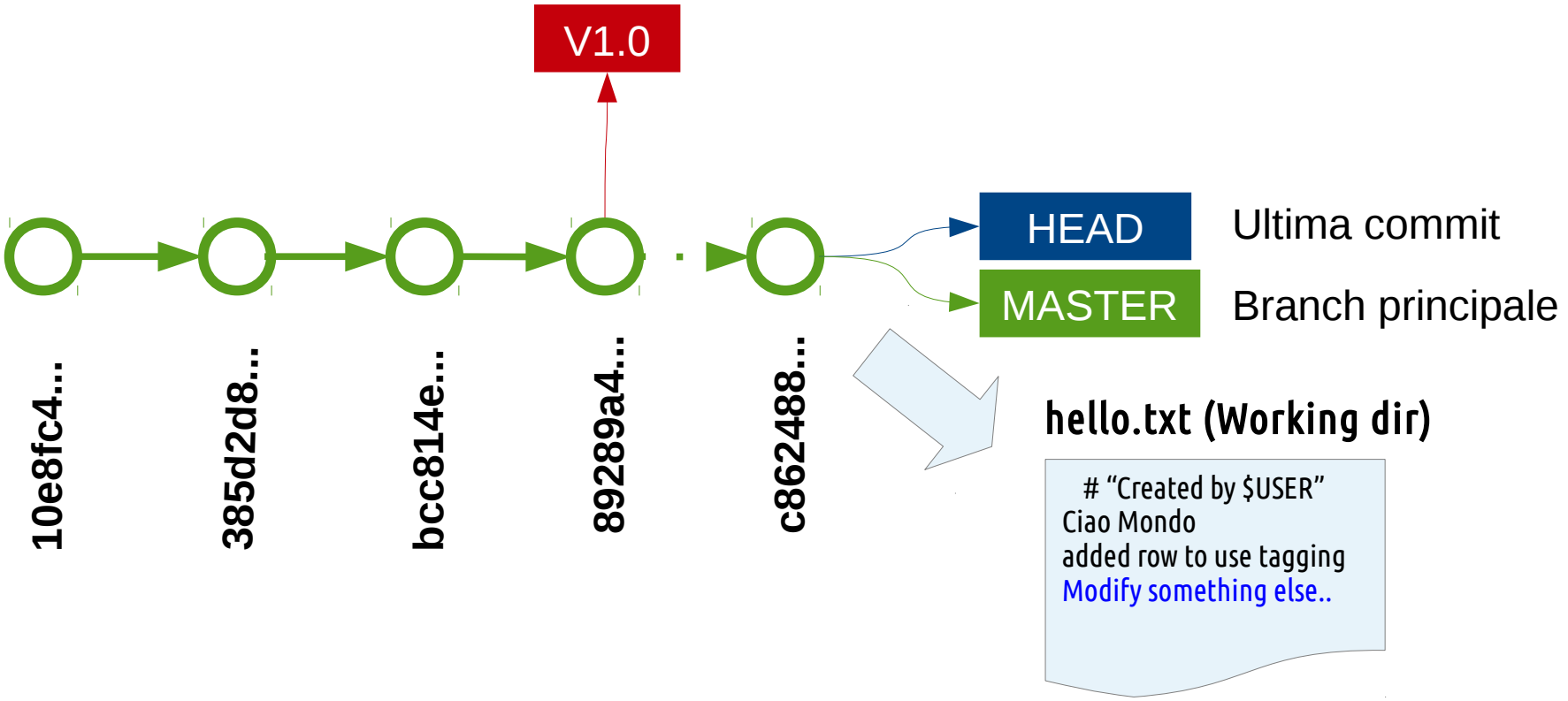
Cancellare una tag




```
$ git tag -d V1.0beta
Deleted tag 'V1.0beta' (was ......)
$
```

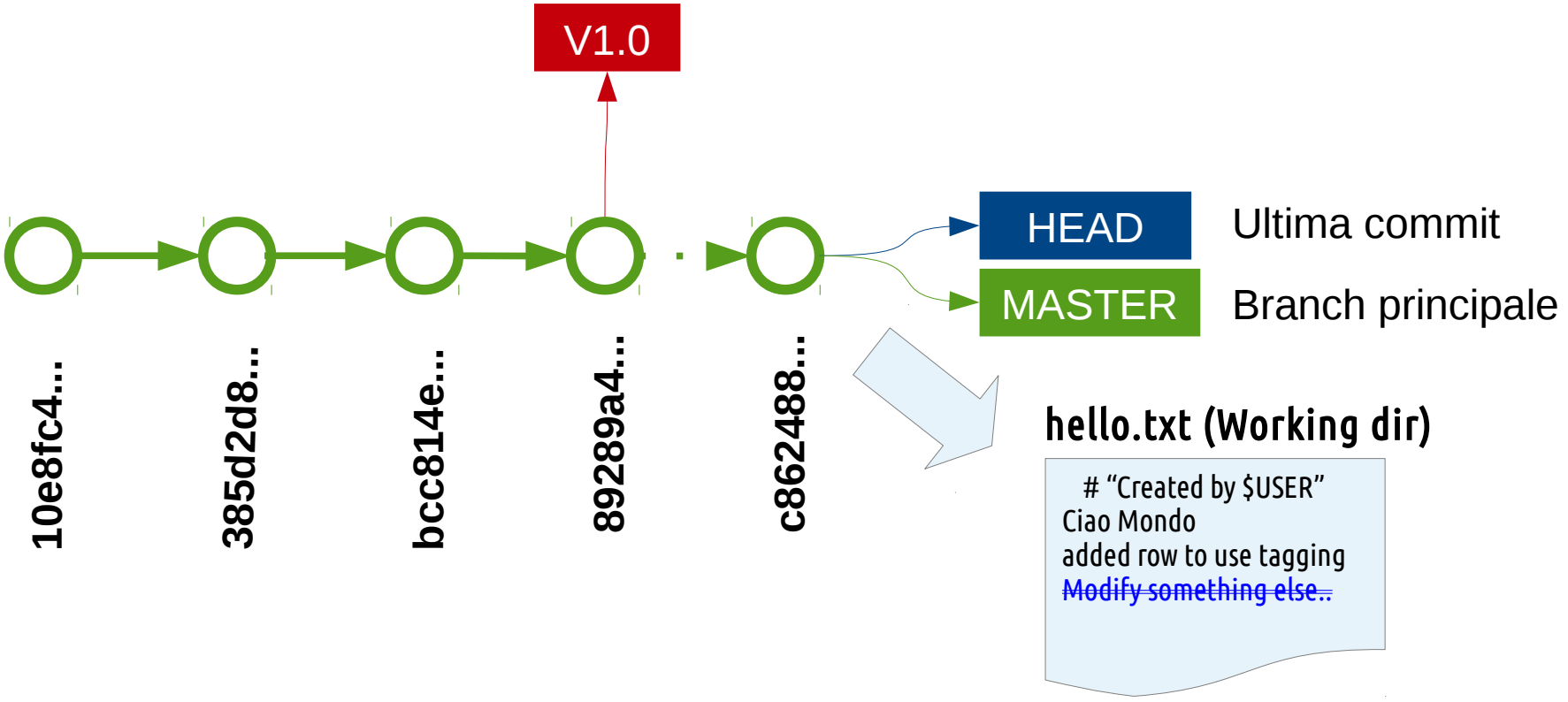
Git

Annullare le modifiche (prima dello staging)



Git

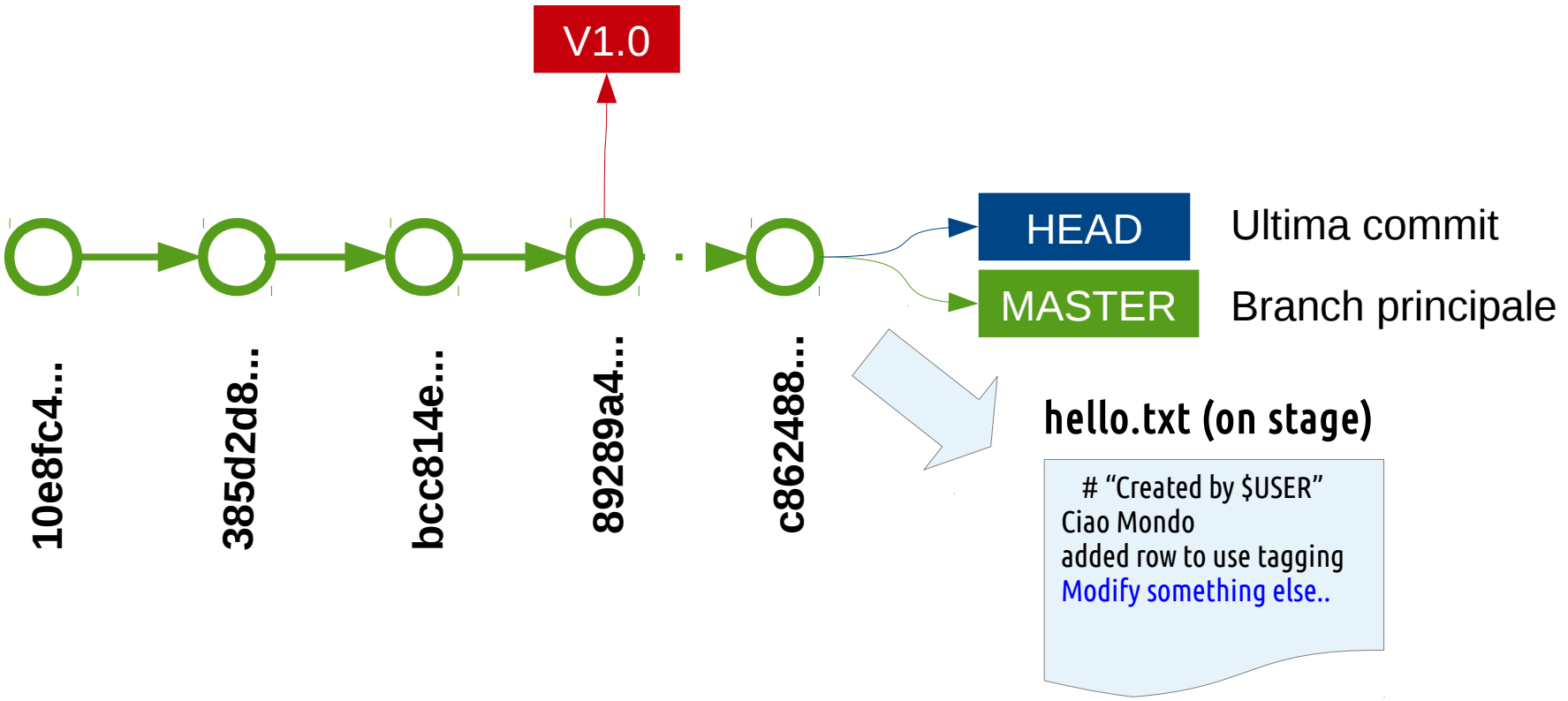
Annullare le modifiche (prima dello staging)



<git checkout hello.txt>

Git

Annullare le modifiche (on stage)

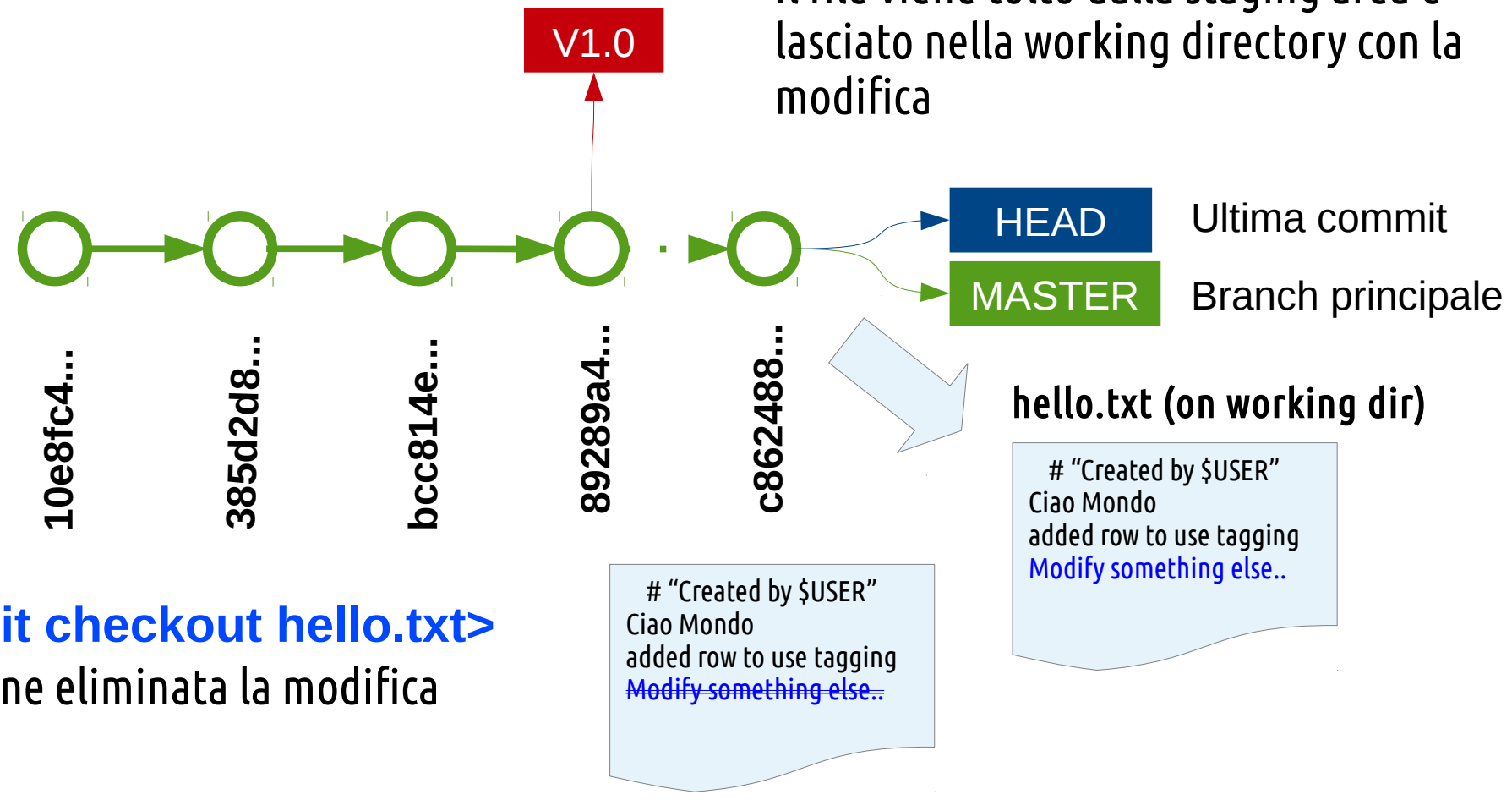


Git

Annullare le modifiche (on stage)

<git reset HEAD hello.txt>

Il file viene tolto dalla staging area e lasciato nella working directory con la modifica



<git checkout hello.txt>

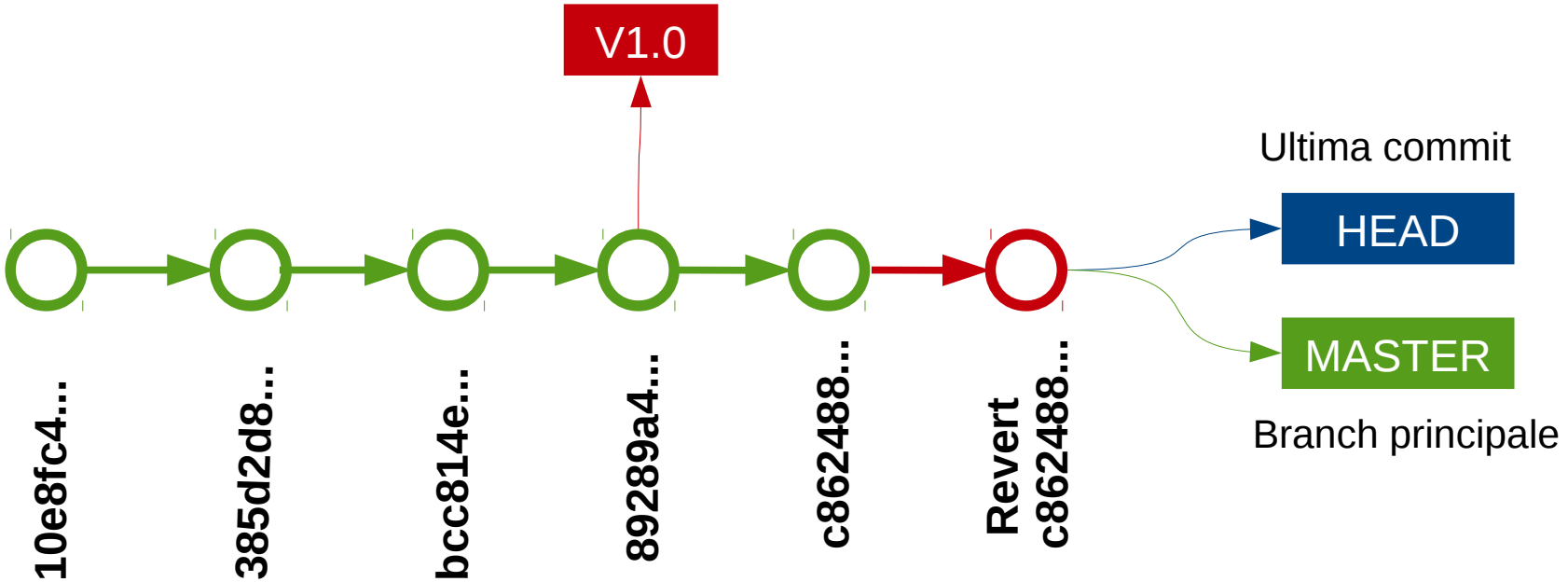
Viene eliminata la modifica

Git

Annullare una commit

<git revert HEAD>

Viene creata una **commit di 'rollback' che annulla gli effetti della commit annullata, che comunque resta**



Git

Eliminare una commit da un branch

Mi accorgo di aver fatto una commit che, per vari motivi, non dovrebbe comparire in assoluto, nemmeno come Revert (per esempio contiene dati sensibili...). In questi casi il comando da utilizzare è

$ git **reset <tag, hash>**

Che succede?

a) Riscrive il branch fino alla commit specificata

b) Reset della staging area, per renderla consistente con la commit specificata (opzione)

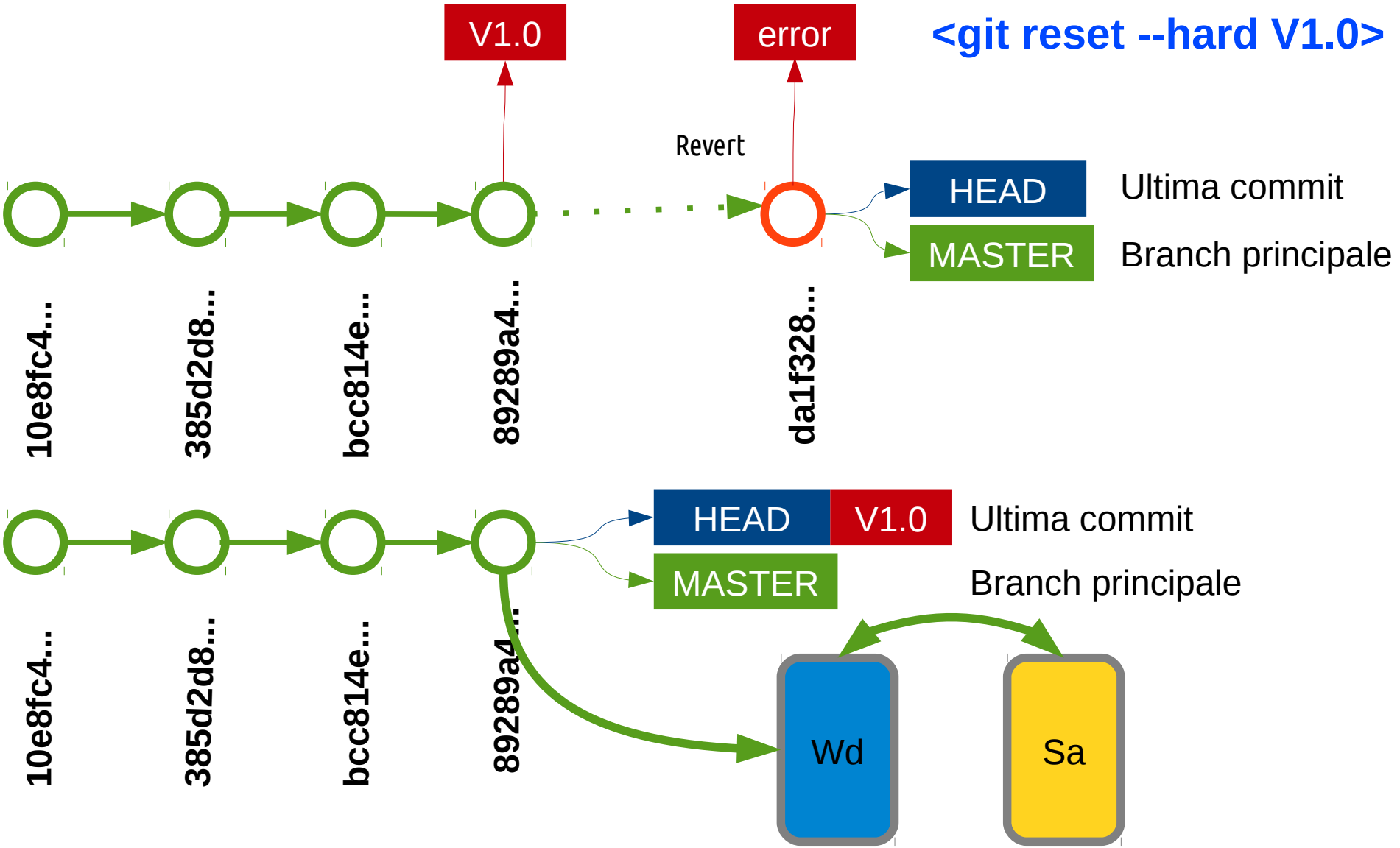
c) Reset della working directory, per lo stesso motivo (opzione)

I punti b,c diventano attivi se utilizziamo

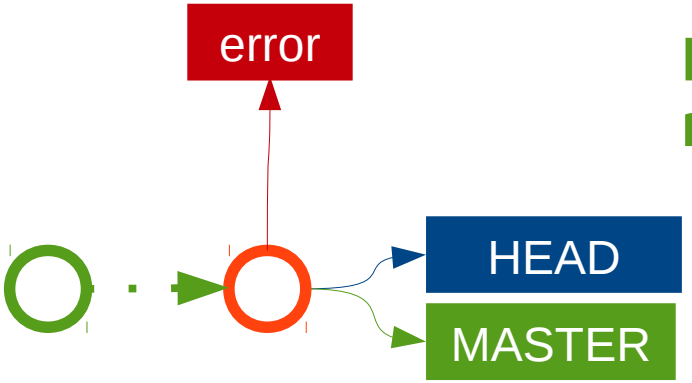
$ git **reset - -hard <tag, hash>**

Git

Eliminare una commit da un branch



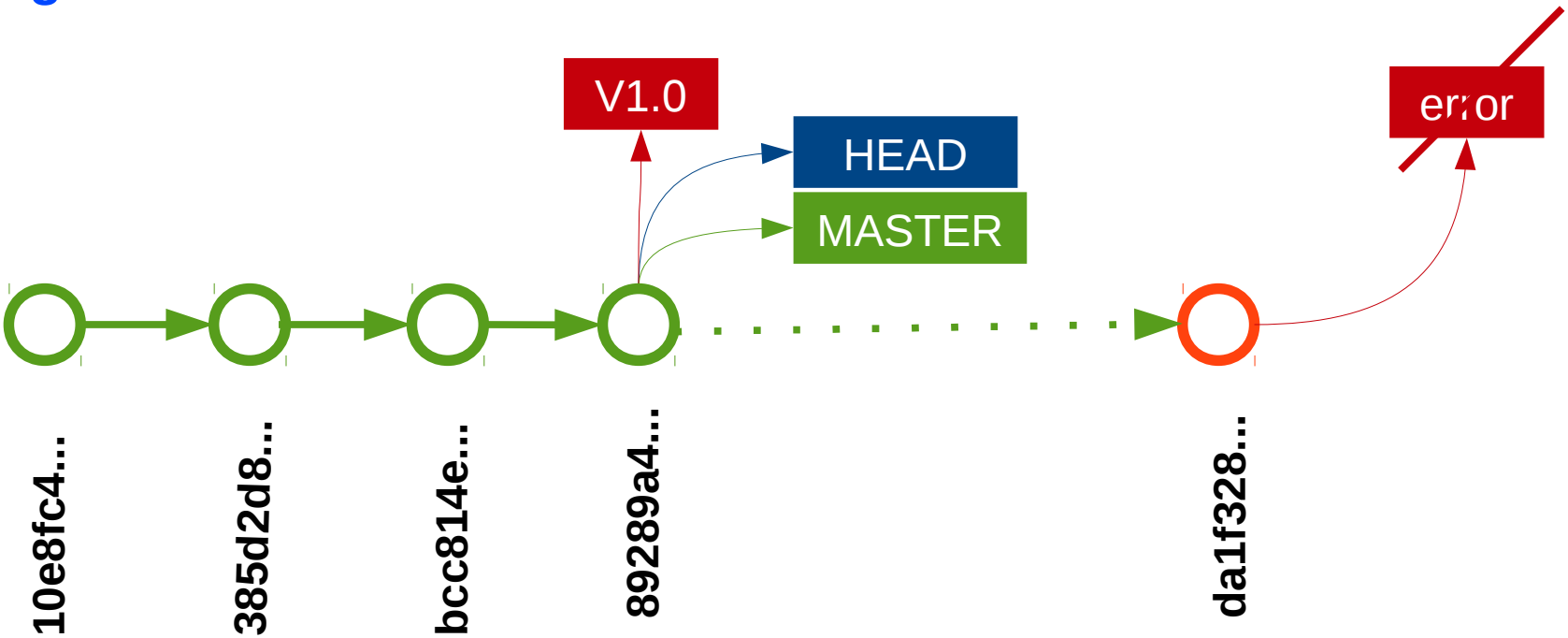
Git



IMPORTANTE: la commit eliminata ma etichettata non viene persa!

Se rimuovo la tag, perdo la commit per sempre

<git tag -d error>



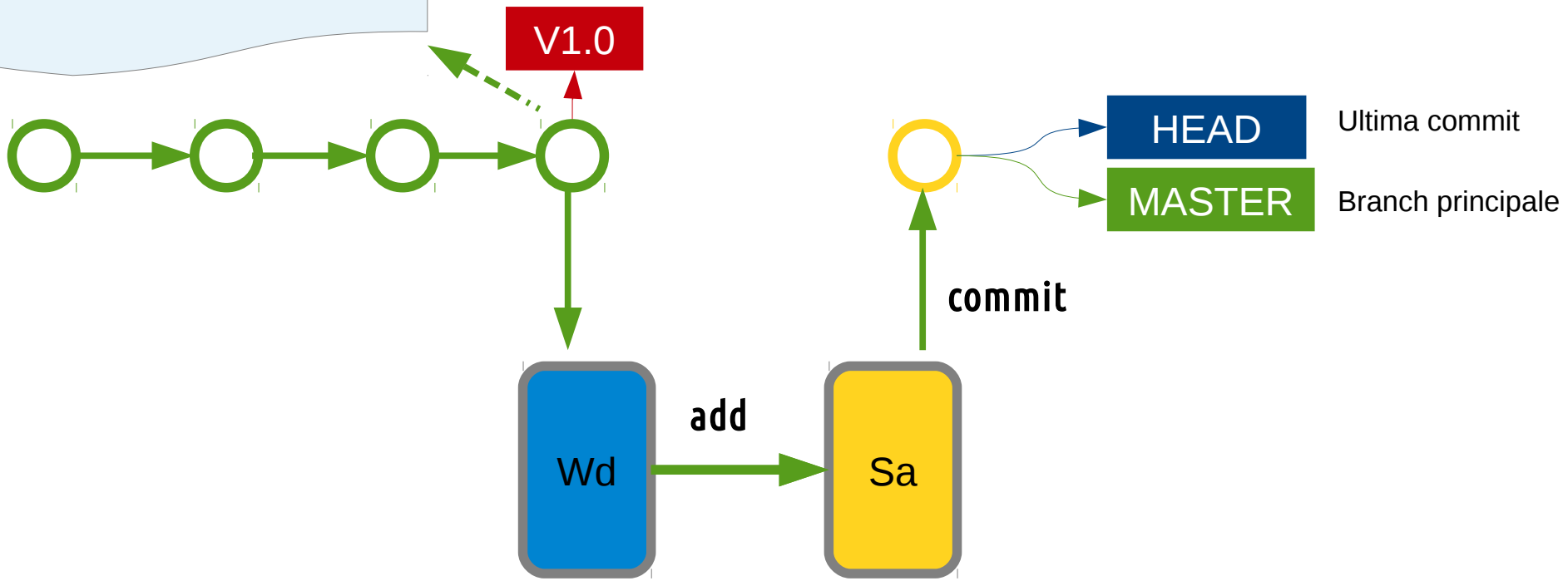
Git

Correggere l'ultima commit

Modifico un file, lo metto on stage (**add**), e poi nella commit area (**commit**)

hello.txt

```
# “Created by $USER” - role developer
Ciao Mondo
added row to use tagging
```

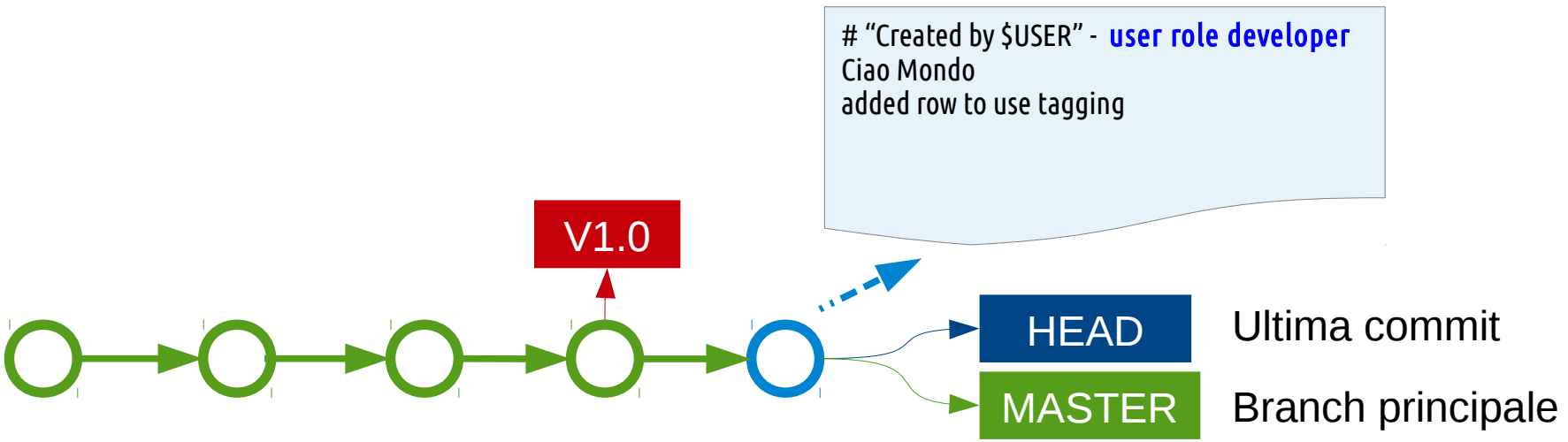


Git

Correggere l'ultima commit

Ho sbagliato: devo correggere il testo che ho aggiunto

hello.txt



Non voglio aggiungere una nuova commit (anche se potrei), ma piuttosto correggere l'ultima...

<git commit -- amend>

Git

Muovere files

Voglio muovere il file hello.txt in una nuova directory

```
$ mkdir newdir
$ git mv hello.txt newdir
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    hello.txt -> newdir/hello.txt
#
$
```

Git

Cancellare files

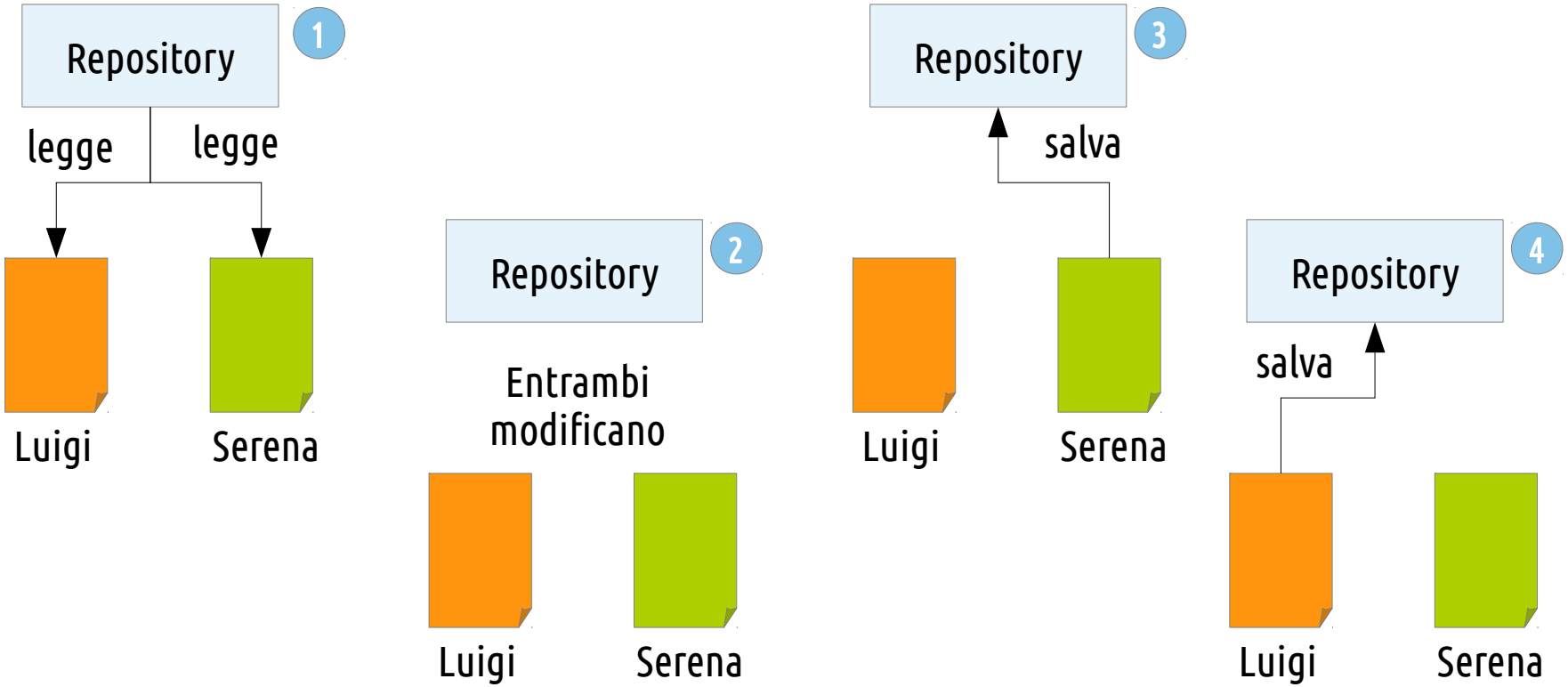
Voglio cancellare un file dal progetto ed eliminarlo dal controllo di versione

```
$ git rm file1.txt
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#     deleted:    file1.txt
#
$ $ git commit -m “cancellato file1.txt”
$
```

Git

Risorse condivise

La possibilità che più sviluppatori lavorino sullo stesso file è concreta, sopratutto per gruppi di lavoro estesi/dislocati. Cosa non deve mai succedere:

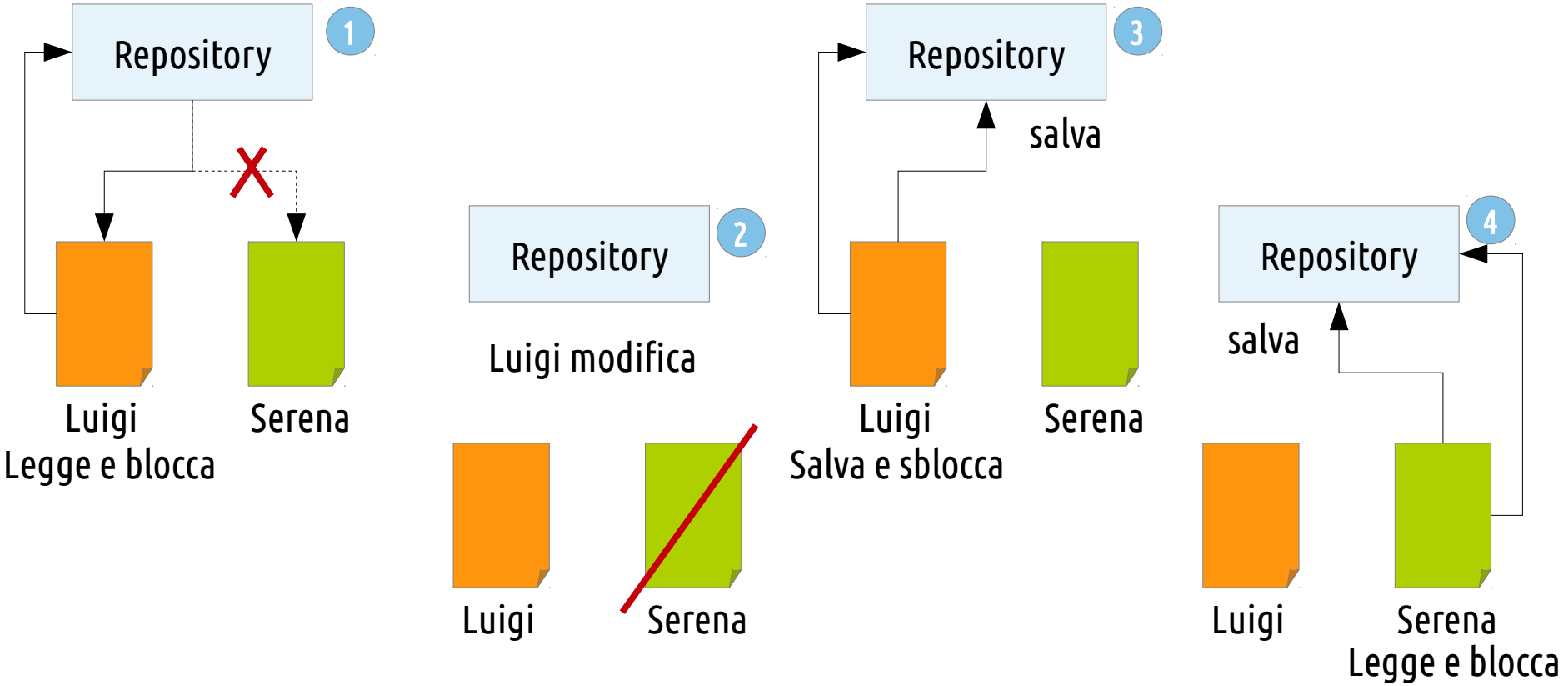


Luigi sovrascrive le modifiche fatte da Serena, e le annulla!!
DA NOTARE: in generale questo è possibile, **ma è molto più difficile che succeda con GIT (unità elementare = modifica).**

Git

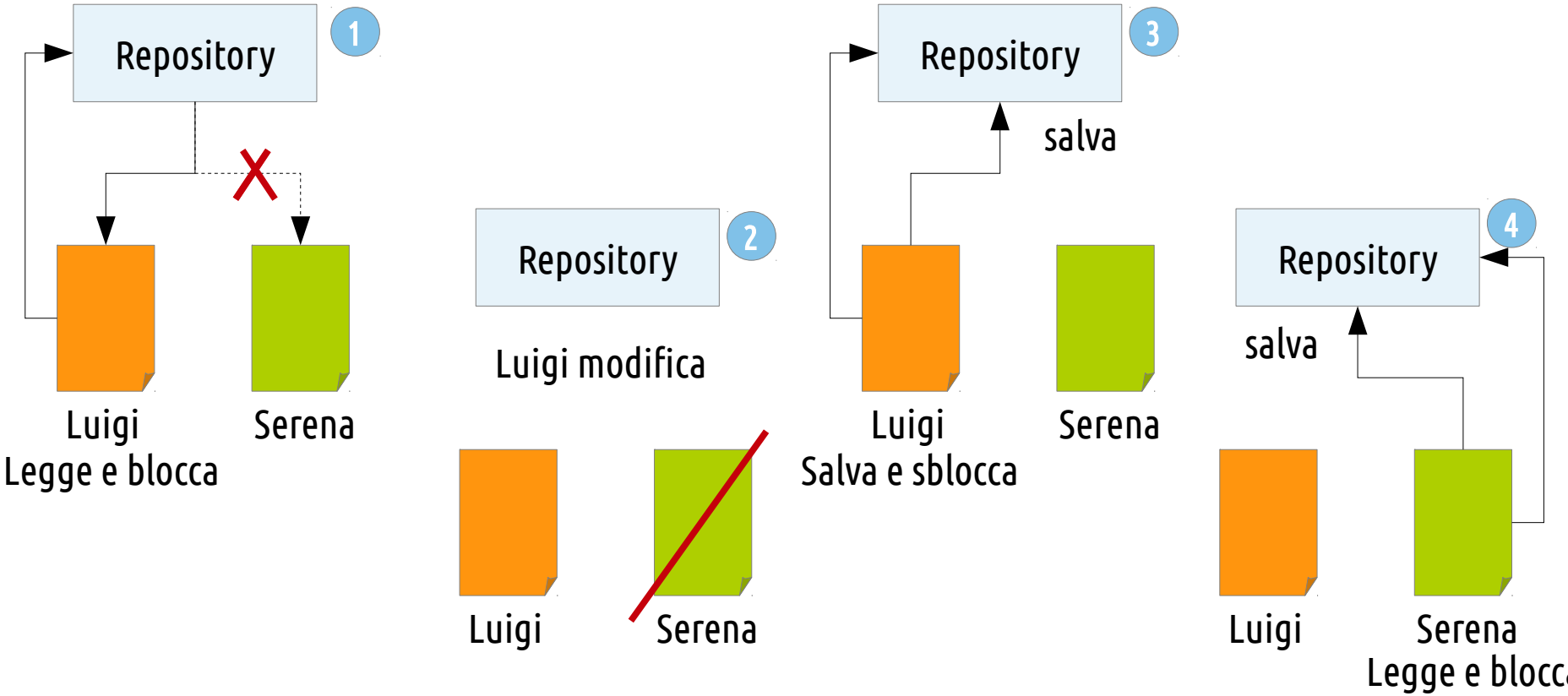
Risorse condivise

Il modello **lock-modify-unlock**: chi primo arriva blocca la lettura agli altri.
Quando ha finito sblocca.



Git

Risorse condivise



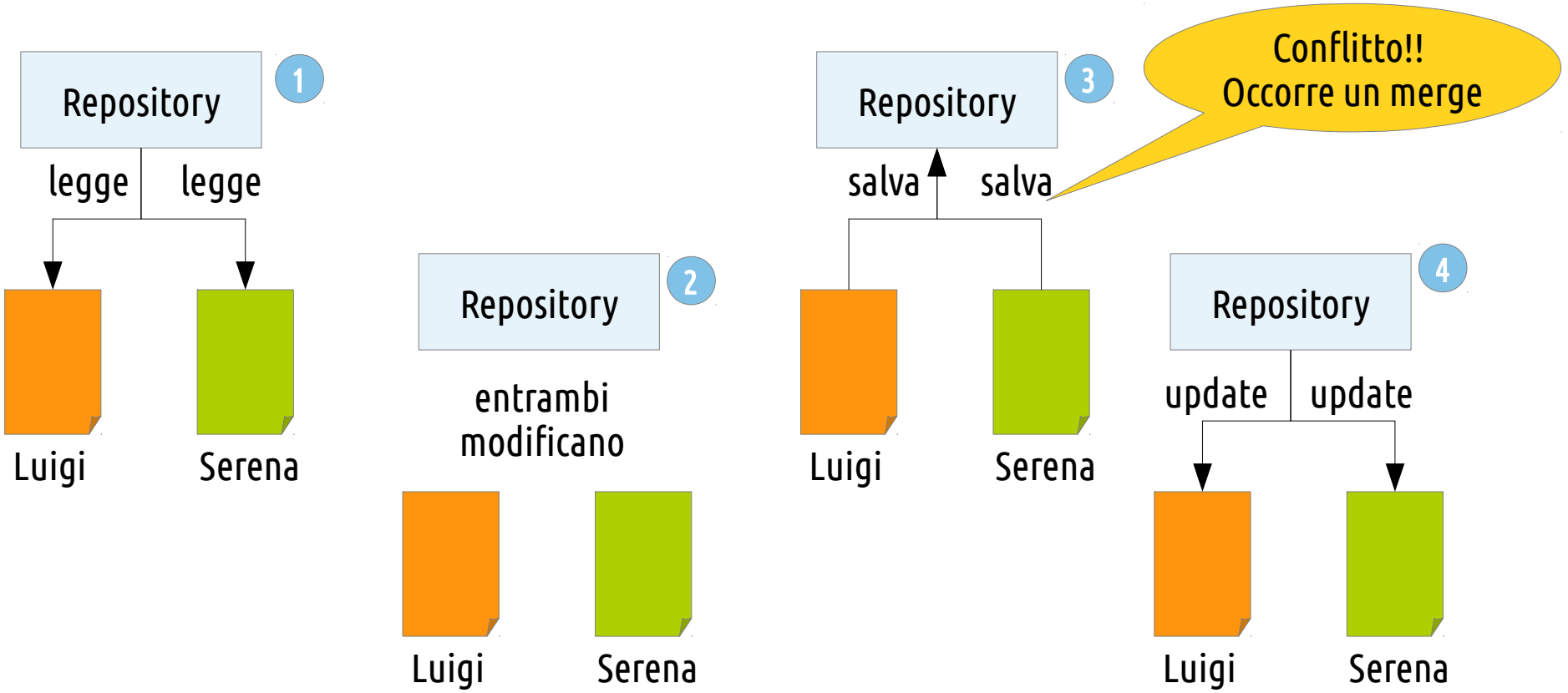
Il Locking può portare a problemi amministrativi. Mi sono dimenticato di togliere un lock ma ora sono in vacanza...

Il Locking può causare una serializzazione inutile: Se Luigi deve modificare l'inizio di un file e Serena la fine? Le modifiche non si sovrapporrebbero, ma...

Corso Git

Risorse condivise

Il modello **copy-modify-merge**: Tutti modificano. Nel caso di sovrapposizioni si attua un merge.



Il sistema di versioning è in grado di reagire di fronte ad una possibile sovrapposizione, e di attivare le successive fasi di merge.

Git

Risorse condivise

Riassumendo:

Il modello **lock-modify-unlock** è superato, e viene utilizzato solo in casi particolari.

Il modello più diffuso è **copy-modify-merge.**

Quando usare l'uno e quando l'altro?

Se un progetto è composto da file di testo, facilmente gestibili durante un conflitto ed il successivo merge, si usa il modello **copy-modify-merge.**

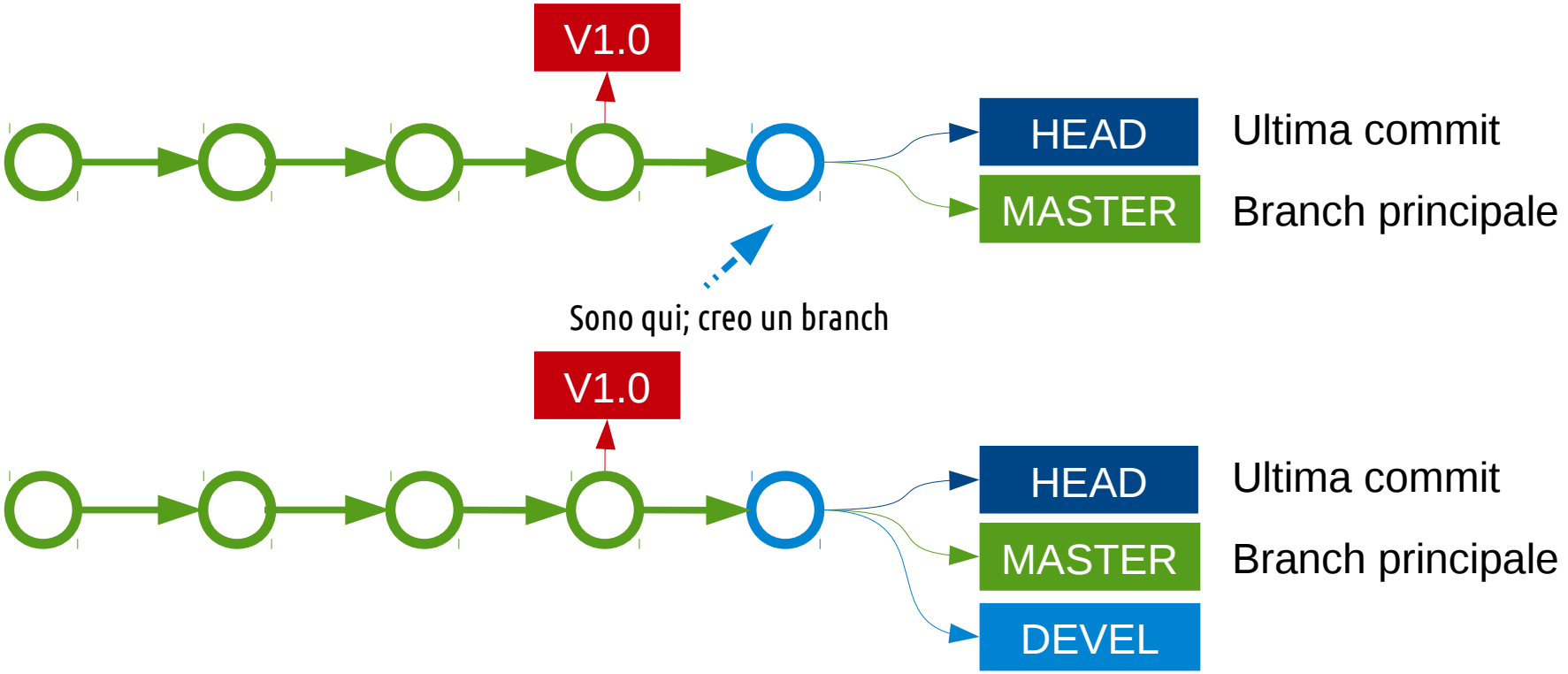
Se, al contrario, un progetto è composto di file binari di varia natura, con i quali è praticamente impossibile fare un merge, si utilizza il modello **lock-modify-unlock**. Come esempio, immaginiamo di gestire il merge tra due file audio/video, due fotografie, due disegni, ecc.

Git

Utilizzare i branches

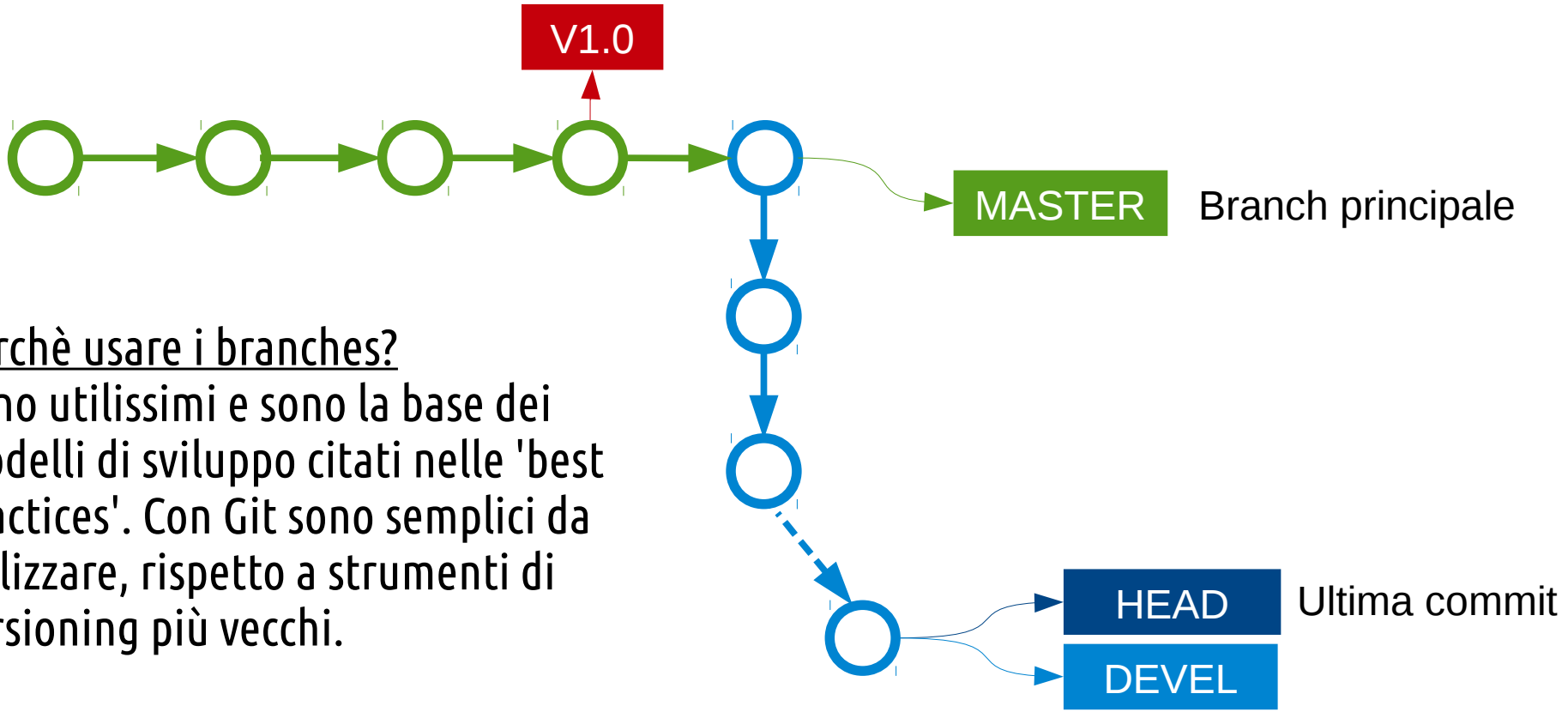
Cos'è un branch?

Un branch in Git è semplicemente un puntatore mobile ad una commit. Quando creo un nuovo branch, creo solo un puntatore ad una commit esistente. Se poi mi posiziono su quel puntatore, e lavorando creo nuove commit, creo un nuovo ramo di sviluppo.



Git

Utilizzare i branches



<u>Perchè usare i branches?</u>
Sono utilissimi e sono la base dei modelli di sviluppo citati nelle 'best practices'. Con Git sono semplici da utilizzare, rispetto a strumenti di versioning più vecchi.

Git

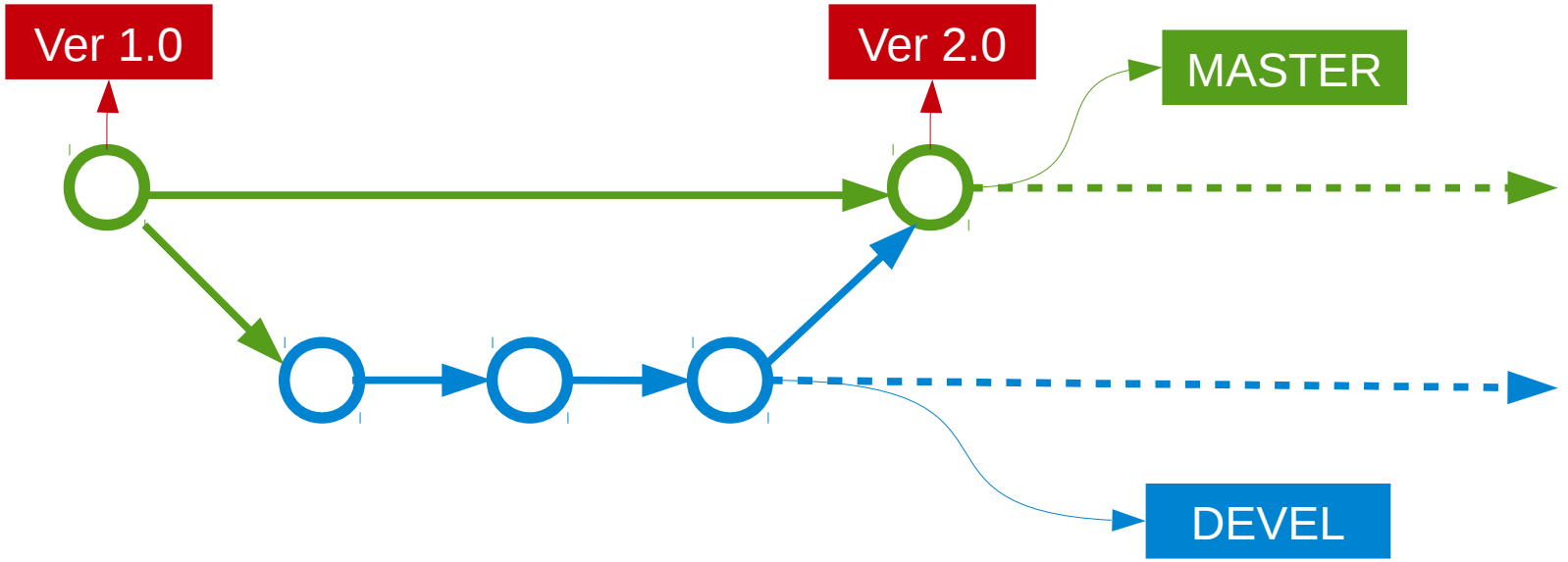
Utilizzare i branches

Flusso normale di sviluppo

Un progetto software in fase di sviluppo mantiene sempre almeno due 'cantieri' aperti:

PRODUZIONE il ramo che genera le release da distribuire (master)

DEVEL Il ramo che contiene i nuovi sviluppi

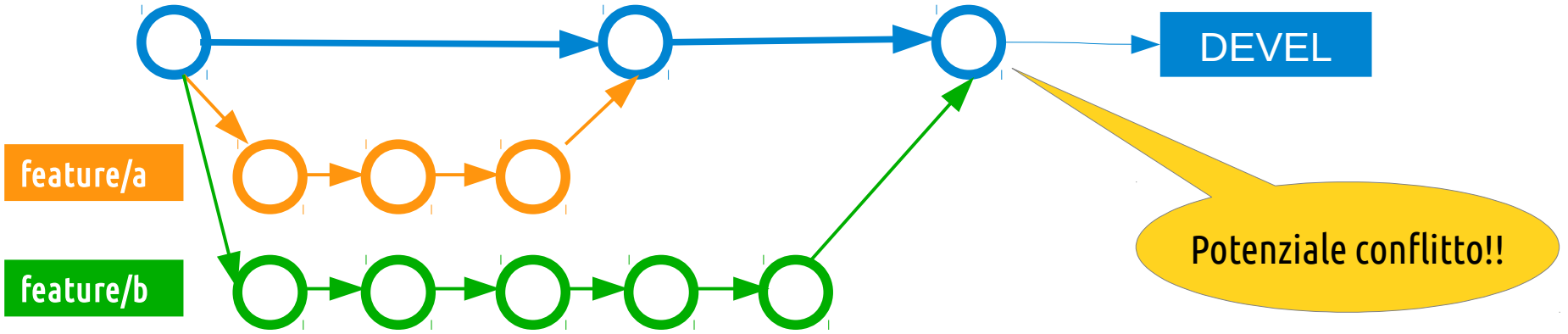


Git

Utilizzare i branches

Feature branch

- Branch di sviluppo di una nuova funzionalità
- Ramifica a partire da **devel** e si innesta su **devel**
- a fine sviluppo viene cancellato

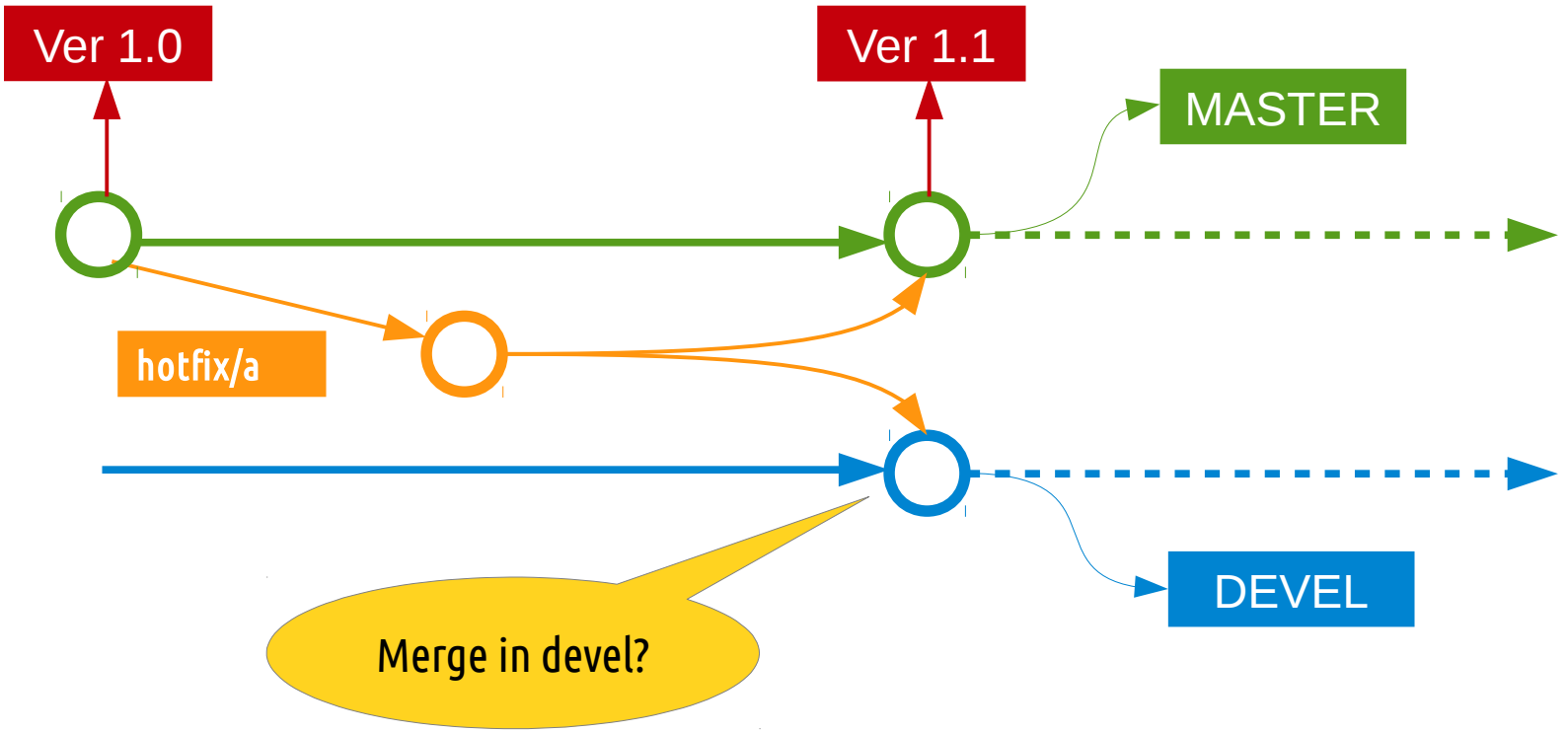


Git

Utilizzare i branches

Hotfix branch

- branch di fix per un bug in produzione
- ramifica da master (produzione) e si innesta su master
- a fine sviluppo viene cancellato (può essere necessario un merge su devel)

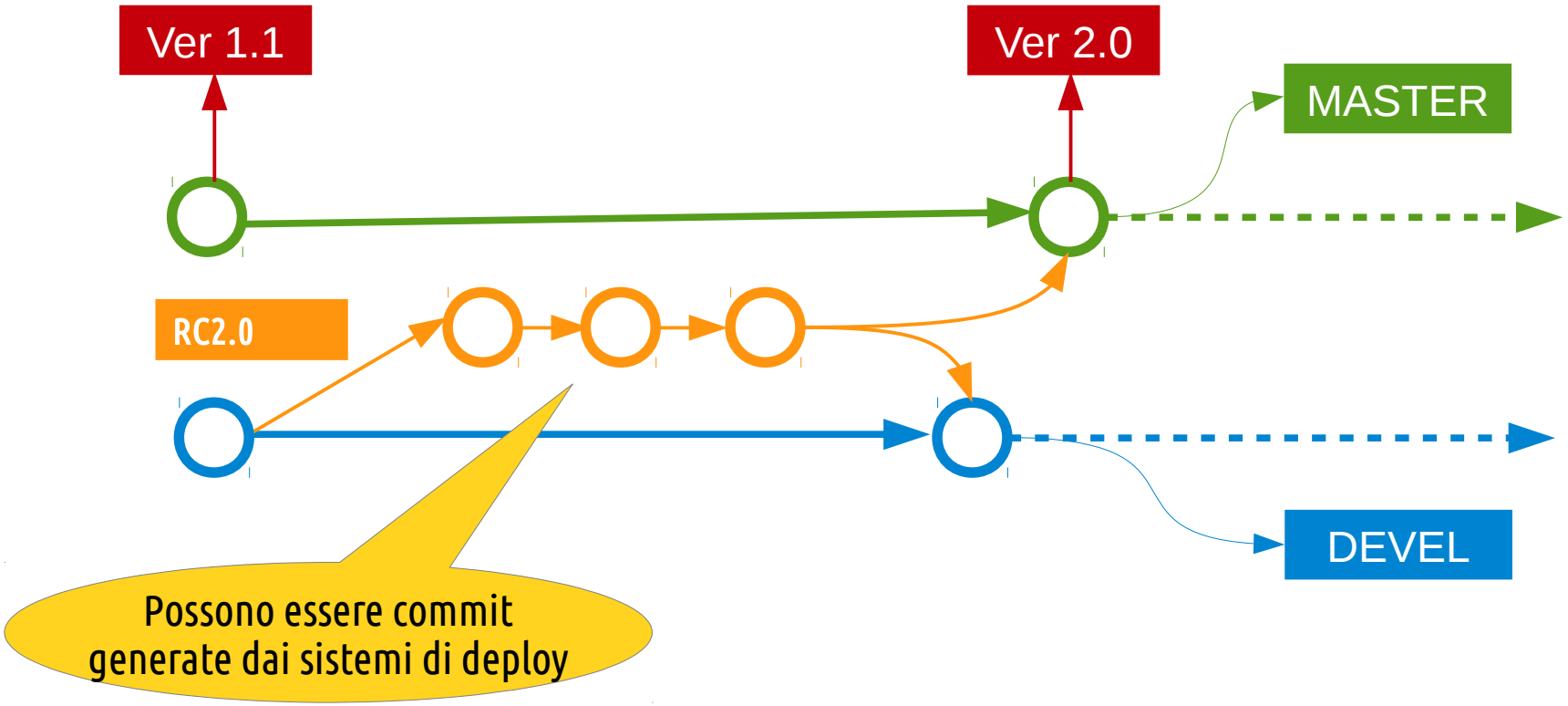


Git

Utilizzare i branches

Release branch

- candidata in test per la prossima versione di produzione
- ramifica da devel e si innesta su master
- viene generalmente “taggata” con un numero di versione



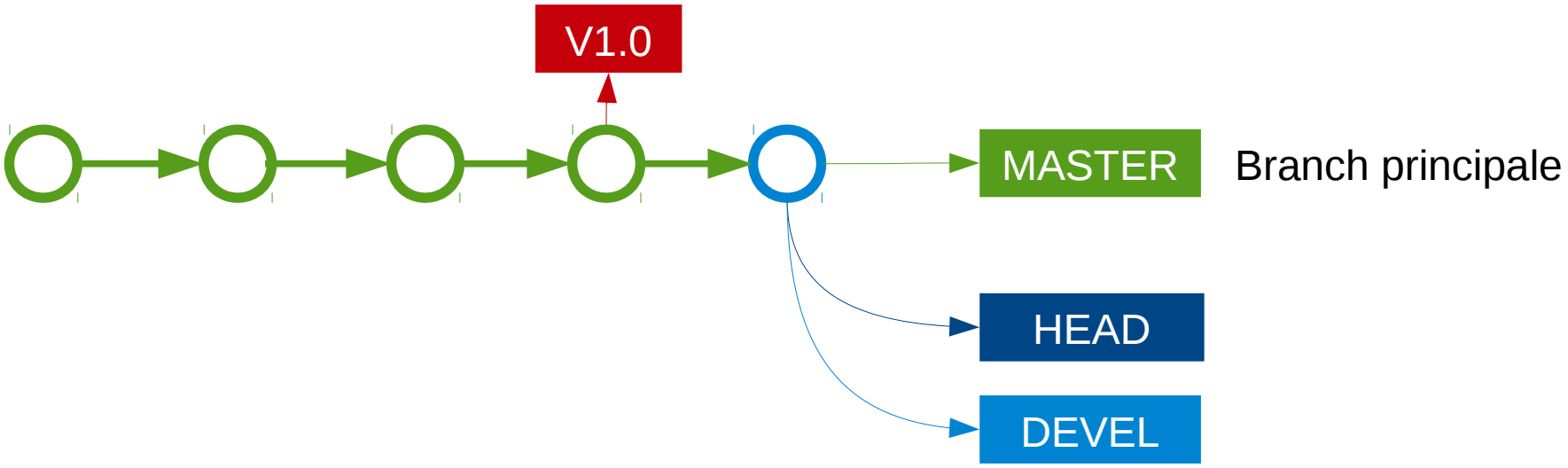
Git

Creare un branch

Supponiamo di partire dal branch MASTER, ultima commit (HEAD)

<git branch DEVEL>
<git checkout DEVEL>

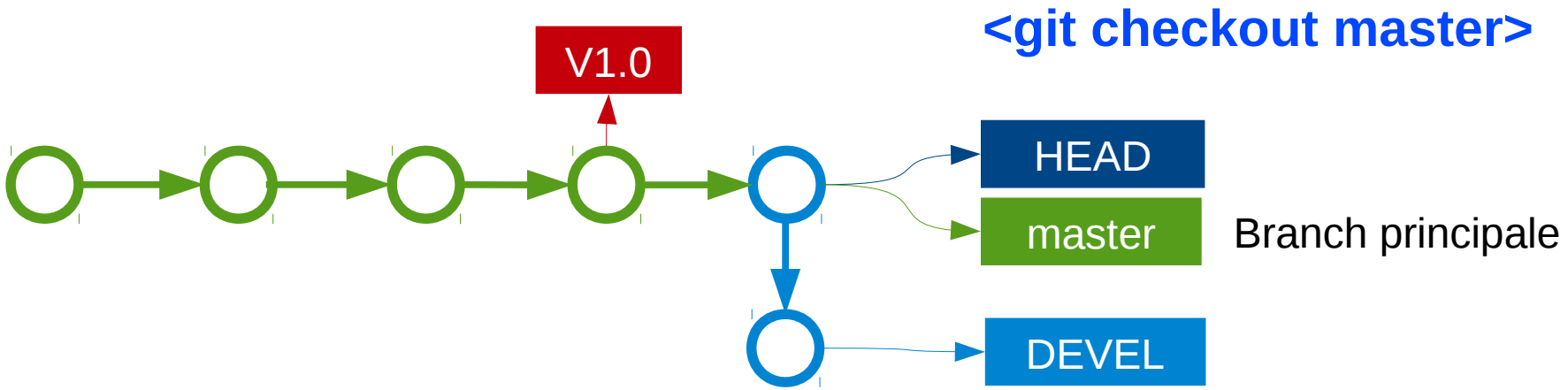
Più brevemente:
<git checkout -b DEVEL>

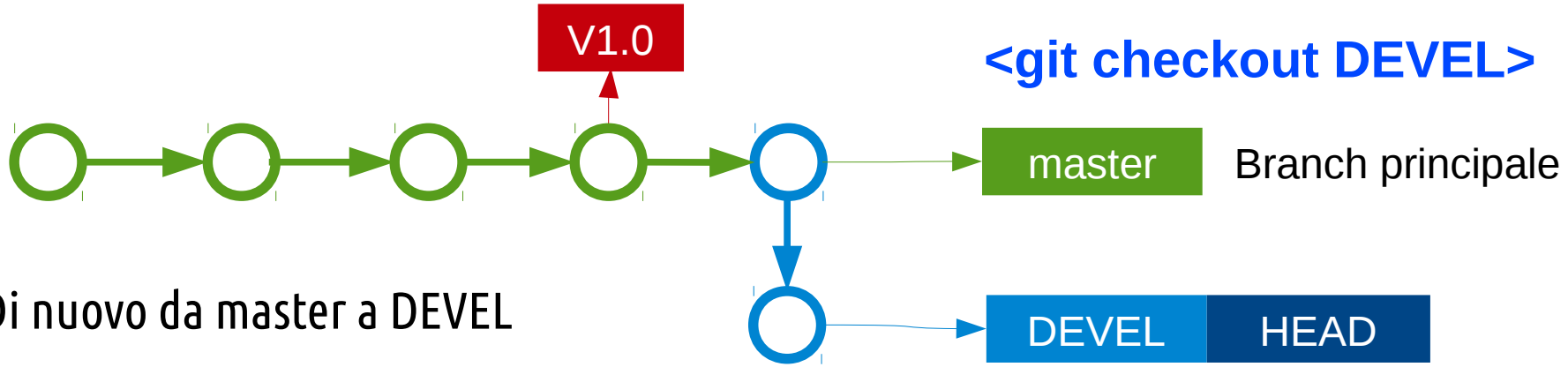


Git

Navighiamo tra i branches

Siamo nel branch DEVEL e vogliamo passare al branch master:





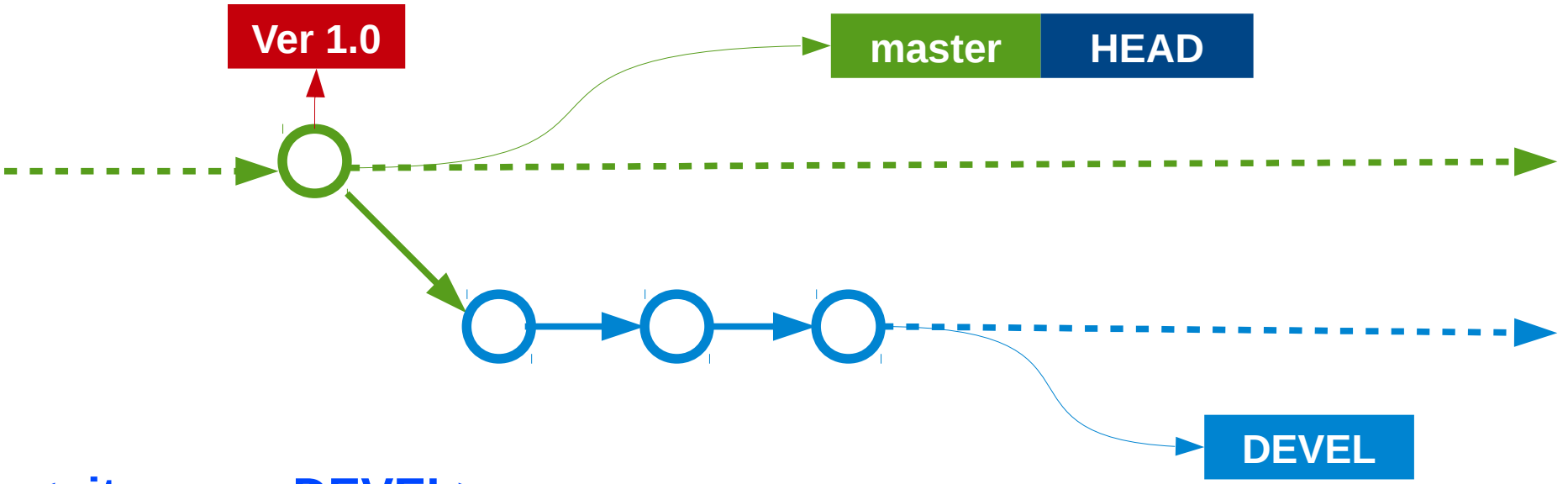
Di nuovo da master a DEVEL

Git

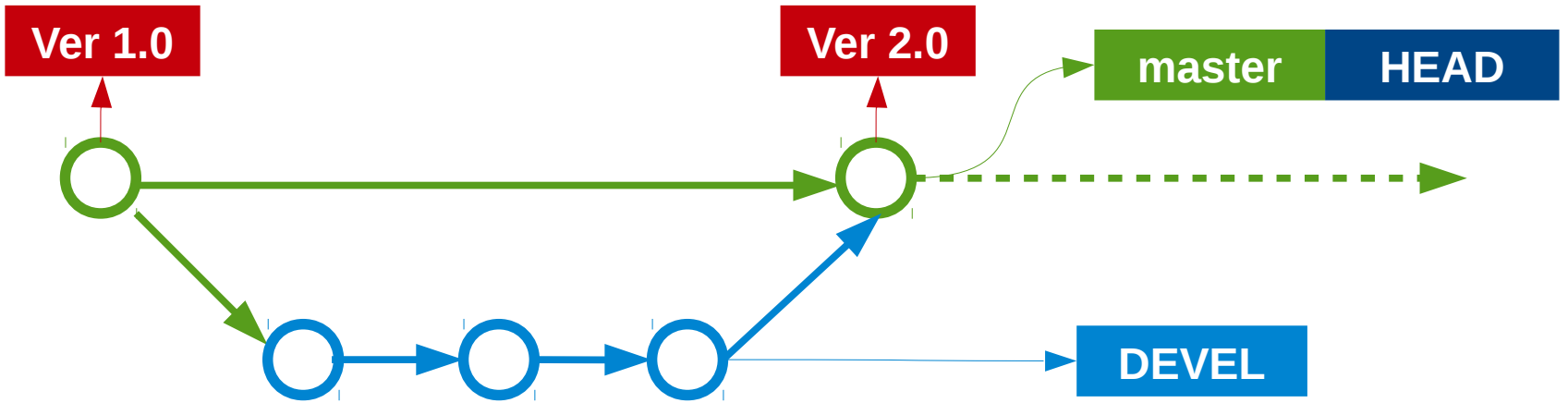
Merging

Siamo al punto di far passare il lavoro svolto da un ramo ad un altro

<git checkout master>



<git merge DEVEL>

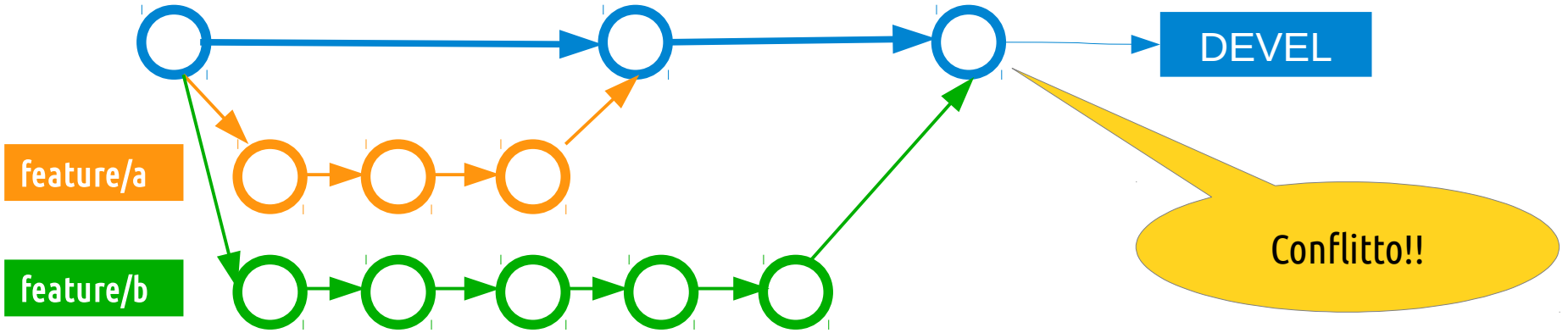


Git

Merging

E se ci fossero dei conflitti? Per esempio, se un file venisse modificato in entrambi i branches?

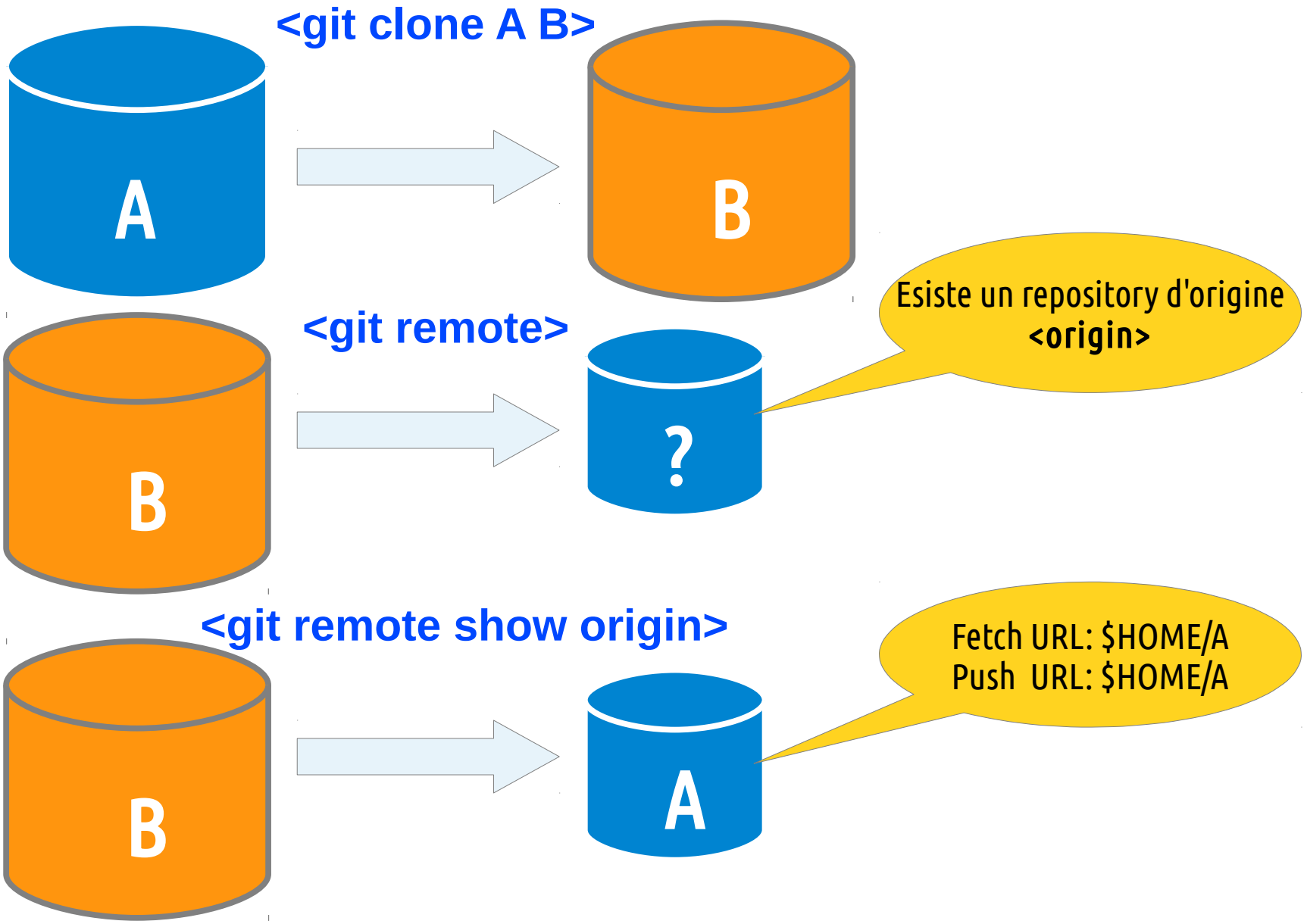
<git merge DEVEL>



In questo caso dovremmo risolvere manualmente il conflitto e poi fare una commit. Questo succede con qualsiasi sistema di versioning, utilizzando il modello **copy-modify-merge**.

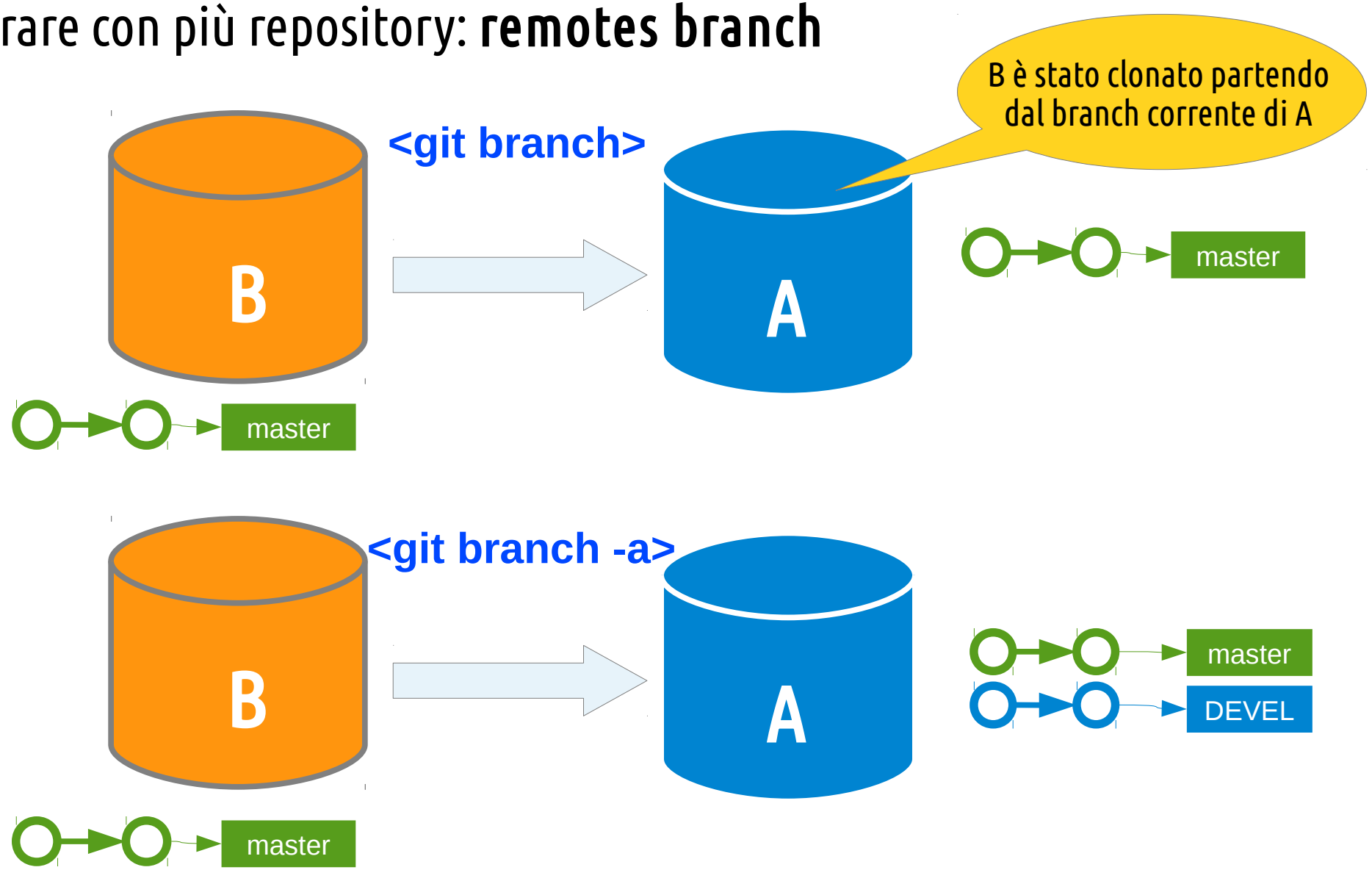
Git

Lavorare con più repository: **clone**



Git

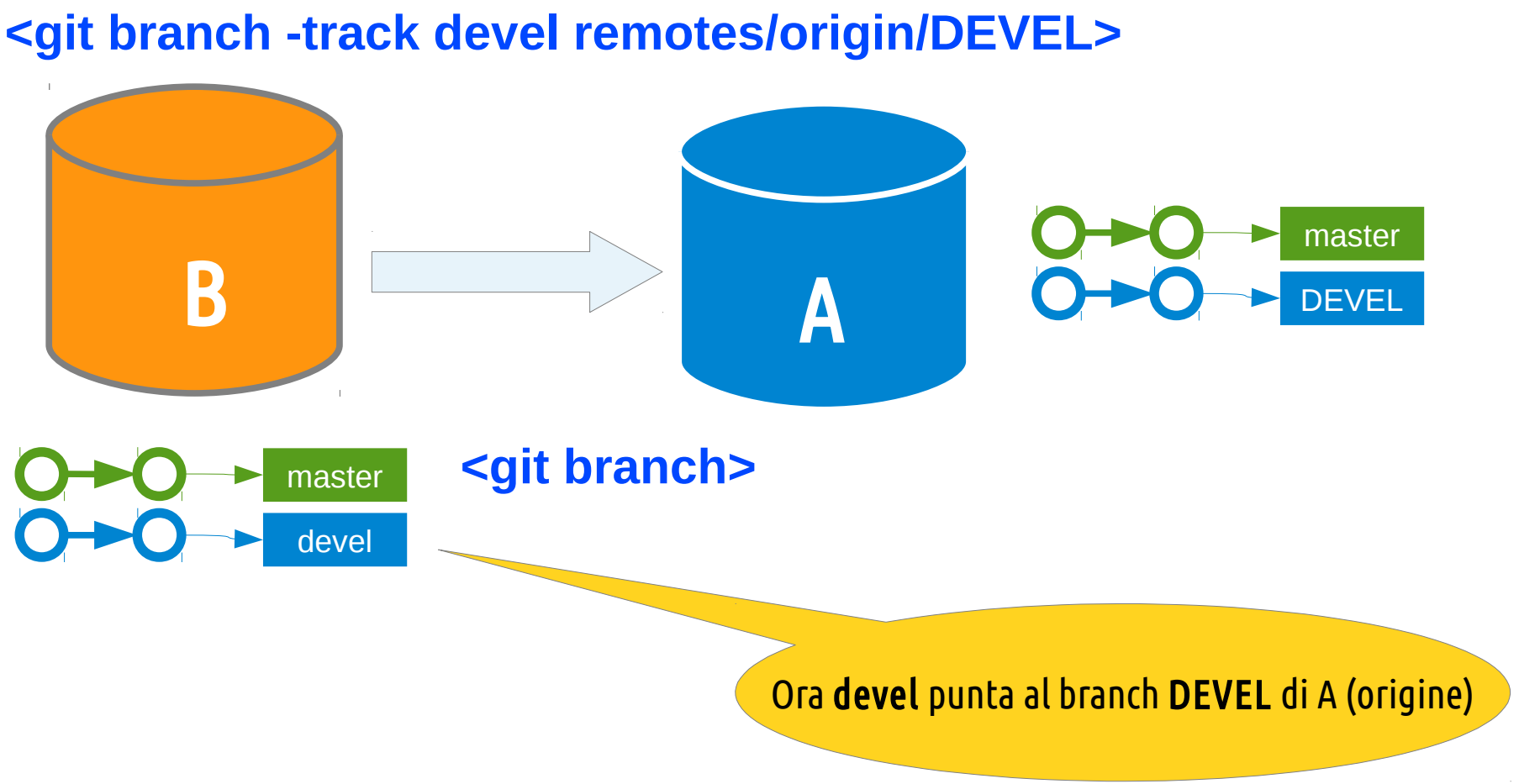
Lavorare con più repository: **remotes branch**



DEVEL esiste nel repository originale, ma deve essere dichiarato per essere utilizzato in B

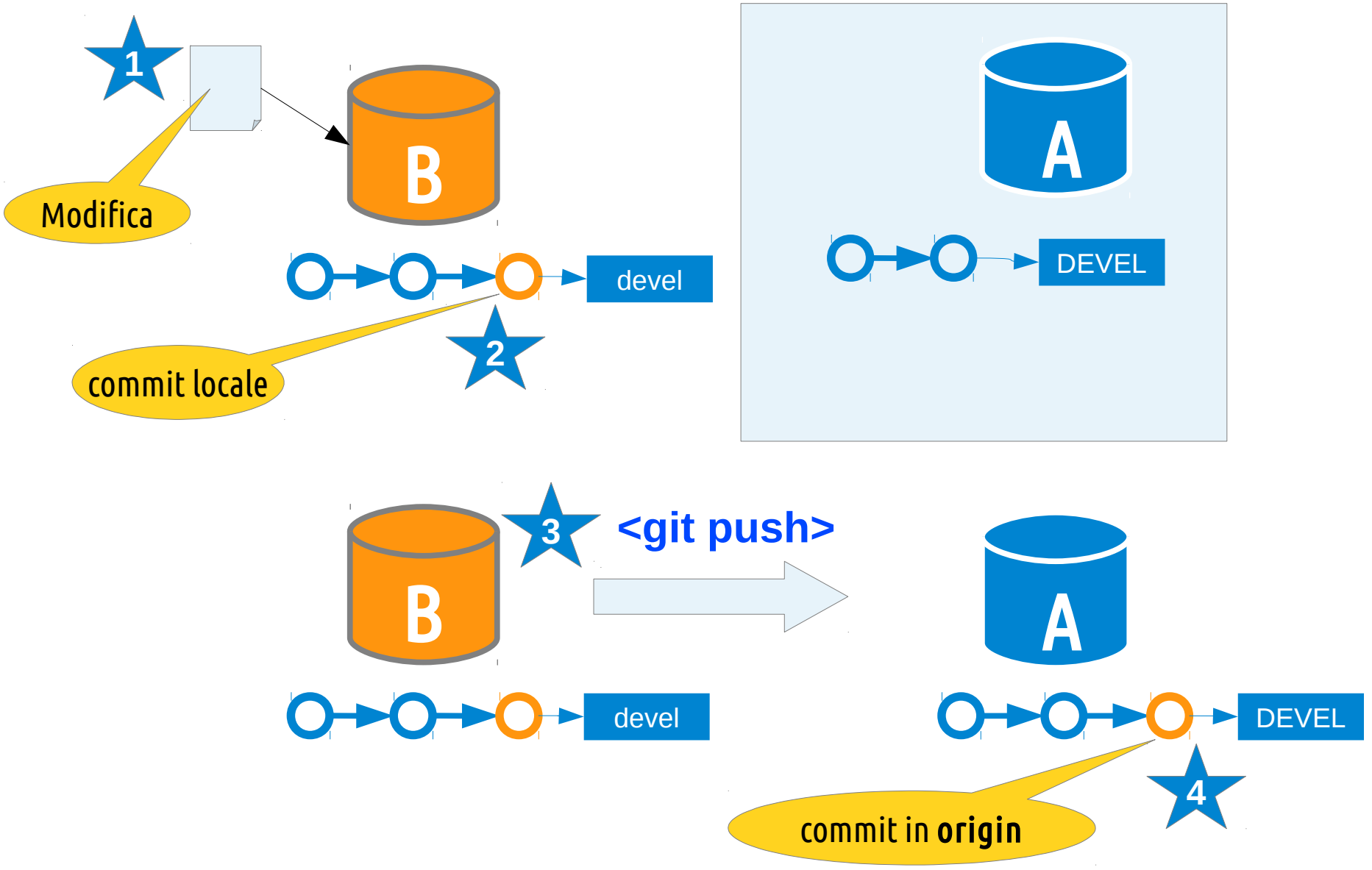
Git

Lavorare con più repository: **remotes branch**



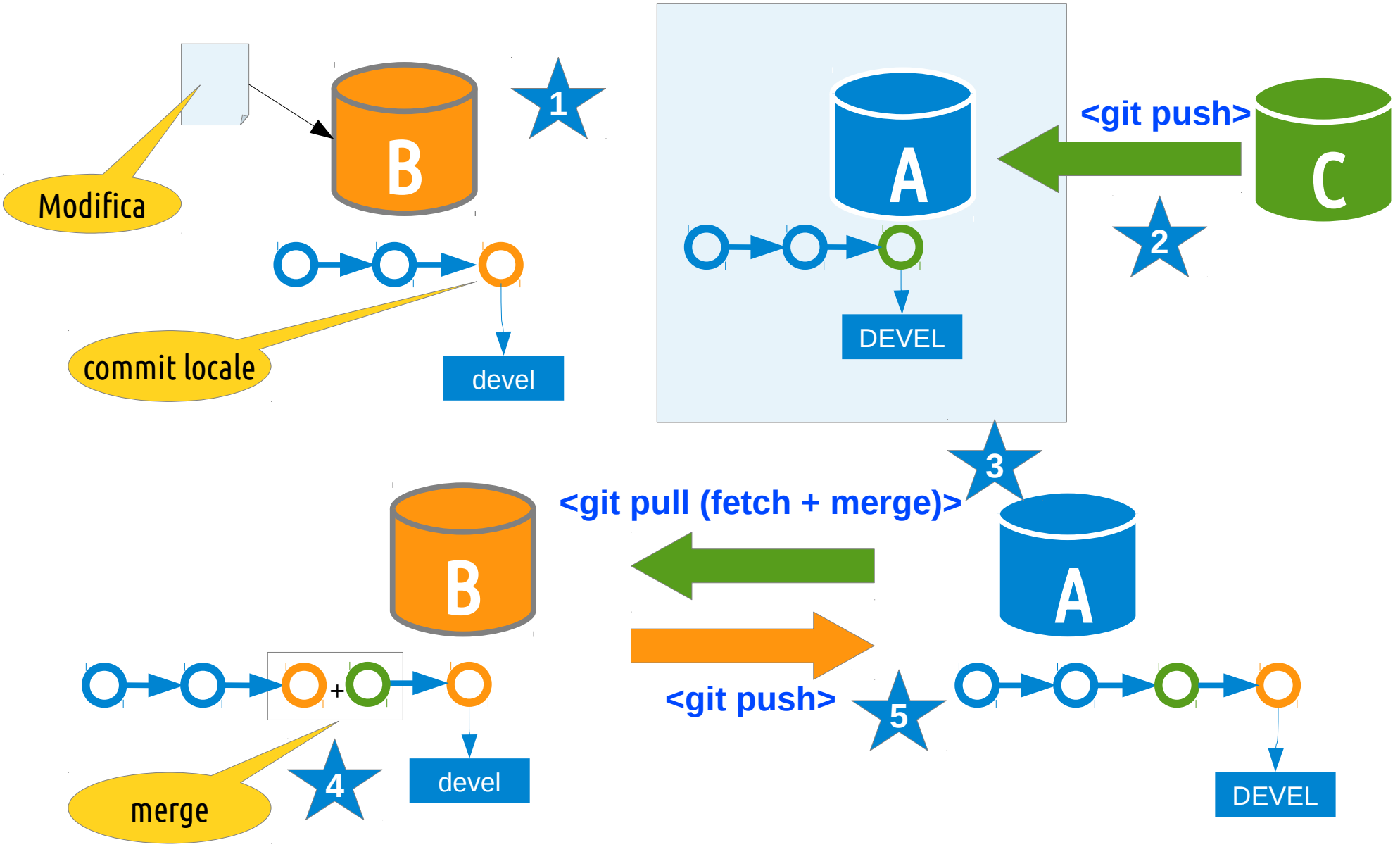
Git

Lavorare con più repository: **push**



Git

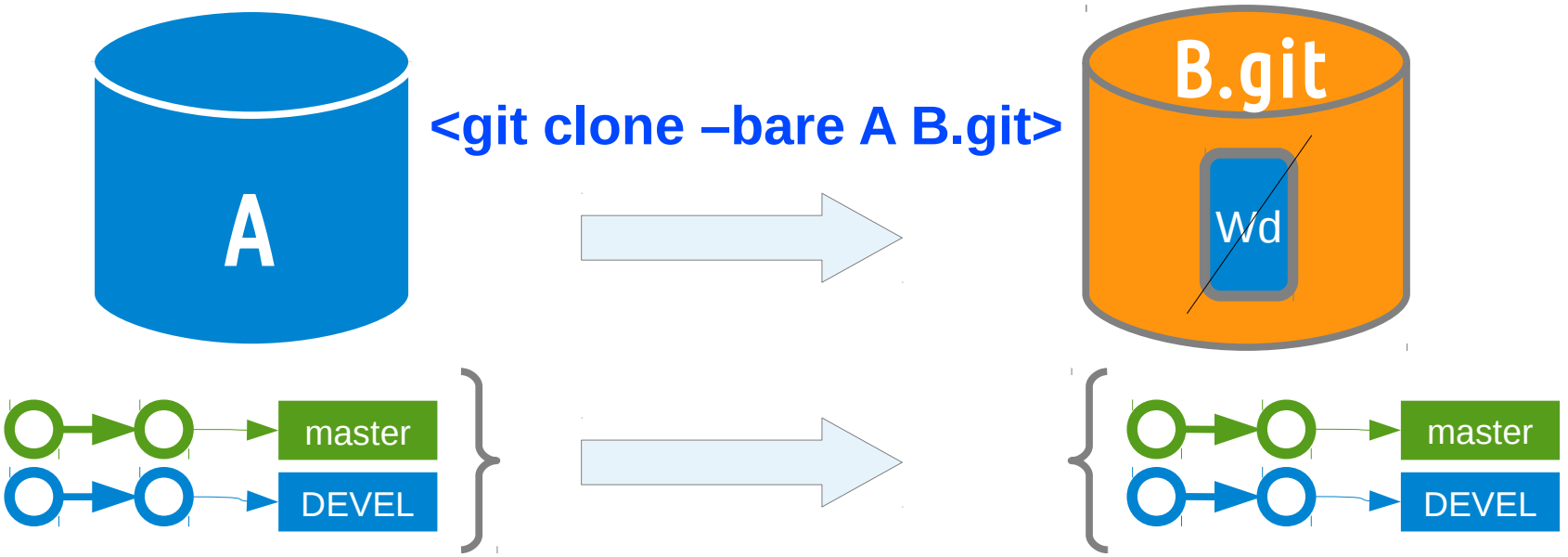
Lavorare con più repository: **fetch, pull**



Git

Lavorare con più repository: **bare**

I repository **bare** (nudo) sono repository che servono solo per la condivisione, e **non hanno la working directory**. I repository creati con questa opzione si denotano convenzionalmente con la desinenza **.git**



Da notare il fatto che un repository clonato con questa opzione **ha tutti i branches** del repository padre

Git

Repository remoti

Tutto quello che abbiamo visto finora ci fa pensare che un repository remoto (in un altro network) non sia altro che un repository **bare,** al quale abbiamo accesso e che possiamo clonare per iniziare le nostre attività.

Ci sono diversi repository managers per Git, in Cloud o installabili in un local network. Un esempio è:

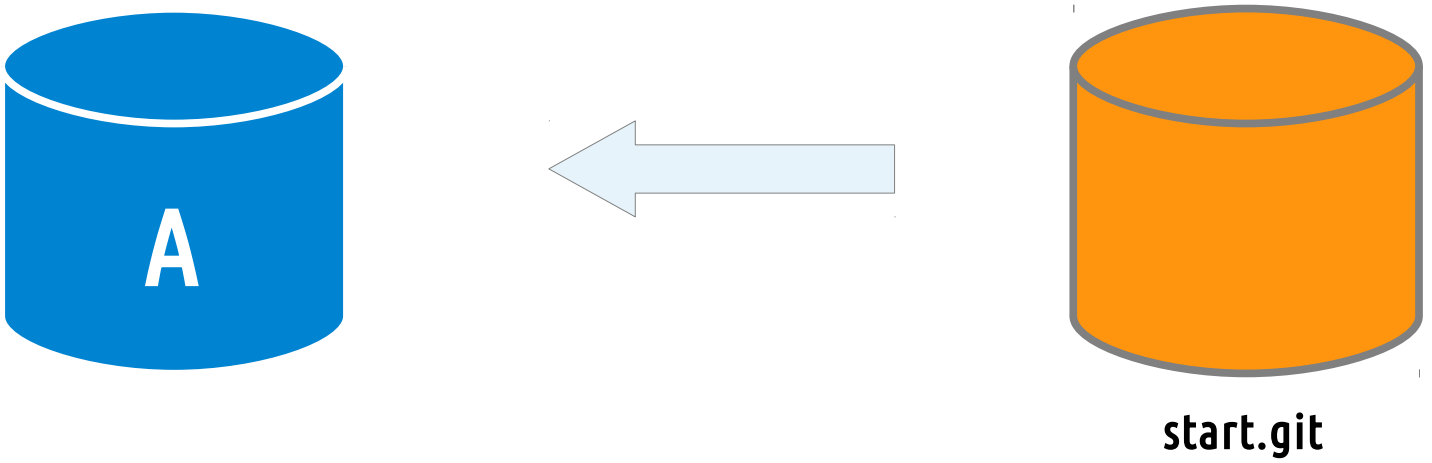
https://github.com/TRZ-EI/start.git

Git

Repository remoti

In ogni caso clonare un repository **bare** per ottenere un repository di lavoro si fa sempre allo stesso modo

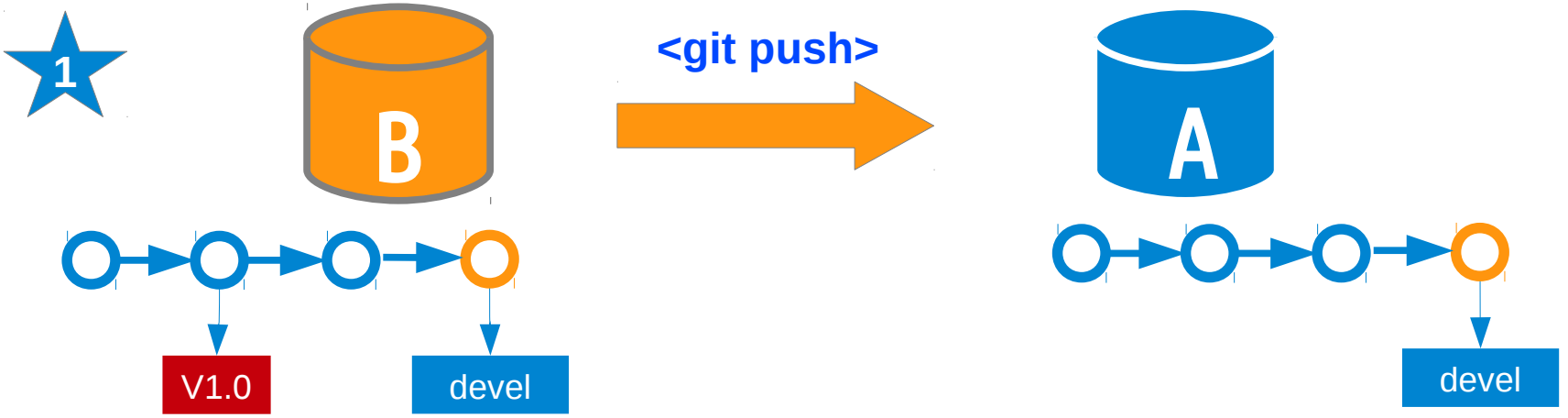
<git clone https://github.com/TRZ-EI/start.git>



Git

Tags remote

Le tags che utilizziamo nel repository locale non vengono automaticamente copiate nel repository remoto

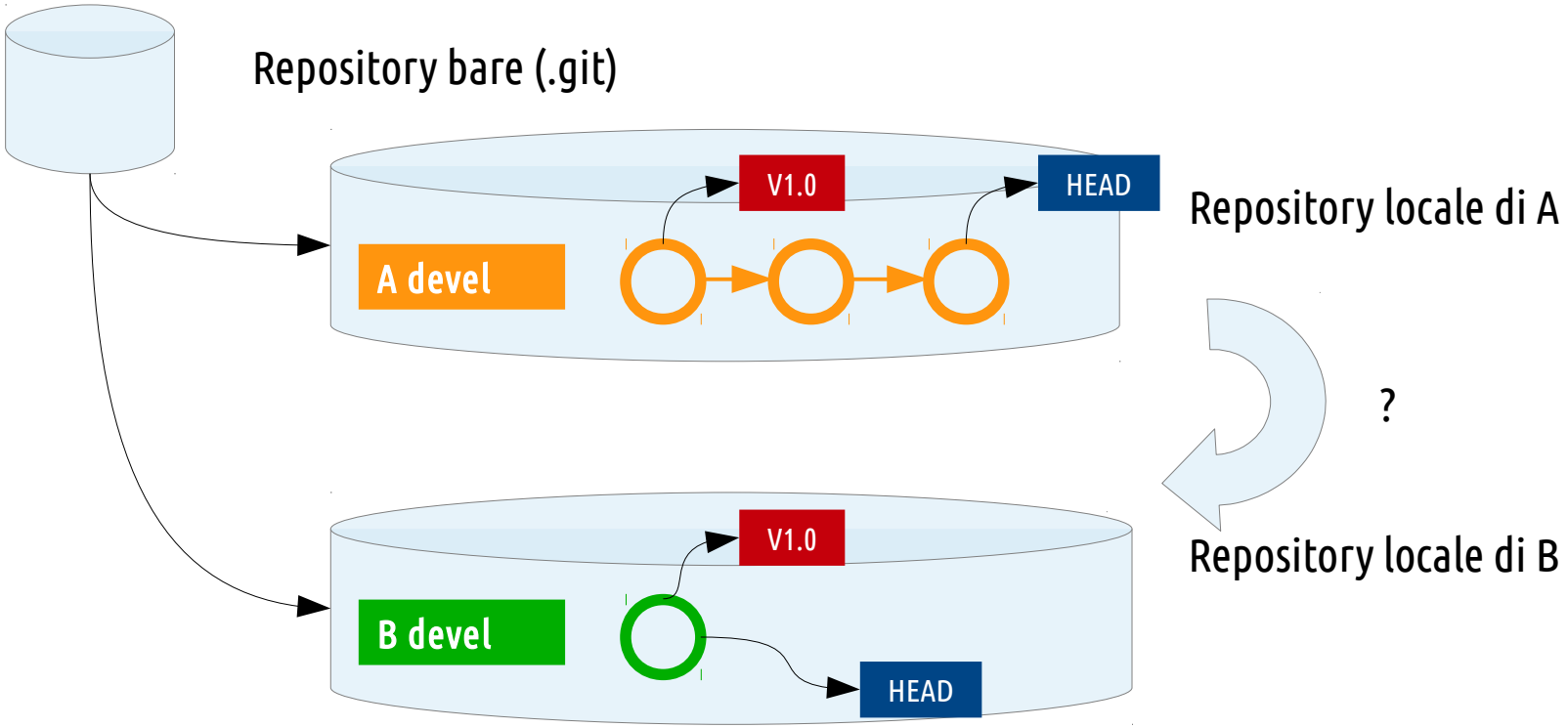


Per memorizzarle anche nel repository remoto, è necessario dichiararlo esplicitamente

Git

Patch

Supponiamo che gli sviluppatori **A** e **B** clonino un repository remoto ed inizino a lavorare. Improvvisamente, per qualche motivo, **A** deve lasciare l'attività in corso e passarla allo sviluppatore **B**...



Git

Patch

Le modifiche fatte da A sono ancora instabili, ed è meglio che non vengano salvate nel repository **origin**. Con Git si può creare una patch, che è un file di testo che elenca le modifiche fatte da A, da un certo punto in poi. B prende questo file e lo applica al proprio repository locale