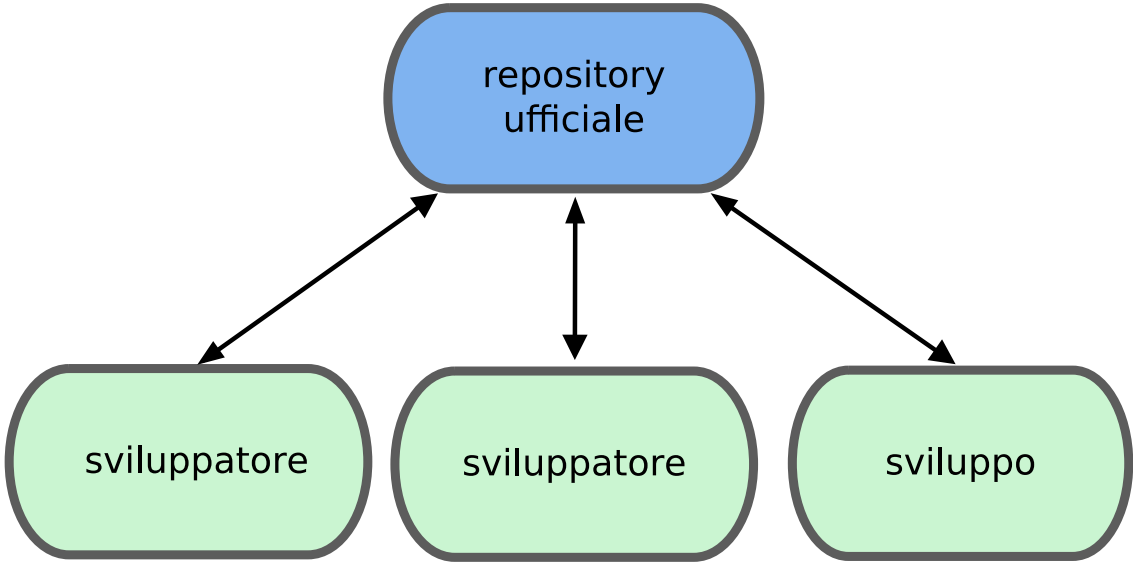
Git: Sviluppo distribuito e funzionalità avanzate

Emanuele Santoro
manu@santoro.tk

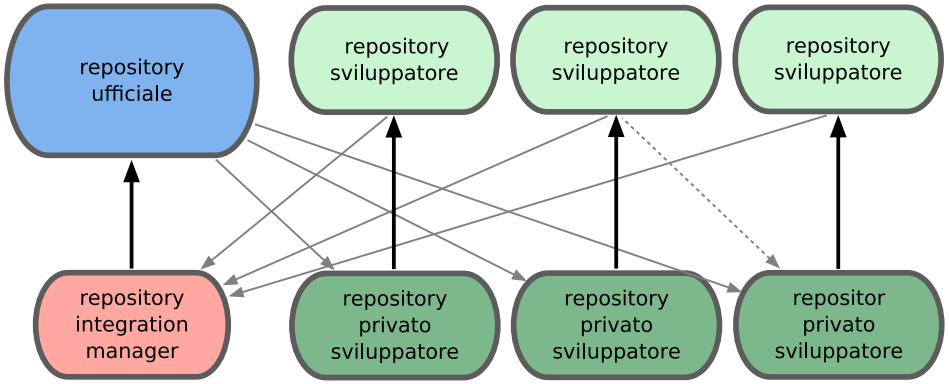
Corso Git 2014

Modello centralizzato



- Ottimo per piccoli team
- Ogni sviluppatore può leggere e scrivere
- Git impedisce di fare confusione: prima di fare un push bisogna integrare le modifiche già pubblicate da gli altri sviluppatori

Modello a «Integration Manager»



- Un repository «**ufficiale**»
- Ogni sviluppatore ha due repository: uno **privato** (tipicamente locale) ed uno **pubblico**
- L' «integration manager» ha un suo repository, privato.
- Gli sviluppatori hanno accesso **solo in lettura** al repository ufficiale
- L'integration manager ha accesso in lettura e scrittura al repository ufficiale

Modello a «Integration Manager»: Come funziona

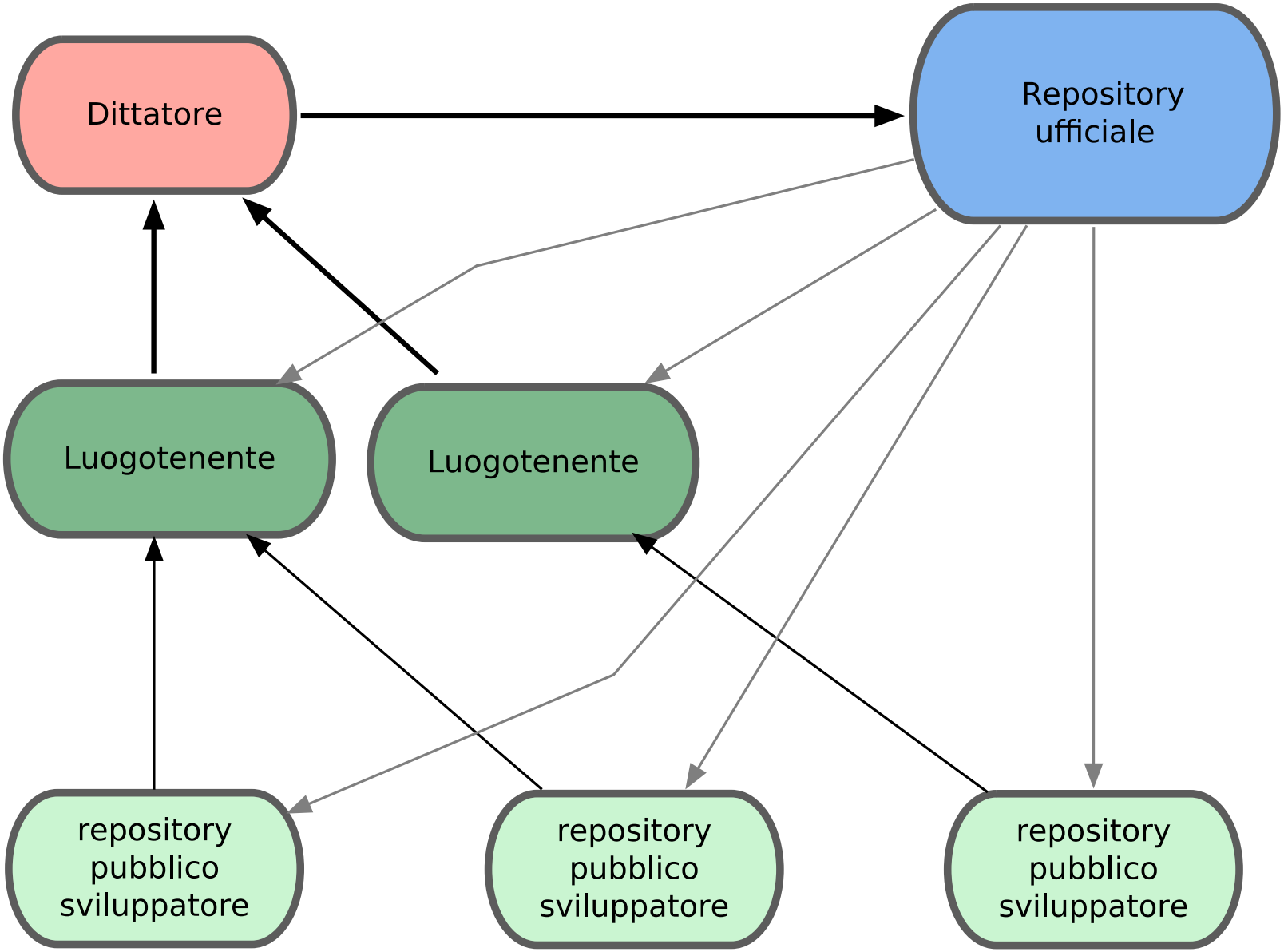
Il capo-progetto (che spesso è anche l'integration manager) ha un suo repository locale (privato), e gestisce il repository ufficiale (pubblico)

- Uno sviluppatore può clonare (**clone**) localmente il repository ufficiale, ottenendo la sua copia privata, e lavorarci sopra
- Fatte le modifiche, lo sviluppatore può pubblicare il suo repository, e chiedere all'integration manager di integrare le sue modifiche (**pull-request**)
- L'integration manager aggiunge il repository dello sviluppatore come repository remoto e scarica le modifiche (**pull**).

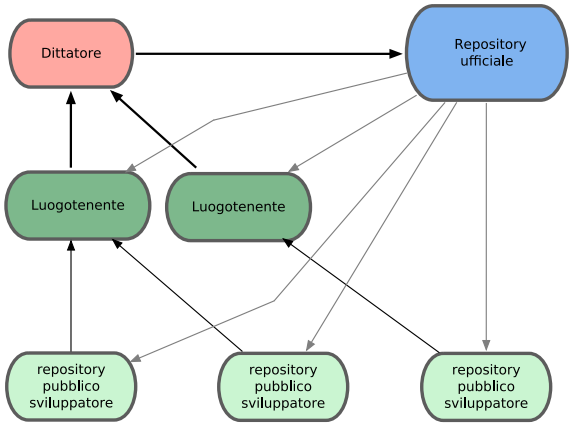
Modello a «Integration Manager»: Come funziona (2)

- L'integration manager verifica la qualità delle modifiche e **decide** o meno **se integrarle**
- L'integration manager integra le modifiche (**merge**)
- L'integration manager aggiorna (**push**) il repository ufficiale affinché contenga le modifiche apportate.

Modello dittatoriale



Modello dittatoriale: come funziona



Cioè?

- Modello utile per progetti di grandi dimensioni
- È un modello «Integration Manager» a più livelli
- Prevede la presenza di uno sviluppatore capo detto «dittatore» che gestisce il repository ufficiale
  - Esempio: Il kernel Linux (con Linus Torvals «dittatore benevolo»)

Modello dittatoriale: come funziona (2)

Come funziona?

- I normali sviluppatori lavorano sul loro **branch monotematico**, avendo come riferimento il repository del dittatore
- I **luogotenenti verificano** ed integrano il lavoro degli sviluppatori normali nei loro branch, verificandone la qualità
- Il **dittatore integra** i branch dei luogotenenti nel repository principale. Il dittatore ha facoltà di rigettare delle modifiche.
- Se il **dittatore accetta** le modifiche, aggiorna il repository ufficiale

Repository remoti

Per collaborare con altre persone è necessario saper:

- Pubblicare il proprio repository
- Accedere ad i repository dei nostri collaboratori
- Gestire (caricare/scaricare/aggiornare) i nostri rami di sviluppo

Git remote

Gestire riferimenti a repository remoti:

- **Aggiungere**: `git remote add <nome> <url>`
  - I remote sono degli url, ad esempio:
    - git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
- **Rimuovere**: `git remote rm <nome>`
- **Rinominare**: `git remote rename <old> <new>`
- **Visualizzare** tutti i repository remoti tracciati:
  - `git remote`
  - `git remote -v`
- Visualizzare i dettagli di un repository in particolare:
  - `git remote show <nome>`

Git fetch

- Il comando git fetch serve per importare commit e banch da un repository remoto al repository locale.
- I commit/branch saranno presenti come remoti, in modo da poterli controllare prima di fonderli con quelli locali.
  - `git fetch <remote>`
  - `git fetch <remote> <branch>`
- Per sapere quali altri rami sono presenti nei repository remoti:
  - `git branch -r`
- Per visualizzare un ramo remoto:
  - `git checkout <remote>/<branch>`
  - Nota: questo crea un ramo "temporaneo". Per scaricare e lavorare su un ramo remoto:
    - ★ `git checkout -b <nome_locale> <remote>/<branch>`

Git pull

Git fetch scarica tutti i commit remoti:

- `git checkout master ## sono sul ramo master`
- `git fetch origin master ## scarico tutti i commit dei miei colleghi`
- `git log origin/master ## controllo le modifiche`
- `git merge origin/master ## fondo i due rami nel repository locale`

Git pull fa tutto questo (fetch+merge), in un **unico comando**:

- `git pull <remote>`
  - Scarica i nuovi commit sul ramo corrente da <remote> e aggiorna il mio repository

Git push

- É la controparte di git fetch: serve per trasferire i commit locali su un repository remoto
  - `git push <remote> <branch>`
- Se il branch corrente non esiste sul server remoto, verrà creato.
- Se il branch remoto esiste, i branch locale e remoto verranno fusi.
- Carica tutti I branch locali su <remote>:
  - `git push <remote> --all`

Git stashing

Git stash: salvare delle modifiche non ancora complete (non abbastanza per fare un commit), lavorare su un altro branch per poi tornare sul ramo precedente, riapplicare le modifiche parziali, ultimarle, fare il commit.

```
user@host :~/code $ ## salvare le modifiche
user@host :~/code $ git stash
user@host :~/code $ ## cambio branch
user@host :~/code $ git checkout master
user@host :~/code $ emacs src/main.c
user@host :~/code $ ## hack hack hack
user@host :~/code $ ## torno sul mio branch
user@host :~/code $ git checkout altro_branch
user@host :~/code $ ## applicare le modifiche
user@host :~/code $ git stash apply
```

Riscrivere la history

- Errore nell'ultimo commit
  - `user@host :~/code $ git commit --amend`
- Abbiamo dimenticato di aggiungere un file al commit precedente:
  - `user@host :~/code $ git add file_mancante`
  - `user@host :~/code $ git commit --amend`
- **Attenzione**: `git commit -amend` non modifica l'ultimo commit, ma lo **sostituisce** completamente con uno che integra le nostre modifiche. Pertanto, non bisogna modificare dei commit già pubblicati!

Debugging

Git non é stato creato pensando ad un linguaggio o tipo di file particolare.
Git ha degli strumenti di debug generici:

- `git blame`
- `git bisect`

Git blame

- Git blame ci aiuta a rispondere a queste domande!
- Sintassi: `git blame [opzioni] file`
- **Esempi:**
  - Chi ha modificato il file src/core/server.c ?
    - `user@host:~/code $ git blame src/core/server.c`
  - Chi ha modificato le linee 10-15 del file src/core/server.c ?
    - `user@host:~/code $ git blame -L 10,15 src/core/server.c`
  - Abbiamo appurato che nel file x e nella linea y c'é un bug. Chi é il responsabile?
    - Per rispondere a questa domanda, bisogna ispezionare il commit, e vederne l'autore.

Git bisect

Git bisect: ricerca binaria di un commit specifico, alla ricerca di bug (o altro).

- Puó aiutarci a rispondere a domande del tipo:
  - Quale commit ha introdotto questo bug? (e chi è il responsabile?)
  - C'è un bug di sicurezza, da quanto tempo è in produzione?
  - Quando è stato aggiunto questo file al repository? E da chi?

Git bisect (2)

Come funziona?

- Git può, considerando il branch corrente come una successione di commit, provare ad individuare il commit che introduce il problema con una ricerca binaria.
- Ci sono quattro comandi fondamentali:
  - `git bisect start`
  - `git bisect good`
  - `git bisect bad`
  - `git bisect reset`
  - Extra: `git bisect run`

Git bisect (3)

All'inizio della ricerca:

- `git bisect start`
  - Si inizia il processo di ricerca binaria
- `git bisect good <commit>`
  - marca lo stato del progetto al momento di un certo commit come positivo
- `git bisect bad [<commit>]`
  - marca lo stato del progetto al momento di un certo commit come negativo (se <commit> non é specificato, viene considerato l'ultimo commit)
- good/bad insieme servono a delimitare l'insieme di commit su cui effettuare la ricerca binaria!

Git bisect (4)

- **Durante la ricerca**:
  - `git bisect good`
    - ★ Marca il commit in esame come positivo
  - `git bisect bad`
    - ★ Marca marca il commit attuale come negativo
- Dopo (relativamente) poche iterazioni, git dovrebbe consentirci di trovare il commit incriminato!
- Una volta individuato il commit incriminato, si puó riportare il repository in uno stato operativo, per poterci lavorare sopra:
  - `git bisect reset`
- `git bisect reset` pone fine al processo di ricerca binaria

Git bisect (5)

Extra: `git bisect run`

- Il test del repository può essere specificato come un test (codice eseguibile) che git eseguirà ad ogni iterazione per determinare lo stato positivo/negativo del repository dopo ogni commit.
- Sintassi:
  - `user@host:~/code $ git bisect run <script> <argomenti>`
  - `<script>` deve essere un eseguibile, che accetta `<argomenti>` sulla riga di comando
- Per segnalare lo stato positivo/negativo del repository, `<script>` usa il codice di uscita:
  - 0 – bug assente
  - != 0 – bug presente

Git bisect (6)

Esempi:

- `git bisect run make` #quale bug introduce un errore di compilazione?
- `git bisect run make test` #quale commit rompe la suite di test?

Git bisect (esempio)

Esempi:

- Quale commit introduce delle chiavi ssh nel repository?
  - Le chiavi sono nella cartella ssh_keys/

```
#!/usr/bin/env python
import os,sys
if os.path.exists('./ssh_keys') :
»       sys.exit(1)
else :
»       sys.exit(0)
```

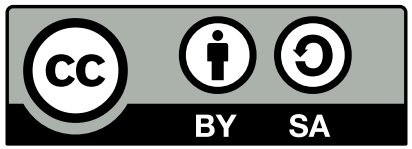
```
user@host:~/code $ git bisect start
user@host:~/code $ git bisect bad
user@host:~/code $ git bisect good
b2a5a24991d0044649128426b8d7600c02f0fc03
user@host:~/code $ git bisect run ./test.py
user@host:~/code $ git bisect reset
```

Fonti

- Pro Git book (Scott Chacon): `http://git-scm.com/book/en/Distributed-Git-Distributed-Workflows`

Fine

Grazie per l'attenzione!



`http://www.poul.org`