10
Strategie di uso efficace dei Decentralized Version Control Systems

Danilo Pianini
Giovanni Ciatto, Angelo Croatti, Mirko Viroli

Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

17 novembre 2017

Outline

Outline

Preparazione dell'ambiente di lavoro

1. Clonare il repository degli esercizi
   - `https://bitbucket.org/danysk/courses-2017-oop-lab-09`
     - Opzionalmente, si fork-i il repository
   - Si copi la URI con cui effettuare il clone dall'interfaccia web di Bitbucket
2. Importare il repository in Eclipse come progetto Java
   2.1 File
   2.2 Import
   2.3 General
   2.4 Existing project into workspace
   2.5 Si selezioni la cartella del progetto
   2.6 Si confermi l'import
3. Configurare correttamente Eclipse, abilitare e configurare i plugin per il controllo di qualità del codice

Modalità di lavoro

1. Seguire le istruzioni del file README.md nella root del repository
2. Tentare di capire l'esercizio in autonomia
   - Contattare il docente se qualcosa non è chiaro
3. Risolvere l'esercizio in autonomia
   - Contattare il docente se si rimane bloccati
4. Utilizzare le funzioni di test per verificare la soluzione realizzata
5. Cercare di risolvere autonomamente eventuali piccoli problemi che possono verificarsi durante lo svolgimento degli esercizi
   - Contattare il docente se, anche **dopo aver usato il debugger**, non si è riusciti a risalire all'origine del problema
6. Scrivere la Javadoc per l'esercizio svolto
7. Assicurarsi che non ci siano warning nel proprio codice
8. Effettuare *almeno* un commit ad esercizio completato
9. **A esercizio ultimato sottoporre la soluzione al docente**
10. Proseguire con l'esercizio seguente

Outline

Outline

Dalle puntate precedenti

DVCS

- DVCS sono strumenti potenti per tenere traccia in maniera efficiente della storia di un progetto
- Nascono in particolare come evoluzione dei tradizionali VCS (SVN, CVS . . .)
- Enfasi su una **miglior gestione del lavoro di team**

DVCS e teamwork

- "La potenza è nulla senza controllo!"
- Ovvero . . . la mancanza di un metodo chiaro e condiviso per utilizzarli può portare a risultati **DEVASTANTI**
  - effort necessario per la parte di gestione diventa presto preponderante e insostenibile
- Ecco perché è bene adottare un **workflow collaborativo**
  - i vostri progetti e i vostri partner di progetto vi ringrazieranno!

DVCS Workflow

Cos'è

- E' una sorta di "protocollo", un insieme di regole e passi da seguire quando si utilizza un DVCS
- Essendo per definizione **collaborativo**, ciascun team member vi partecipa con un **ruolo**
- Associa le diverse fasi del ciclo di vita del software a una serie di attività
  - Ogni ruolo ha la/le proprie attività, ognuna codificata in un insieme di passi da seguire

Come deve essere

- In tre aggettivi
  - Semplice
  - Chiaro
  - Condiviso

Outline

L'importanza dei ruoli

- Occupandosi di collaborazione in un team, è naturale che il workflow definisca dei ruoli
- Nei progetti del "mondo reale" il set minimo di ruoli è composto da:
  - i **developer**: sviluppano codice, implementano nuove funzionalità, bugfixing di funzionalità già rilasciate
  - il **team leader**: coordina l'attività dei developer, sviluppa egli stesso (spesso si prende carico degli aspetti più delicati)
  - il **release manager**: si occupa di gestire i rilasci, ovvero "pubblicazione" delle nuove funzionalità del software, rese così disponibili ai suoi utenti

Le attività coperte

- Sono quelle tipiche del cosiddetto *software lifecycle*
  - Sviluppo del software (delle nuove funzionalità)
  - Rilascio delle nuove funzionalità (dopo opportuni test funzionali)
  - Fix di bug riscontrati nelle versioni già rilasciate (a seguito di segnalazioni da parte degli utenti)
- Versioni? Cioè?

Le versioni

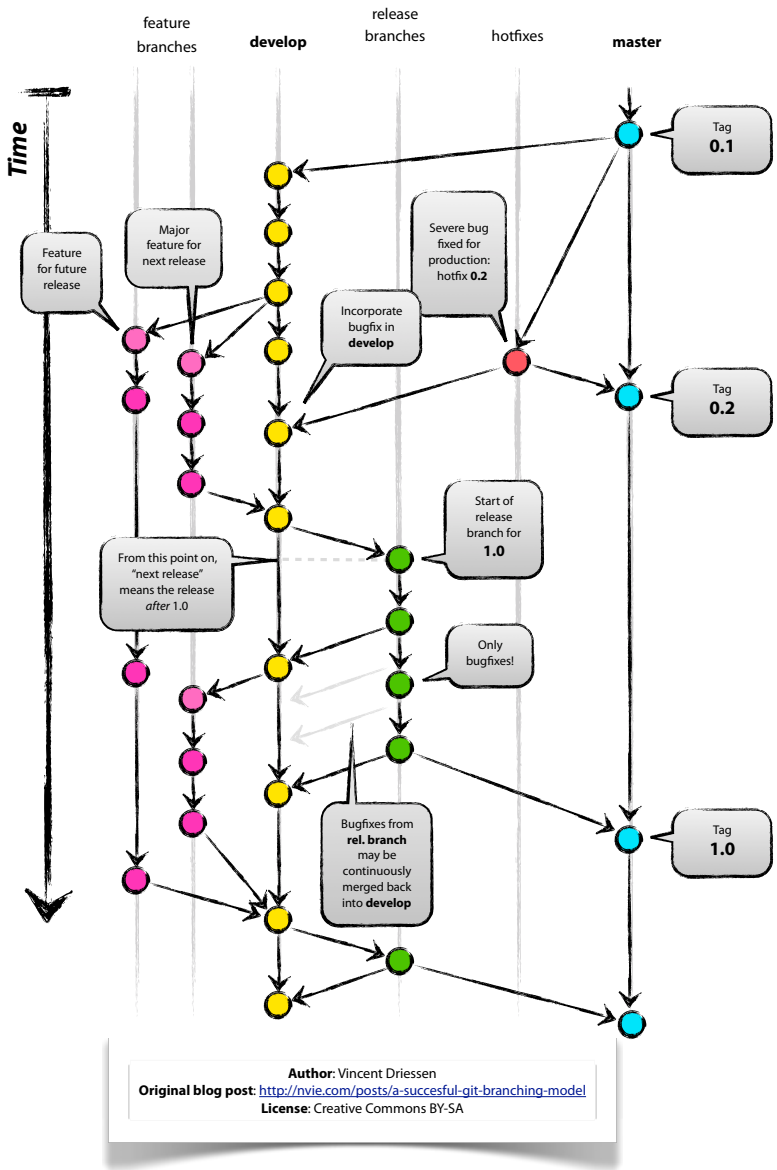
- Ogni rilascio del software è caratterizzato da una versione, identificata da un **numero**
- il numero ha (o dovrebbe) avere un senso, non è un numero estratto a sorte!
- Il pattern tipico è: **X.Y.Z** (numeri naturali)
  - **X** - **major version**. Identifica una evoluzione importante del software, nella quale funzionalità precedenti possono in generale aver subito modifiche o essere state eliminate.
    - Retrocompatibilità non garantita!
  - **Y** - **minor version**. Identifica rilasci che aggiungono nuove funzioni, senza toccare quelle già presenti.
    - Retrocompatibilità garantita!
  - **Z** - **bugfix version**. Identifica la risoluzione di difetti su una certa release $X.Y$, senza aggiunta, modifica e/o cancellazione di funzionalità
    - Retrocompatibilità garantita!
- Un buon riferimento è: `http://semver.org/`

Quale workflow

- Come si sceglie un workflow?
- Abbiamo parlato di semplicità...
- In realtà è più corretto parlare di giusto **trade-off** tra semplicità ed esigenze

Outline

Lo stato dell'arte I



Lo stato dell'arte II

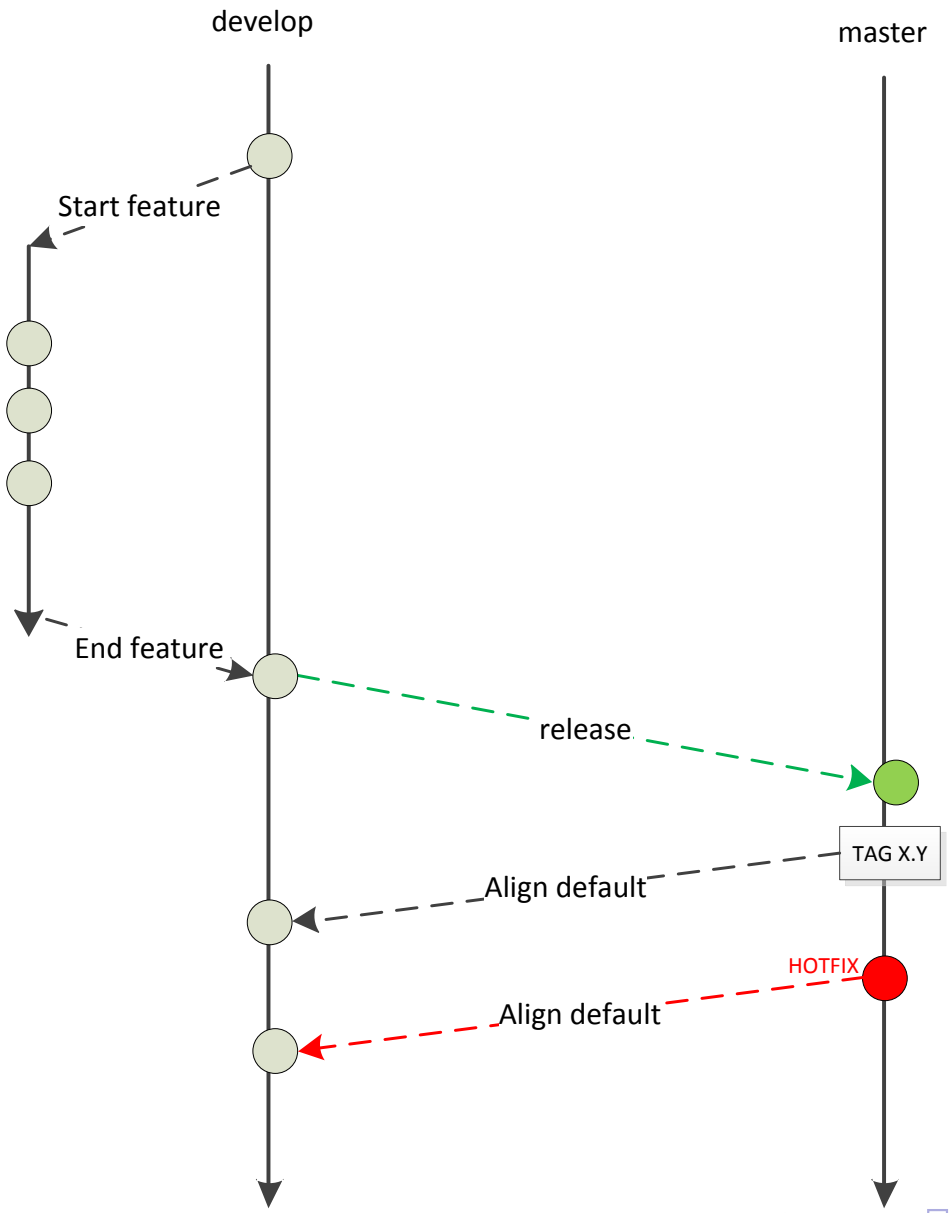
Alcune considerazioni

- Non lo useremo
  - troppo complicato per i nostri scopi
- Comunque molto interessante perché racchiude tutti gli aspetti di un DVCS workflow

I branch

- Sono il supporto fondamentale alle fasi del ciclo di vita del software
- Ogni fase ha il proprio branch!
- Branching e merging all'ordine del giorno!

Un modello più semplice



Rivediamo ruoli e compiti

- Sviluppatore:
  1. aprono un nuovo **feature branch** da `develop` ogni volta che cominciano lo sviluppo di nuova funzionalità
  2. le **feature** in realtà si aprono solo per funzionalità "grandi", altrimenti si lavora su `develop`
  3. effettuano commit regolari sul feature branch
  4. terminato lo sviluppo della funzionalità effettuano il merge della feature in `develop`
- Technical leader:
  1. supervisiona la qualità del codice prodotto
  2. è responsabile della qualità del codice che finisce sulla `develop` branch
- Release manager:
  1. effettua i rilasci
  2. quando una nuova versione è pronta per il rilascio, effettua il merge della `develop` in `master`
  3. effettua il tagging della release

Sulle feature branch

Ha senso utilizzarle:

- per modifiche grandi o a rischio fallimento
  - In particolare in termini di tempo richiesto per completare l'implementazione
- Quando l'attività di analisi abbia già portato ad una classificazione accurata delle funzionalità
- Per separare e gestire meglio le attività di sviluppo in progetti articolati
  - chi fa cosa
  - più sviluppatori impegnati sulla stessa funzionalità usano lo stessa branch

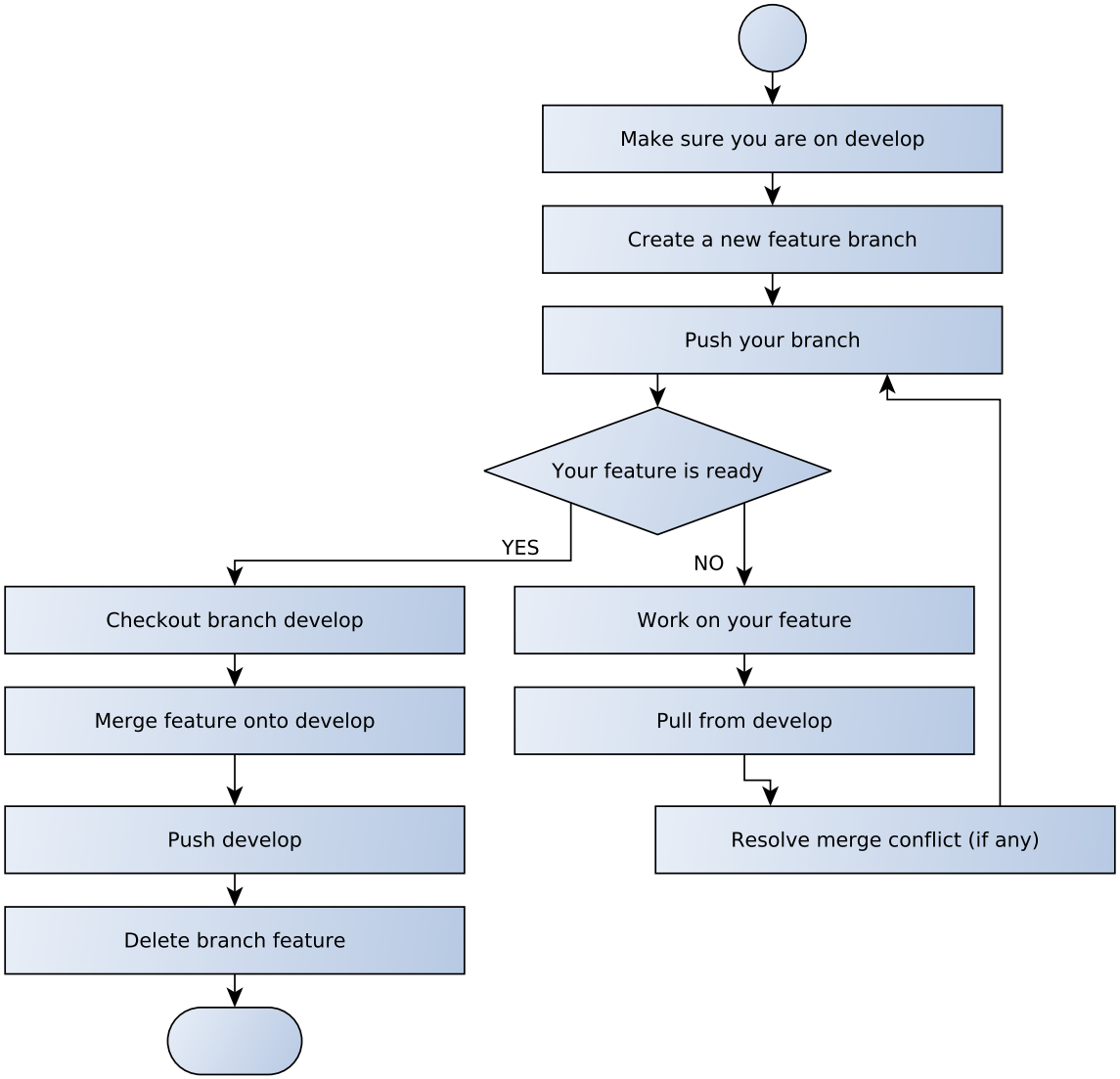
In contesti articolati può quindi aiutare a tener traccia dello stato di avanzamento di determinate attività

Outline

Inizializzazione del repository

```
# Create a new work directory
mkdir myawesomeproject
cd myawesomeproject

# Initialize your repository
git init

# Create a README.md file and a .gitignore file
# Initialize the branch master with them
git add README.md .gitignore
git commit -m "Initialize project"

# Configure the remotes and share your project stub
git remote add origin your_project_url
git push -u origin master

# Switch to develop, and push the stub
git checkout -b develop
git push -u origin develop
```

Feature branch I



Feature branch II

```
# Create a new feature branch from develop and share it
git checkout develop
git checkout -b feature-mynewfeaturename
git push origin feature-mynewfeaturename

# WHILE your_feature_is_unfinished
# work on your feature:
git add mynewfiles
git add mymodifiedfiles
git add mydeletedfiles
git commit -m "my commit message"
# Merge develop in to prevent big merge conflicts!
git pull origin/develop
# You may need to solve a merge conflict here!
# Share and save your work
git push
# END-WHILE

# Merge feature onto develop
git checkout develop
git merge feature-mynewfeaturename

# Push develop
git push

# Delete feature branch
git branch -d feature-mynewfeaturename
```

Associazione di nomi simbolici ad un commit I

In generale

È necessario poter associare ad un commit della meta-informazione personalizzata. Ad esempio, per associare un certo commit ad una versione, in modo da poterla facilmente richiamare.

In Git

- `git tag -a MyTag -m "My tag information"` Associa al commit corrente il nome simbolico `MyTag` ed il messaggio aggiuntivo `My tag information`
  - Non rimpiazza il commit message, ma lo integra
  - È possibile invocare comandi come `git checkout MyTag` per tornare velocemente ad un tag precedente
- `git push --tags`
  - Fa `push` delle metaiformazioni aggiunte col comando `tag`
  - La `push` "normale" non invia queste informazioni!

Fare una release

```
# Make sure that develop contains the version that you want to become the new stable
# Switch to master
git checkout master

# Merge the last develop in and share your work
git merge develop
git push

# Name your release using a tag and share this information
git tag -a 0.1.0 -m "New version 0.1.0, that includes this amazing new feature"
git push --tags

# Integrate the new information onto develop to begin a new development cycle
git checkout develop
git merge master
git push

# Back to regular development!
```

Il repo ufficiale del vostro progetto I

Workflow raccomandato

- Qualcuno di voi agirà come "repo maintainer"
- Creerà quindi il repository su BitBucket
- Gli altri membri del team faranno la `clone`
- Ciascuno lavorerà parallelamente sul proprio repository locale (working copy), condividendo tramite `push` e `pull` il proprio lavoro con gli altri

Il repo ufficiale del vostro progetto II

Workflow avanzato

Ottimo per progetti di grosse dimensioni e/o per team molto eterogenei, dove qualcuno deve assicurarsi della qualità del codice prodotto da altri.

- Il maintainer crea il repository, ed è l'unico col diritto di scrittura
- Gli altri membri del team hanno una fork a testa
- Ciascuno lavora su una working copy, facendo pull dal repository "centrale" e push sulla propria fork
- Quando una feature è completa, o si arriva ad un buon grado di sviluppo, si apre una **pull request**
- Il maintainer revisiona il codice, assegna eventuali modifiche, e quando è soddisfatto accetta la pull request facendo il merge del codice nel repository principale

Questo workflow è un overkill per il progetto di OOP

- Ma è possibile che vi chiederemo di lavorare così, se farete tesi o un tirocini relativi ad alcuni nostri software

Best practice per la condivisione I

Sviluppatore

- In generale, push regolare delle proprie modifiche su repo ufficiale (e pull regolare di quanto presente in remoto)
  - Se il push riguarda il `develop` branch: le modifiche che condividete via push **devono compilare**!
- Eseguire sempre un pull e update tutte le volte che cominciate a lavorare su qualche cosa in una certa branch
- Per eventuali feature branch? Stesse regole di `develop`

Release manager

- Pull di `develop` prima di cominciare una release
- Push di `master` e `develop` al termine della release

Per approfondimenti...

A successful git branching model:
`http://nvie.com/posts/a-successful-git-branching-model/`