Maven - Forge - Intégration continue

Matthieu Moy (Transparents originaux d'Emmanuel Coquery)

https://matthieu-moy.fr/

2020

Outline

Autour du développement

Au delà du code:

- Tests (unitaires, intégration, fonctionnels)
- Documentation
- Partage des sources
- Suivi de bugs / évolutions
- Qualité du code
- Distribution

$\rightarrow$ cycles de développement lourds à gérer

Outils

- Frameworks de tests
- Générateurs de documentation
- Gestionnaires de version
- Gestionnaires de tickets
- Outils d'audit de code
- Scripts, *builders*

Outline

Maven

Automatisation du traitement des phases du cycle de vie

- Peut être vu comme un “super Makefile”
  - Java comme langage de script
- Lance l’exécution d’outils:
  - Compilation
  - Test automatisés
  - Archives, Déploiement
  - Génération de documentation
  - ...

Alternatives: CMake, Premake, Grunt, Gulp, etc

Architecture

- Basée sur un système de plugins
  - Un plugin $\leftrightarrow$ un script Java
    - ⋆ i.e. une classe avec une méthode particulière
    - ⋆ paramétrable via un morceau de XML
  - Une exécution de maven $\leftrightarrow$ suite d'exécution de plugins

- Nombreux plugins disponibles
  - Pas tous installés au départ
  - Système de téléchargement de plugins à la demande

Phases et cycles de vie

- Une phase regroupe un ensemble de tâches (goals)
  - 1 tâche $\rightarrow$ 1 plugin
- Un cycle de vie est une suite de phases
  - Le déclenchement d'une phase déclenche les phases précédentes du cycle de vie
- Le cycle de vie dépend du *packaging* (jar,war, ...)
  - *packaging* = type de projet
  - Format d'archive
  - Ordre des phases
  - Affectation tâches $\rightarrow$ phases
  - Préconfiguration des tâches
  - peut être reconfiguré selon les besoins du projet

Exemple: phases du *packaging* jar

Phase	Tâche(s)
process-resources	`resources:resources`
compile	`compiler:compile`
process-test-resources	`resources:testResources`
test-compile	`compiler:testCompile`
test	`surefire:test`
package	`jar:jar`
install	`install:install`
deploy	`deploy:deploy`

Projet maven: organisation des fichiers

- pom.xml ← config. du projet
- src/ ← sources
  - main/ ← à distribuer
    - java/ ← code Java
    - resources/ ← fichiers à distribuer (config appli, images, etc)
    - webapp/ ← ressources web (pour les war: html, jsp, js, images)
    - antlr4/ ← grammaire pour générer les *parsers*
    - ...
  - test/ ← uniquement pour les tests
    - java/, resources/, antlr4/, etc
- target/ ← tout ce qui est généré, il est supprimé par `clean`
  il ne faut **pas** le versionner (utiliser `.gitignore`)

Projet Maven: le pom.xml

```
<project ...>
  ...
  <groupId>fr.univ-lyon1.info.m1</groupId>
  <artifactId>monProjet</artifactId>

  <dependencies>
    <dependency>...</dependency>
    <dependency>...</dependency>
  </dependencies>

  <build>
    <plugins>
      <plugin>...</plugin>
      <plugin>...</plugin>
    </plugins>
  </build>
</project>
```

Projet Maven: dépendances dans le pom.xml

```
<project ...>
  <dependencies>
    <dependency>
      <groupId>org.openjfx</groupId>
      <artifactId>javafx-controls</artifactId>
      <version>11.0.2</version>
    </dependency>
  </dependencies>
  ...
</project>
```

En pratique, morceaux de XML à copier-coller depuis https://search.maven.org/, Maven s'occupe du reste !

Repository Maven

- Dépôt contenant:
  - Des plugins
  - Des bibliothèques (en général Java)
- Sur le web
  - Téléchargement automatique à la demande
  - Défaut: `http://repo1.maven.org`
  - Miroirs (Nexus, Archiva, etc)
- Local: `~/.m2/repository`
  - contient les archives des projets locaux
    - ⋆ phase `install`
  - cache pour les *repository* web

Classpath et dépendances

Utilisation de libs externes

- Téléchargement
- Gestion des versions ($\Rightarrow$ build reproductible)
- Transitivité des dépendances
- Configuration du CLASSPATH

Également utilisé pour les plugins

Dépendances: *scope*

Dépendances dans le CLASSPATH dans chaque phase selon la porté (scope)

scope	compilation	test	exécution	déploiement
compile	x	x	x	x
provided	x	x	x	
runtime		x	x	x
test		x		

(+ system, import)
Exemple avec JUnit:

```
<dependency>
  <groupId>org.junit.jupiter</groupId>
  <artifactId>junit-jupiter-engine</artifactId>
  <version>5.7.0-M1</version>
  <scope>test</scope> <!-- not available in
                           'mvn exec:java' -->
</dependency>
```

Archetypes

- Complexité inhérente aux projets maven
  - Difficultés de mise en œuvre
- Archetype = mini-projet de départ
  - D'un type particulier
  - Préconfiguré
- Exemples
  - maven-archetype-quickstart
  - spring-mvc-jpa-archetype
- En ligne de commande: `mvn archetype:generate`

Intégration dans les EDI

Un bon IDE a besoin de connaître les dépendances (erreurs de typage, suggestions, ...). `pom.xml` contient toutes les informations.

- IntelliJ : par défaut
- VSCode :
  - Plugin Maven for Java, inclus dans Java Extension Pack
  - File → Open Folder sur le répertoire contenant le `pom.xml` configure le projet automatiquement.
- Eclipse :
  - Plugin m2e + connecteurs
  - `mvn eclipse:eclipse`
    - ⋆ configure un projet Eclipse
    - ⋆ qui correspond au projet maven
- Netbeans: par défaut

Outline

Forges logicielles

Outil de travail collaboratif pour le développement :

- Espace collaboratif
  - Partage de documents
  - Wiki
  - Dépôt (SVN, Mercurial, Git, etc)
- Gestion des tâches
  - Bug tracking
  - Support, tâches diverses
  - Calendrier, Gantt
  - Suivi via un système de tickets (Issues)

forge.univ-lyon1.fr

- Forge Gitlab (logiciel libre installé sur nos serveurs)
- Dépôts Git (logiciel sur lequel repose GitLab, mais on peut utiliser Git sans GitLab)
- Intégration SI Lyon 1 (LDAP + CAS)
  - Disponible aux étudiants et personnels
  - Utilisable pour les TPs
  - Obligatoire pour le projet transversal (ex-MultiMIF)

Outline of this section

Git

- Gestionnaire de versions distribué
  - À la mercurial / darcs / bazaar /etc
- Utilisable
  - En ligne de commande (git)
  - Via une interface dédiée
    - ⋆ Tortoise git, SourceTree, etc
  - Dans un EDI
    - ⋆ Intégration VSCode, Eclipse, Netbeans, IntelliJ, Emacs, etc

Conseil : apprendre la ligne de commande
(maîtriser les concepts, les noms des commandes),
puis choisir l'outil qui vous convient.

Commandes de base

- Création
  - init, clone
- Fichiers
  - add, remove, mv, status
- Versions
  - commit, checkout
- Branches
  - branch, merge, rebase
- Synchronisation de dépôts
  - pull, fetch, push

Scénario simple

1. Début du travail
   1. Clone d'un dépôt distant
   2. Modification d'un fichier
   3. Commit
      - Pour l'instant, pas de modification du dépôt distant
   4. Push vers le dépôt distant
2. Plus tard ...
   1. Pull du dépôt distant
      - ou bien `fetch` + `rebase` (≈ `git pull --rebase`)
   2. GOTO 1.2

Outline of this section

Suivi des tâches (gestionnaire de tickets)

- Les tâches ont
  - Une description
  - Un statut: fermé ou ouvert
- Les tâches peuvent avoir:
  - Une échéance
  - Une personne assignée
  - Des étiquettes
  - Une étape (milestone)
- Liens commit (hash 32fb54de)/tâche (numéro 1234):
  - Fixes #1234 dans le message de commit $\Rightarrow$ ferme le ticket + lien hypertexte
  - ref 32fb54de dans les commentaires de la tâche $\Rightarrow$ lien vers le commit

Outline of this section

Branches: Why and How

- 1 branch = 1 named ref to a commit
- Think of a branch as a set of commits
- Typical uses
  - maintenance branch (bugfix only, will lead to next minor release) vs development branch (new features, will lead to next major release)
  - Topic branch: 1 branch per feature
    - ★ Create the branch
    - ★ Work on it (`commit`)
    - ★ Request a merge (`push` + pull-request, ...)
    - ★ (Delete the branch when it's merged)

Branches and Tags in Practice

- Create a local branch and check it out:
  `git checkout -b branch-name`[1]
- Switch to a branch:
  `git checkout branch-name`
- List local branches:
  `git branch`
- List all branches (including remote-tracking):
  `git branch -a`
- Create a tag:
  `git tag tag-name`

[1] Since git 2.23, `git switch -- create`

Outline of this section

Merging With Upstream

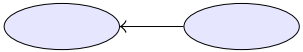
Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 1: merge (default with `git pull`)

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

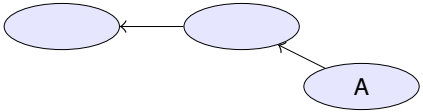
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

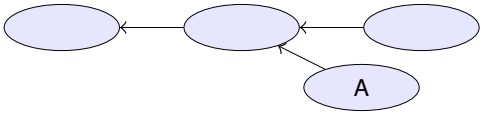
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

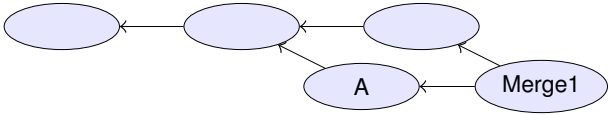
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

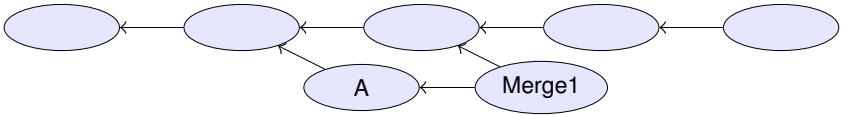
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

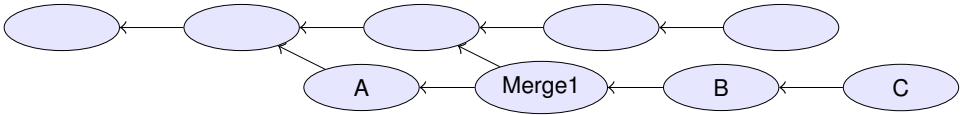
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

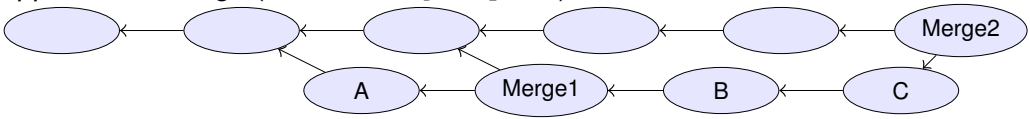
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 1: merge (default with `git pull`)



- Drawbacks:
  - Merge1 is not relevant, distracts reviewers (unlike Merge2).

Merging With Upstream

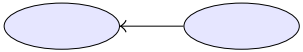
Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 2: no merge

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

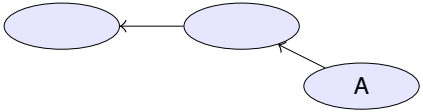
- Approach 2: no merge



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

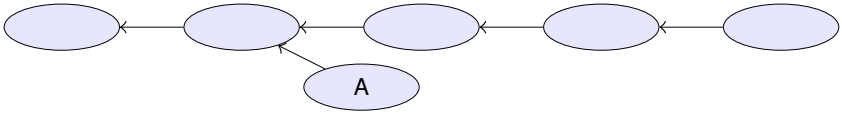
- Approach 2: no merge



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

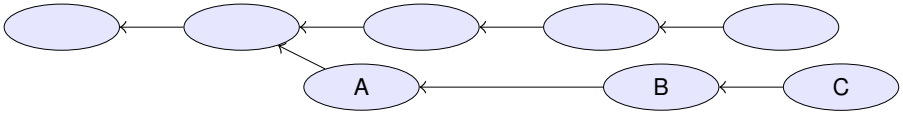
- Approach 2: no merge



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

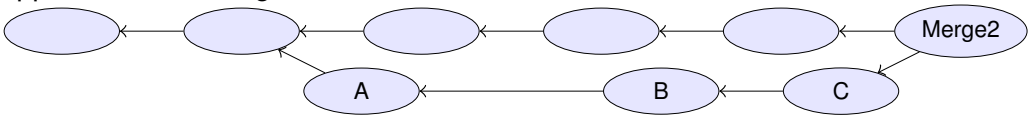
- Approach 2: no merge



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 2: no merge

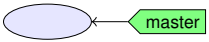


- Drawbacks:
  - In case of conflict, they have to be resolved by the developer merging into upstream (possibly after code review)
  - Not always applicable (e.g. “I need this new upstream feature to continue working”)

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

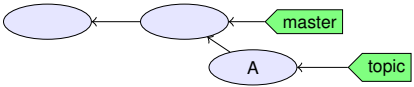
Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 3: rebase (`git rebase` or `git pull --rebase`)

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

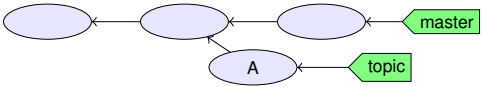
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

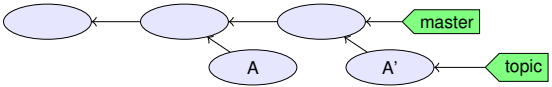
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

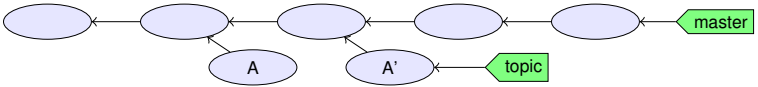
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

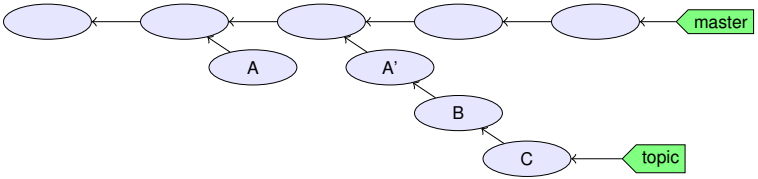
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

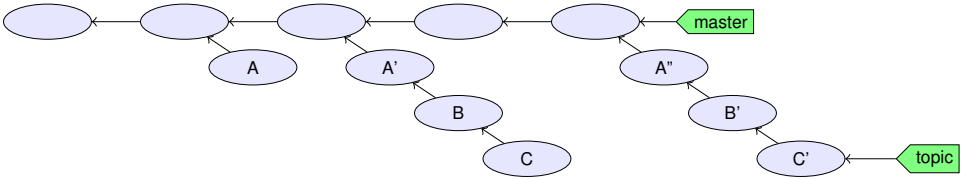
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

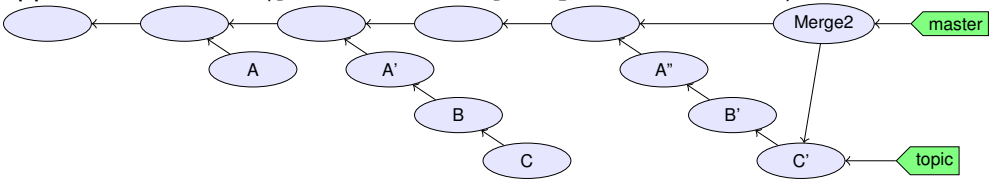
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

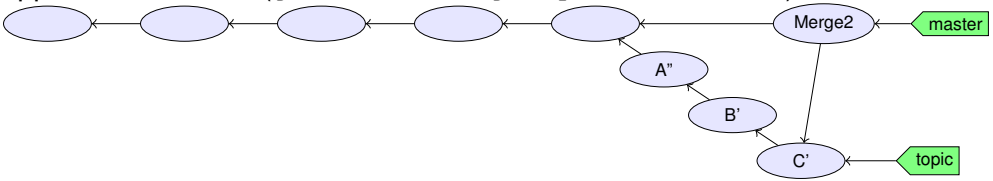
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

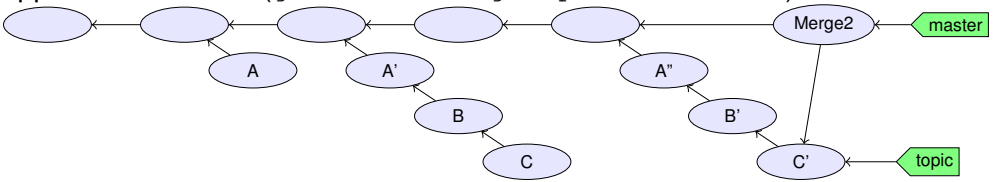
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 3: rebase (`git rebase` or `git pull --rebase`)



- Drawbacks: rewriting history implies:
  - A', B, C, A", B' probably haven't been tested (never existed on disk)
  - What if someone branched from A, A', B or C?
  - Basic rule: don't rewrite published history

Outline of this section

Rewriting history with `rebase -i`

- `git rebase`: take all your commits, and re-apply them onto upstream
- `git rebase -i`: show all your commits, and asks you what to do when applying them onto upstream:

```
pick ca6ed7a Start feature A
pick e345d54 Bugfix found when implementing A
pick c03fffc Continue feature A
pick 5bdb132 Oops, previous commit was totally buggy

# Rebase 9f58864..5bdb132 onto 9f58864
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
```

`git rebase -i` commands (1/2)

p, pick use commit (by default)

r, reword use commit, but edit the commit message
Fix a typo in a commit message

e, edit use commit, but stop for amending

- Once stopped, use `git add -p`, `git commit --amend`, ...

s, squash use commit, but meld into previous commit

f, fixup like "squash", but discard this commit's log message

- Very useful when polishing a set of commits (before or after review): make a bunch of short fixup patches, and squash them into the real commits. No one will know you did this mistake ;-).

`git rebase -i` commands (2/2)

x, exec run command (the rest of the line) using shell

- Example: `exec make check`. Run tests for this commit, stop if test fail.
- Use `git rebase -i --exec 'make check'`[2] to run `make check` for each rebased commit.

[2] Implemented by Ensimag students!

Outline of this section

Git Reflog: TL; DR

Git's reflog = detailed history

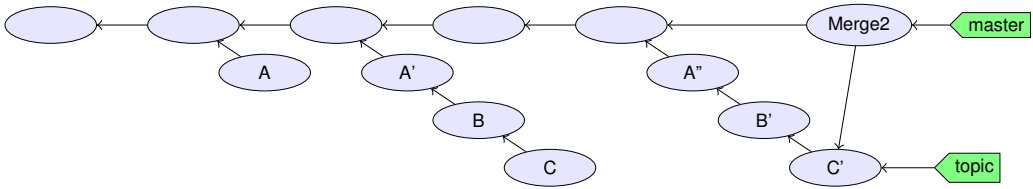
(makes `git rebase` less dangerous)

Good news:

if you don't know how to use it, a Git expert around you may do and recover data you thought was lost :-)

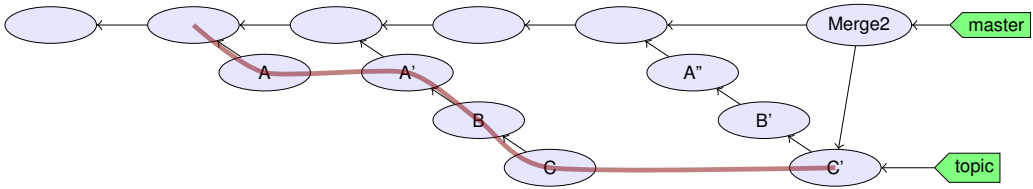
Git's reference journal: the reflog

- Remember the history of local refs.
- $\neq$ ancestry relation.



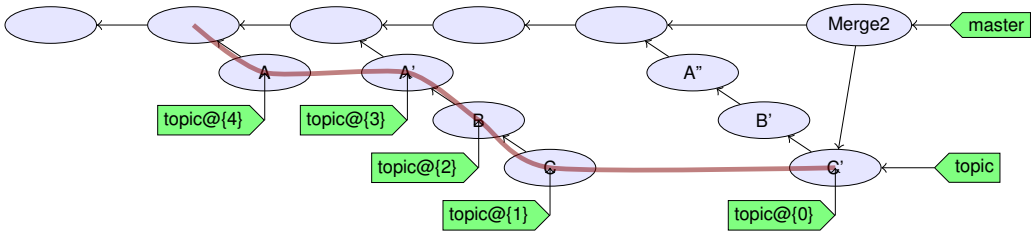
Git's reference journal: the reflog

- Remember the history of local refs.
- $\neq$ ancestry relation.



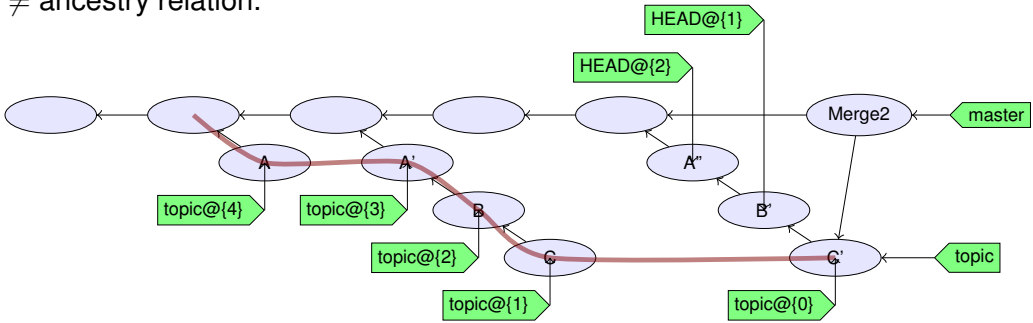
Git's reference journal: the reflog

- Remember the history of local refs.
- $\neq$ ancestry relation.



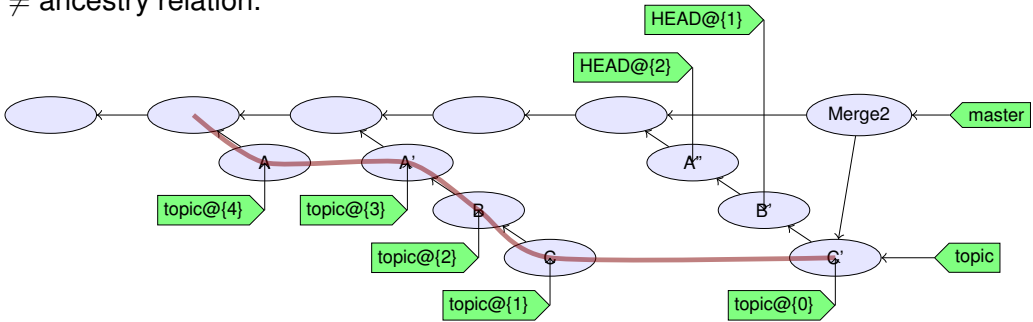
Git's reference journal: the reflog

- Remember the history of local refs.
- $\neq$ ancestry relation.



Git's reference journal: the reflog

- Remember the history of local refs.
- $\neq$ ancestry relation.



- `ref@{n}`: where *ref* was before the *n* last ref update.
- `ref~n`: the *n*-th generation ancestor of *ref*
- `ref^`: first parent of *ref*
- `git help revisions` for more

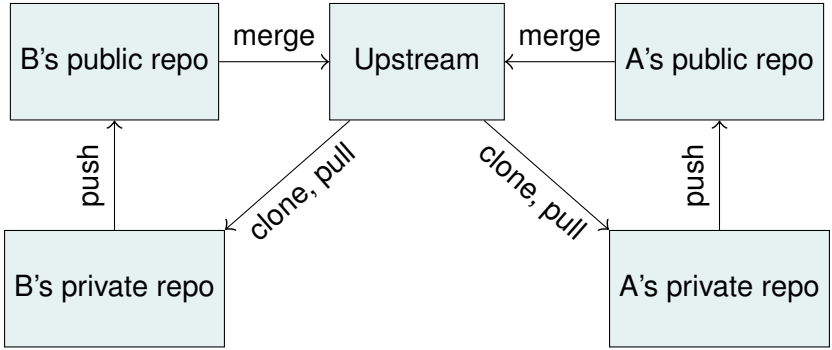
Outline of this section

Centralized workflow

```
do {
   while (nothing_interesting())
      work();
   while (uncommited_changes()) {
      while (!happy) { // git diff --staged ?
         while (!enough) git add -p;
         while (too_much) git reset -p;
      }
      git commit; // no -a
      if (nothing_interesting())
         git stash;
   }
   while (!happy)
      git rebase -i;
} while (!done);
git push; // send code to central repository
```

Triangular Workflow with pull-requests

- Developers pull from upstream, and push to a “to be merged” location
- Someone else reviews the code and merges it upstream



Pull-requests in Practice

Contributor	create a branch, commit, push
Contributor	click “Create pull request” (GitHub, GitLab, BitBucket, ...), or `git request-pull`
Maintainer	receives an email
Maintainer	review, comment, ask changes
Maintainer	merge the pull-request

Code Review

- In a perfect world:
  1. A writes code, commits, pushes
  2. B does a review
  3. B merges to upstream
- What usually happens:
  1. A writes code, commits, pushes
  2. B does a review
  3. B requests some changes
  4. ... then ?

Iterating Code Reviews

- At least 2 ways to deal with changes between reviews:
  1. Add more commits to the pull request and push them on top
  2. Rewrite commits (`rebase -i`, ...) and overwrite the old pull request
     - ⋆ The resulting history is clean
     - ⋆ Much easier for reviewers joining the review effort at iteration 2
     - ⋆ e.g. On Git's mailing-list, 10 iterations is not uncommon.

Triangular Workflow: Advantages

- Beginners integration:
  - start committing on day 0
  - get reviewed later
- In general:
  - Do first
  - Ask permission after
- For Open-Source:
  - Anyone can contribute in good condition
  - “Who’s the boss?” is a social convention

Outline

Intégration Continue : l'idée

- Principe :
  - Intégrer le nouveau code au fur et à mesure qu'il est écrit
  - Garder une branche principale toujours fonctionnelle
- Outil indispensable : test automatisé à chaque `push`
- Abus de langage : « intégration continue » = « tests automatisés »

Intégration Continue - contexte

Historiquement réservé aux projets

- de grande taille
- impliquant de nombreuses personnes
- avec des itérations courtes

Technologies actuelles $\rightarrow$ accessible sur de petits projets $\rightarrow$ une fois qu'on y a goûté, on en fait partout !

Intégration Continue - principes

→ Automatisation des phases du cycle de vie
- Compilation, test, mise à disposition de binaires

→ Institutions de bonnes pratiques
- Commit réguliers
- La branche par défaut compile
- ...

→ Surveillance
- Tableaux de bord, etc

Continuous Integration: example with GitLab-CI

https://gitlab.com/moy/gitlab-ci-demo

- Configuration (`.gitlab-ci.yml`):

```
junit:
  # docker image with Maven
  image: maven:3.6.1-jdk-11
  script:
    - cd my-project/ && mvn test --batch-mode
```

- Use: work as usual ;-)
  - Tests launched at each `git push`
  - Pass/failed indicator for each merge-request

Serveurs d'IC

Permet d'exécuter régulièrement:

- Checkout
- Compilation
- Test
- Audit de code

Pour gitlab:

- peut servir de serveur d'IC;
- nécessite de gérer et de configurer des "runners" qui exécuteront les tâches (merci Manu !)

Alternatives: Jenkins, Travis-CI, GitHub actions, etc

SonarQube

Outil d'audit de code

- Analyse exécutée lors du cycle de vie
  - Via e.g. un goal maven
- Fournit des tableaux de bord:
  - Qualité du code
  - Couverture des tests unitaires

Outline

Merge-requests (aka pull-requests sur GitHub)

- Principe:
  - Un contributeur fait un “fork” du projet
  - Il fait un `push` dans une branche perso
  - Il demande à intégrer (`merge`) son code
- Intérêts:
  - Liste des branches à intégrer facile à voir
  - Discussion sur le code avant intégration
  - Possibilité de rejeter le mauvais code

Tout ça ensemble?

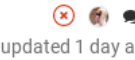
- Début du travail:

```
git checkout -b mvc
...
git commit
git push -u origin mvc
```

- Création de la pull-request sur l'interface web
- Gitlab-CI fait passer les tests :

```
# Fichier .gitlab-ci.yml
...
junit:
  script:
    - cd my-project/ && mvn test --batch-mode
```

- Maven s'occupe de télécharger les [illegible] tester, vérifier le style, et voilà :



Outline

Références

- https://maven.apache.org/
- https://git-scm.com/
- https://about.gitlab.com/