Git Workflows

Sylvain Bouveret, Grégory Mounié, Matthieu Moy

2017

[first].[last]@imag.fr

`http://recherche.noiraudes.net/resources/git/git-workflow-slides.pdf`

Goals of the presentation

- Global history: multiple workflow
- Global history: branching, rebasing, stashing

Workflows

Branches

Branching exists in most VCS. The goal is to keep track separately of the various parallel works done on an on-going software development.

This is Git killing feature: Git branch management (and merge) is lightweight and efficient. Thus it is quite easy to split the work among the developpers and still let them work easily together toward the common goal.

But it is not a magic bullet: the splitting of the work should be handle with care.

Example of bad splitting: per developer branches

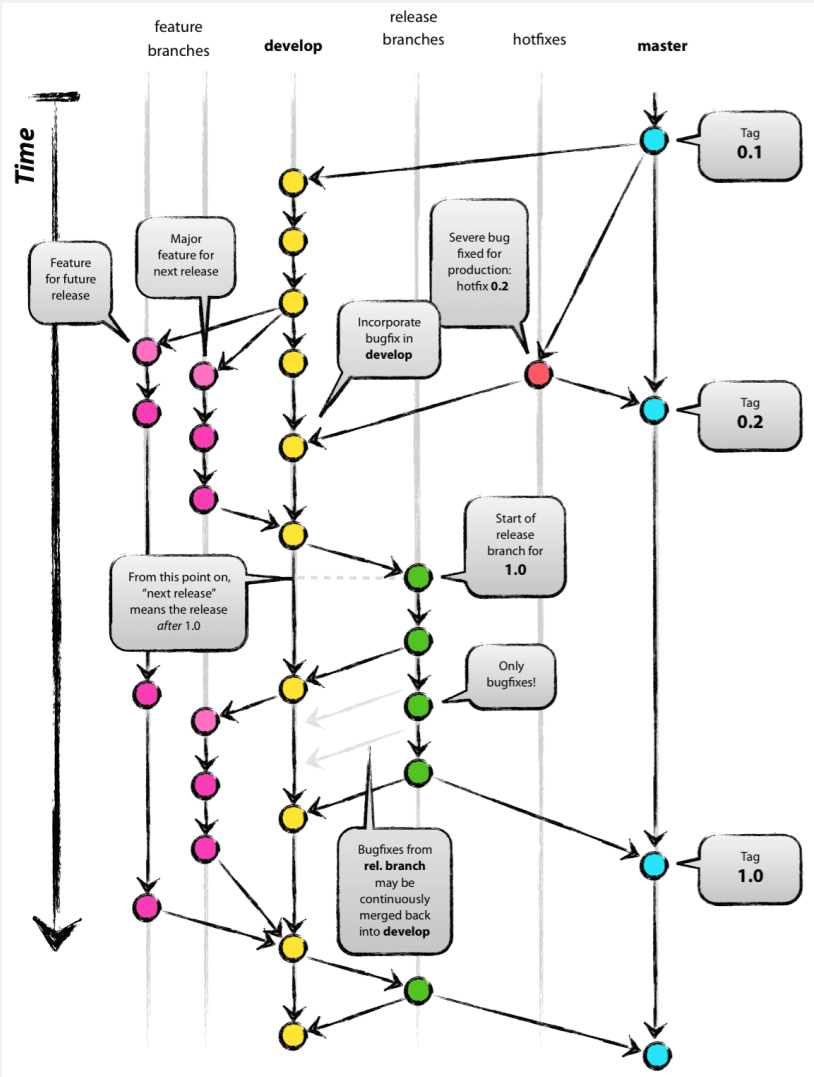
Pro

- simple to define and set-up ✓

Cons

- missing information: to find something, you need to know who did it ✗
- redundant information: committer and branch name are strongly related ✗
- confusing and error prone code sharing, blending new feature development and bug correction ✗
- developers use old version of common code ✗
- As it does not enforce convergence of code ⇒ the multiple incompatible development are increasingly difficult to merge ✗

Better splitting: per topic branches



- *master* branch contain release versions
- *develop* branch contain ongoing development
- *release* branch to prepare new release in master
- *hotfix* branch for bug correction of release version
- *feature* branch to develop new features

[Vincent Driessen,

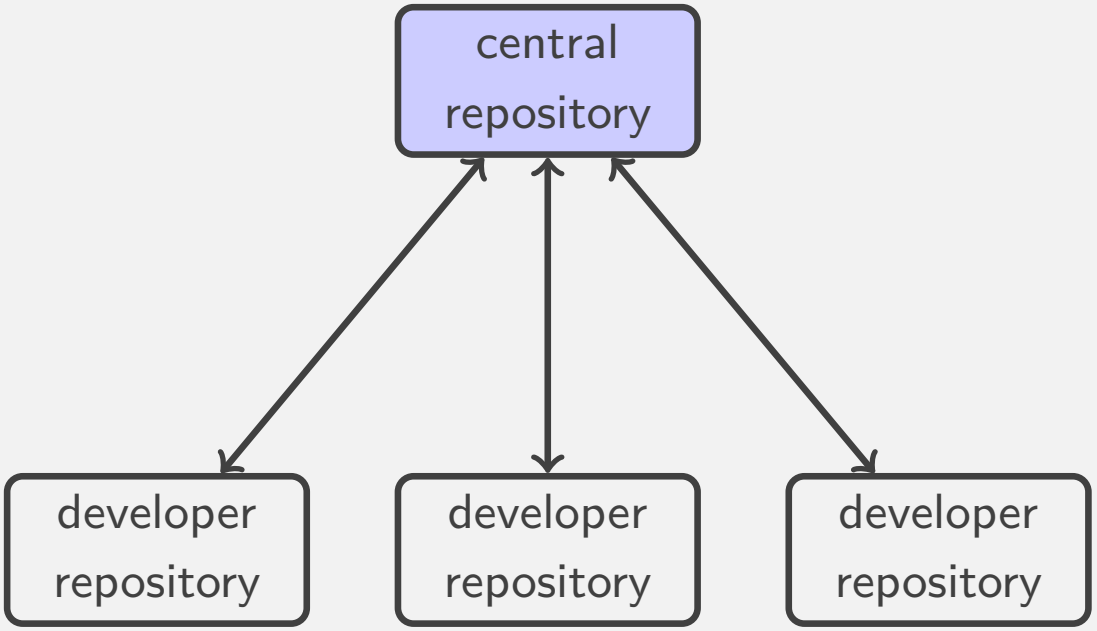
`http://nvie.com/posts/a-successful-git-branching-model/`]

Code sharing models

Distributed VCS such as Git are ... distributed.

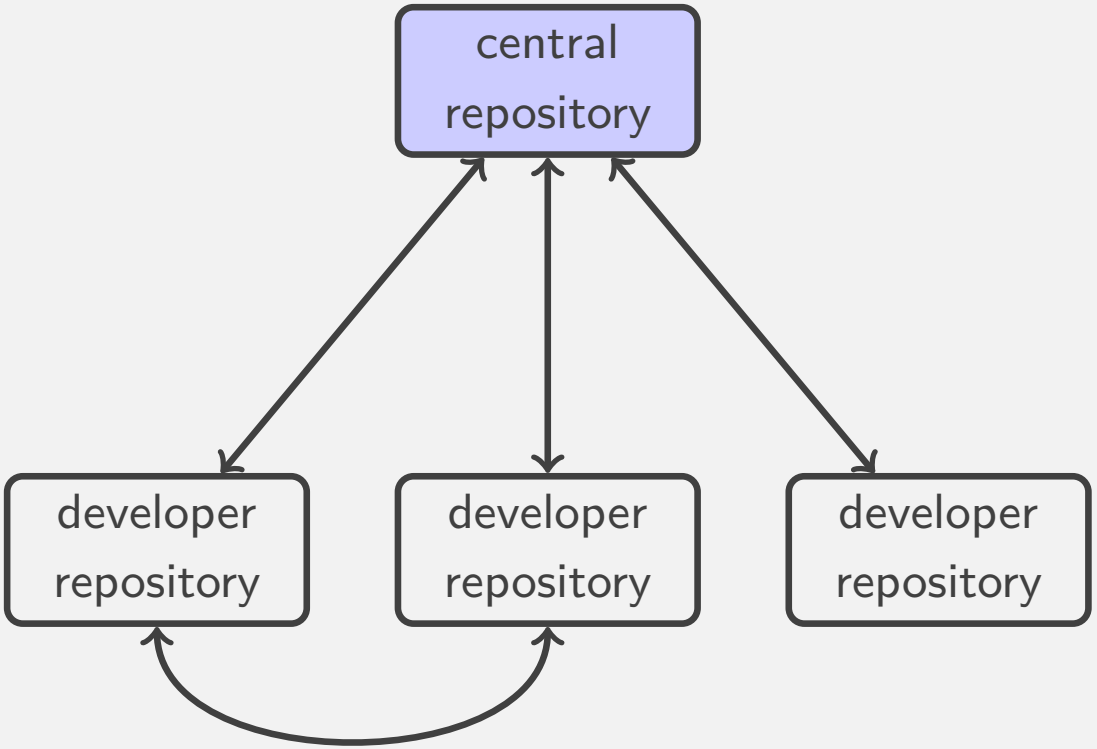
The code is thus modified at several places at the same time.
There are many different ways to share the code modifications between repositories.

Code sharing: centralized workflow



Similar to SVN way of life, just fully operational with branches, merge and off-line work. Quite good for small teams.

Code sharing: centralized workflow



Similar to SVN way of life, just fully operational with branches, merge and off-line work. Quite good for small teams.

Additional transfers using git are easy and safe !

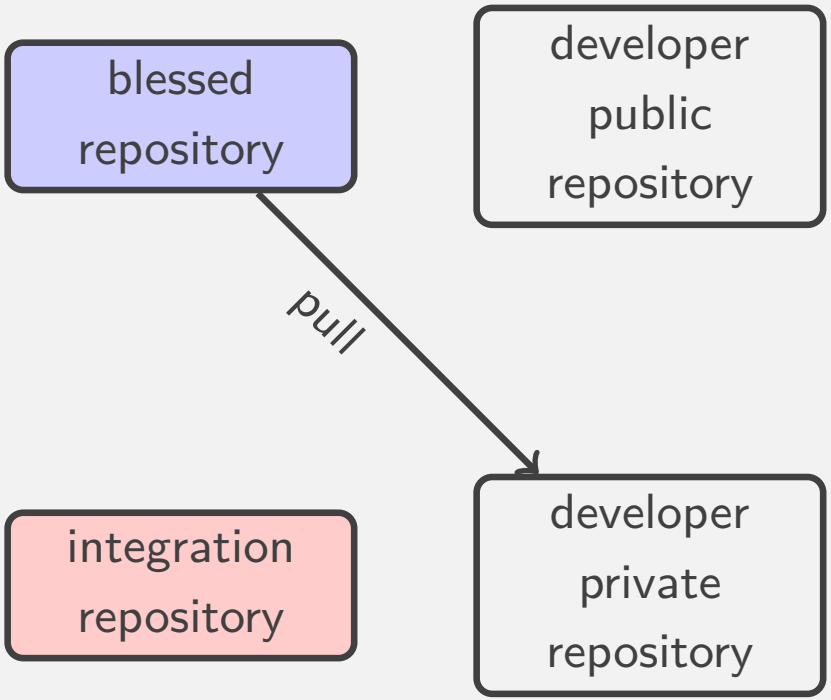
The tricky part

The most important point is *to use git* for transfer patches. Any transfer outside git can not be taken into account in futur merge.

Centralized workflow

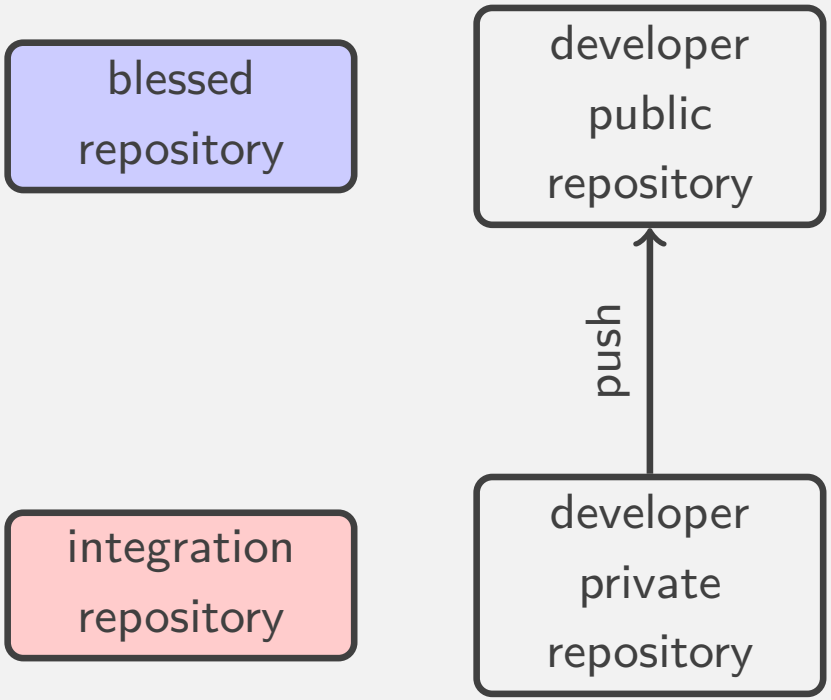
```
1  do {
2      while (nothing_interesting())
3          work();
4      while (uncommited_changes()) {
5          while (!happy) { // git diff --staged ?
6              while (!enough) git add -p;
7              while (too_much) git reset -p;
8          }
9          git commit; // no -a
10         if (nothing_interesting())
11             git stash;
12     }
13     while (!happy)
14         git rebase -i;
15 } while (!done);
16 git push; // send code to central repository
```

Code sharing: Github workflow



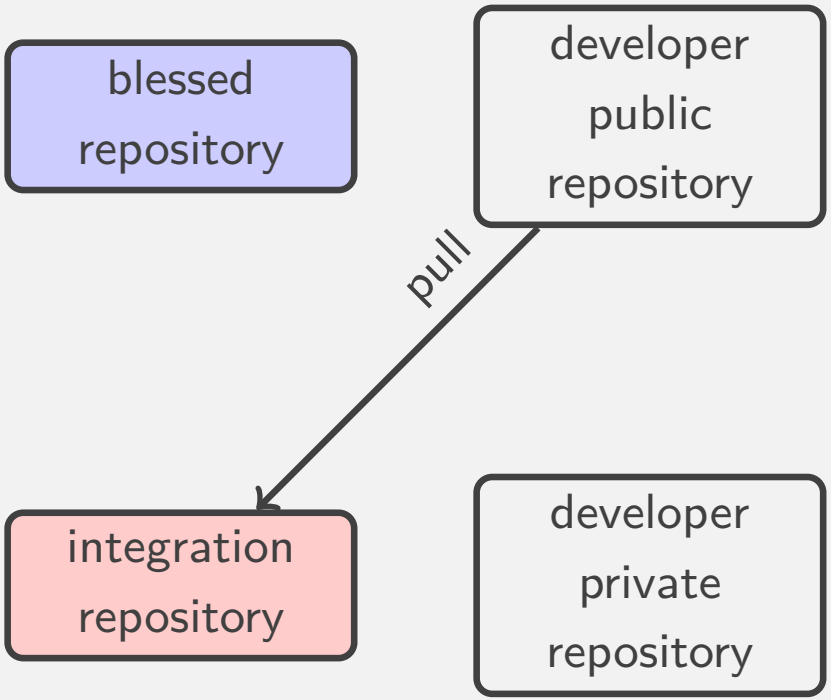
- Released and official branches are stored in the blessed repository.
- Contributors forks and works privately
- Contributors publish their work and ask for merge
- Integrators merges then publish the contributions

Code sharing: Github workflow



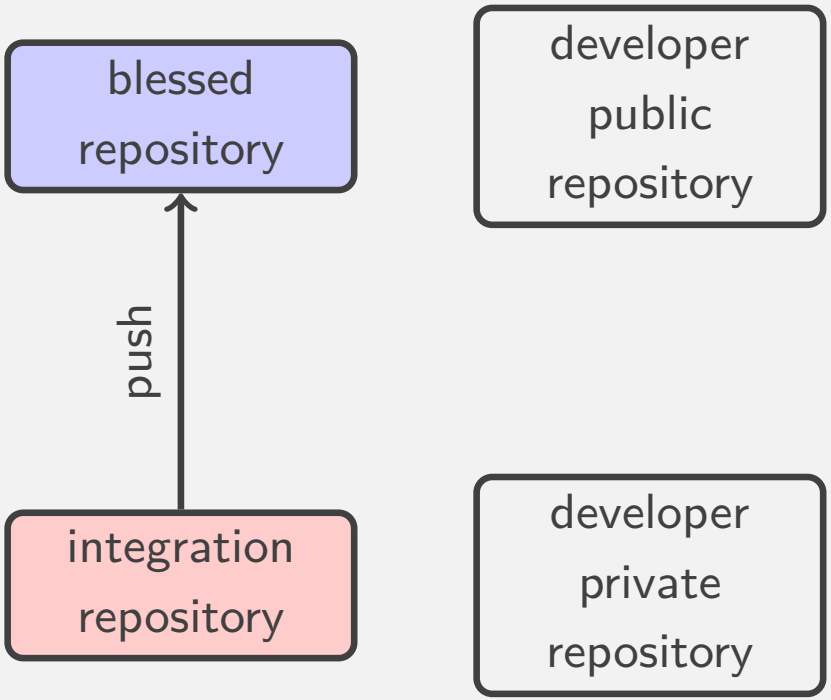
- Released and official branches are stored in the blessed repository.
- Contributors forks and works privately
- Contributors publish their work and ask for merge
- Integrators merges then publish the contributions

Code sharing: Github workflow



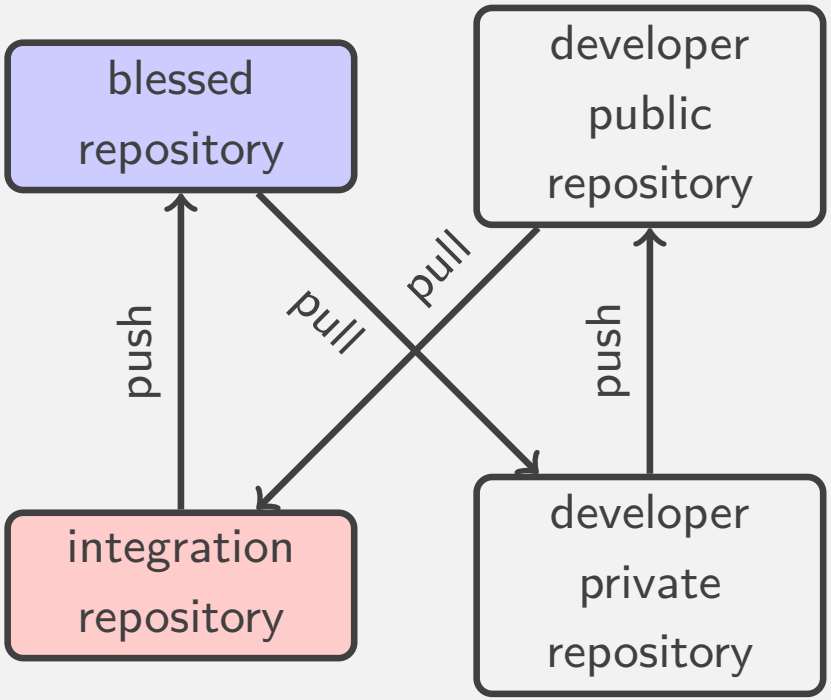
- Released and official branches are stored in the blessed repository.
- Contributors forks and works privately
- Contributors publish their work and ask for merge
- Integrators merges then publish the contributions

Code sharing: Github workflow



- Released and official branches are stored in the blessed repository.
- Contributors forks and works privately
- Contributors publish their work and ask for merge
- Integrators merges then publish the contributions

Code sharing: Github workflow



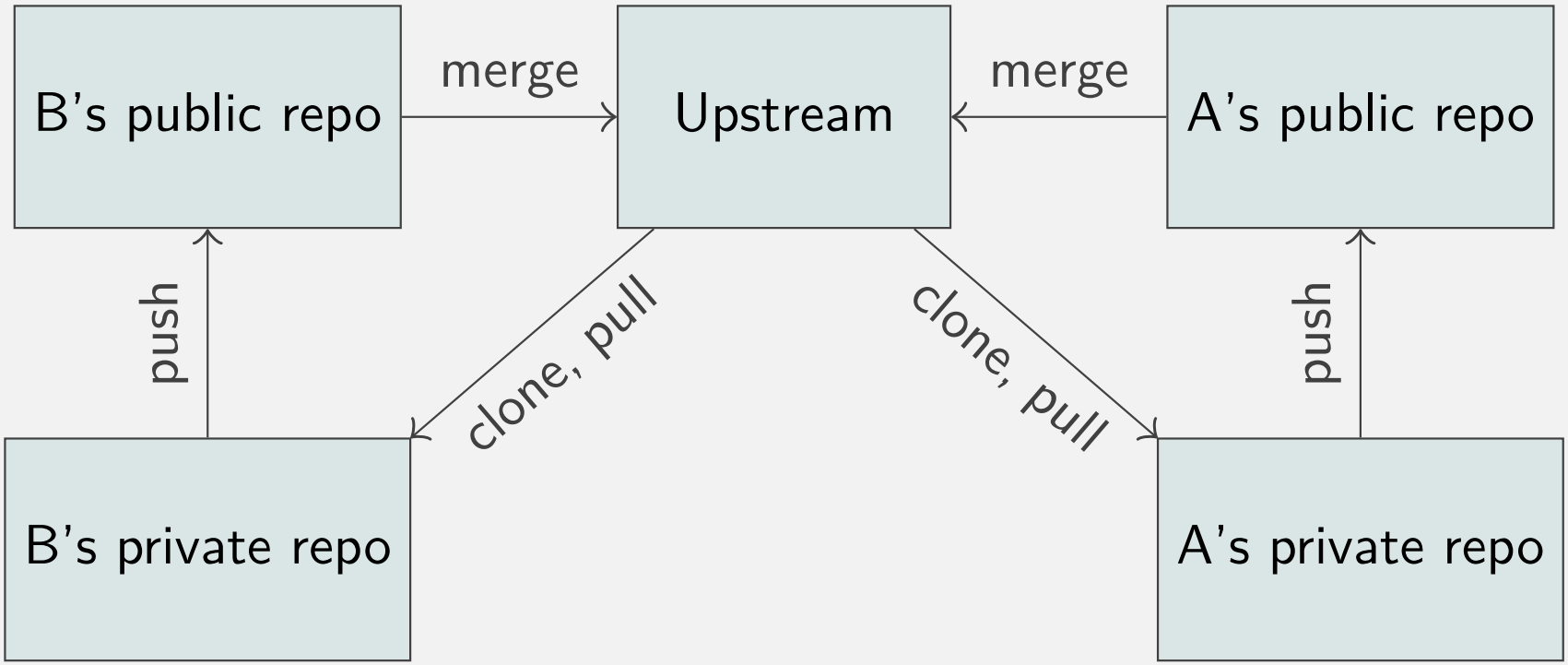
- Released and official branches are stored in the blessed repository.
- Contributors forks and works privately
- Contributors publish their work and ask for merge
- Integrators merges then publish the contributions

Public mailing list as public repository

Git patches can be send by email (`git format-patch -1`), published in mailing-list (eg. bug-gnu-emacs@gnu.org), then integrated (`git am`)

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

The code review is the major point for code quality insurance !

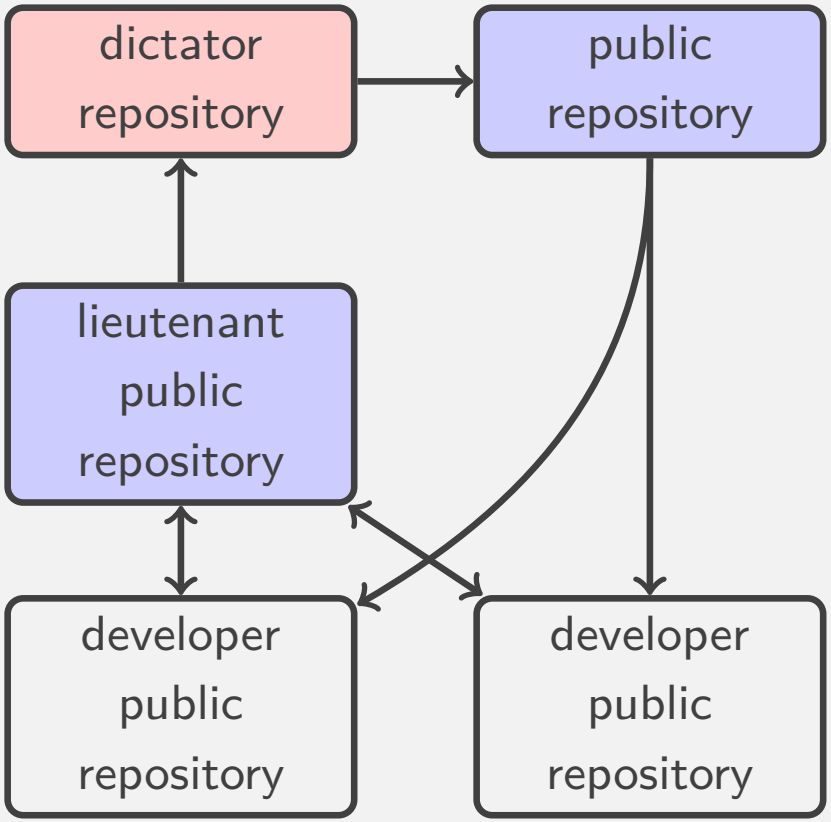
Code Review

- What we'd like:
  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.

Code sharing: Linux kernel workflow



- Code review and basic filtering of contributions is done by the lieutenants
- Final decision is done by the benevolent dictator
- Lieutenant repositories are the testbeds of new ideas that mature in it before upstream submission

Code sharing: remote management

Remote allows to work with several sources and sink

```
$ git remote add lieutenant git://.../public.git
$ git remote
origin
lieutenant
$ git fetch lieutenant
...
$ git branch -r
 origin/HEAD -> origin/master
 origin/master
 lieutenant/master
$ git checkout -b lieutenant lieutenant/master
```

Branches and tags in practice

Branches and Tags in Practice

- Create a local branch and check it out:
  `git checkout -b` *`branch-name`*
- 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`*