

GIT ESSENTIALS

October 2011

This image is the Linux kernel as visualised by Gource

Why Distributed Version Control?

- Builds that never break
- Work that is always backed-up
- Safe local updating and merging
- Flexibility around adopted workflows
- No single point of failure
- Knows the 'fallacies of distributed computing'



Branching as a process enabler

- You cannot have stable code without branches
stable lines must be isolated from development lines
- You cannot have code reviews without (some form of) branch
otherwise you cannot continue to work while waiting for reviews to happen
- For a DVCS, branching is mandatory since every local commit is a branch that potentially needs merging
- A DVCS is designed to be good at branching and merging



Why GIT?

- Seems to be where the momentum is
- Already very stable and mature
- Beautifully simple semantic model
- Fast, especially under Linux
- Stable tools, e.g. Eclipse support
- Branch per task is practical



But...

- Git is harder to learn than a typical centralised VCS, it has more concepts and more commands
- Git is extremely flexible, but that demands disciplined processes and conventions



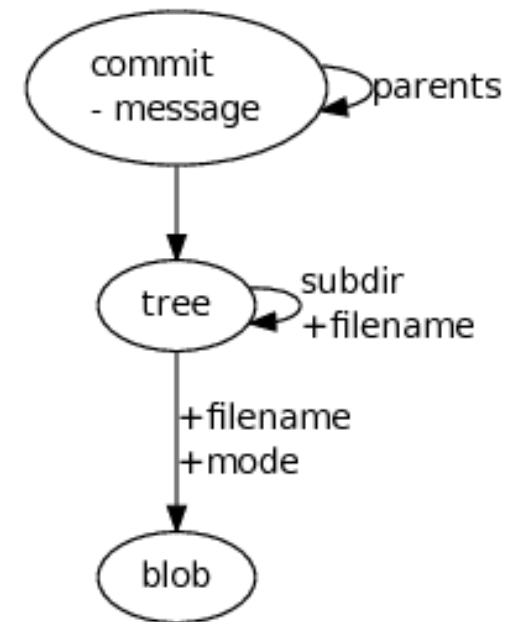
Git tracks content, not files

It stores three types of data separately:

- content is stored in blob objects
- history is stored in commit objects
- folder structure is stored in tree objects

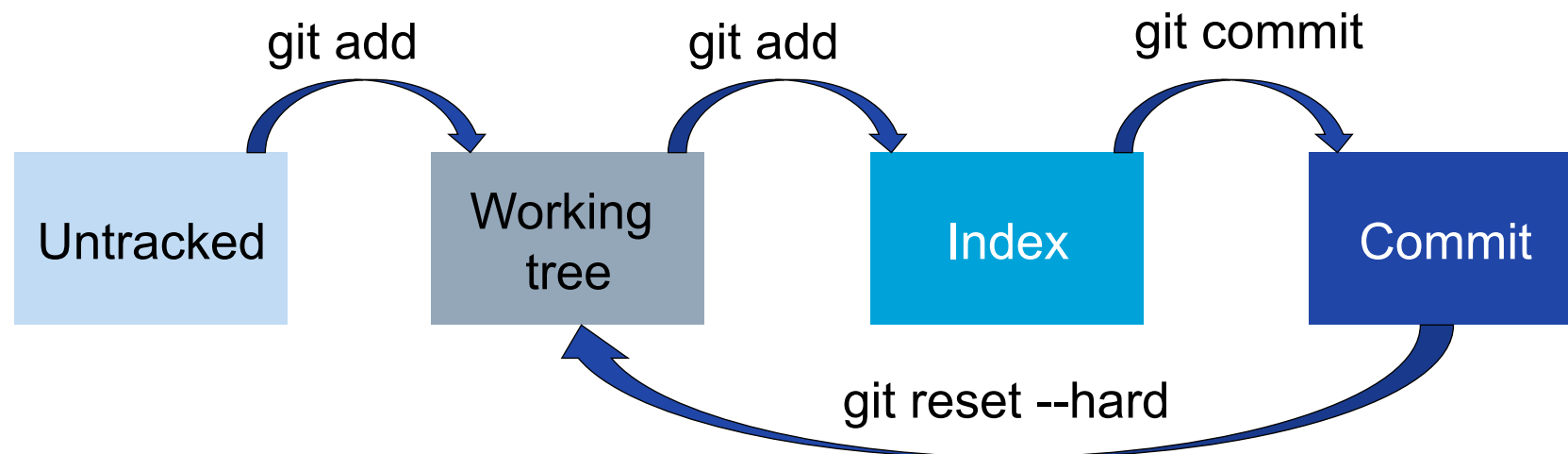
This allows:

- full merge accounting of non-linear histories
- tracking the history of code, which may pass through many files
- fast path-limited revision traversal



The Working Tree and Index

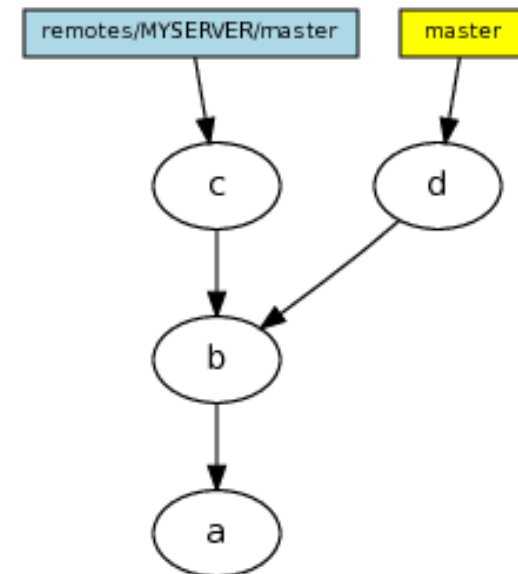
- Git commands such as “git add” and “git rm” work against the index, which is used to generate the next commit
- Changes to your working tree do not affect the index, changes are **staged** using the above commands
- Provides a place to store an unfinished merge, so you can try various strategies, including hand-editing, to finish it



Commits

- A single, atomic change-set with respect to the previous state
- Represents the entire repository, since we snapshot the index to create a new tree object representing the repository root
- Represents an entire line of development, since each commit points to its predecessor

- form a directed-acyclic graph, when we branch
- self-identifying and secure using SHA1 hashes



Branches

There are two types of branches in Git:

- Local branches

represent **your** branches, use “git branch” to see them. They can be set to track remote-tracking branches

- Remote a.k.a. “remote-tracking” branches

represent a snapshot of **someone else's** branch, use “git branch -r” to see them and “git fetch” to update them

To create and checkout a local branch that tracks a remote:

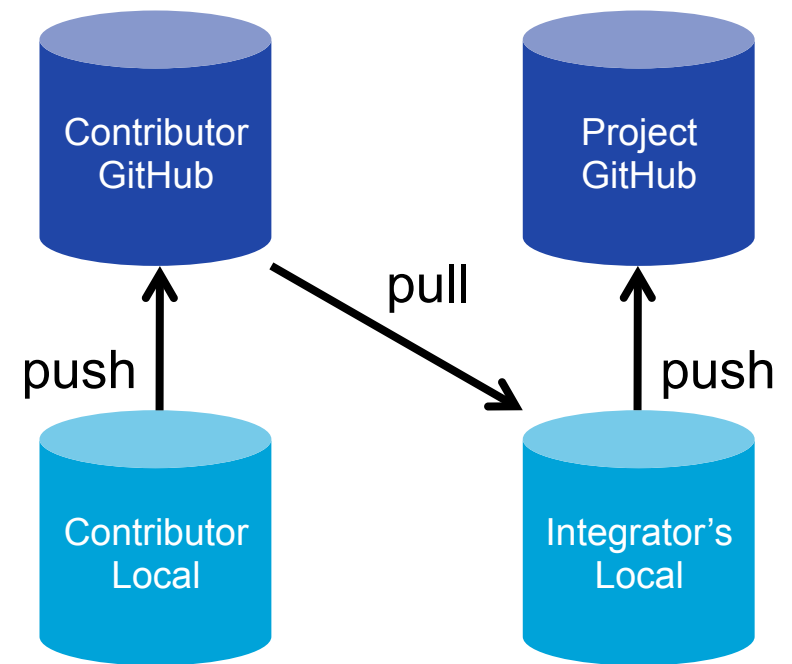
```
git checkout --track -b experiment origin/experiment
```

Remotes

- Git can have many peers
- these peers, called remotes, can thought of as simple aliases for long URLs

To add a new remote:

```
git remote add github <url>
```



Refs

- a ref is a SHA1 hash pointing to a git commit
- named refs are stored in `.git/refs` according to their fully qualified names
For example `.git/refs/remotes/origin/master` contains the (last known) SHA1 commit of the origins master branch
- special refs exist, e.g. `HEAD` which means the latest commit on the current branch
- relative commits can be accessed using a tilde
For example `HEAD~2` references two commits before `HEAD`
- ranges can be specified using double dots
For example `HEAD..HEAD~2`
- branches and tags are just named refs
Note branch refs can move, tags cannot



Tags

- first class citizens in Git
- can be used to start new branches or simply mark milestones in the code's lifetime
- by default, “git tag” creates a simple named ref, essentially a branch that never moves
- better to create annotated tags using “git tag -a” or signed tags
- use “git describe --tags” to show how many commits you are past the last or supplied tag



Common commands

- Creating

- git init
- git clone

- Querying

- git status
- git show
- git log

- Updating

- git add
- git commit
- git fetch
- git merge
- git pull

- Undo

- git reset
- git clean
- git revert

- Powertools

- git rebase
- git cherry-pick
- git bisect
- git stash
- git blame



Subversion equivalents

Old world

svn checkout <url>
svn update
svn update -r <rev>
svn revert
svn add/rm/mv
svn commit

New world

git clone <url>
git pull
git checkout <rev>
git checkout
git add/rm/mv
git commit



Merging

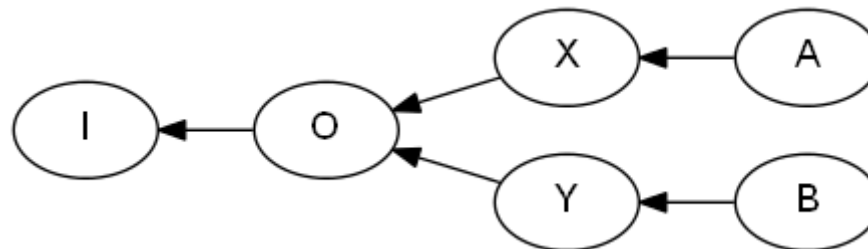
- Happens whenever we “git pull” or “git merge”
- No Conflicts:
 - Git creates a new merge commit, if the merge is non-trivial.
 - If the merge is trivial, ie. just an update, it **Fast-Forwards** the commits
- Conflicts:
 - changes alter the same line of the same file
 - must be resolved before a merge commit can be created



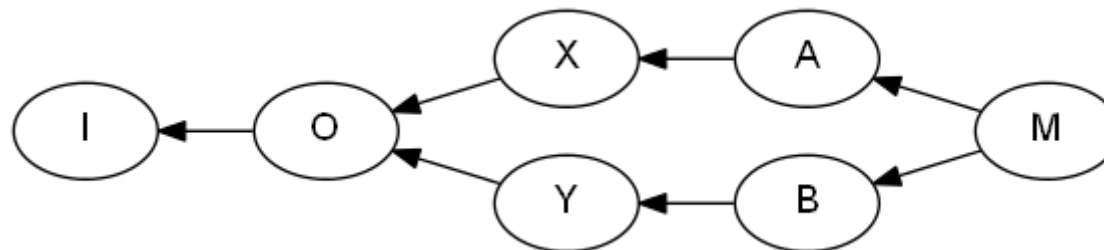
Merging

- Git has pluggable merge strategies and many to choose from
- By default Git uses the 'recursive' strategy to perform a basic three-way merge. It applies it to whole files, and then to lines within files.

To do a basic three-way merge, you need three versions of a file. The versions A and B you want to merge, and a common ancestor O.



We want the file O, plus all the changes made from O to A and from O to B.



Merging: common strategies

- **Fast-forward** (default trivial)

- simply replays the commits onto a common parent
- used, for example, to update a developer's remote copy
- use "--no-ff" if you explicitly want the merge in your history when doing "git pull" or "git merge"

- **Recursive** (default non-trivial)

- performs a basic three-way merge, unless there are multiple common ancestors, in which case it attempts to merge the ancestors and then use the result as a common base

- **Ours**

- abandon any conflicting changes in the feature branch, but keep them in the history

- **Subtree**

- for merging an independent project into a subdirectory of a superproject

Merging: Resolving conflicts

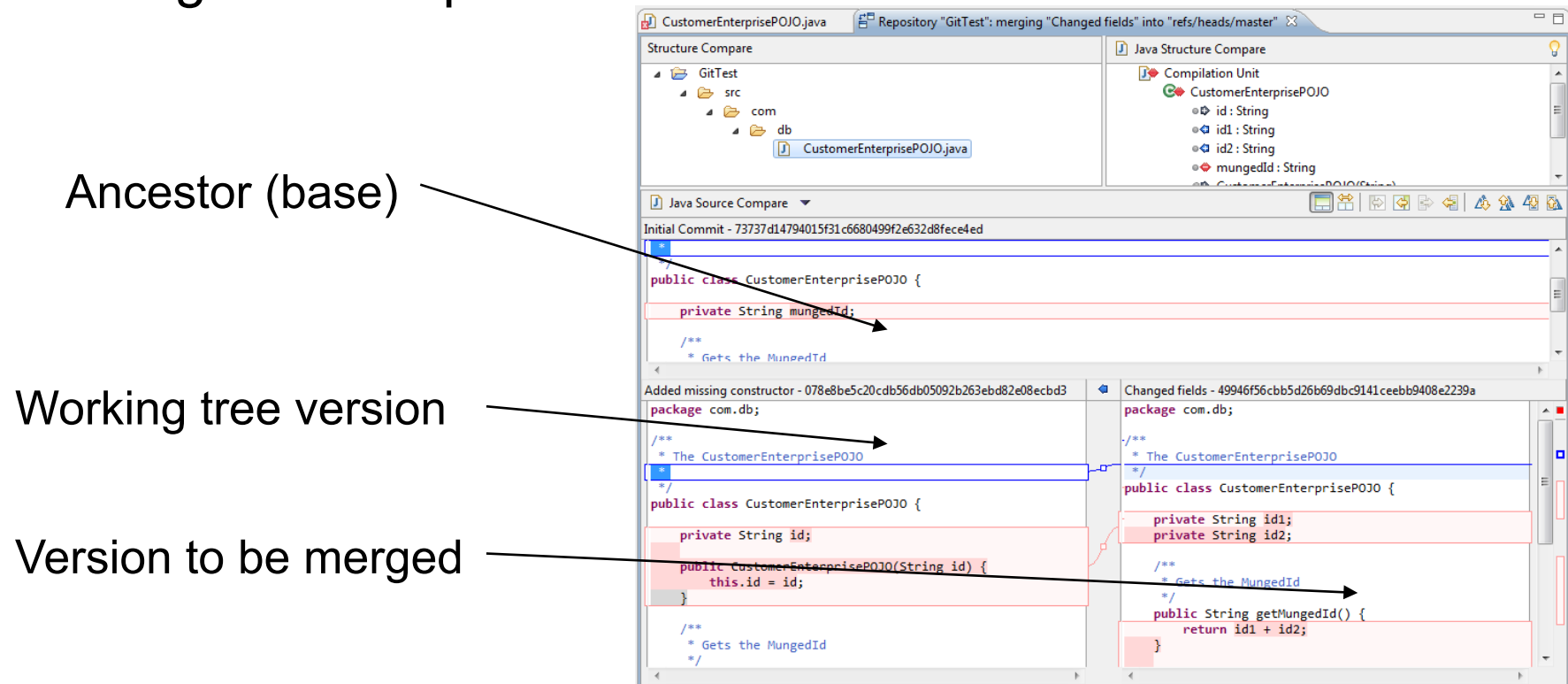
- a merge (via git pull or git merge) may result in a conflict

```
Auto-merging DemoServer/Java/pom.xml
CONFLICT (content): Merge conflict in DemoServer/Java/pom.xml
Auto-merging WebServer/Java/run.bat
CONFLICT (content): Merge conflict in WebServer/Java/run.bat
Auto-merging Bandwagon Examples.iws
CONFLICT (delete/modify): Bandwagon Examples.iws deleted in 682a683d05f763bb246a
439033e3e1e63ccff7b6 and modified in HEAD. Version HEAD of Bandwagon Examples.iw
s left in tree.
```

- while in a conflicted-merge state, the index holds three versions of each conflicted file: base, ours and theirs
- the conflicted files in the working tree also contain markers, showing the conflicted lines
 - “git status” will list all the modified files brought in by the non-conflicting commits. It will also list the conflicted files.
 - “git reset --hard” aborts the merge

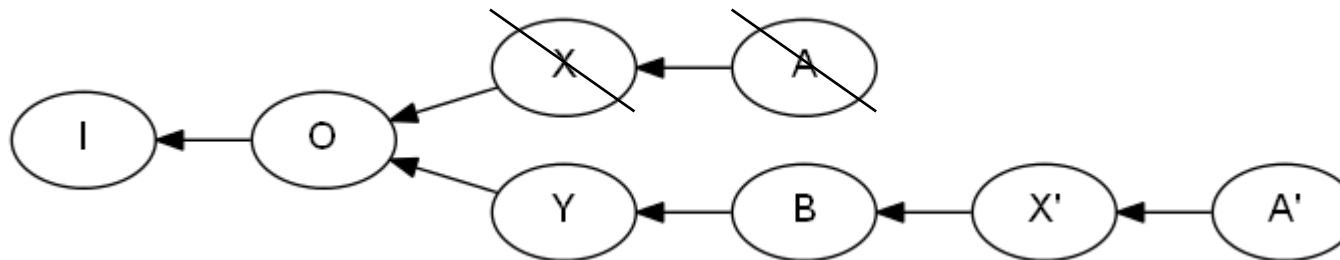
Merging: Eclipse and EGit

1. right click a conflicted file
2. select Team -> MergeTool
3. select the merge mode
use HEAD (the last local version) of conflicting files" and click OK
4. the merge editor opens



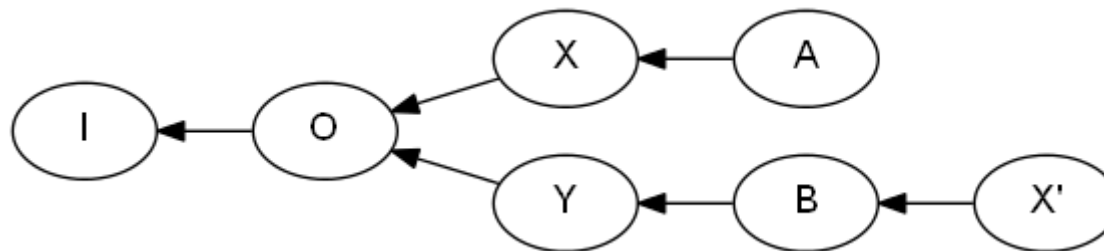
Rebasing

- best thought of as re-writing history
- should not be done to commits already published!
- useful for cleaning up a noisy and confusing private history before publishing
 - especially if some bad intermediate commits may cause problems for tools such as “git bisect”
- the interactive rebase “git rebase -i” can be useful for squashing a series of recent commits into one bundle for publishing



Cherry picking

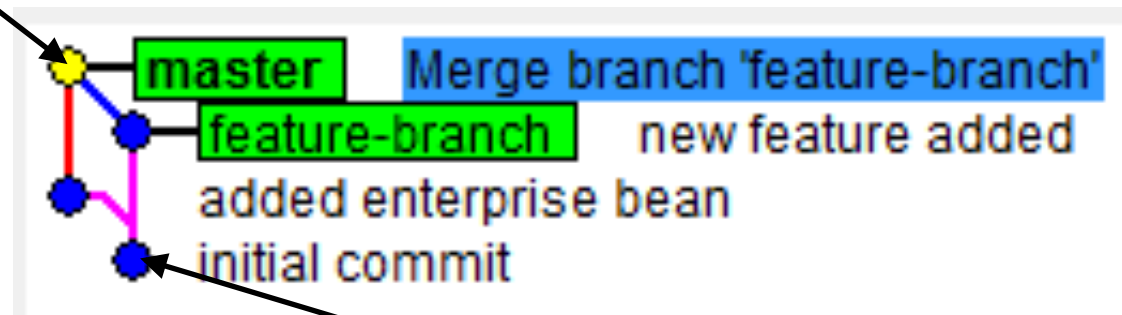
- allows you to "cherry-pick" one or more commits from within an arbitrary development line
- creates a new commit on top of your current HEAD
- if it cannot apply the change, conflicts are resolved similarly to "git merge"
- often an alternative to rebase, which can be thought of as a series of cherry-picks, followed by a branch reset



Visualisation

- gitk included with git
- Run using “gitk” or “gitk --all” for all branches
- Eclipse EGit offers similar graph views in the History view
For example, Team -> Show in History

Merge commit



Common ancestor

Renames

- Git doesn't record any rename tracking information at commit time
- Renames are detected using heuristics. To make sure this works, always commit moves separately from content changes.
- Use “git log --follow <filename>” to view the history of a file across renames



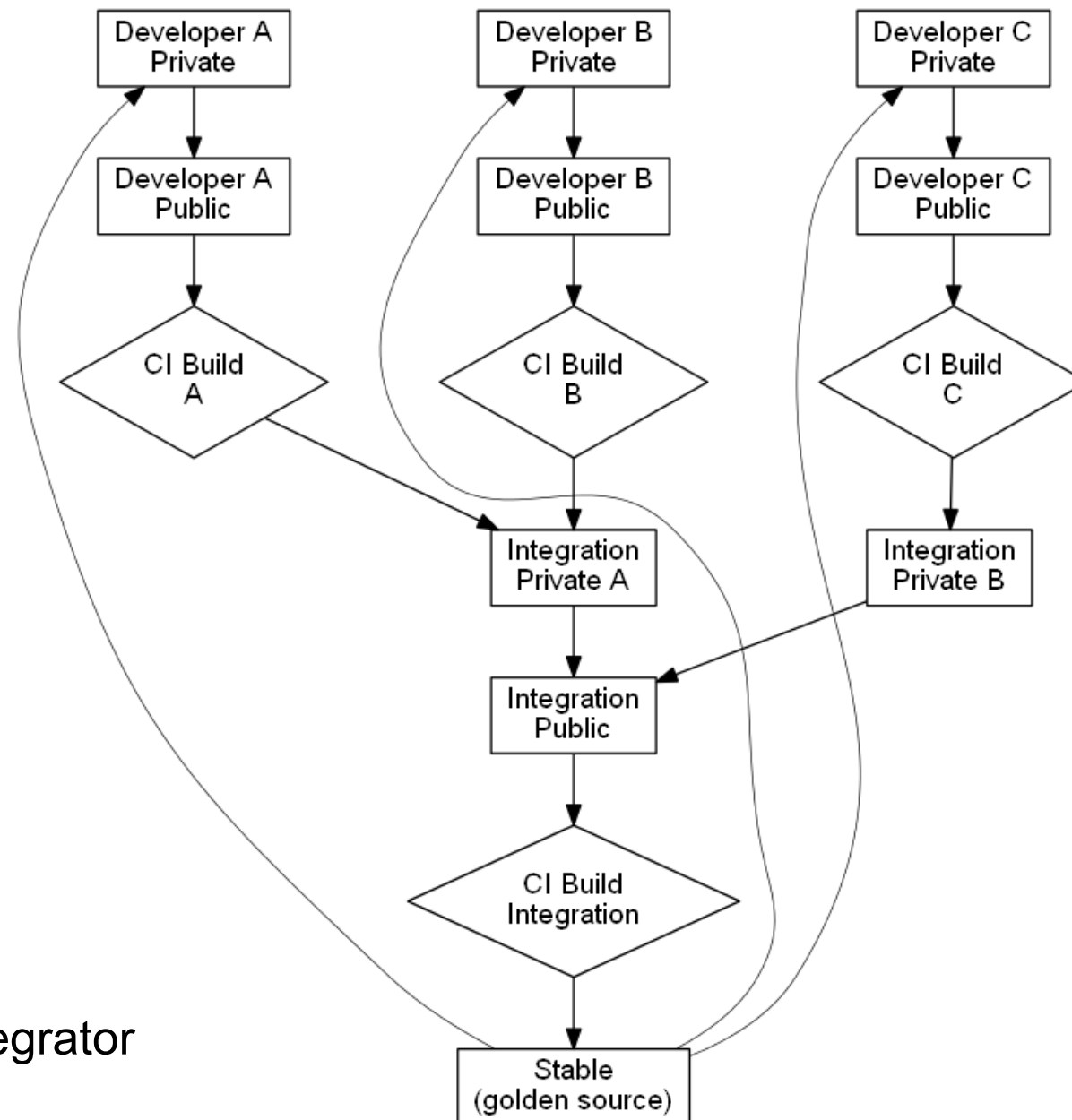
Workflow

An ideal workflow for moderately sized teams would feature:

- **A stable “golden source” repository** that developers pull from and releases are cut from, that always builds
- **In-progress work is backed-up remotely**
 - this includes branches that may never make it into the golden source.
- **A Branch-per-task methodology**
 - made practical by Git's full merge accounting and scalable architecture.
- **First class code reviews**
 - using a collaboration platform like GitHub, code review is a trivial add-on.



Workflow



Roles:

- Developer
- Reviewer/integrator

Best practice

- all work should be done on ticket-linked branches
- commit and push regularly, especially after renames
- the person responsible for the pull request should resolve conflicts should their branch fall behind master
- work is not done and tickets are not closed, until code has at least made it to stable



Setting up

1. Set up your configuration:

```
git config --global user.name "Tim Williams"
```

```
git config --global user.email tim@timphilipwilliams.com
```

- Three levels of config: system, global and local to the repository
- You can view your configuration by doing "git config -l"
- On Windows it is worth checking that "core.autocrlf" is set to false

2. Set up ssh keys:

```
ssh-keygen -t rsa
```

- add your keys here %HOME%\ssh\id_rsa
- private key should really have a password

3. Upload your public key to GitHub

- taking care to avoid copy-paste errors!



Resources

- Details of various Git documents and books
 - <http://git-scm.com/documentation>

