Faculty of Engineering and Technology
Department of Computer Science

4116COMP Computer Science Workshop

Dr Gabor Kecskemeti
g.kecskemeti@ljmu.ac.uk
Room 701, Byrom Street

1

Version control with Git

In this session…

- Version control in general
- Version control approaches
- Git essentials
- Branching and merging
- Advanced git
- Git workflows
- Things to remember
- Version control issues with computer science workshop

Version control in general

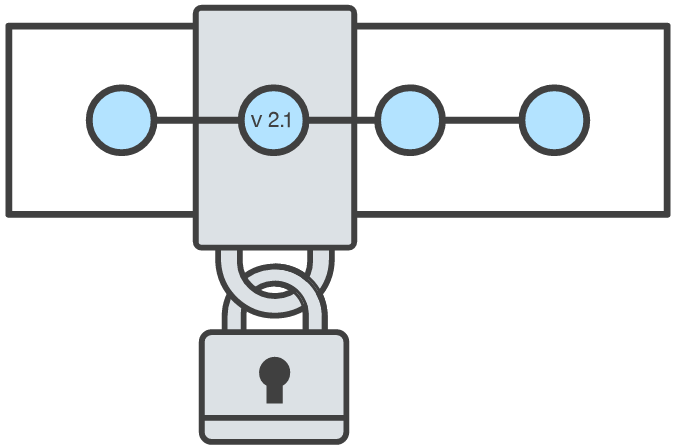
Version Control

Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later.

Version control is like a big undo button for your project. It allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover.

Goals of Version Control

- Be able to search through revision history and retrieve previous versions of any file in a project
- Be able to share changes with collaborators on a project
- Be able to confidently make large changes to existing files
- Work on new features but still be able to go back and fix a bug in the released version of the software.

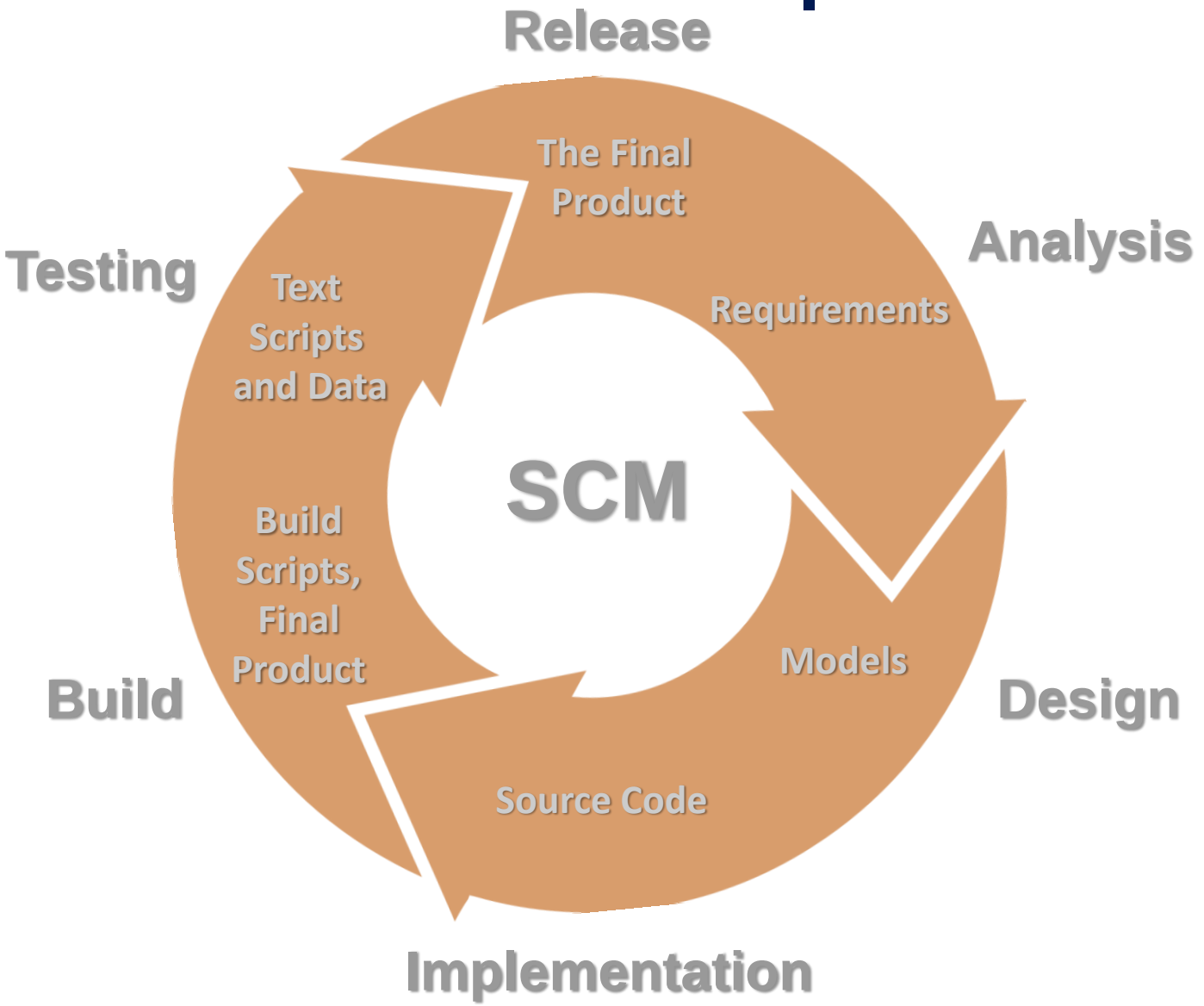


Development and Maintenance Use Cases

Version control is essential when you have multiple developers working on the same code base.

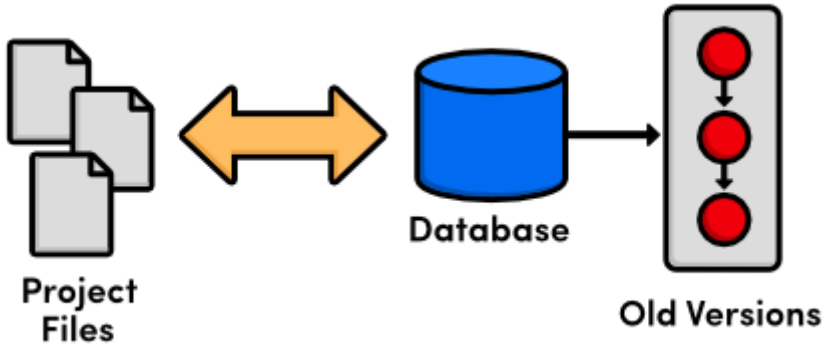
Version control is essential when you are supporting multiple versions of a software system over time. When a bug is reported in one version of the product you have to rebuild that version of the product, fix the bug and ship an updated version.

SCM and the Software Development Lifecycle

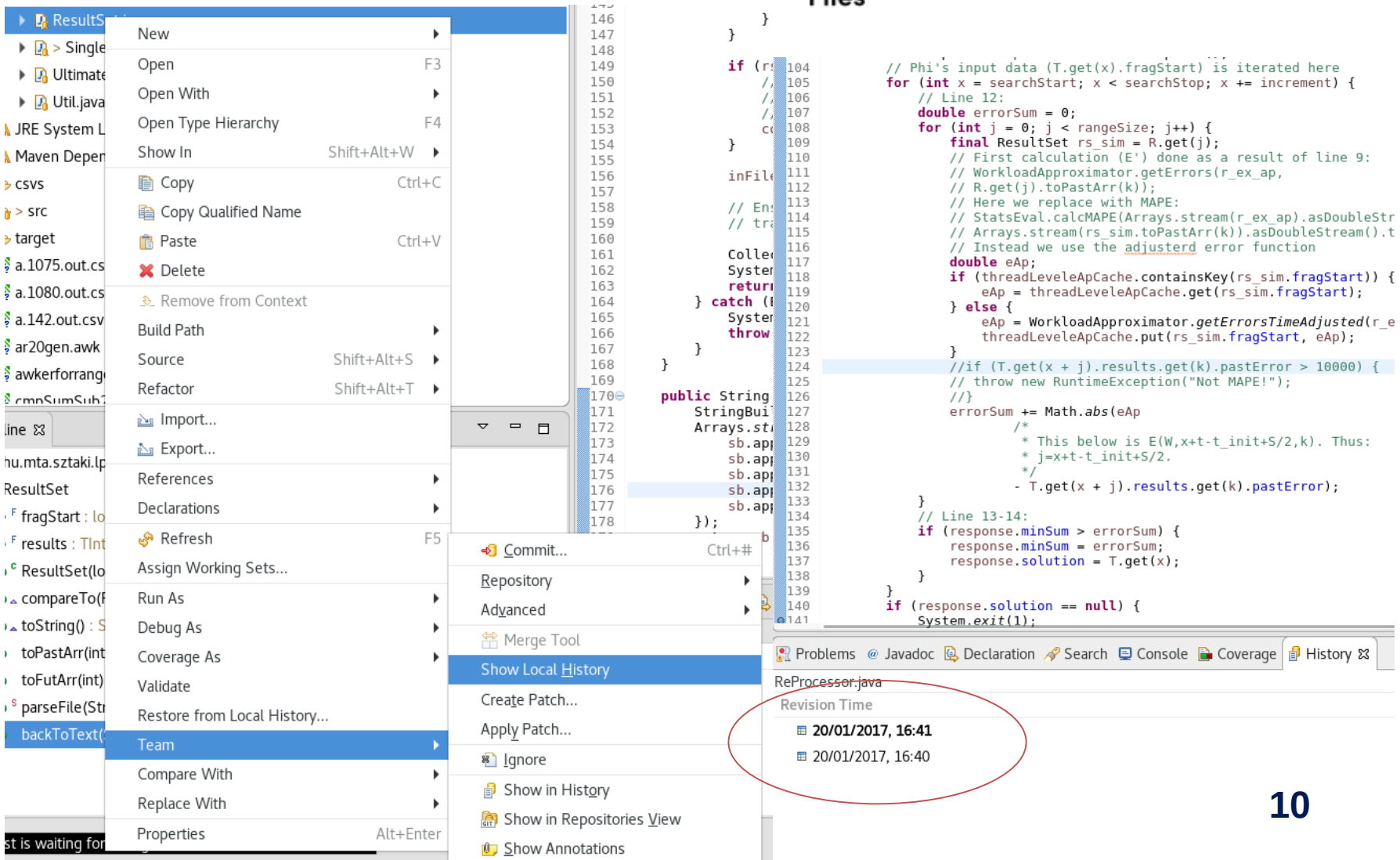


Approaches of version control

Local Database of Versions Approach



- Provides an abstraction over finding the right versions of files and replacing them in the project
- Can’t share with collaborators
- Eclipse has it (see on the side)



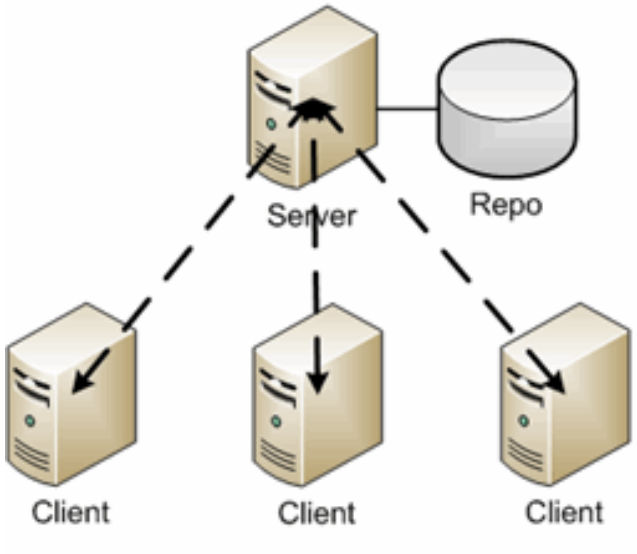
Lock-Modify-Unlock Model – Multi-user approach #1

- Only one user works on a given file at a time
  - No conflicts occur
  - Users wait each other for the locked files → works for small development teams only
  - Pessimistic concurrency control
- Lock-modify-unlock is rarely used
- Problems:
  - Someone locks a given file and forgets about it
  - Time is lost while waiting for someone to release a file → works in small teams only
  - Unneeded locking of the whole file

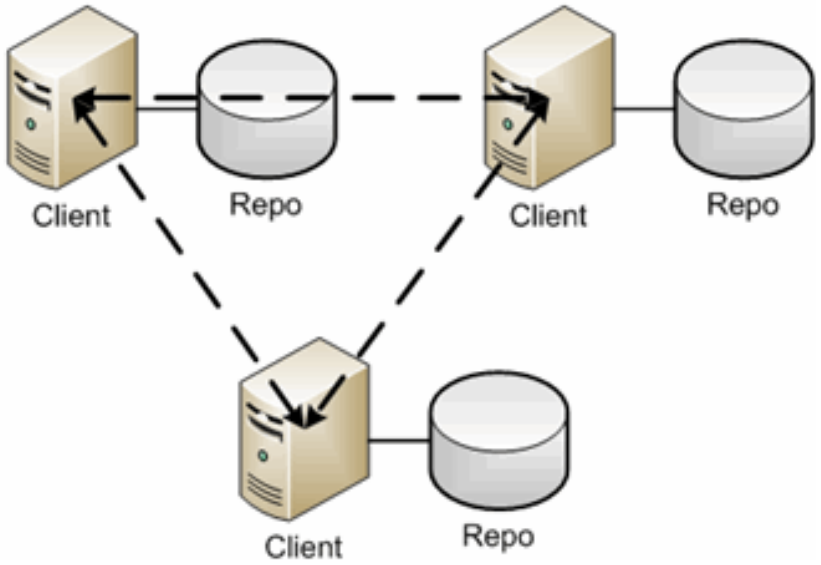
Copy-Modify-Merge Model – Multi-user approach #1

- Users make parallel changes to their own working copies
- Conflicts are possible when multiple user edit the same file
  - Conflicting changes are merged and the final version emerges (automatic and manual merge)
- Optimistic concurrency control

Two main approaches for Version Control



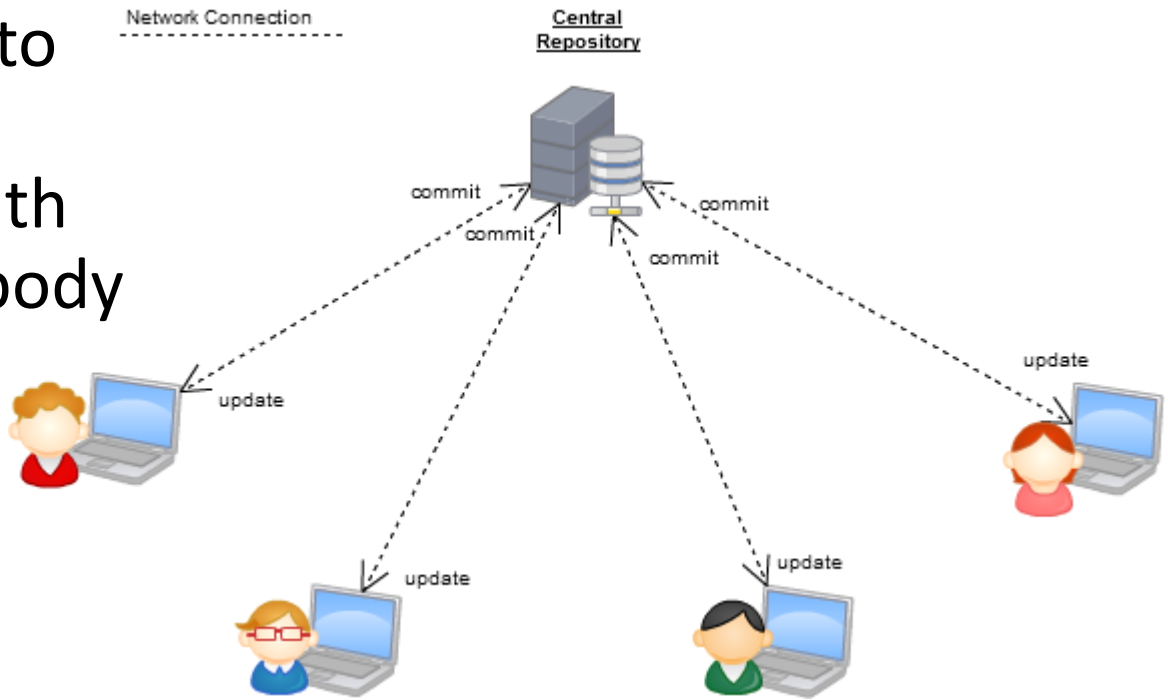
Centralized



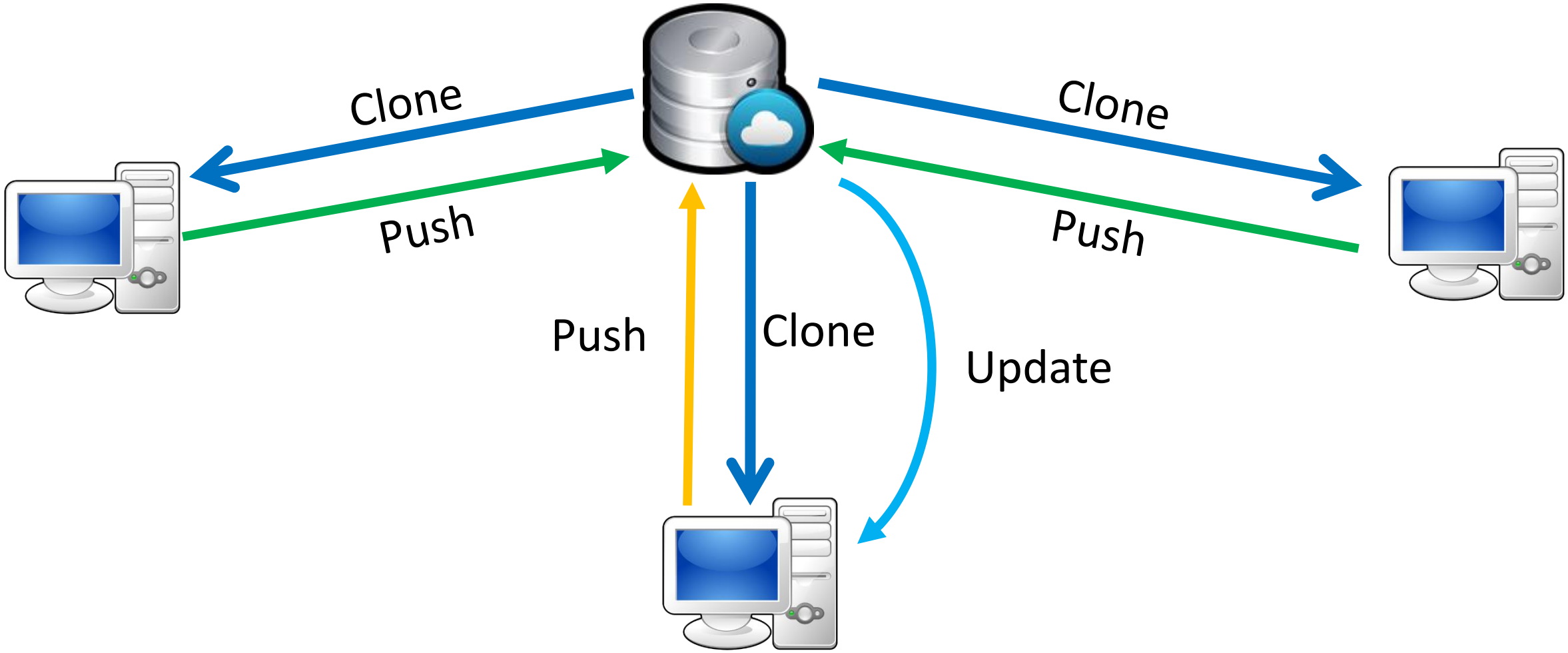
Distributed

Centralized Version Control Systems

- A central, blessed repository determines the **order** of commits (“versions” of the project)
- Collaborators “push” changes (commits) to this repository.
- Any new commits must be compatible with the most recent commit. If it isn’t, somebody must “merge” it in.
- They often discourage contribution:
  - My code is too ugly !
  - I’m not finished just yet!
  - Nobody cares about what I do anyway …
- Examples: SVN, CVS, Perforce

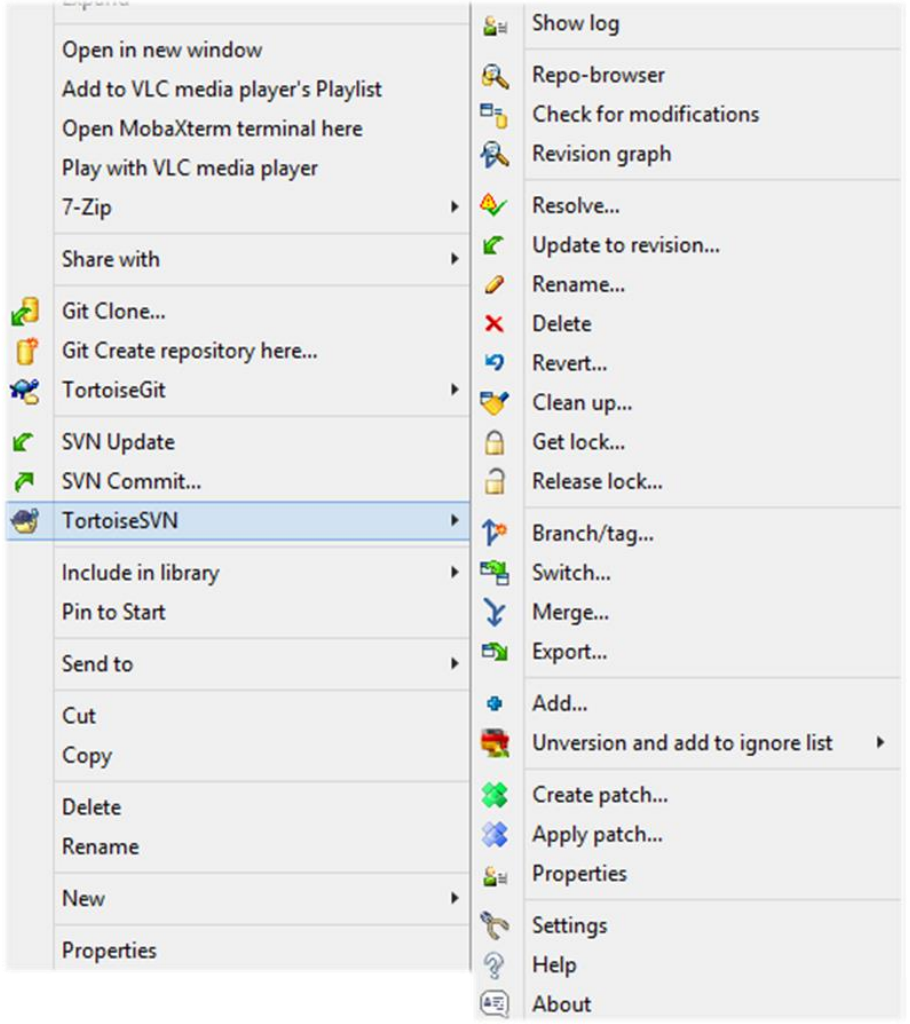


How it works?



Subversion (SVN)

- Subversion (SVN)
  - Open source SCM repository
  - http://subversion.tigris.org
  - Runs on Linux, Windows, Mac OS
- Console client
  - **svn**
- GUI client – TortoiseSVN
  - http://tortoisesvn.tigris.org
- Visual Studio / Eclipse plug-ins



Distributed Version Control

- Users work in their own repository
  - Using the Lock-Modify-Unlock model
  - Local changes are locally committed
  - No concurrency, no local conflicts
- From time to time, the local repository is pushed to the central repository
  - Conflicts are possible and merges often occur
- Example of distributed version control systems:
  - Git, Mercurial

What is Git?

- Free Software Development Tool
  - Speedy tool for distributed revision control and source code management
- Designed by Linus Torvalds for Linux kernel development, design goals:
  - Speed
  - Simple Design
  - Strong Non-linear Development Support
  - Fully Distributed
  - Suport for Large Projects
- Git directories feature full revision tracking functionality
- Not dependent on network access or central servers

git

git has changed the way developers think of branching and merging.

Branching and merging tends to be a big scary event with centralized repositories.

Branching and merging are part of the daily workflow of developers using git. git supports non-linear development.

Why not SVN?

“If it’s not distributed, it’s not worth using”
– Linus Torvalds, 2007

- Git repositories have a much smaller footprint compared to SVN
  - Mozilla switched from SVN to git – reduced size from 12 Gb to 400 Mb
- Git is significantly faster than SVN
  - Commits are local
  - Branches are almost instant, even for large projects

Why not SVN? (cont'd)

- Centralized means some people have commit access – some don't
  - How do you determine who gets access?
- Git assumes other developers are idiots*
- All developers can make their own changes independently of each other
- Developers can decide to pull changes from each other

* As explained by Linus Torvalds

Why not SVN? (still cont'd*)

- All repositories are full backups
  - Complete with entire commit history
  - Theoretically, no repository is more important than any other
- Branches are made almost instantly
- Branches carry entire repository history

Purpose of introducing GIT

- Required for learning outcome (i.e., version control)
- Show the versions of your code in the course of development
- Show versions of your progress and aids your documentation efforts.

Git essentials

Git terminology (1/4)

- ***Git Directory***
  The .git directory that holds all information in the repository
  Hidden from view
- ***Working directory***
  The directory in which .git resides
  Contains a "snapshot" of the repository
  Will be changed constantly eg. when reverting or branching
  Your changes will be made to this directory
- ***Index***
  Changes have to be added to the index from the working directory in order to be saved into a commit
  Could be thought of as a “loading bay” for commits

Git terminology (2/4)

- ***Branch***

  An alternate line of development
- ***Working copy***

  The branch you are in now that you make your changes in
- ***Master***

  The default branch of the repository

Git terminology (3/4)

- ***Commit***
  - A set of changes that have been saved to the repository
  - Can be reverted **and even modified to some extent**
  - Identified by a hash of the changes it contains
- **Tag**
  - A certain commit that has been marked as special for some reason
  - For example used to mark release-ready versions

Git terminology (4/4)

- **HEAD**

  The latest revision of the current branch

  Can be used to reference older revisions with operants

  HEAD^^2 == 2 revisions before latest

  HEAD^ == 1 revision before latest

  HEAD~3 == 3 latest revisions

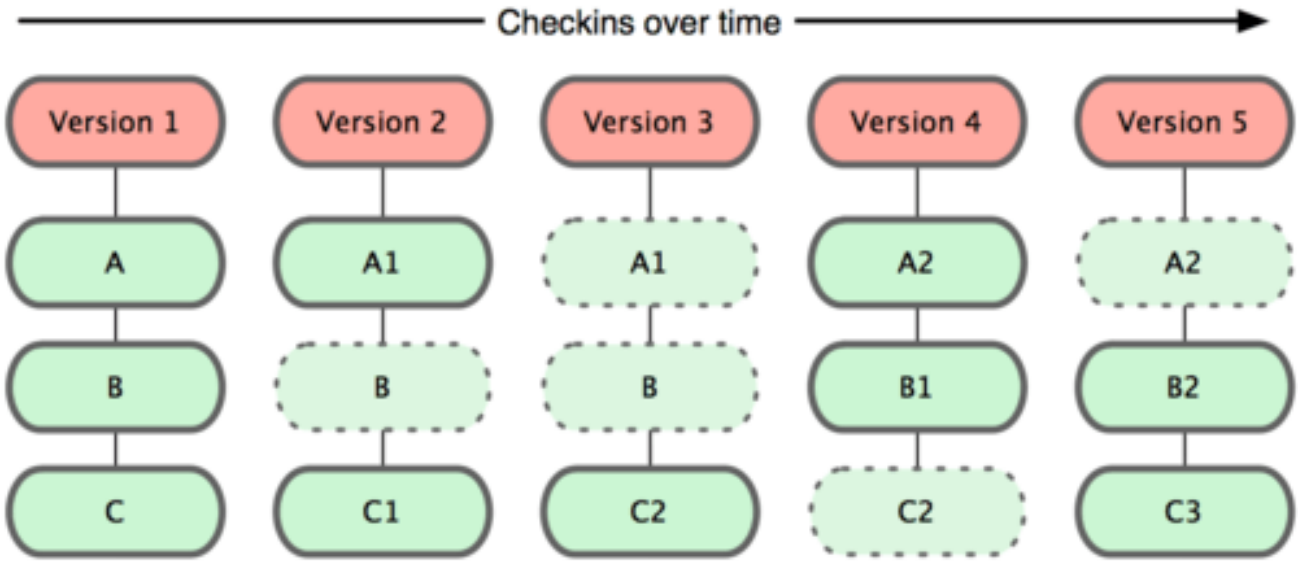
  HEAD^^2..HEAD == 2 revisions before the latest to the latest

- **Origin**

  Default name for a remote repository when cloning an existing repository

How does it work?

- You: Files files files; Git: Snapshot!
- Every '***commit***' is saving the state of your project
  - A snapshot of how the files look like at that moment and stores a '**reference**' to that snapshot
  - File is stored ONLY if it has changed



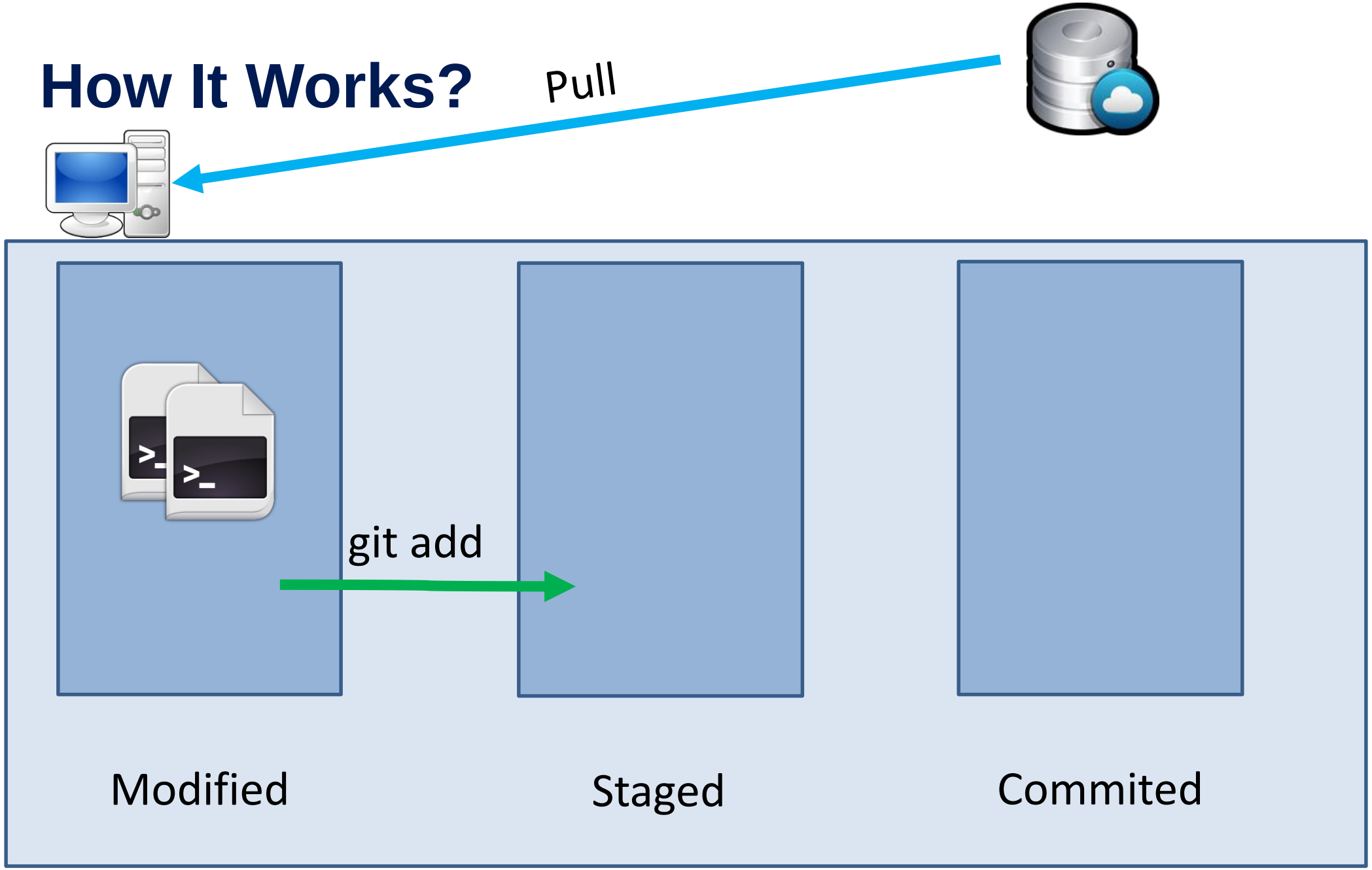
Basic Git Workflow

1. **Modify** files in your working directory.
2. **Stage** files, adding snapshots of them to your staging area.
3. Make a **commit**, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

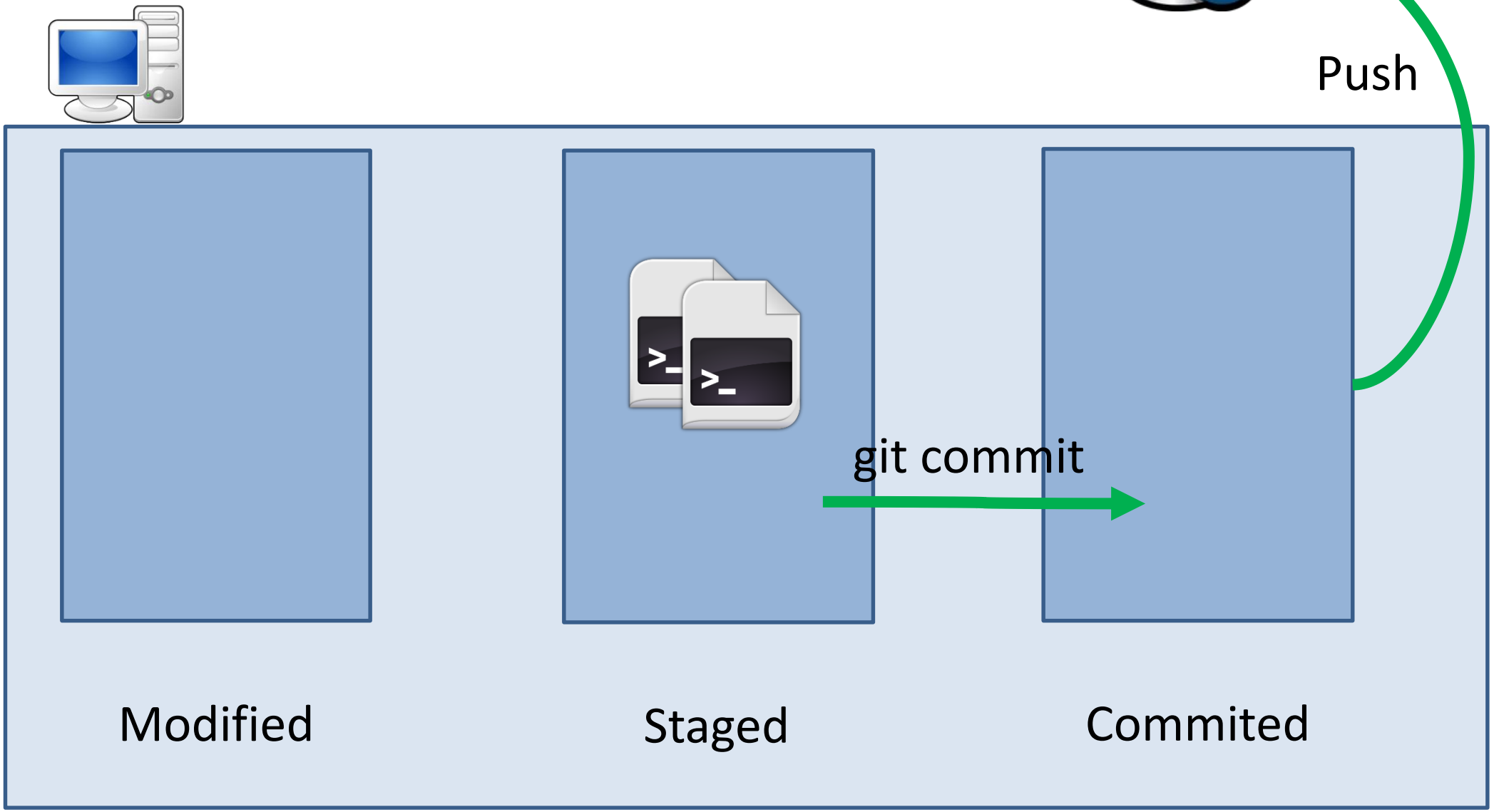
Best practice: commit your work whenever you've gotten one part of your problem working, or before trying something that might fail.

If your new stuff is screwed up, you can always "revert" to your last good commit. (Remember: always "revert", never "roll back")

How It Works?



How It Works?



How does it work?

- All operations happen locally
  - No need for a network as long as you are coding on your own (though, you can’t collaborate without it)
- EVERYTHING is check summed
  - Impossible to change the contents of any file or directory without Git knowing about it
  - We’ll know if you crashed the build!
- Once you have committed, you CAN’T go back or delete
  - Git believes in the true meaning of the word ‘commit’

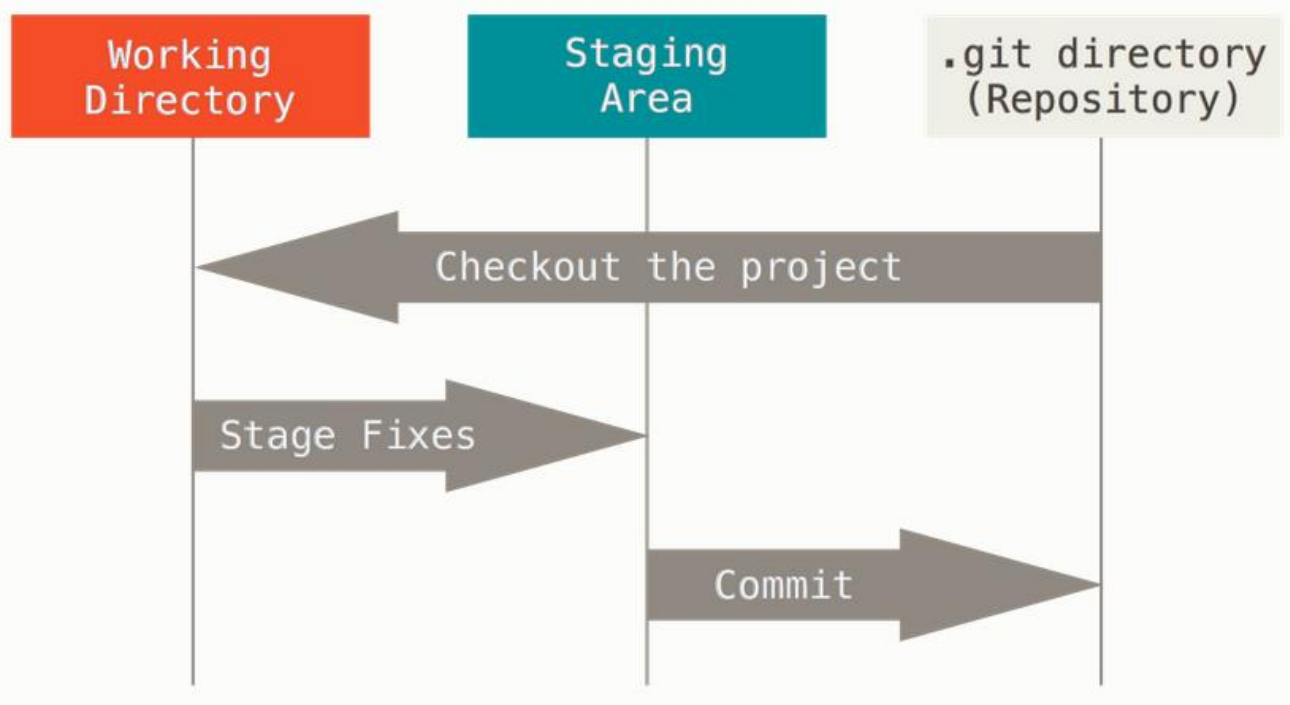
When to Commit?

A. Only when I am done.
B. Every fixed amount of time.
C. Every fixed amount of change.
D. When I think that’s good enough.
E. Before I do something risky.
F. Before I go away on vacation
G. Before I get hit by a bus.

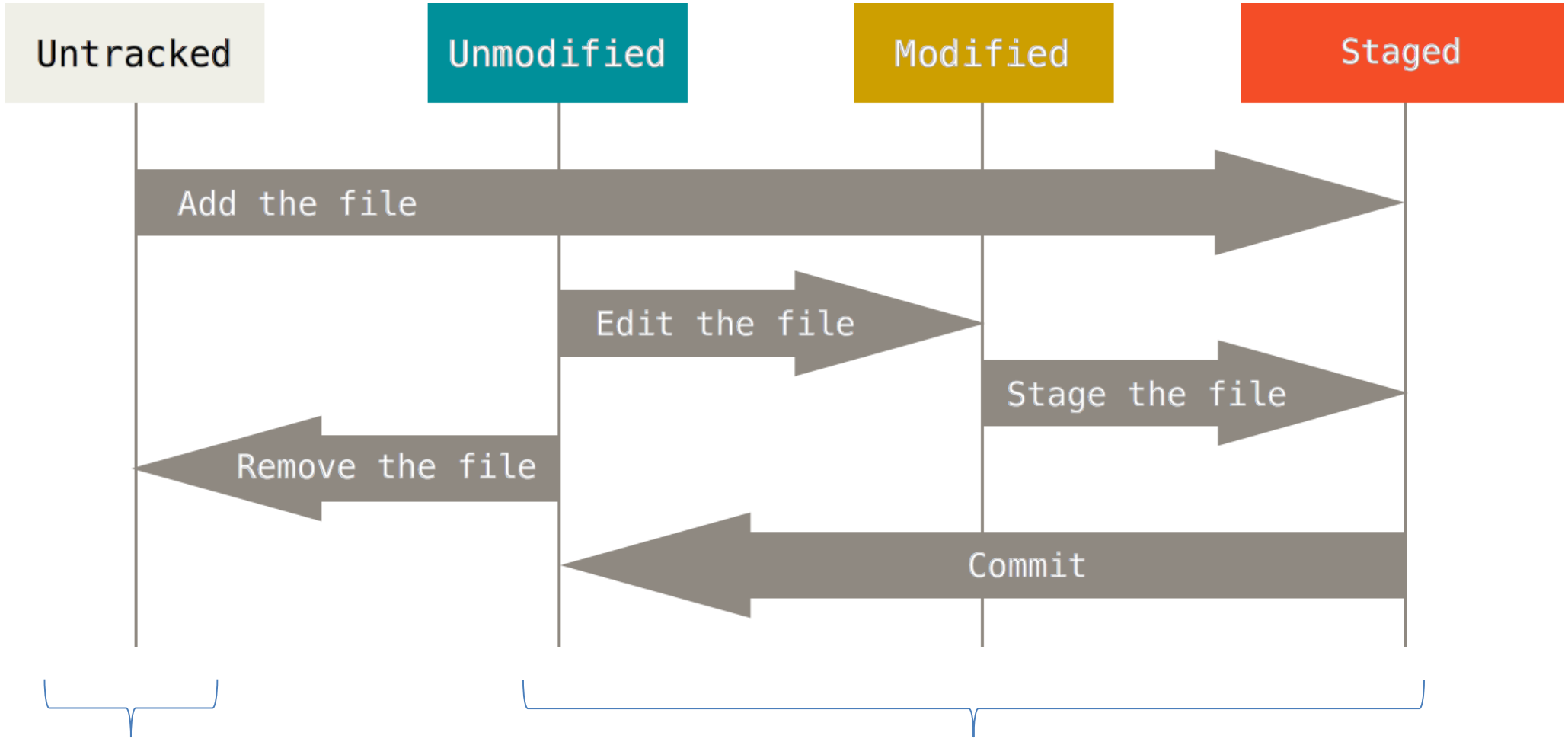
Three main sections of a git project

Files tracked by git are in one of three states: committed, modified, or staged.

This leads us to the three main sections of a Git project: the .git directory, the working directory, and the staging area.



git file stages



SHA-1 checksums in git

git stores data in one big hash table or key-value data store. The keys are SHA-1 checksums. You see these checksums everywhere. For example, if you run git log --oneline you will see something like:

```
$ git log --oneline
940680b add readme file
f118e0e update license text
6beda33 fixed issue 123
```

The hex values above are abbreviated SHA-1 checksum values or keys that point to commits.

- The hash is a SHA1 hash of the contents of the object
  - => All hashes are unique
  - => Identify commits
  - => Are used to check if the files are valid
  - => Identical objects can be identified because they will have identical hashes

Getting started: create a local repo

Two common scenarios: (only do one of these)

a) To **clone an already existing repo** to your current directory:

```
$ git clone <url> [local dir name]
```

This will create a directory named *local dir name*, containing a working copy of the files from the repo, and a `.git` directory (used to hold the staging area and your actual repo)

b) To **create a Git repo** in your current directory:

```
$ git init
```

This will create a `.git` directory in your current directory.

Then you can commit files in that directory into the repo:

```
$ git add file1.java
$ git commit –m “initial project version”
```

Local Commits

1. ‘git status’ to check which files are modified
   -’git diff <filename>’ shows line-by-line changes
2. ‘git add <filename>’ stages all desired files
3. ‘git commit’ creates new snapshot of staged files and adds to the history
4. ‘git log’ pulls up history of branch; should see your latest commit

replace local changes

- In case you did something wrong (which for sure never happens ;) you can replace local changes using the command
  git checkout -- <filename>
  this replaces the changes in your working tree with the last content in HEAD. Changes already added to the index, as well as new files, will be kept.
- If you instead want to drop all your local changes and commits, fetch the latest history from the server and point your local master branch at it like this
  git fetch origin
  git reset --hard origin/master

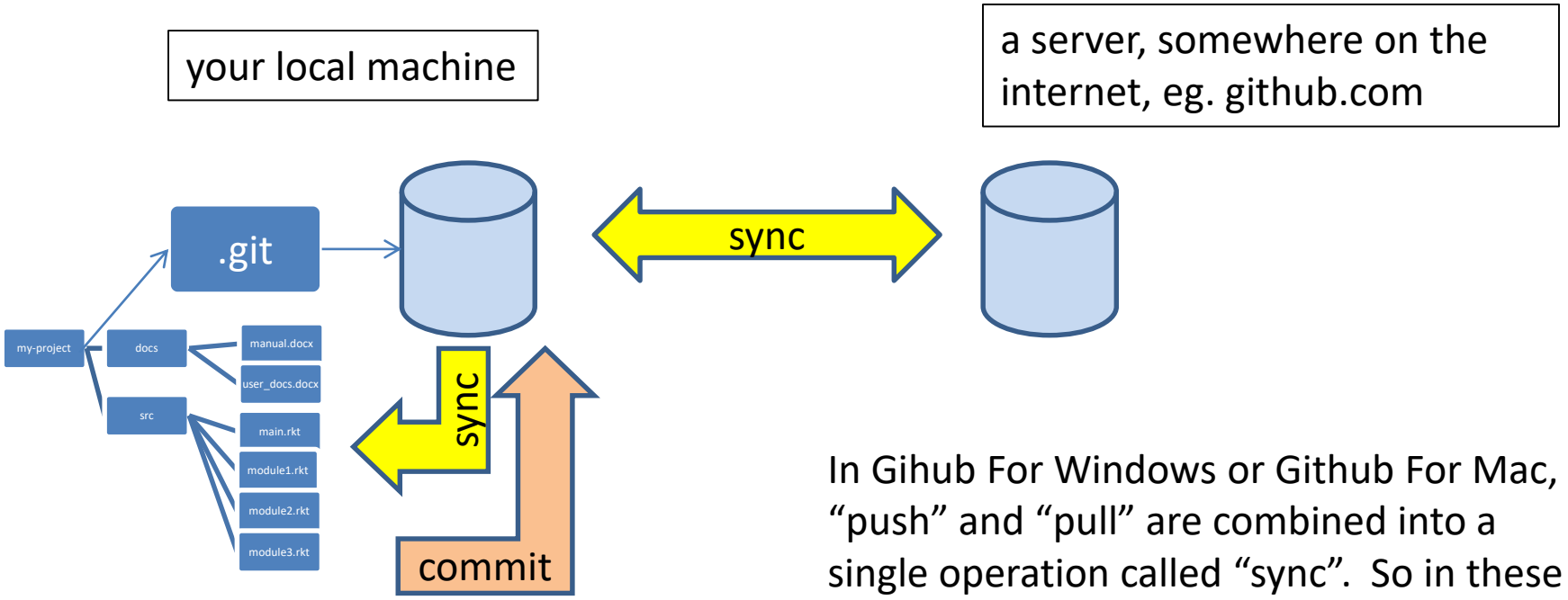
Writing Commit Messages

- **First line** is <80 character **summary**
- Followed by **detailed description**
  - List of all additions/changes
  - Motivation
  - Implementation details
- Examples of bad git messages:
  - “Typo”
  - “Add database entries”
- After saving message a unique commit string is created for each entry

Remotes

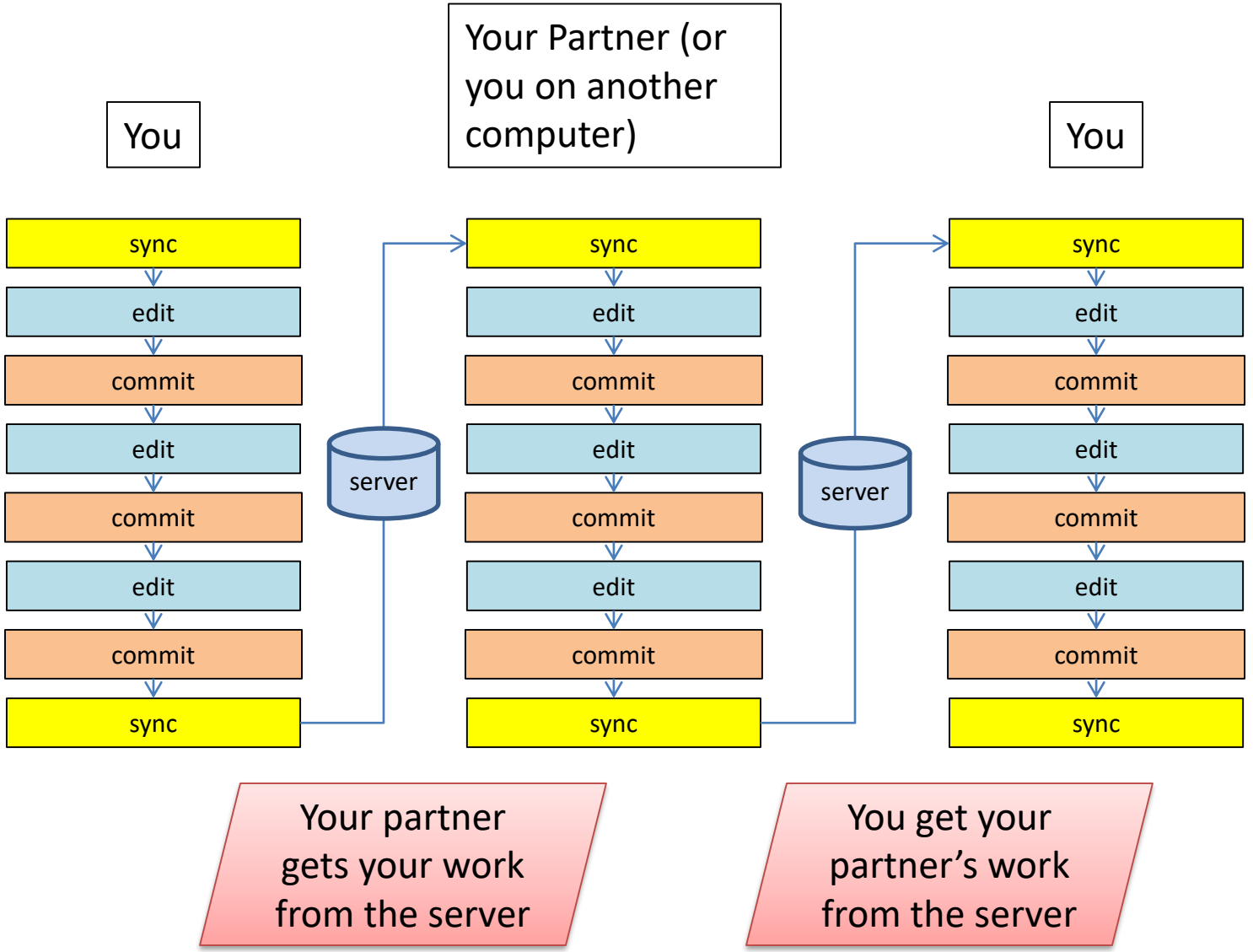
- Remotes are what make Git **Distributed** and **Confusing**
- Remotes are just a **URL** that's been **aliased**
- Commands
  - remote
  - push
  - fetch (or pull)
  - merge
- Git protocols
  - Local – read/write a file system (local, share, friends computer)
  - Git – generally read-only
  - SSH – Git protocol over SSH
  - HTTP/S

The whole picture using GHFW

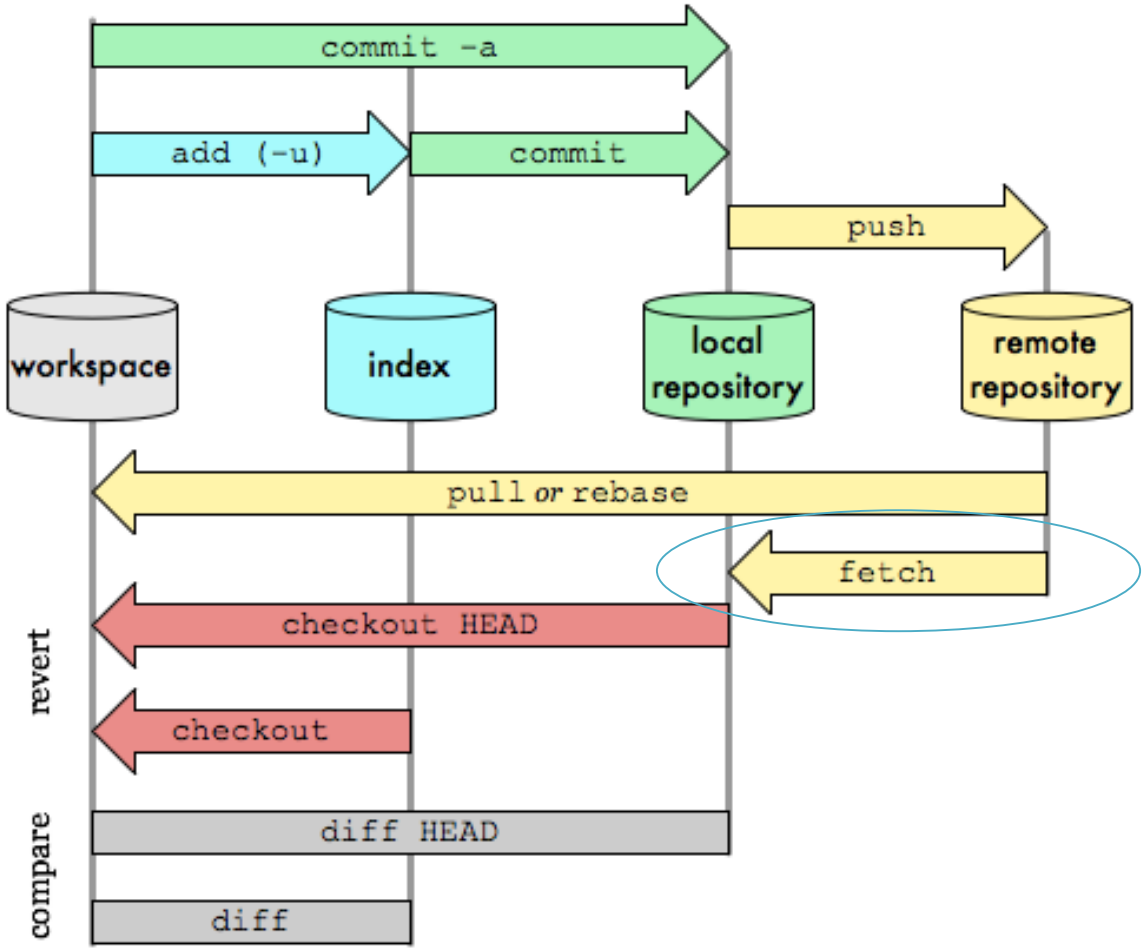


In Gihub For Windows or Github For Mac, “push” and “pull” are combined into a single operation called “sync”. So in these clients, there are only two steps (“commit” and “sync”) to worry about, not three.

Your workflow with a partner



Git Command guide



Branching & Merging

Without Branching - Scenario

- All team member push to master
- Susan is updating the backend
  - Will take weeks
  - Stable when complete
  - Intermediate changes have unpredictable consequences elsewhere
- John is making changes to the frontend
  - Sees side effects caused by Susan’s pushes
  - Thinks his code caused the changes

Without Branching - Scenario

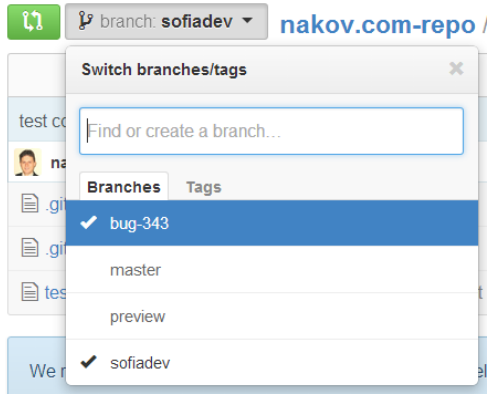
- Tempting solution
  - Don’t push to the remote repo until all changes are complete
  - Changes only exist locally during development
    - One hard drive error away from losing weeks of work!
- If there are 10+ developers with different tasks on the same codebase?
  - Complex interactions between modules can multiply issues

Without Branching – Worse Scenario

- Someone pushes code with errors
  - Code breaks
  - Who broke it?
  - Everyone’s workflow halts and the search begins
- It happens
  - All code functions locally
  - Combined pushes doesn’t contains error
  - ex: A function header is changed

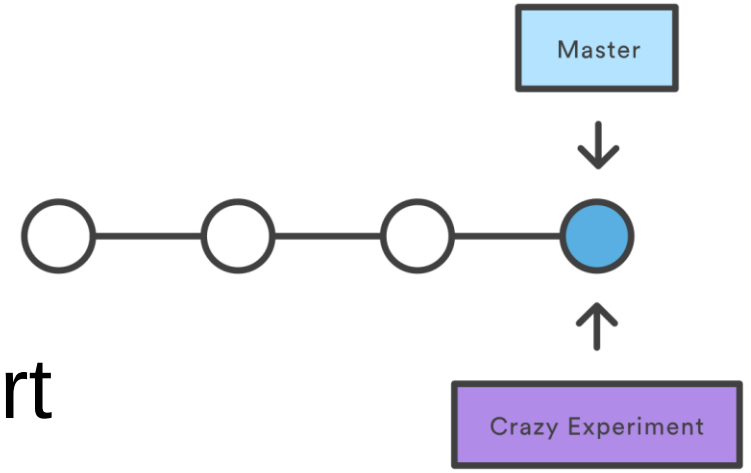
Branching

- Branching allows splitting the development line into separate branches
  - Different developers work in different branches
- Branching is suitable for:
  - Development of new feature or fix in a new version of the product (for example version 2.0)
    - Features are invisible in the main development line
    - Until merged with it
  - You can still make changes in the older version (for example version 1.0.1)



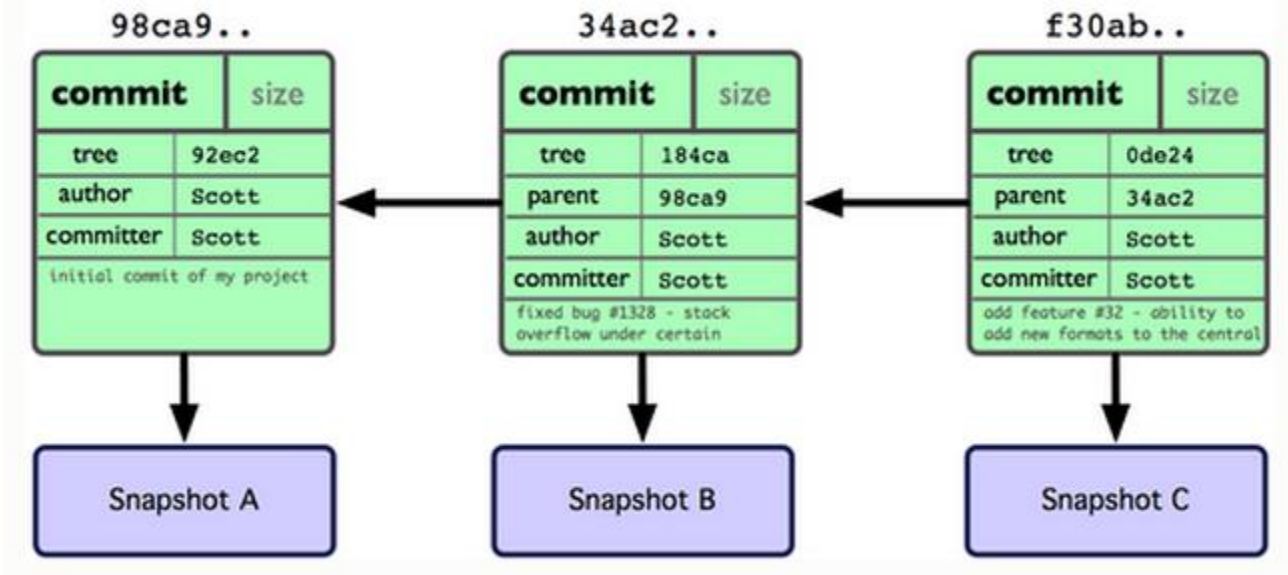
Branches

- independent line of development
- exact copy of remote repo
- Great for dividing work/experimenting
- Redundancy reduces likelihood of revert
- Can merge branches back
- Can be leveraged for modularized dev



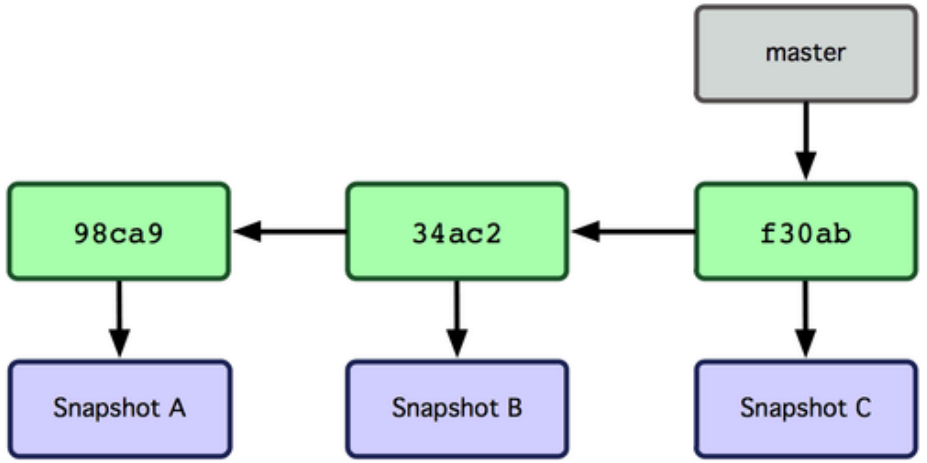
Branching

- When you commit in Git, Git stores a commit object that contains a pointer to the snapshot of the content you staged, the author and message metadata, and zero or more pointers to the commit or commits that were the direct parents of this commit: zero parents for the first commit, one parent for a normal commit, and multiple parents for a commit that results from a merge of two or more branches.



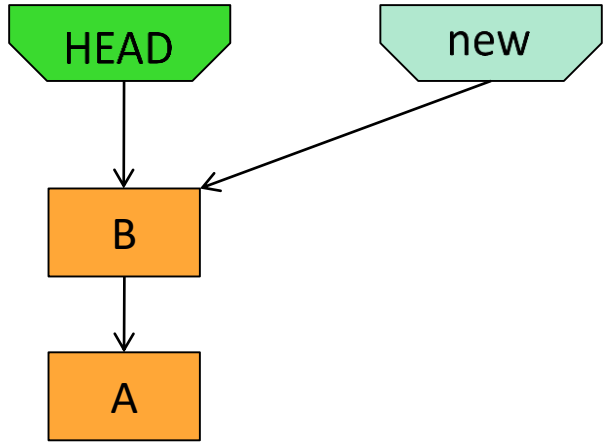
Branching

- A branch in Git is simply a lightweight movable pointer to one of the commits. The default branch name in Git is ‘master’. As you initially make commits, you’re given a master branch that points to the last commit you made. Every time you commit, it moves forward automatically.



Creating branches

git branch <name> <commit>




```
git branch new HEAD
```

Local branches

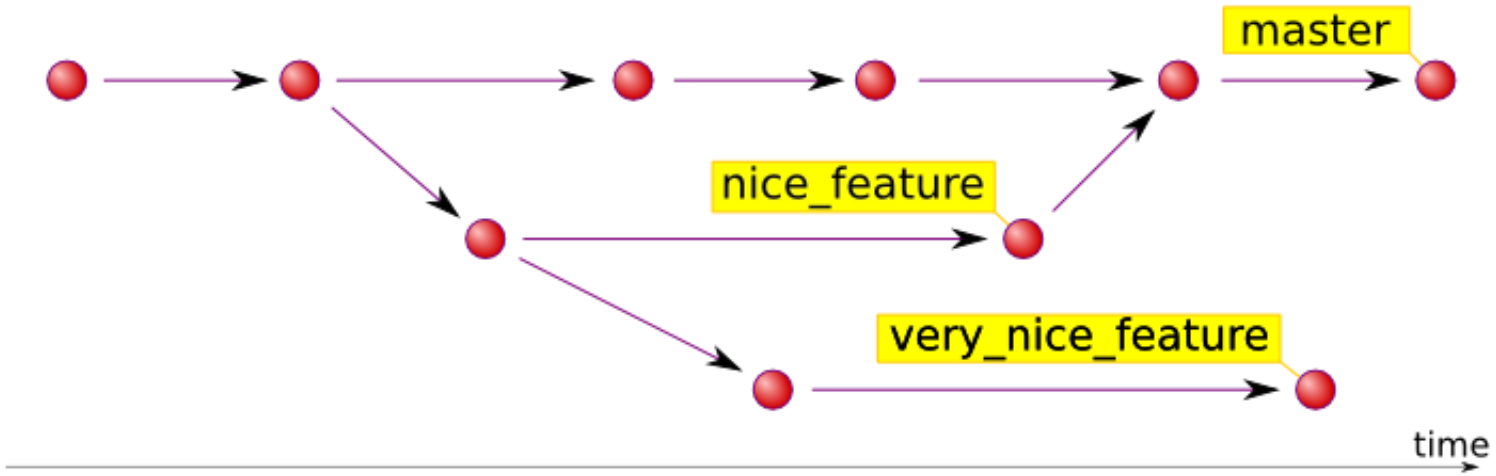
To list them: `git branch -l`

```
    branch1
    branch2
  * master
```

Or look at files in `.git/refs/heads/`

Git Branches

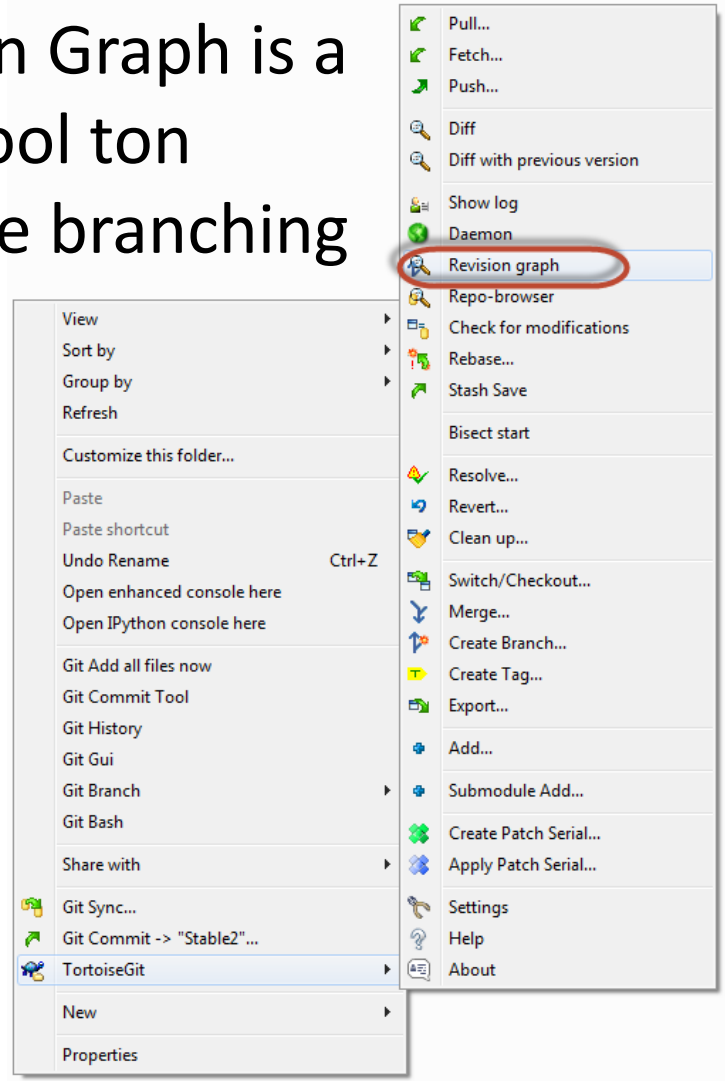
- Branches start a new history to make experimental features

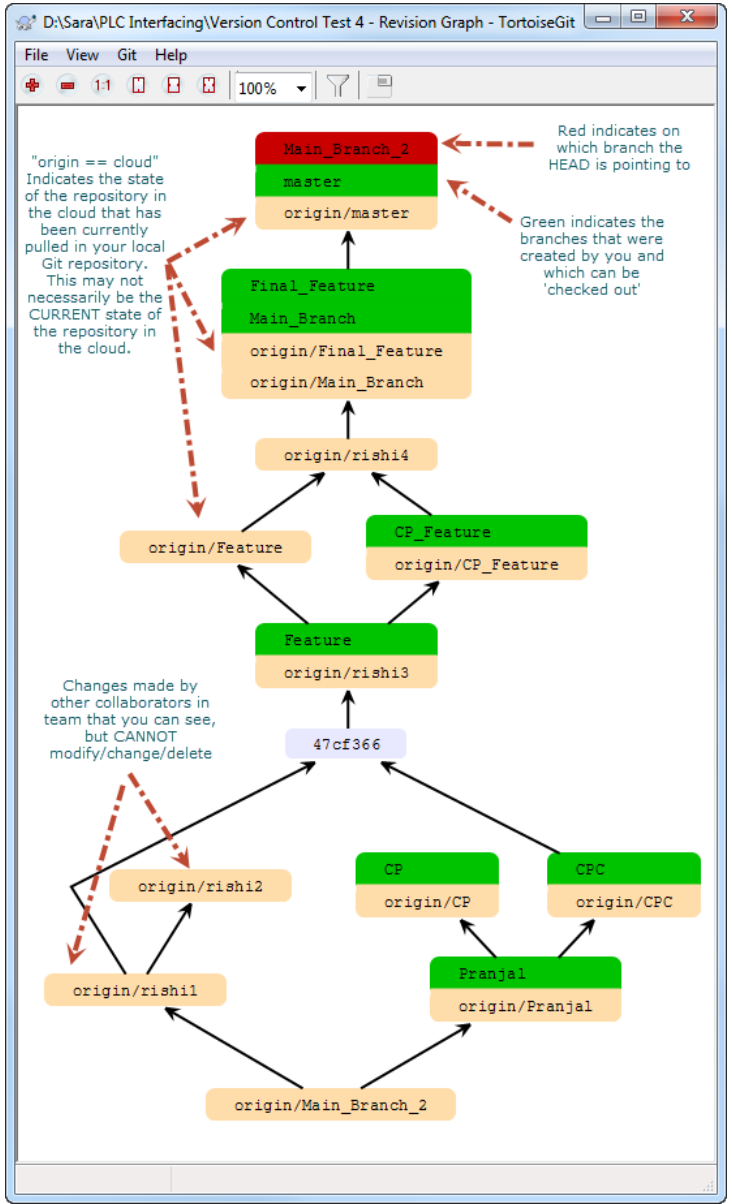


- Allows experimentation without fear of “messing up the code”

Branching

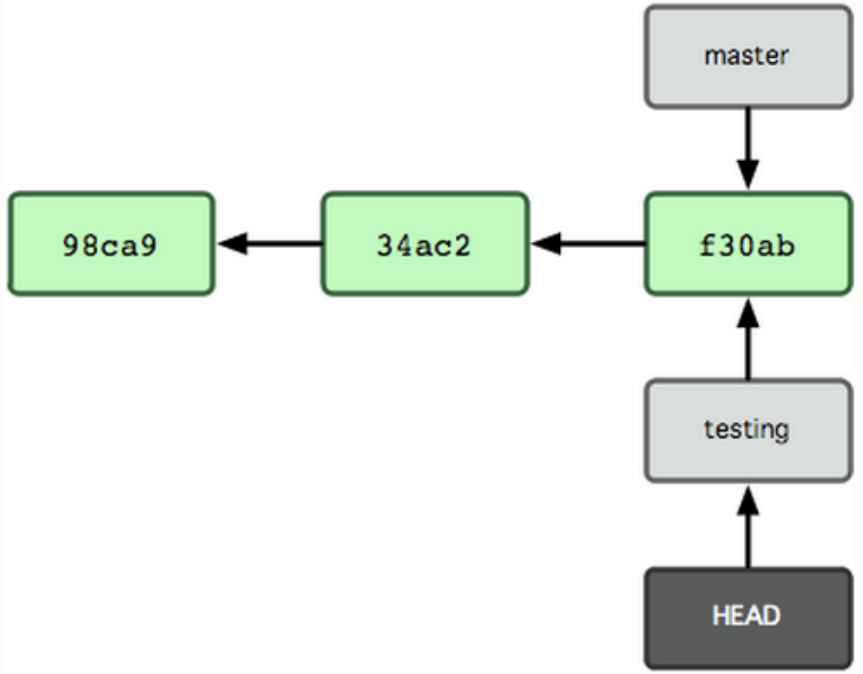
- Revision Graph is a great tool ton visualize branching in Git

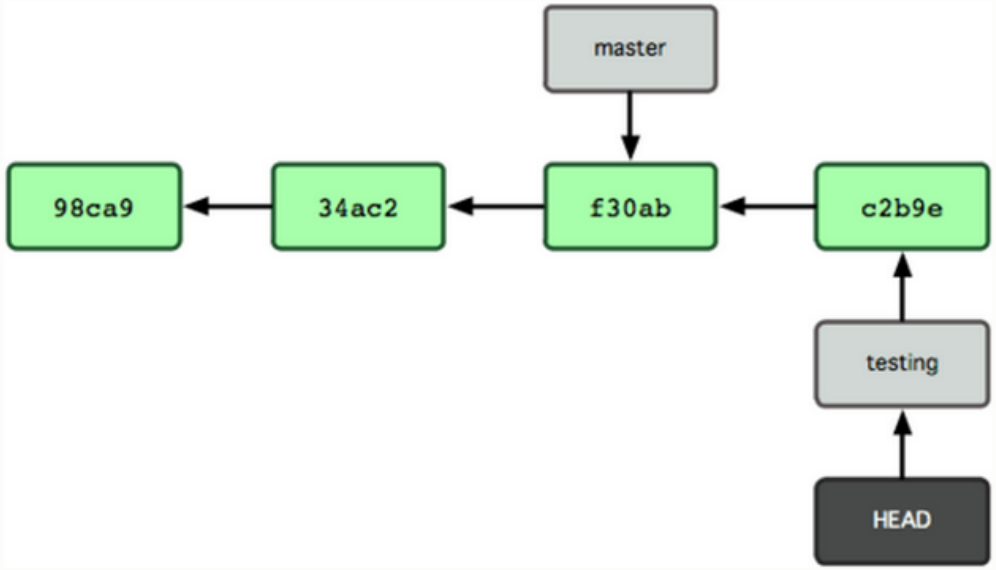




Branching

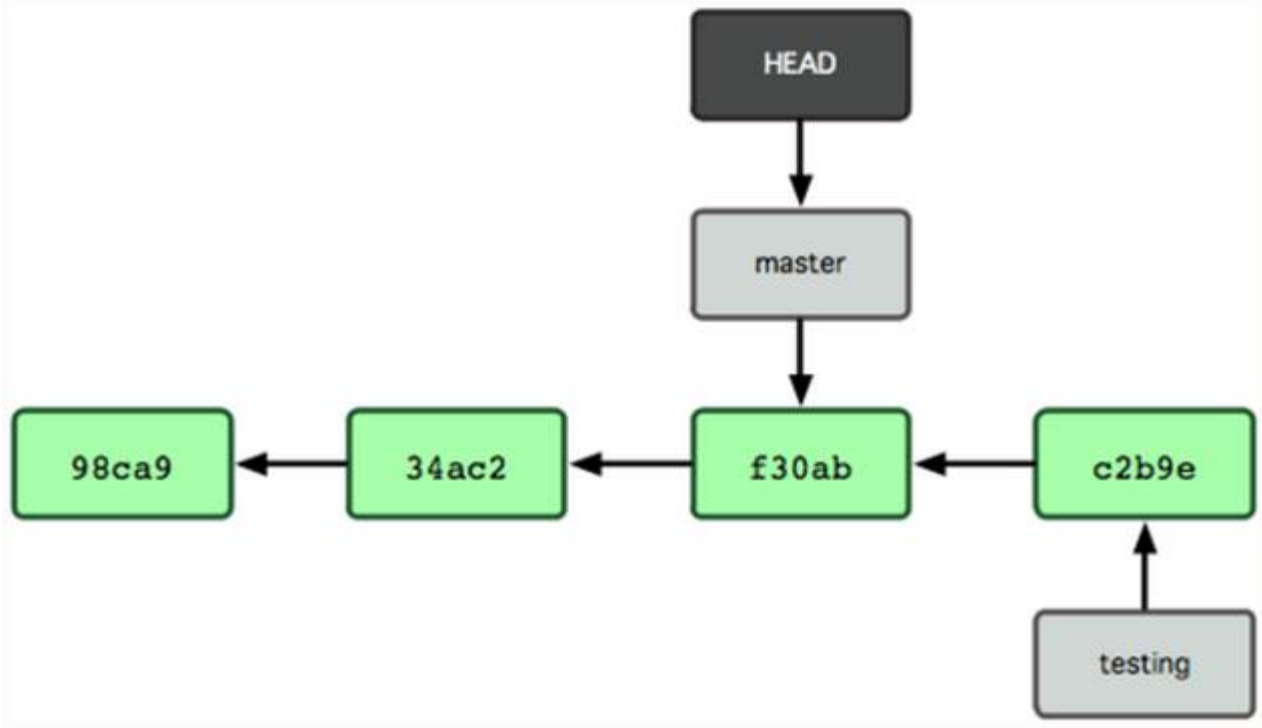
- Switching the HEAD to the new branch and making a commit





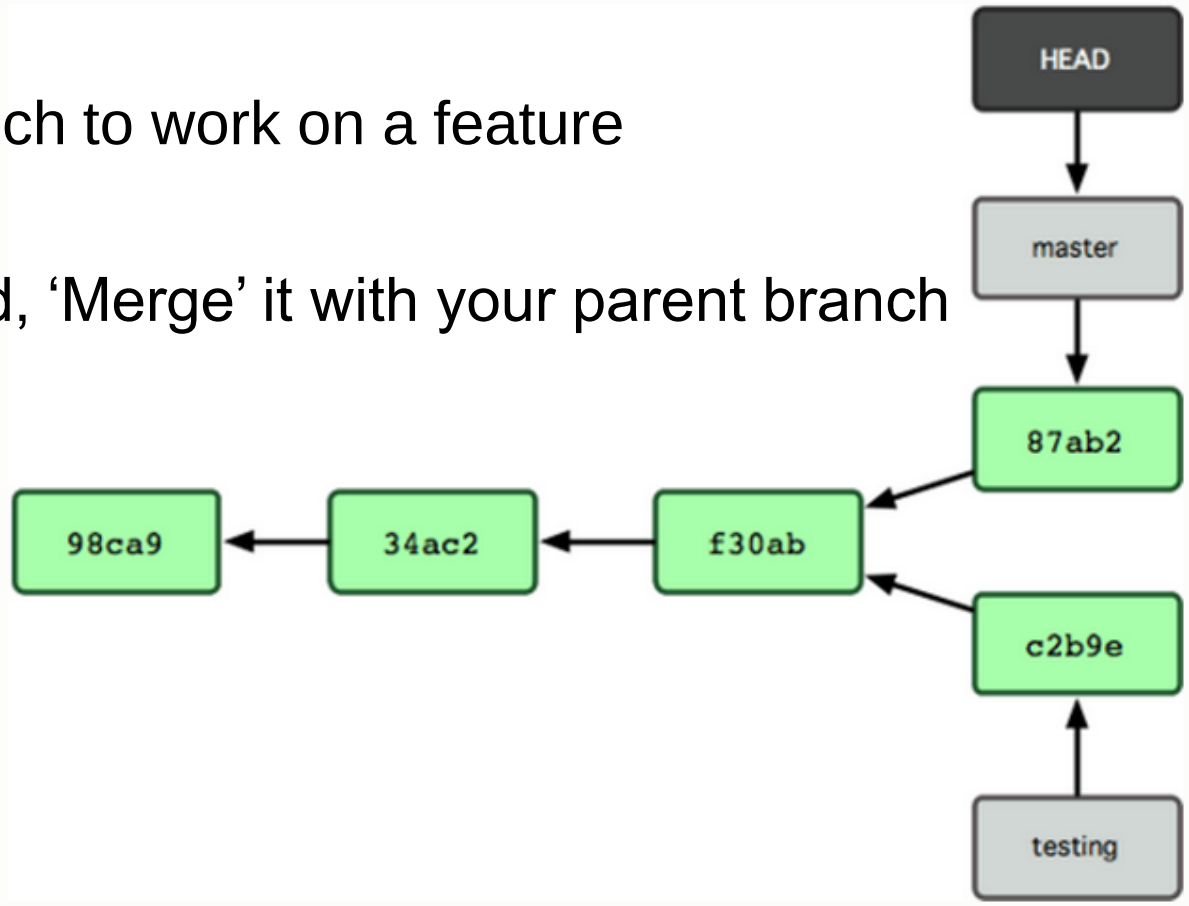
Branching

- Now if you ‘checkout’ the ‘master’ branch, Git moves the HEAD back to the master + rewinds the files in the working directory to the snapshot that ‘master’ points to



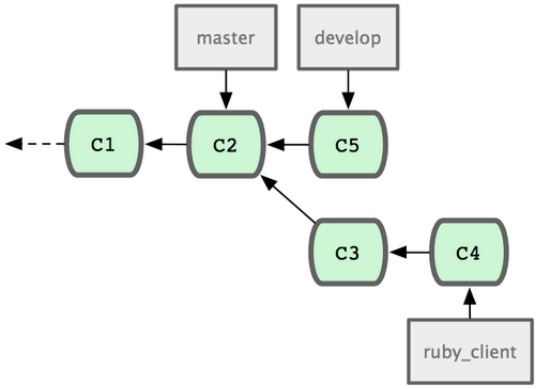
Branching

- If you make a few more commits in the ‘master’ branch, the project history will diverge
- Typically,
  - Branch off from parent branch to work on a feature
  - Make a few commits
  - Once the feature is finalized, ‘Merge’ it with your parent branch



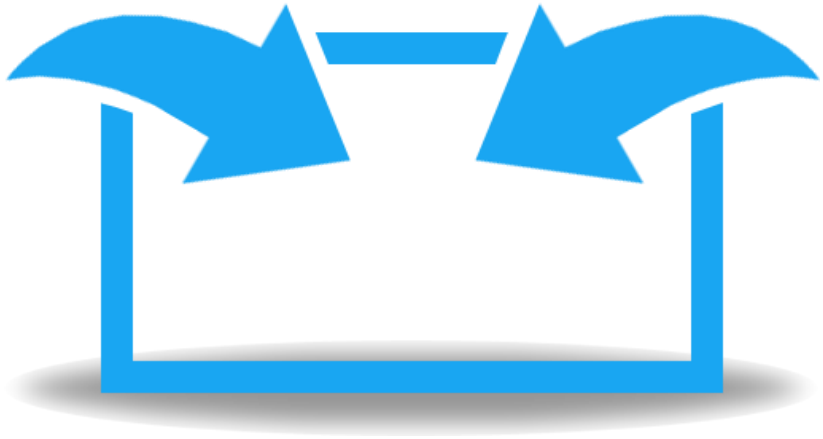
Merging Branches

- Some companies work in separate branches
  - For each new feature / fix / task
- Once a feature / fix / task is completed
  - It is tested locally and committed in its branch
- Finally it is merged into the main development line
  - Merging is done locally
  - Conflicts are resolved locally
  - If the merge is tested and works well, it is integrated back in the main development line



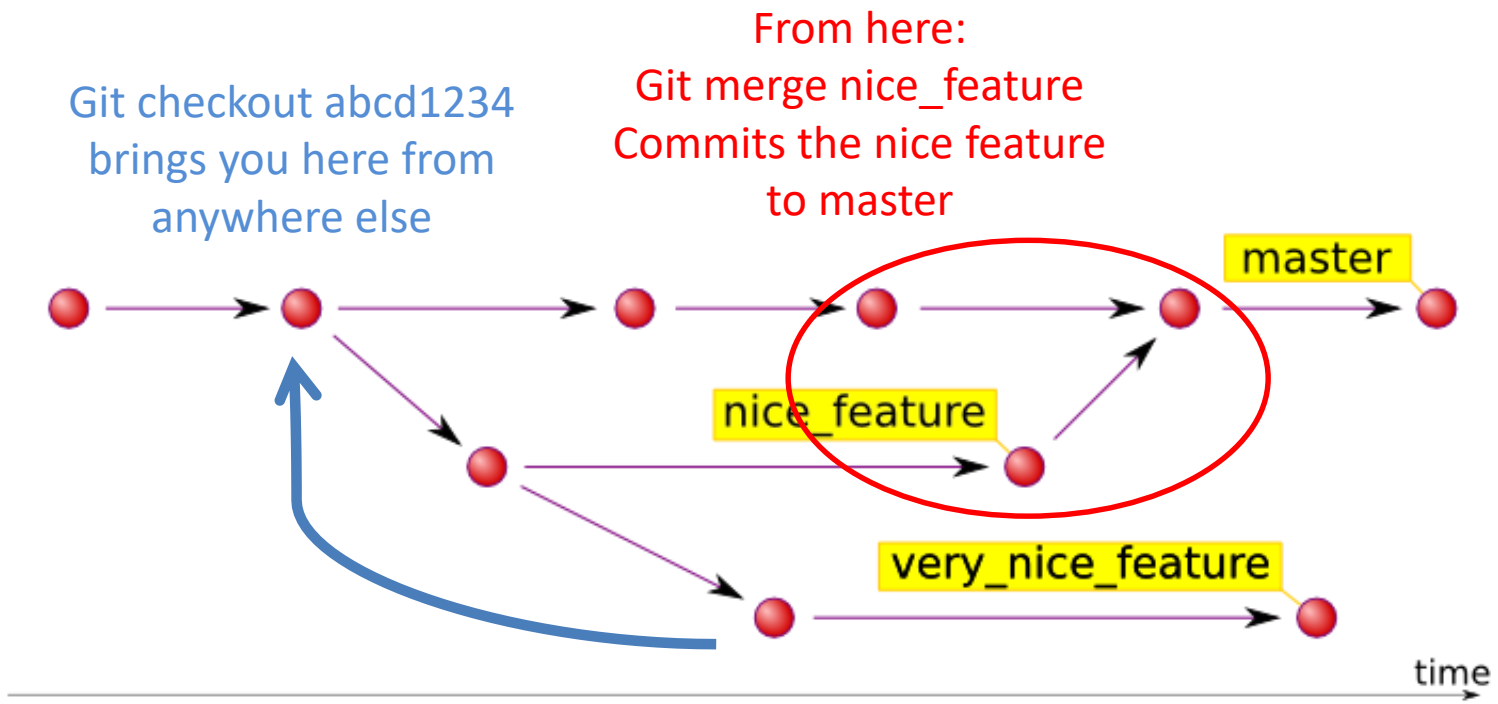
Merging Problems

- When a file is concurrently modified, changes should be merged
  - Merging is hard!
  - It is not always automatic process
- Coordination and responsibility between the developers is required
  - Commit changes as early as finished
  - Do not commit code that does not compile or blocks the work of the others
  - Leave meaningful comments at each commit



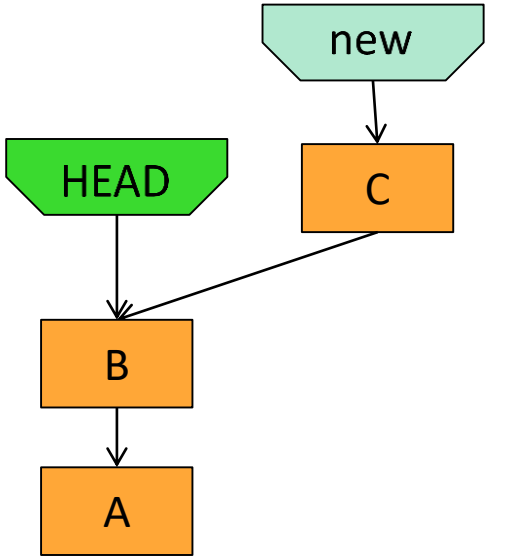
Commands used with Branches

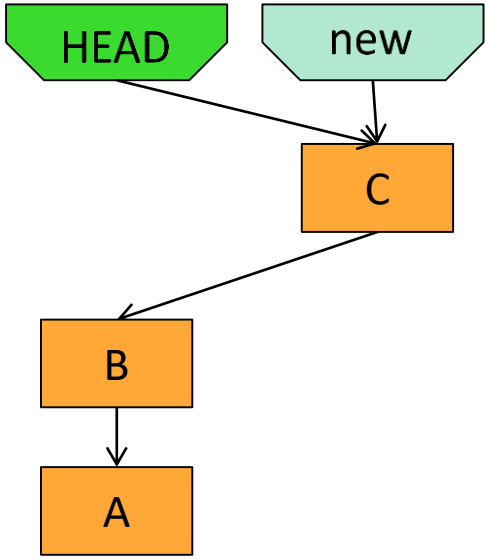
- Git checkout <location>: moves the head to location (can be a commit string or branch name)
- Git merge <branch>: merges all commits from branch



Fast-forward merge

- Current head of the branch to which you are merging is an ancestor of the branch you are merging to it.
- The branch head is just moved to the newer commit.
- No merge commit object is created
  unless “-- no-ff” is specified

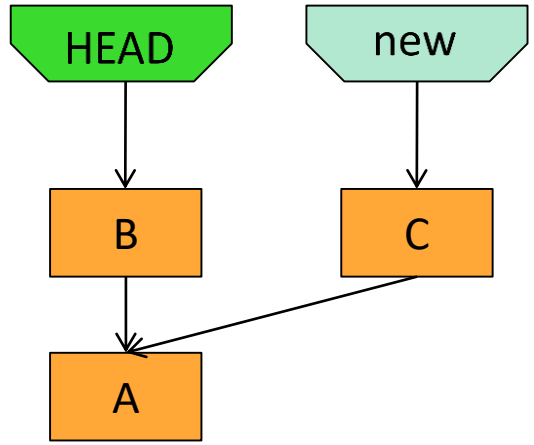


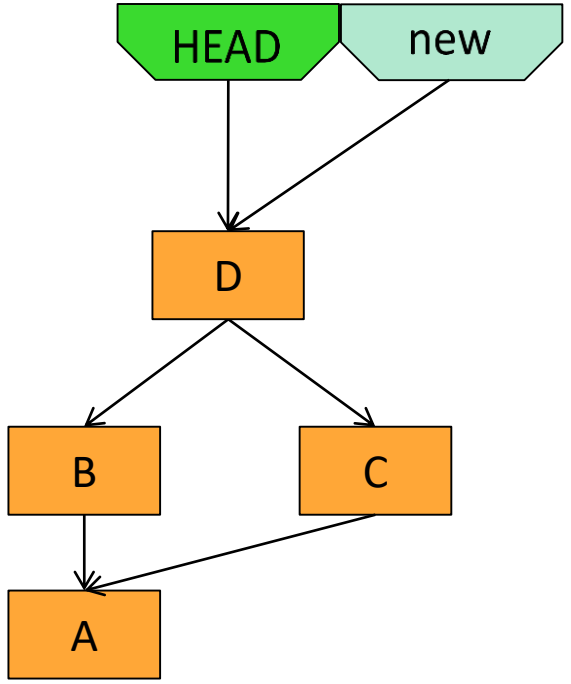



```
git merge new HEAD
```

True merge

- Not fast-forward
- New commit object must be created for new head
- 2 cases:
  - No overlapping changes are detected
    - Merge proceeds normally
  - Overlapping changes are detected
    Manual intervention is required





Beware of false security!

- Just because there are no overlapping changes **does not mean** the changes are semantically compatible.
  - It only means they do not modify the same region of the same file.
- So, unless the merge is a fast-forward, there is a good chance that your merge will break the software.
- This is the reason for following a discipline that forces all merges to be fast-forward.

Merge with conflicts

- HEAD pointer is unchanged
- MERGE_HEAD points to the other branch head
- Files that merged cleanly are updated in the index file and working tree
- 3 versions recorded for conflicting files:
  - Stage 1: common ancestor version
  - Stage 2: MERGE_HEAD version
  - Working tree: marked-up files (with <<< === >>>)
- No other changes are made
- You can start over with `git reset --merge`

How merge marks conflicts

```
Here are lines that are either unchanged from the common ancestor, or cleanly resolved
because only one side changed.
<<<<<<< yours:sample.txt
Conflict resolution is hard; let's go shopping.
=======
Git makes conflict resolution easy.
>>>>>>> theirs:sample.txt
And here is another line that is cleanly resolved or unmodified.
```

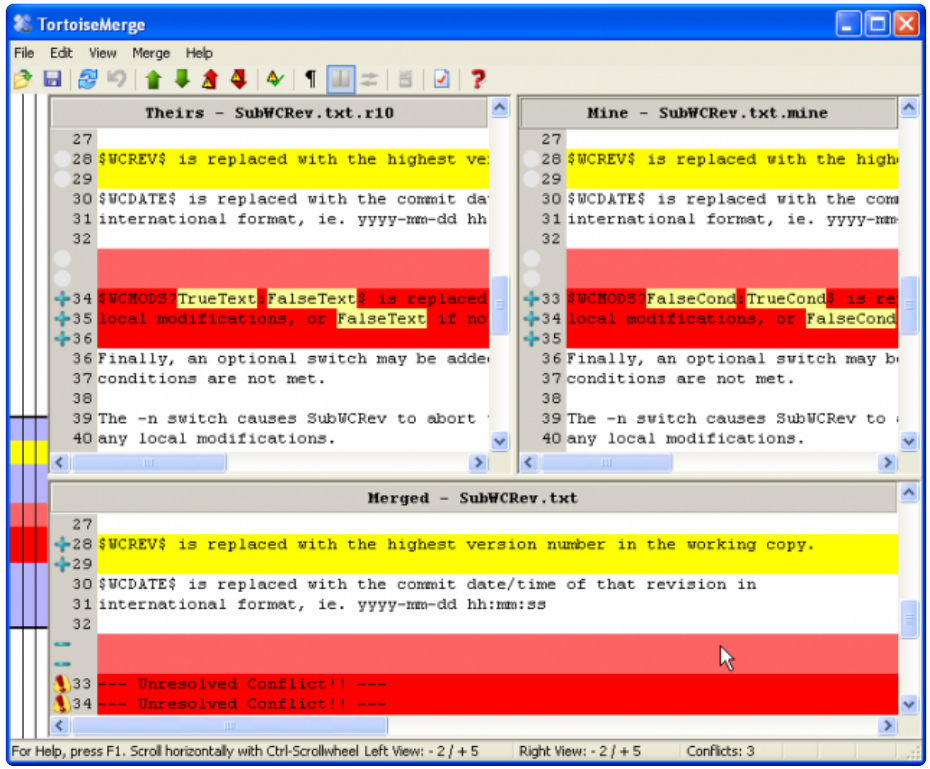
There is an alternate, 3-way, output option that also shows the common ancestor text.

Resolving merge conflicts

- Only two choices
  a. Decide not to merge: git-reset --hard
  b. Resolve the conflicts
- Resolution tools
  – Use a mergetool: git mergetool
  – Look at the diffs, and edit: git diff
  – …

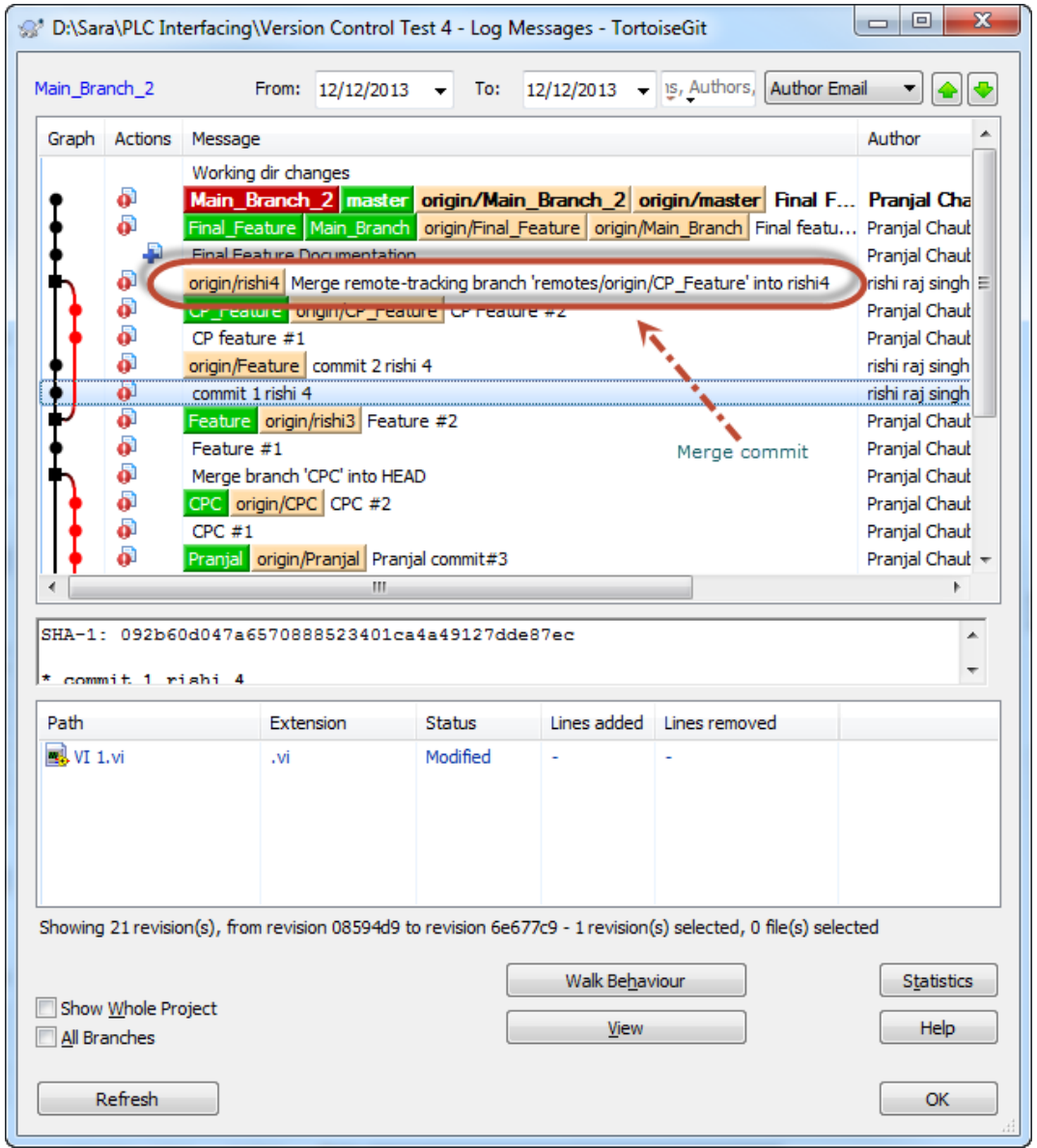
File Comparison / Merge Tools

- During manual merge use file comparison
- There are visual comparison / merge tools:
  - TortoiseMerge
  - WinDiff
  - AraxisMerge
  - WinMerge
  - BeyondCompare
  - CompareIt
  - Kdiff3
  - Tkdiff
  - Meld
  - Xxdiff
  - Emerge
  - Vimdiff
  - Gvimdiff
  - Ecmerge
  - Diffuse
  - Opendiff
  - P4merge
  - Araxis
  - Etc…



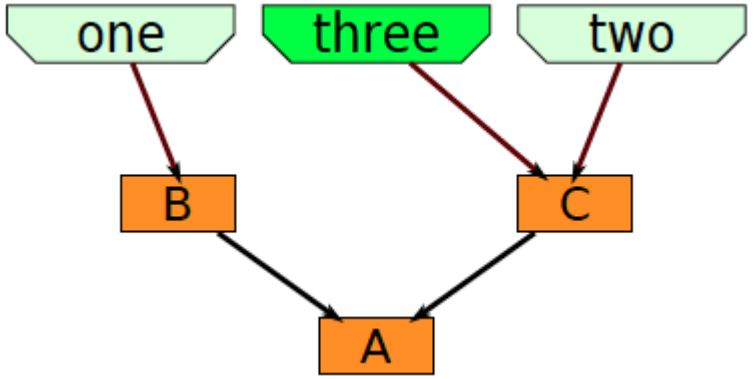
Branching

- Merging
  - Merge Commit

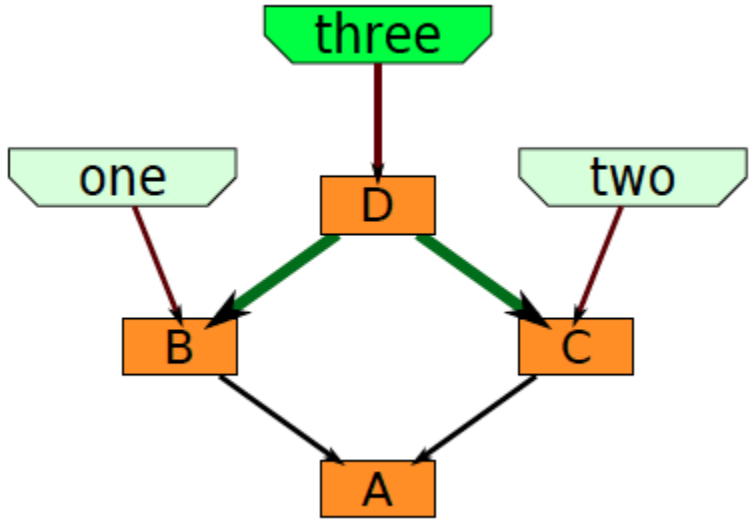


2-way merge

```
git checkout -b three two
```

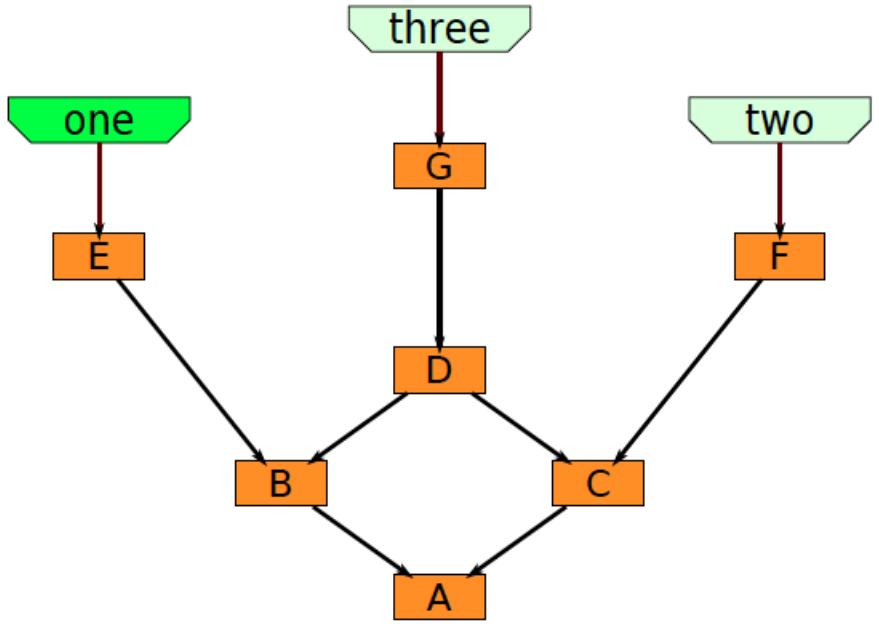


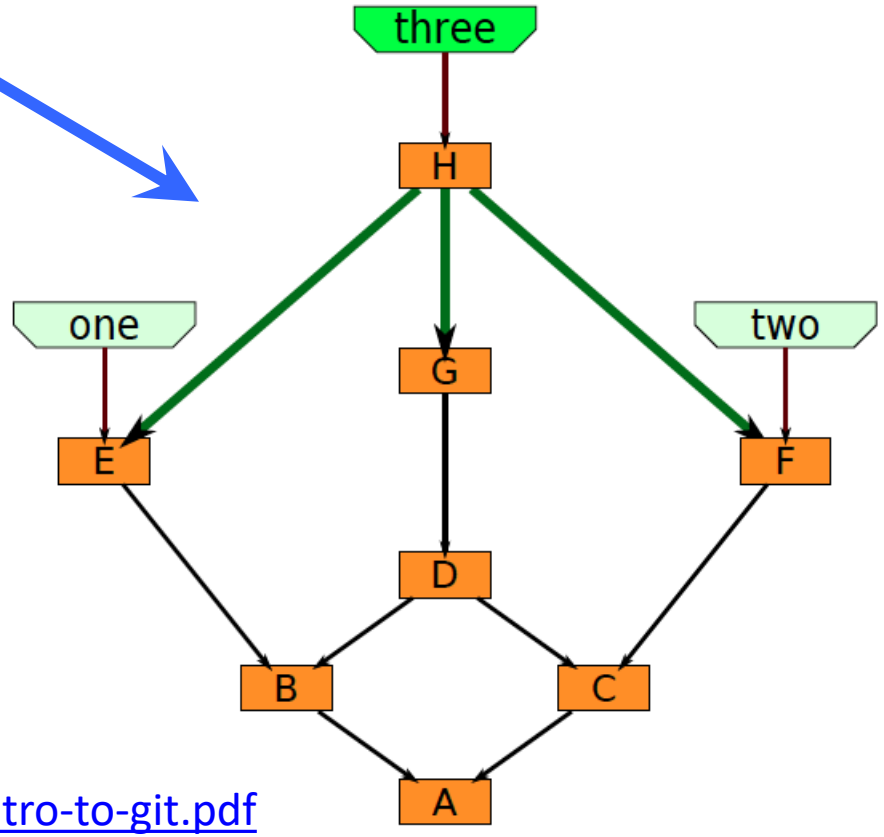

```
git merge one
```



http://edgyu.excess.org/git-tutorial/2008-07-09/intro-to-git.pdf

3-way merge

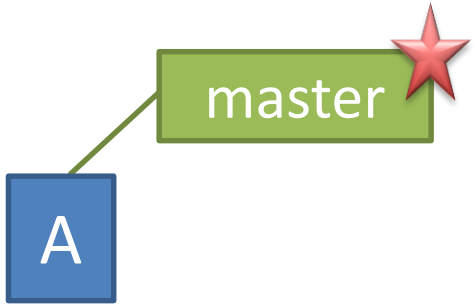





```
git checkout three
git merge one two
```

http://edgyu.excess.org/git-tutorial/2008-07-09/intro-to-git.pdf

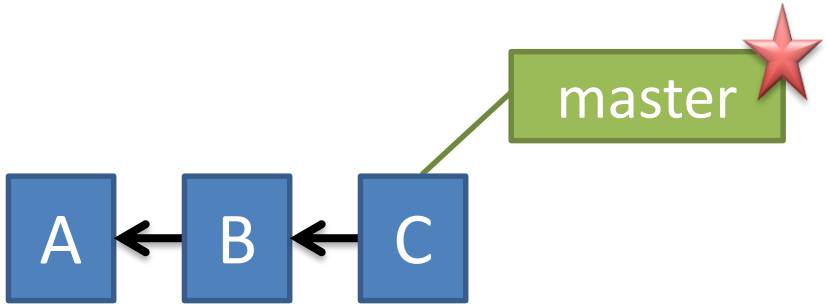
Branches – the master is your first branch



NOTE: we are showing you the command line interface for git.. Note you can use GUI GIT tools for local repositories like SourceTree or others.

```
> git commit –m ‘my first commit’
```

Adding commits to master

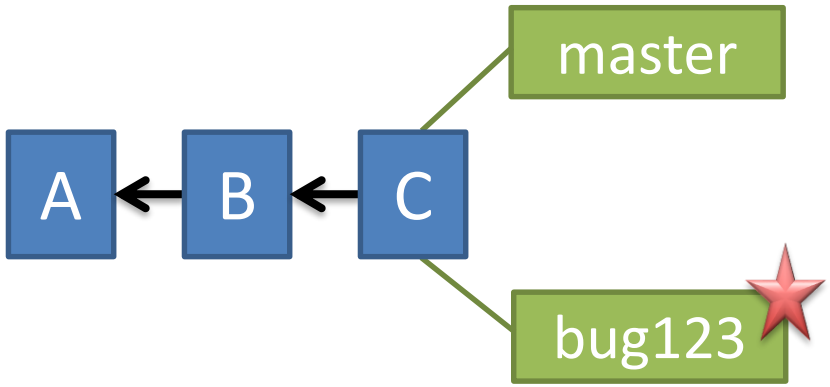


NOTE: we are showing you the command line interface for git.. Note you can use GUI GIT tools for local repositories like SourceTree or others.

```
> git commit *****
```

Creating a New Branch and having master point to it

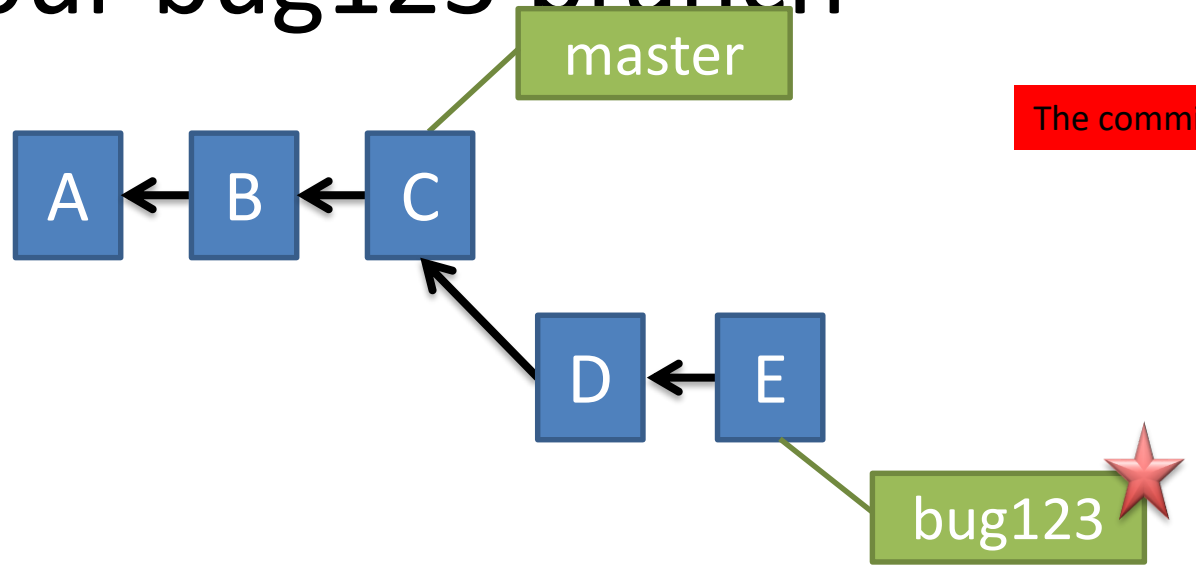
NOTE: we are showing you the command line interface for git.. Note you can use GUI GIT tools for local repositories like SourceTree or others.



At end of checkout master (start) points to bug123 branch

```
> git branch bug123
> git checkout bug123
OR short cut
> git checkout -b bug123
```

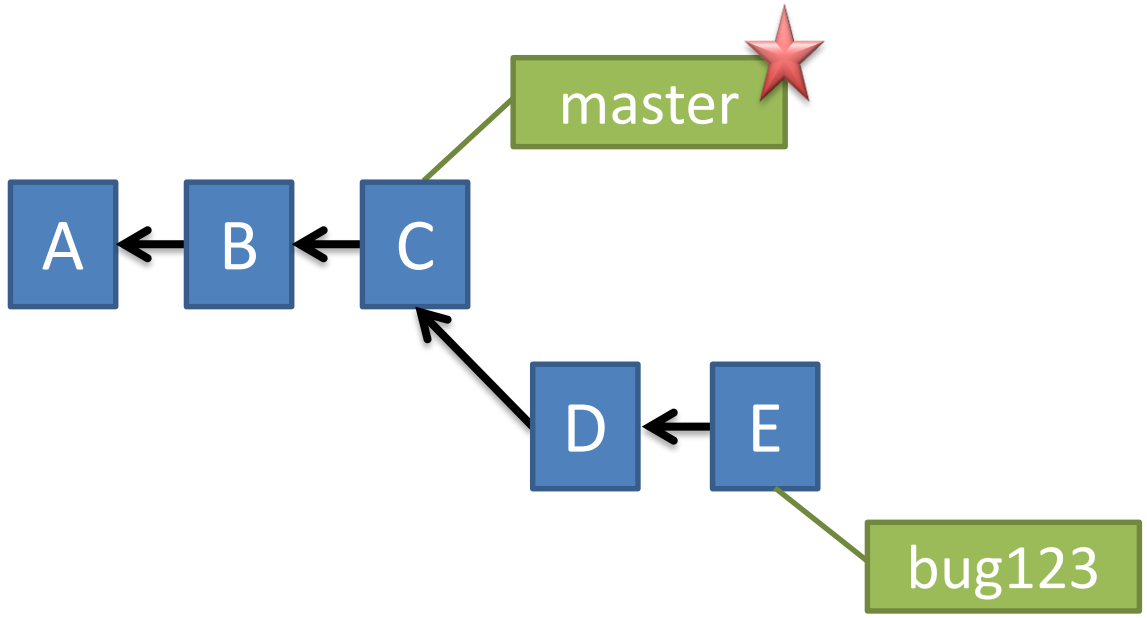
Continuing commits –will be where the head is pointing to –which is our bug123 branch



The commits are going to bug123

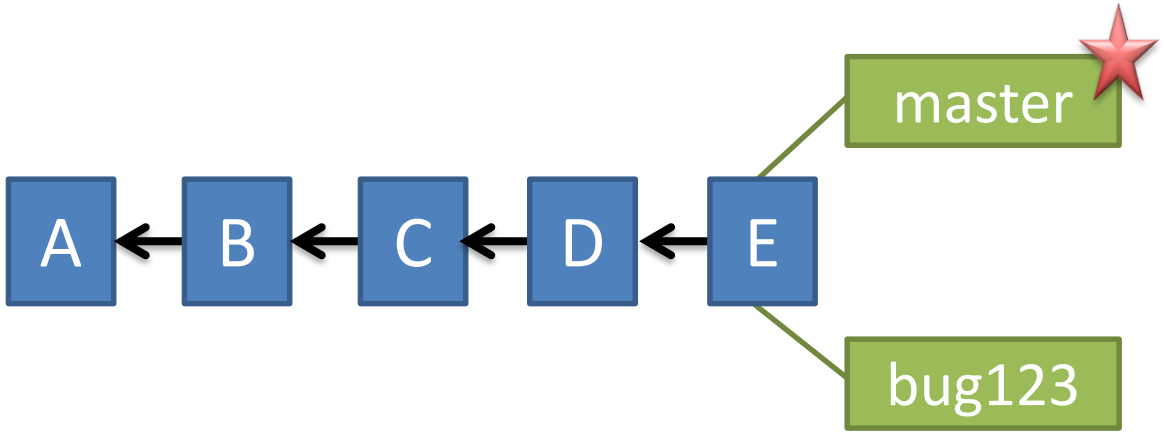
```
> git commit ******
```

Lets point back to master



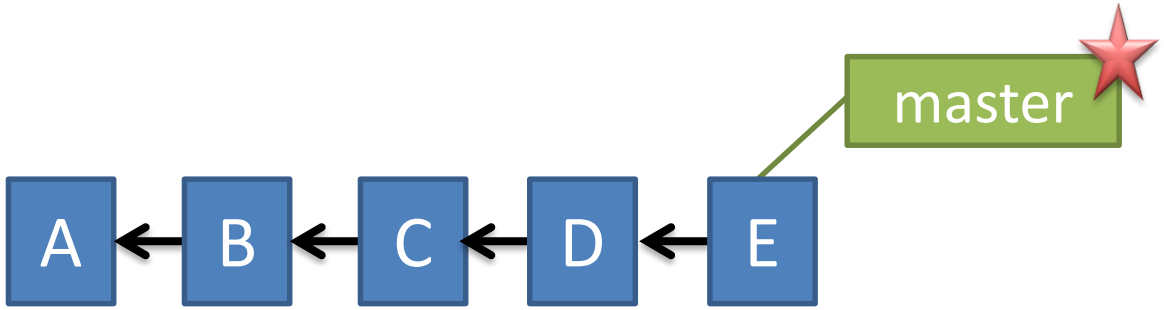

```
> git checkout master
```

Now we want to merge bug123 into our head (current pointing to master)




```
> git merge bug123
```

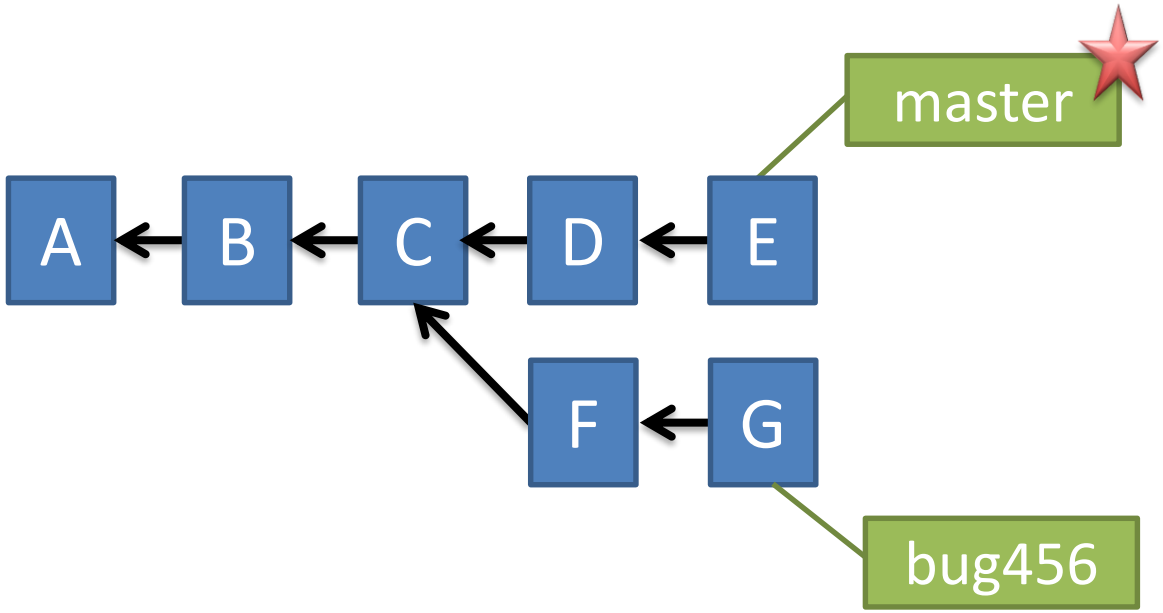
The results of merging



Now lets delete our branch bug123

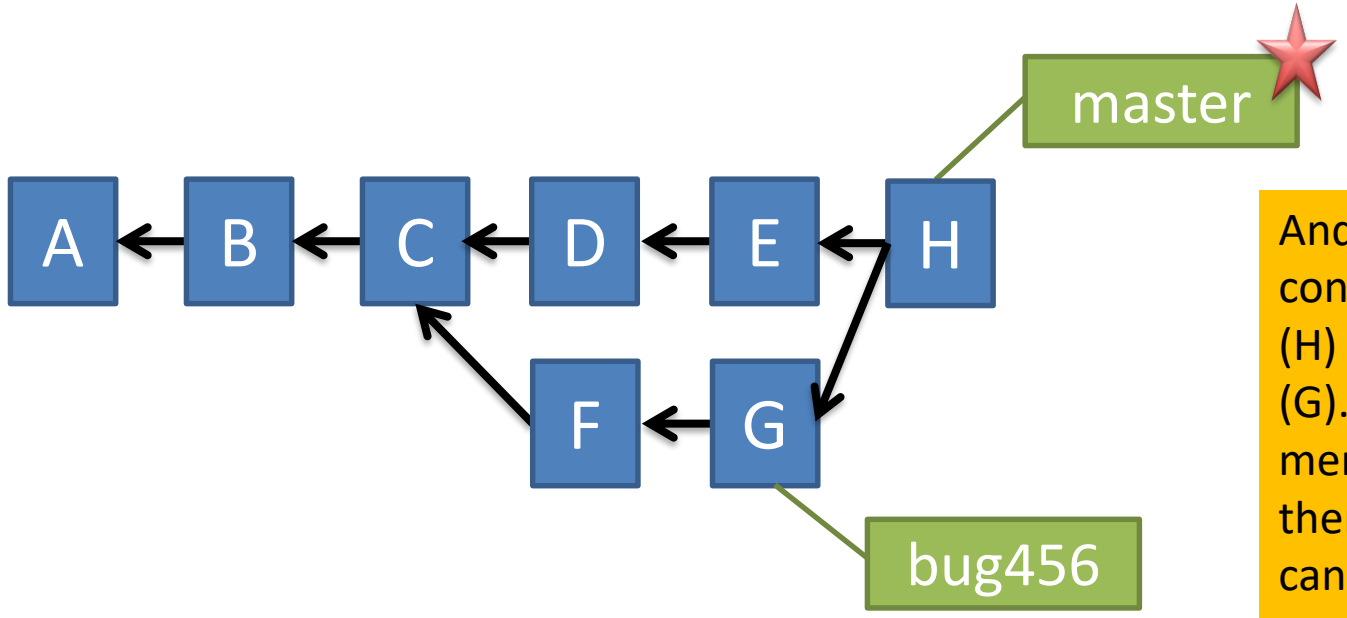
```
> git branch -d bug123
```

Check out master so we can merge next




```
> git checkout master
```

Merge bug456 branch into head (master)



And now we merge, connecting the new (H) to both (E) and (G). Note that this merge, especially if there are conflicts, can be unpleasant to perform.

```
> git merge bug456
```

Setting up a Remote

Adding a remote to an existing local repo

```
C:\CoolProject > git remote add origin
https://git01.codeplex.com/coolproject
C:\CoolProject > git remote -v
origin  https://git01.codeplex.com/coolproject (fetch)
origin  https://git01.codeplex.com/coolproject (push)
```

Setting up a Remote

Clone will auto setup the remote

```
C:\> git clone https://git01.codeplex.com/coolproject
Cloning into 'coolproject'...
remote: Counting objects: 3, done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
C:\> cd .\coolproject
C:\CoolProject> git remote -v
origin  https://git01.codeplex.com/coolproject (fetch)
origin  https://git01.codeplex.com/coolproject (push)
```

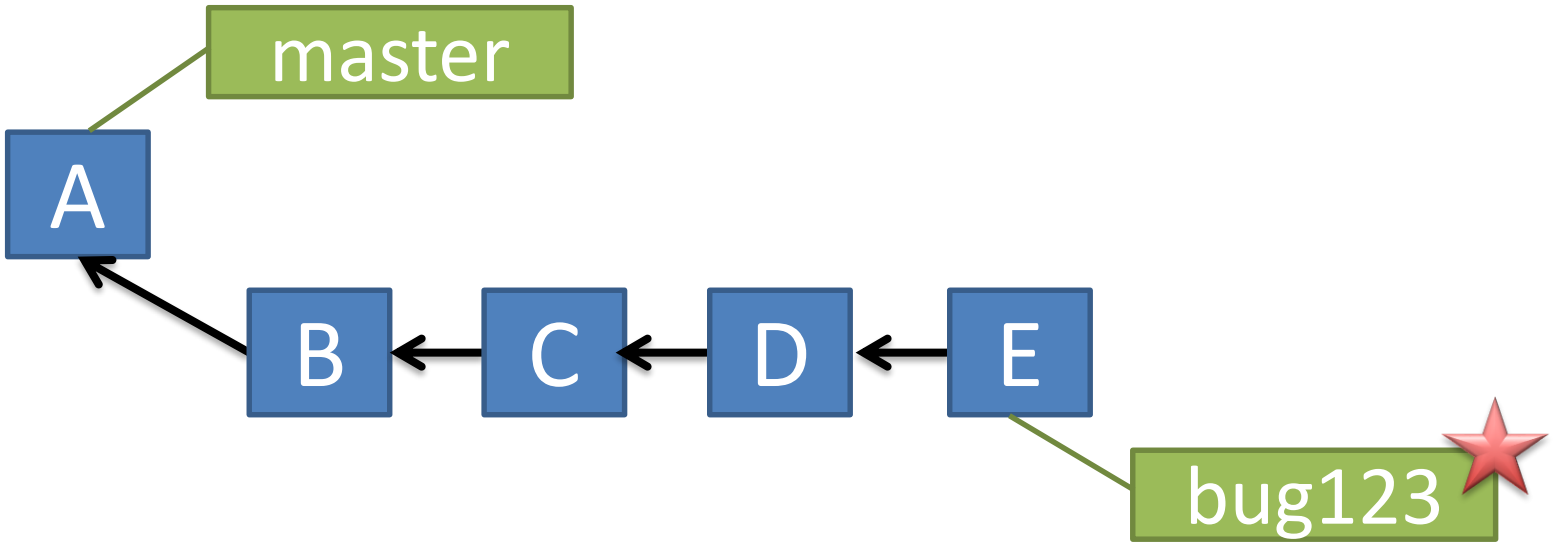
Remote branches

To see them: `git branch -r`

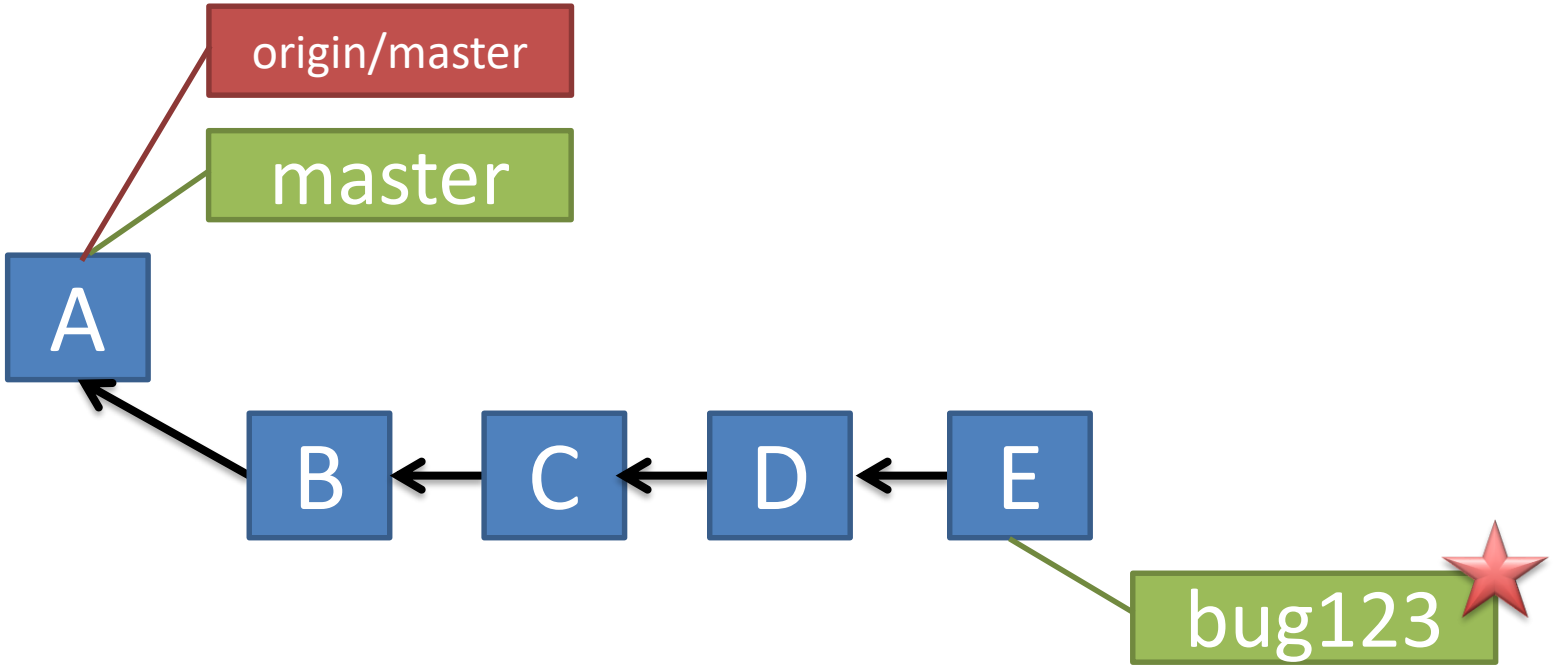
```
origin/HEAD -> origin/master
origin/master
origin/update
```

Or look at files in `.git/refs/remotes/`

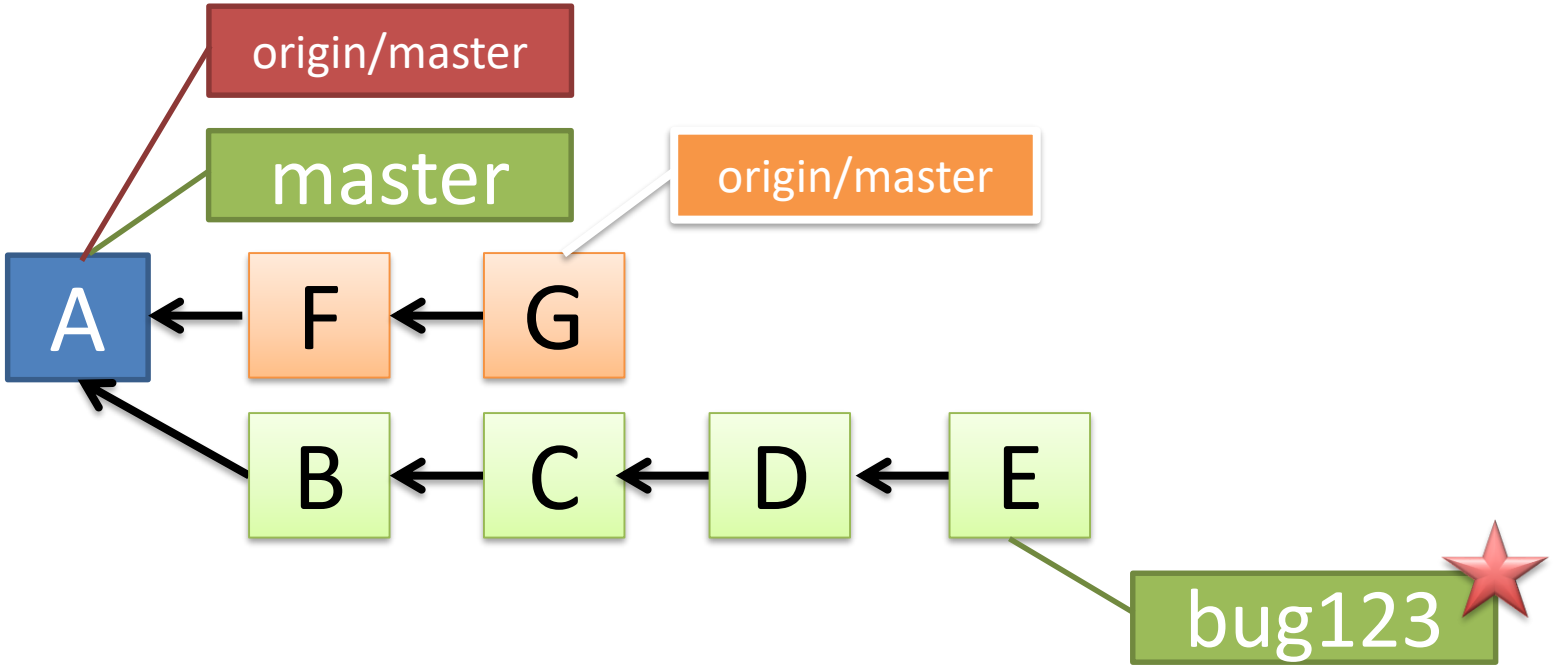
Branches Illustrated



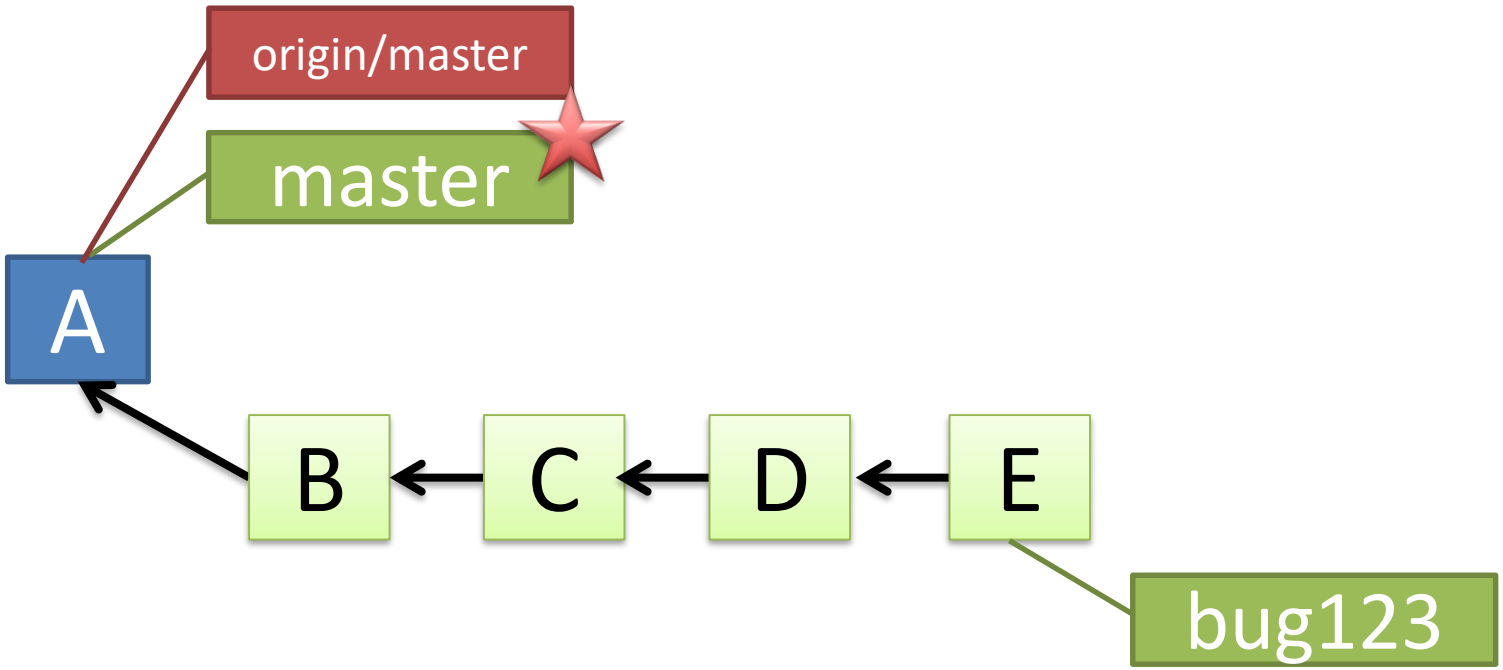
Branches Illustrated



Branches Illustrated

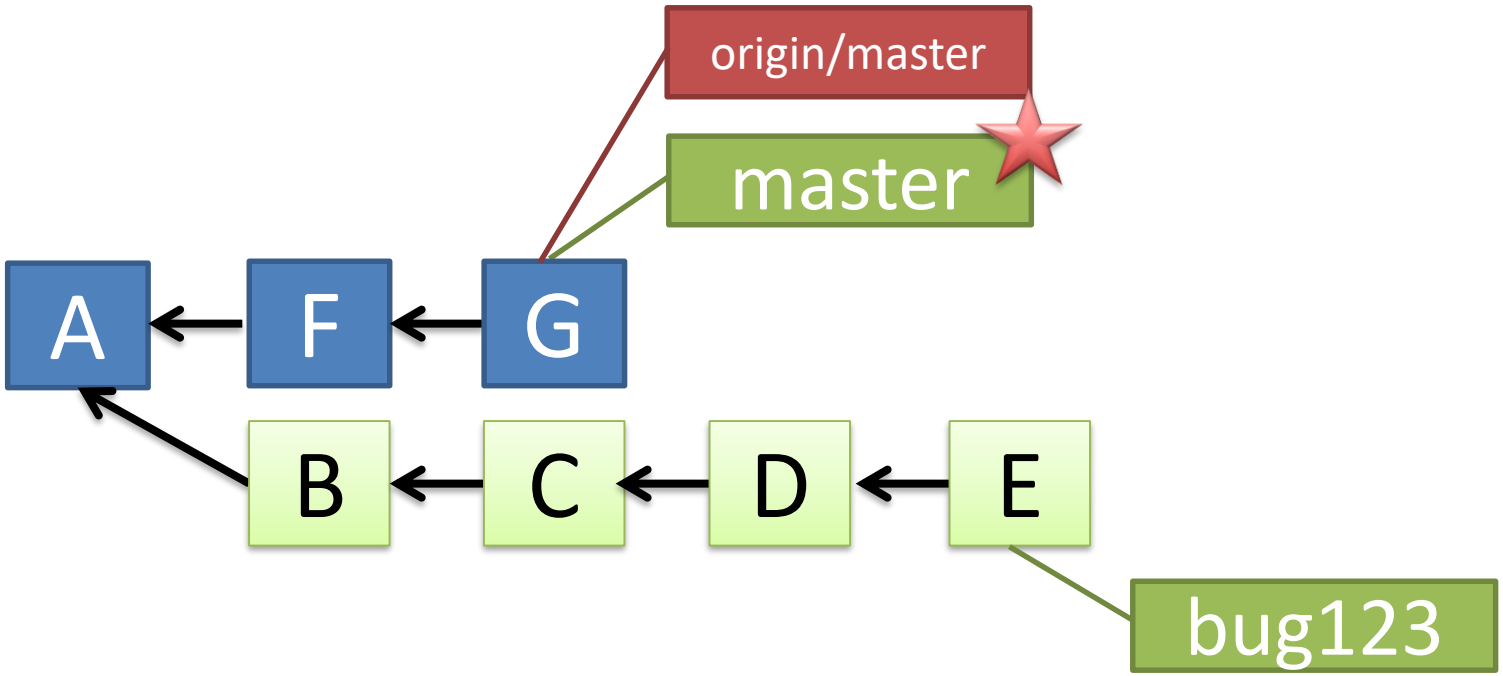


Branches Illustrated



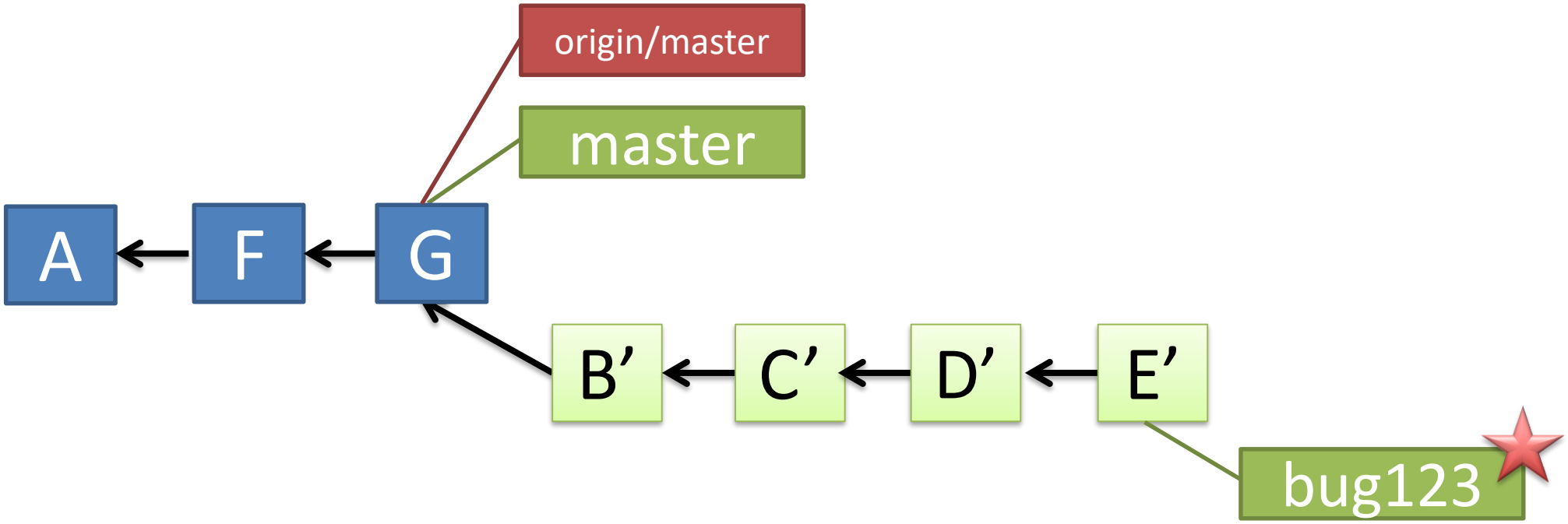

```
> git checkout master
```

Branches Illustrated



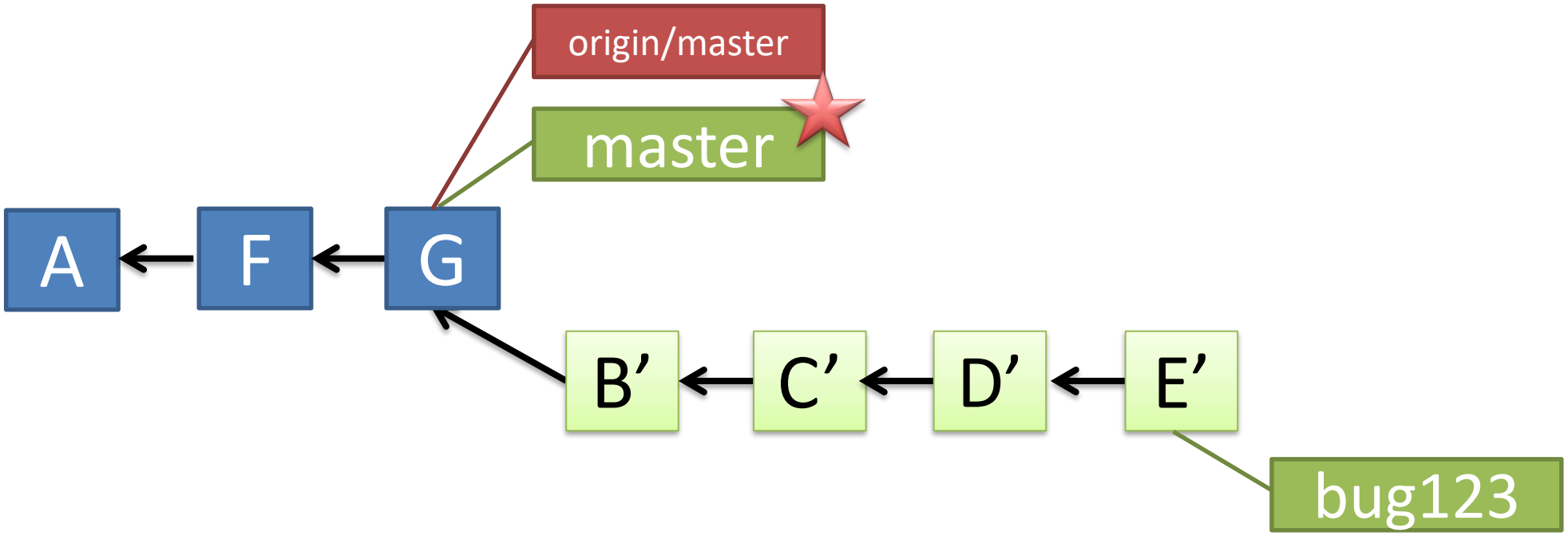

```
> git pull origin
```

Branches Illustrated



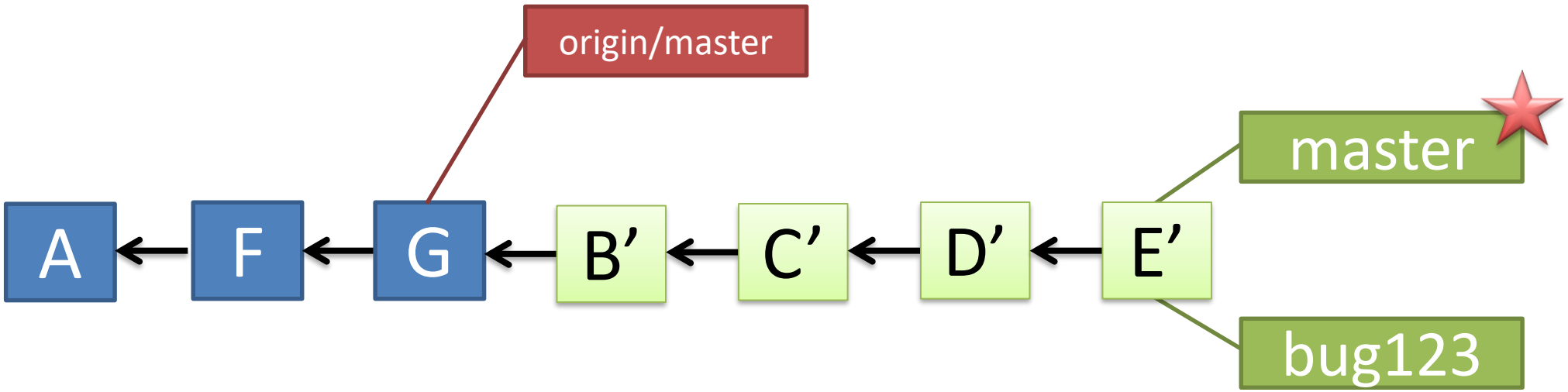

```
> git rebase master
```

Branches Illustrated



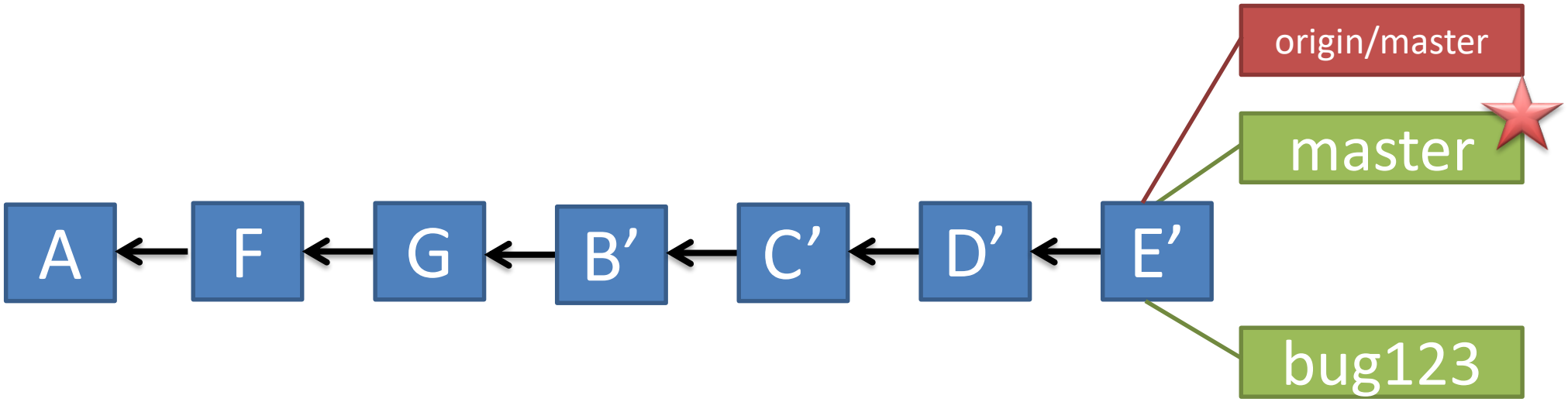

```
> git checkout master
```

Branches Illustrated



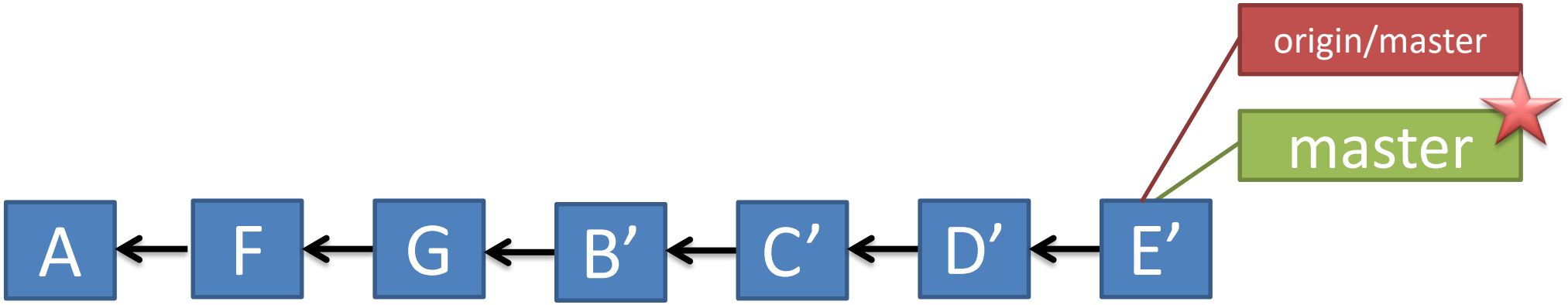

```
> git merge bug123
```

Branches Illustrated



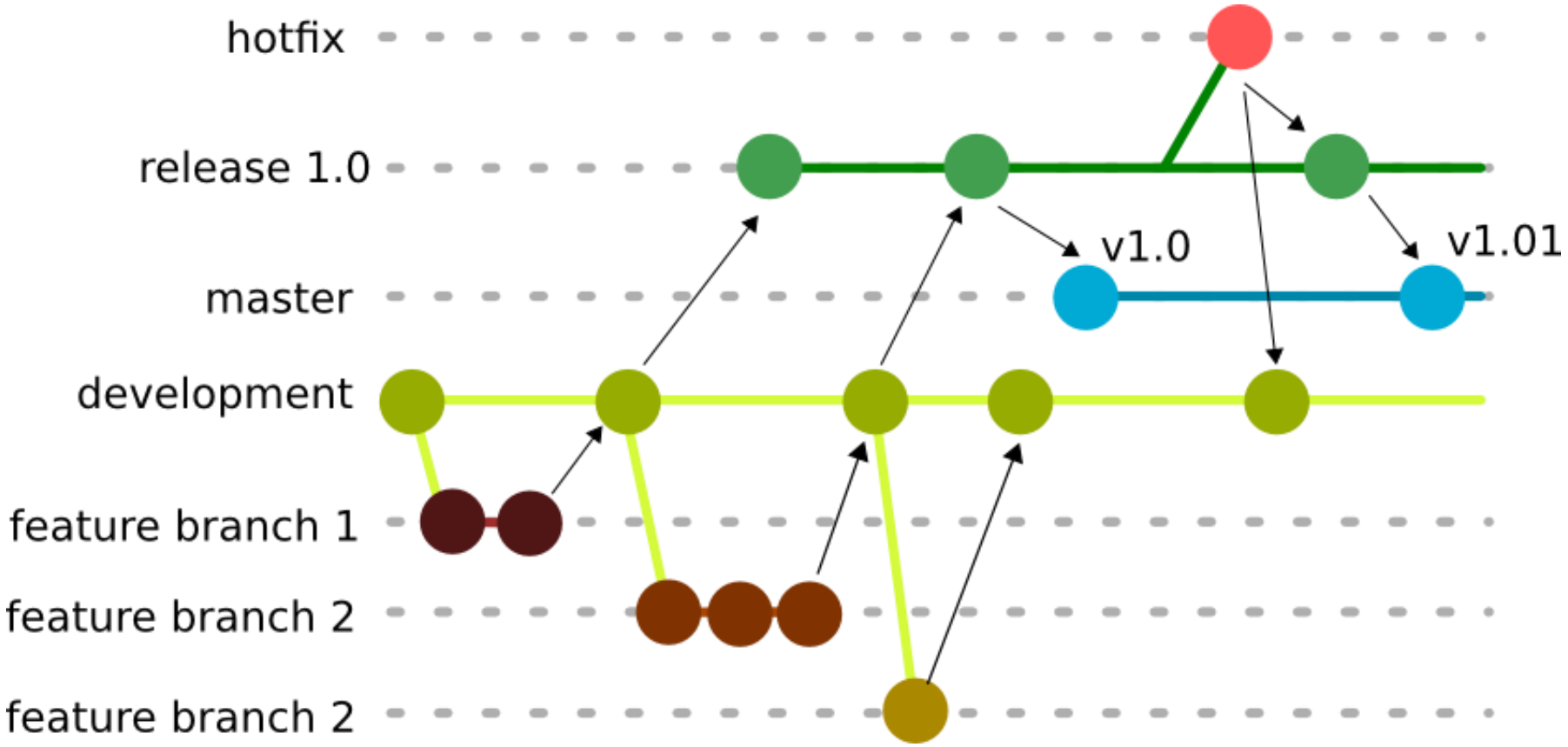

```
> git push origin
```

Branches Illustrated




```
> git branch -d bug123
```

Visualization



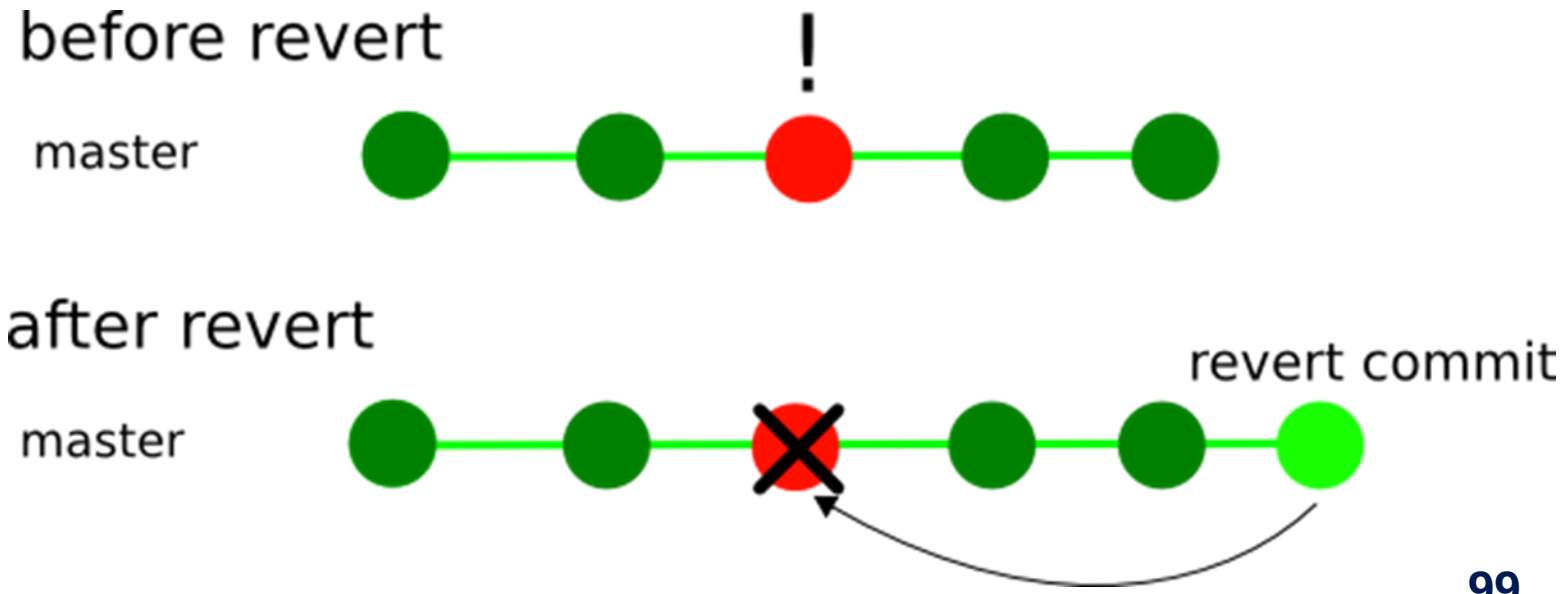
Advanced git usage

Change existing commits

- Git commit support the “amend” option to overwrite committed data
- What can be amended
  - File changes
  - Author / comment
  - Date
- The original commit **will disappear**: amend is NOT revertible (but just amended again)
- Should be avoided if a push has already happened on the amended commit!

Git revert

- You can undo commits by using revert
- Revert undoes a single commit and leaves other commits intact
- This means that you can revert a change that broke something, but reverting it doesn't affect commits made after the "faulty commit"
- Adds a commit that undoes the given commit
  - Revert = negative commit
    (eliminates effect of reverted commit)
- usage example:
  $ git revert <commit-id>
- Git revert is revertible



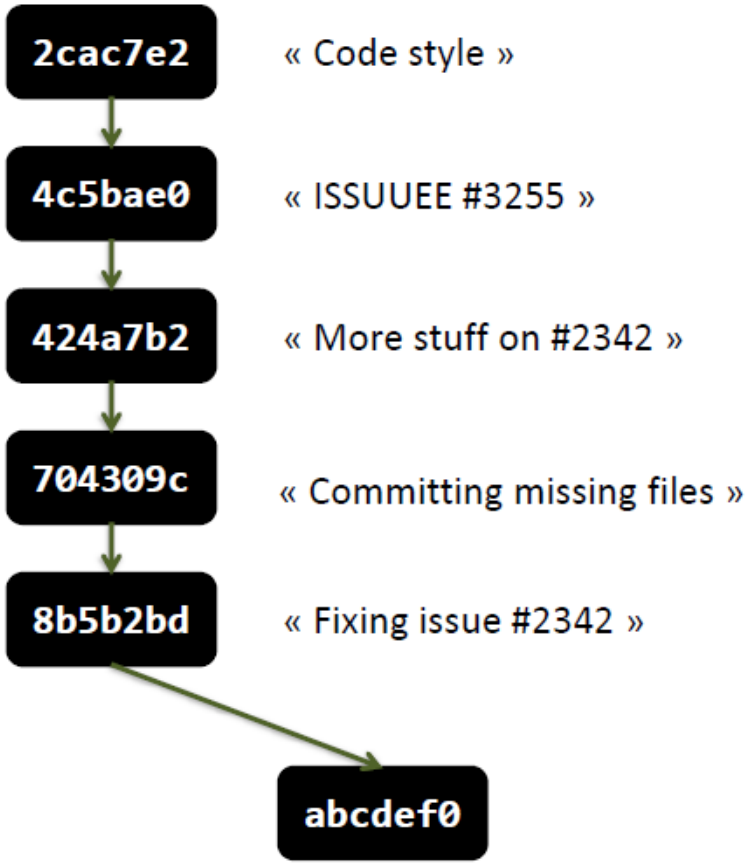
Rebase

- Contrasts to merge
  - Merge joins two branches (results in a more detailed/cluttered and easier to understand history)
  - Rebase preserves branches (results in a clean history, needs more learning)
- Rolls changes from one branch into the other
  - Changes now are now relative to newer baseline
  - Allows tracking changes to baseline while developing new branch
  - Prevents surprises later
  - Avoids conflicts with eventual merges
  - Changes history!
- Rebase frequently to avoid merge conflicts

Rebase’s capabilities

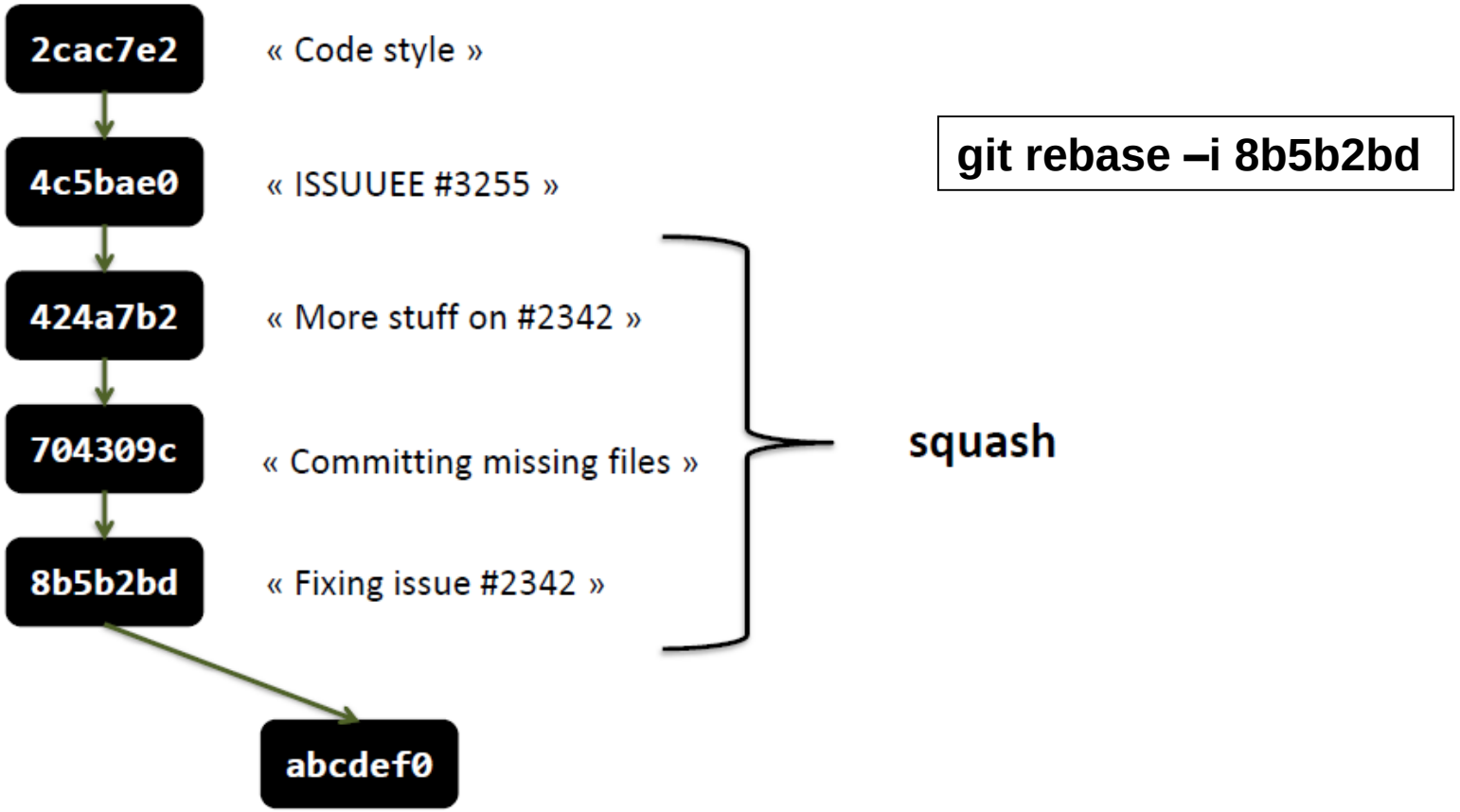
- Allows full rewriting of history:
  - Delete or reorder commits
  - Squash commits together
  - Make changes to a commit
  - Rewrite commit messages
- **WARNING**: do not use this to rewrite history you have pushed

Rebase: Simplifying history



Rebase: Simplifying history

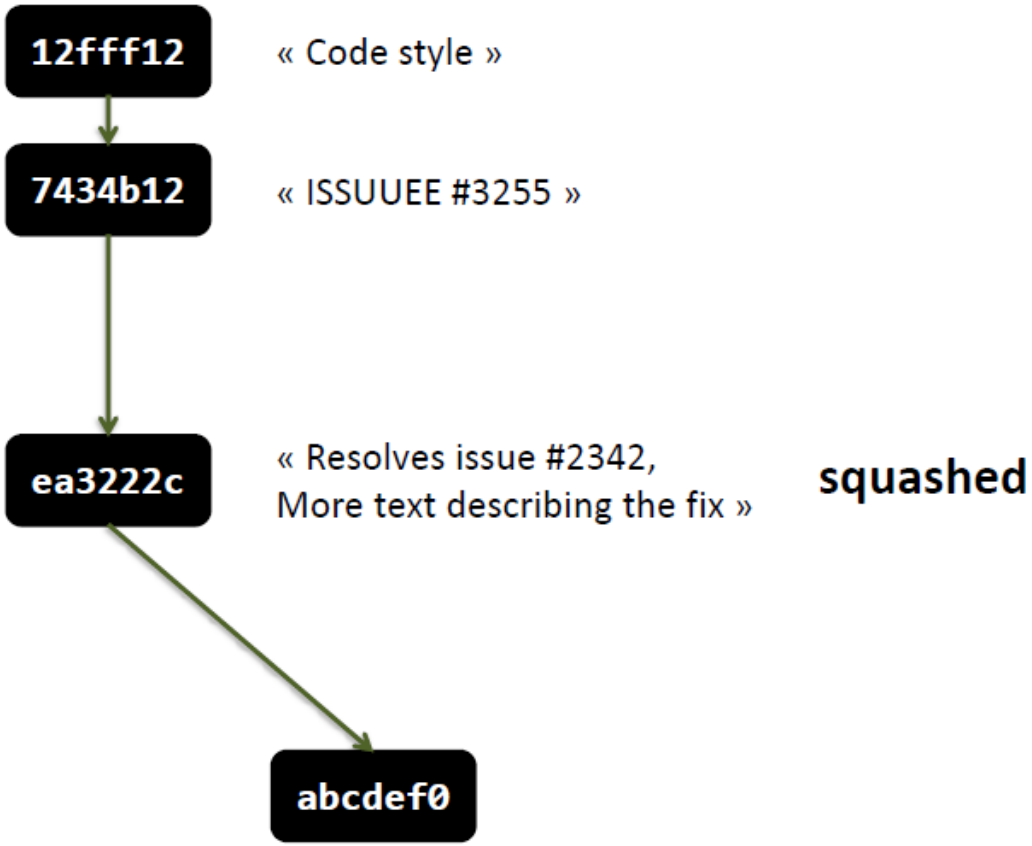
Squashing commits



git rebase –i 8b5b2bd

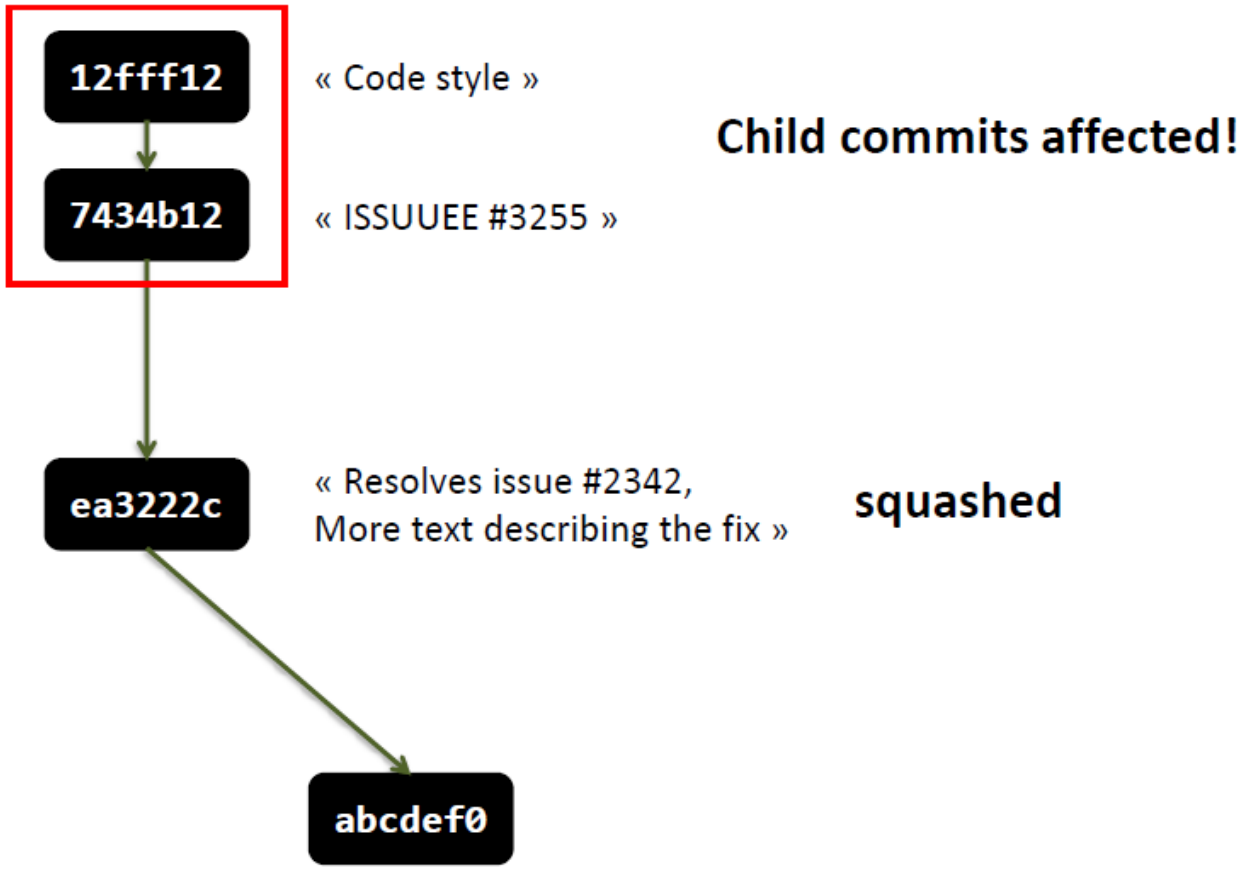
Rebase: Simplifying history

Squashing commits



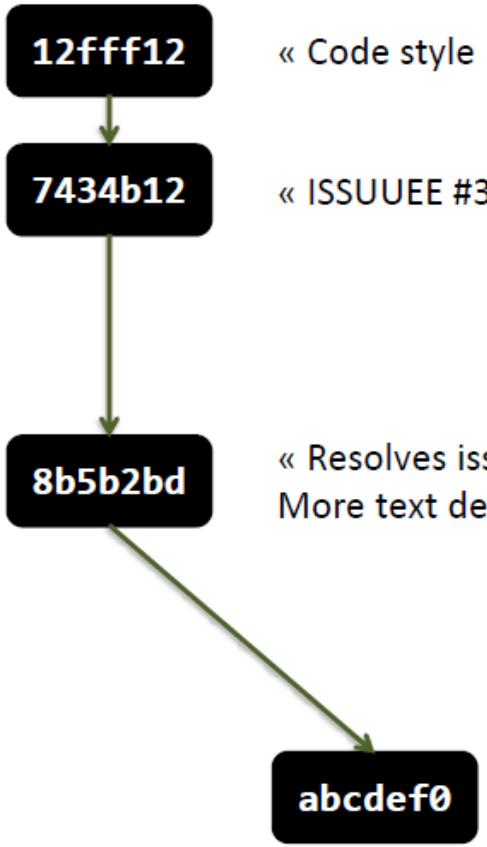
Rebase: Simplifying history

Squashing commits



Rebase: Simplifying history

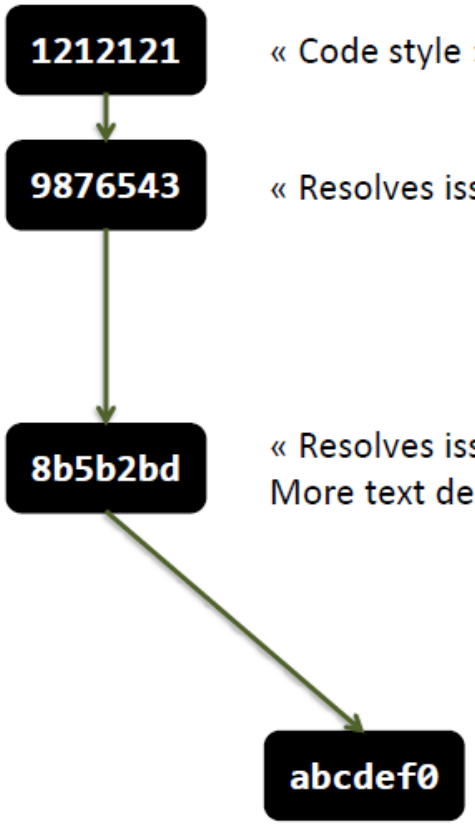
Squashing commits



Amend the commit message

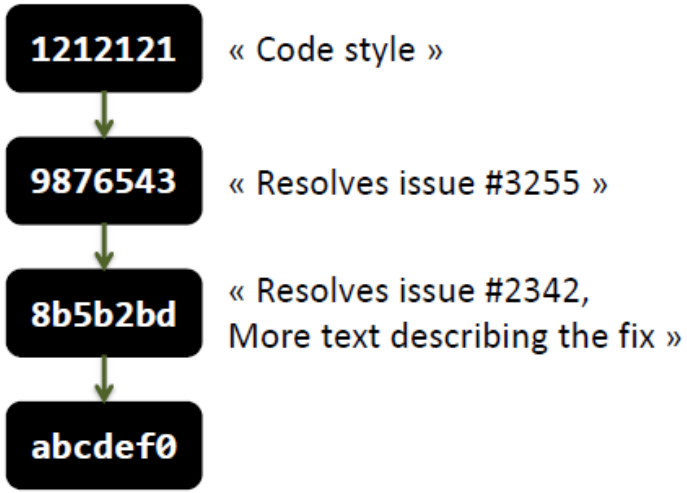
Rebase: Simplifying history

Squashing commits

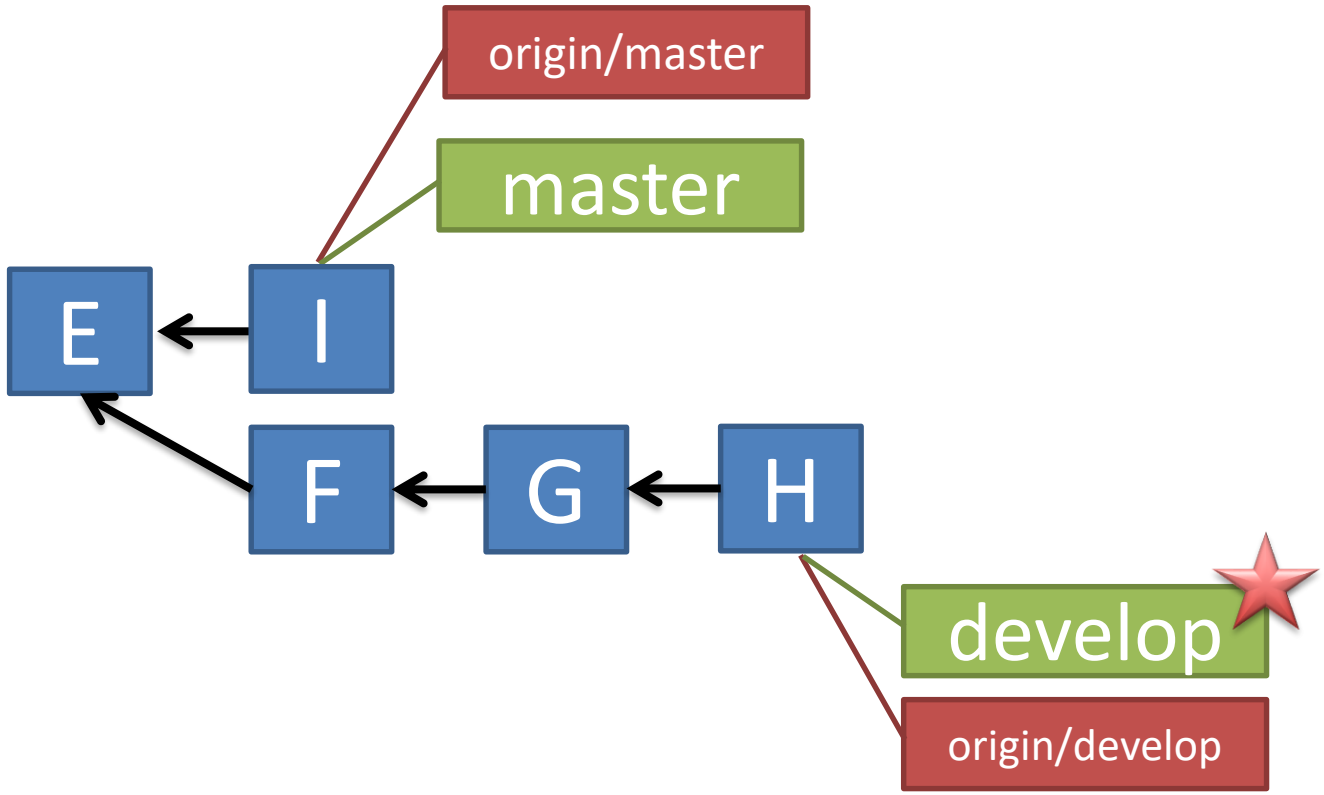


Clean History,
can be merged

Rebase: Simplifying history

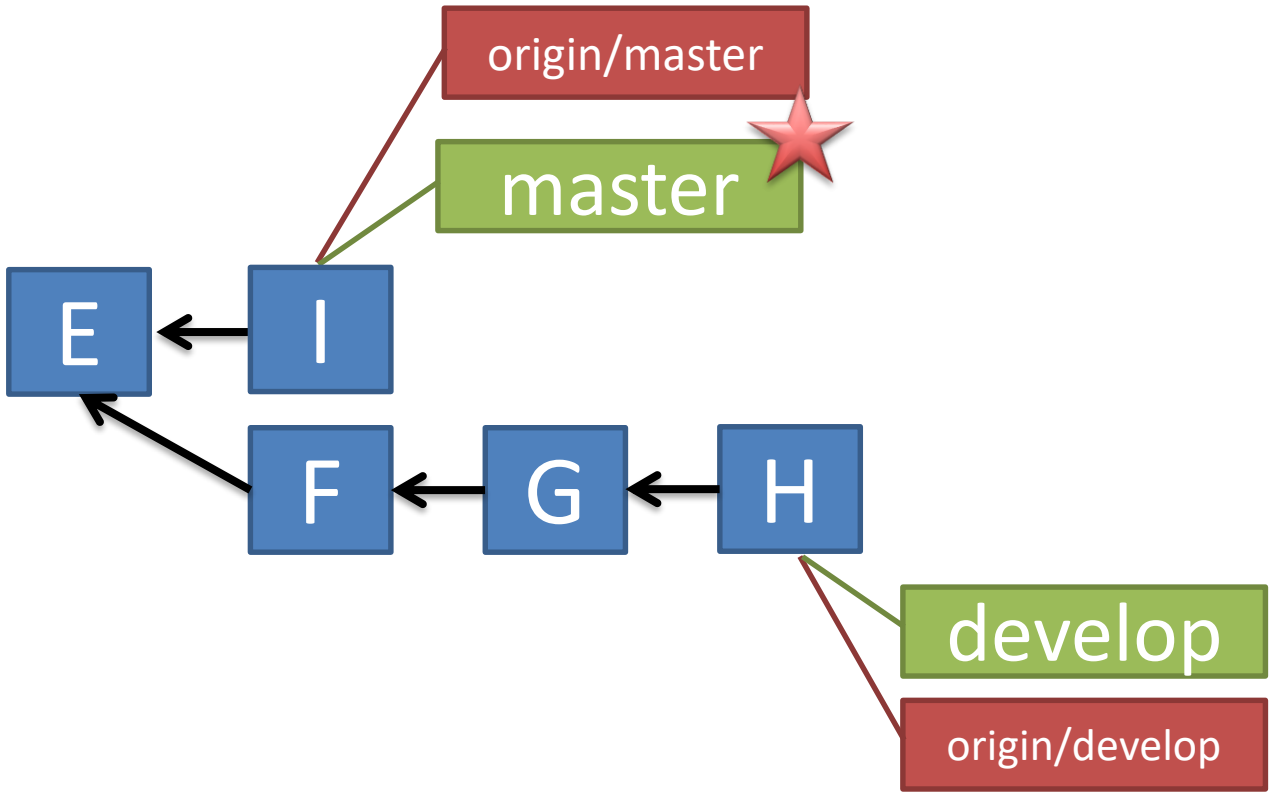


Merge Flow vs. Rebase Flow



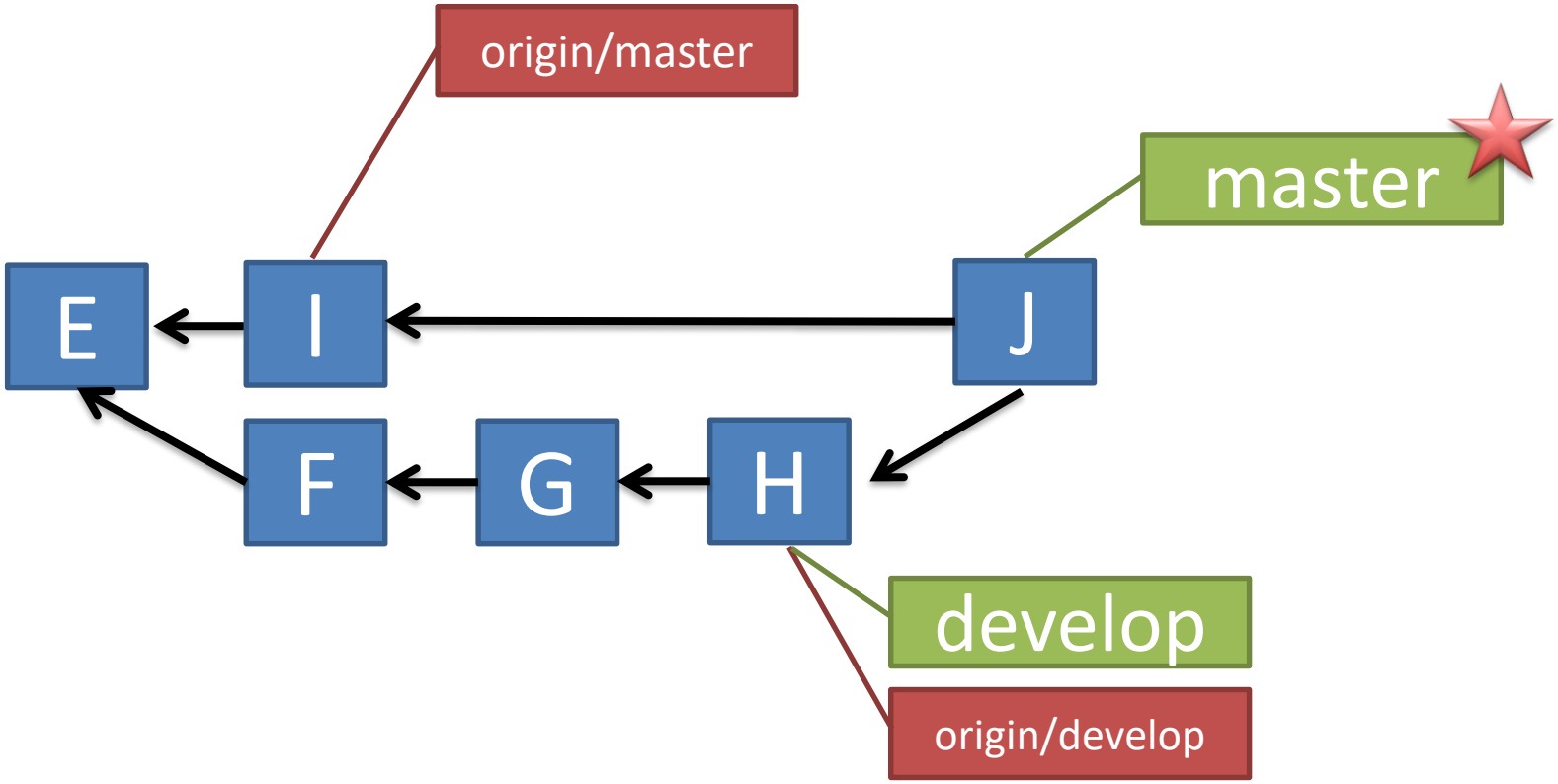

```
> git push origin develop
```

Branches Illustrated – Merge Flow



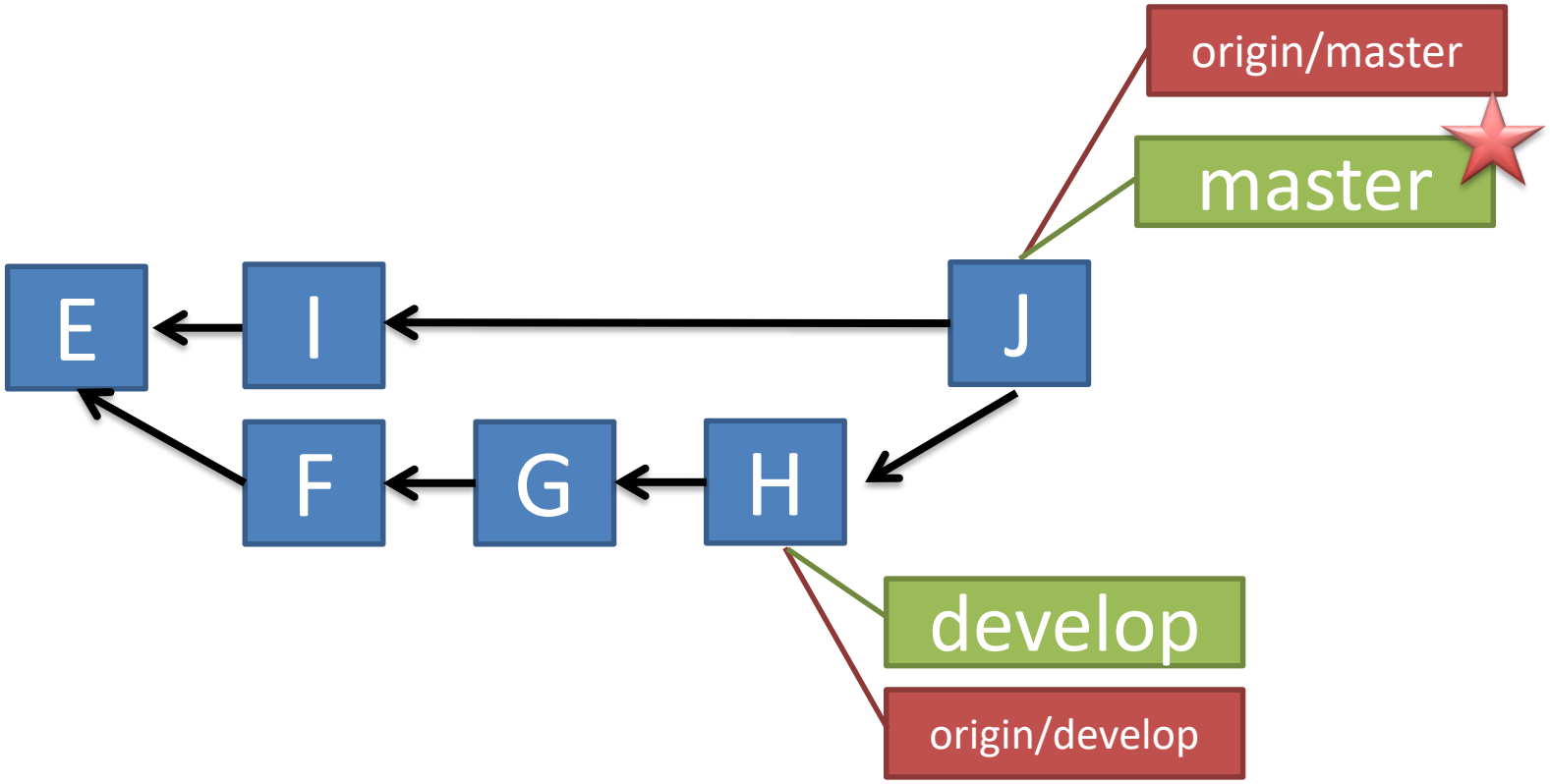

```
> git checkout master
```

Branches Illustrated – Merge Flow



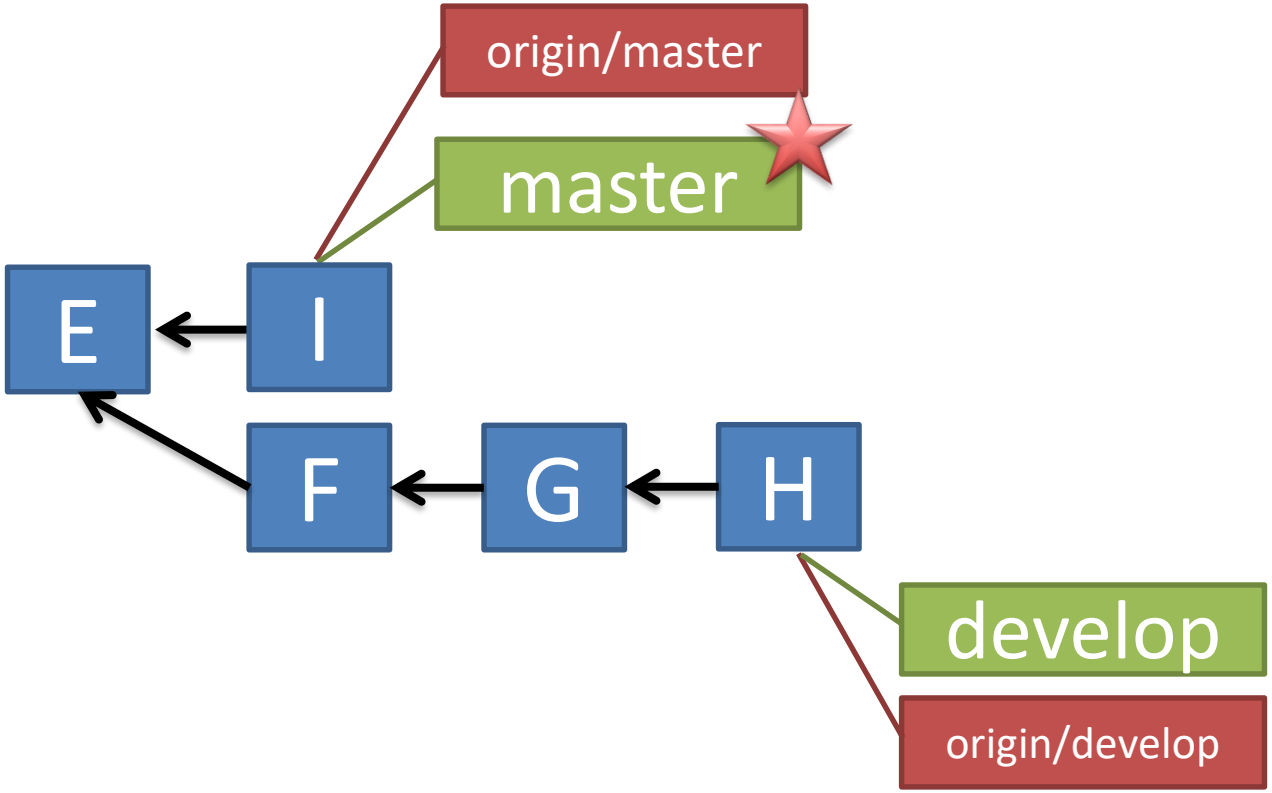

```
> git merge develop
```

Branches Illustrated – Merge Flow



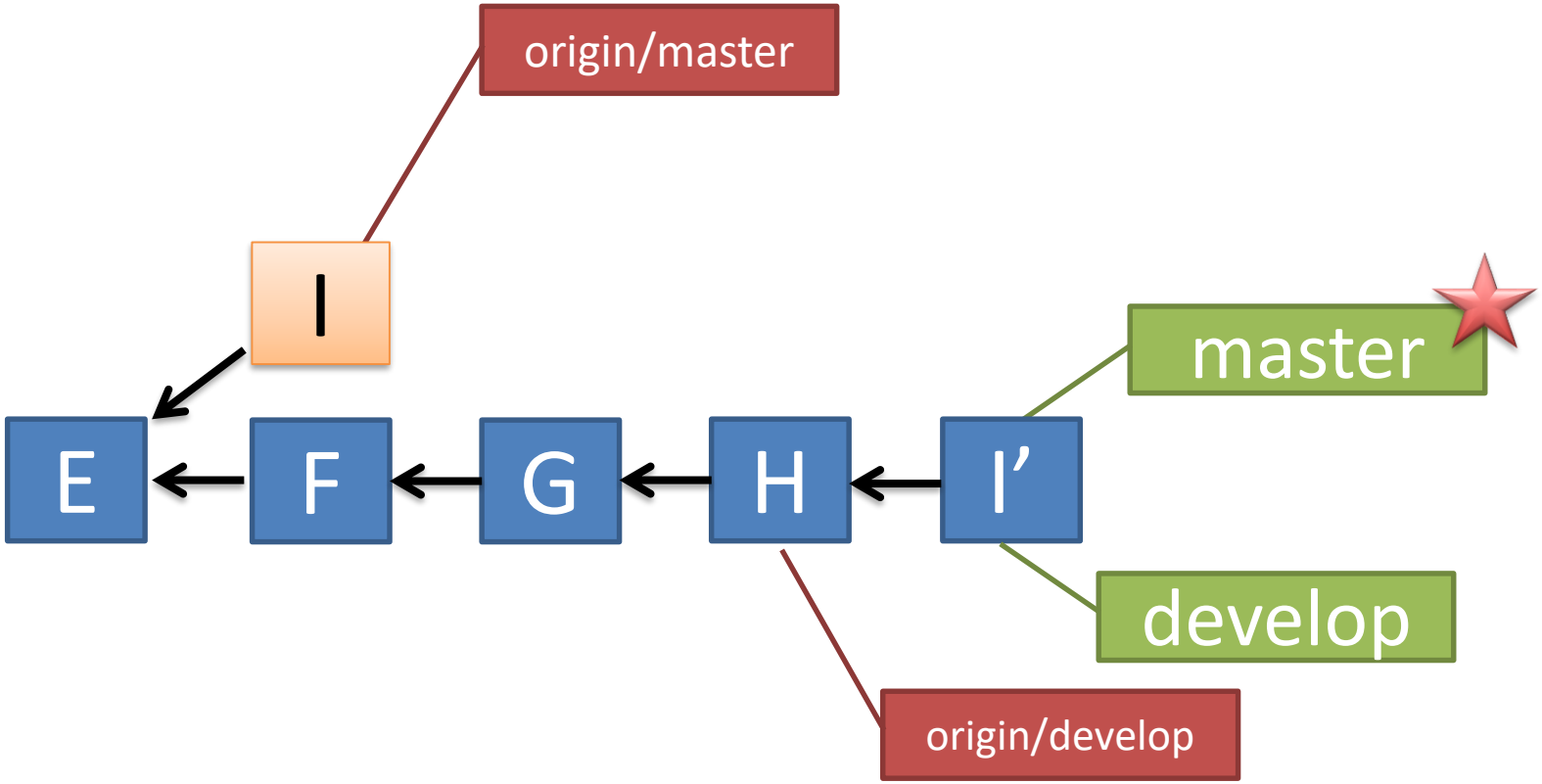

```
> git push origin
```

Branches Illustrated – Rebase Flow



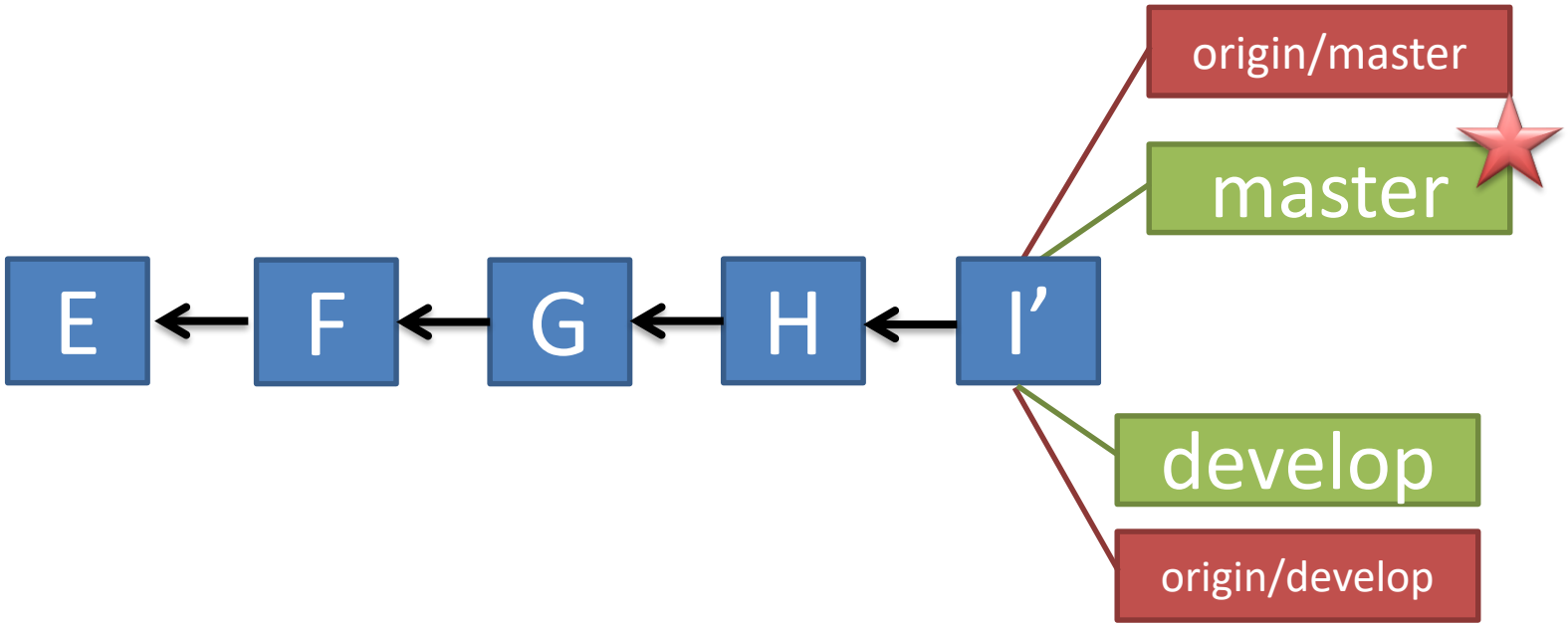

```
> git checkout master
```

Branches Illustrated – Rebase Flow



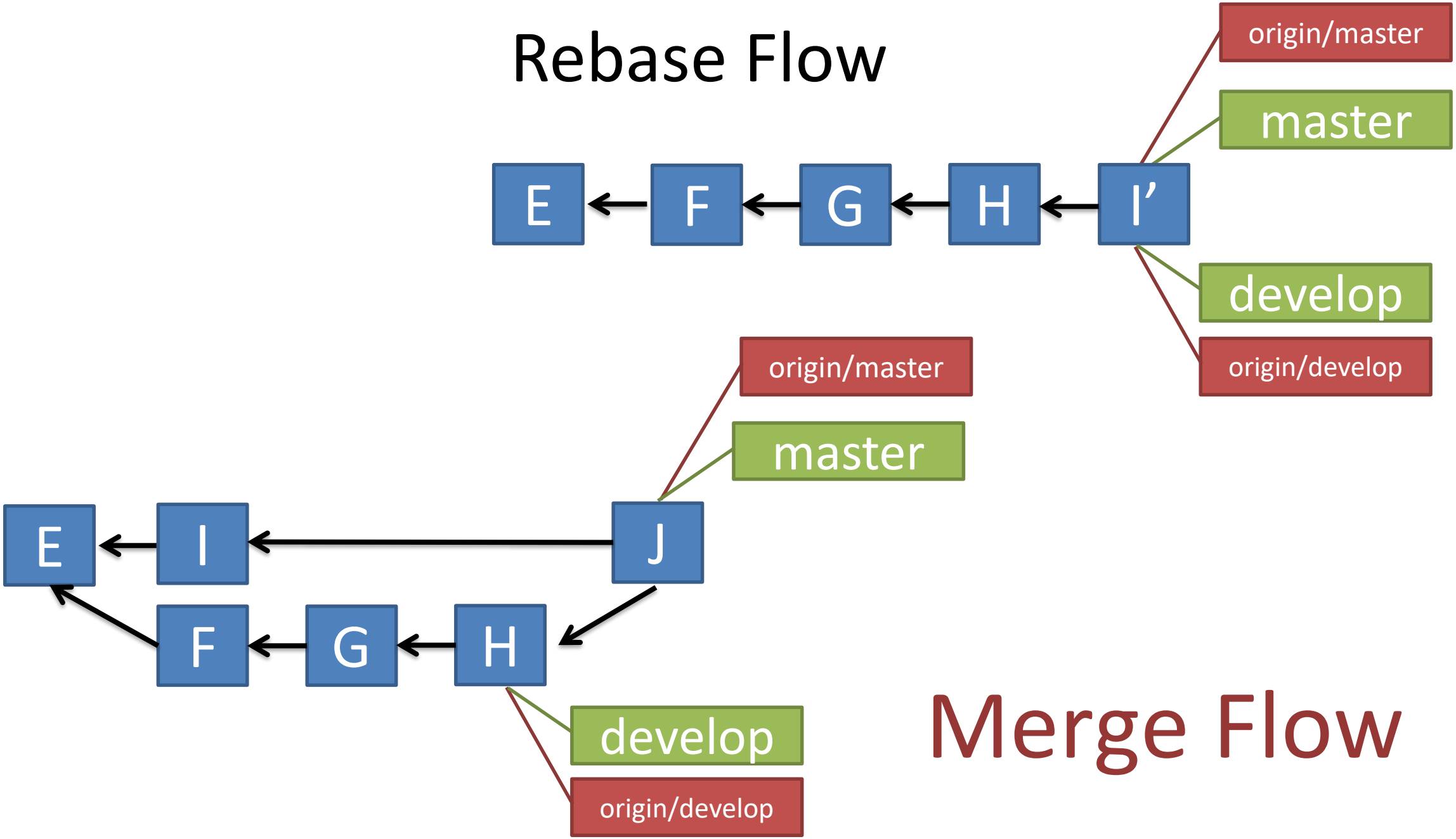

```
> git rebase develop
```

Branches Illustrated – Rebase Flow




```
> git push origin
```

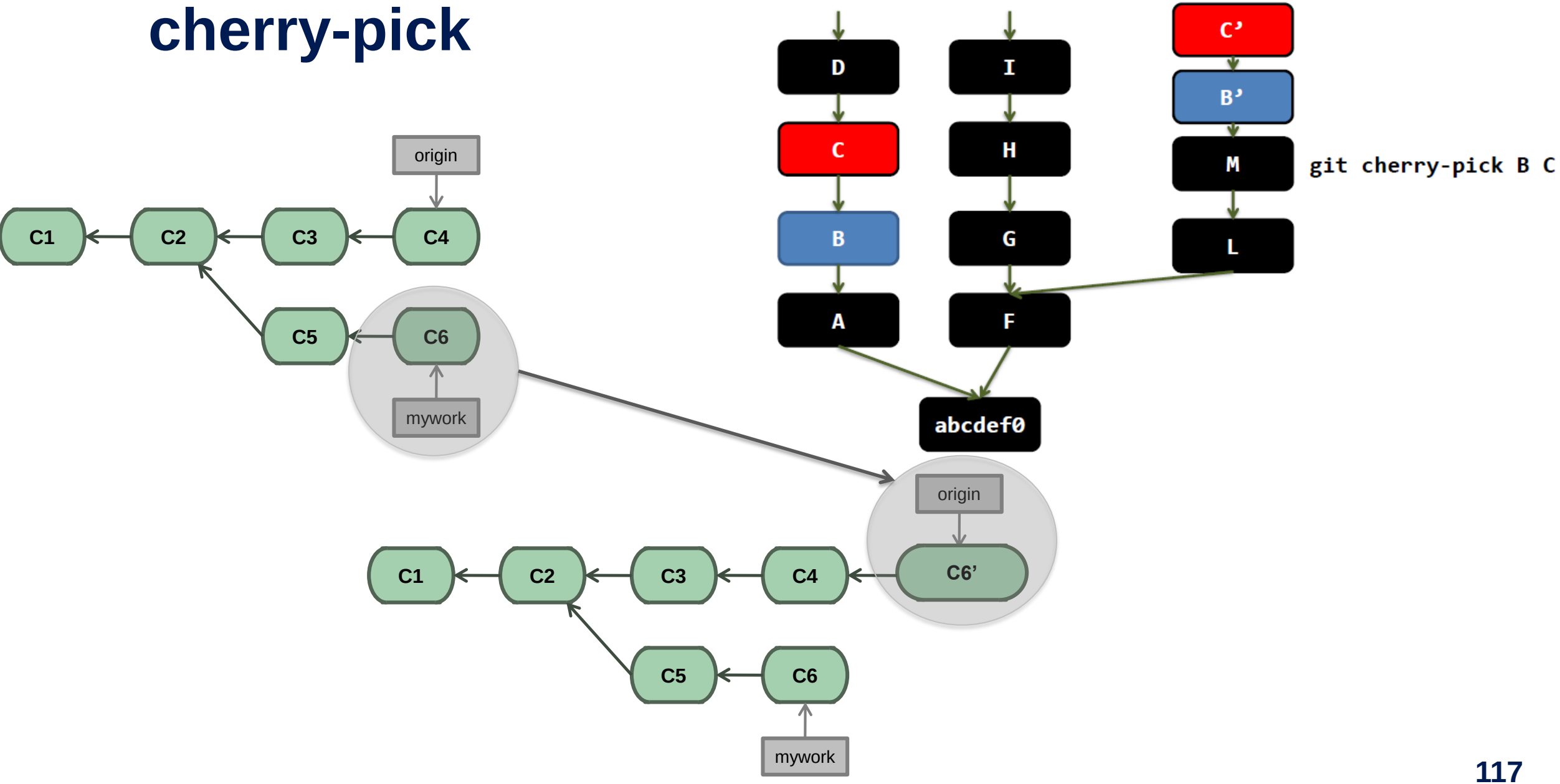
Rebase Flow



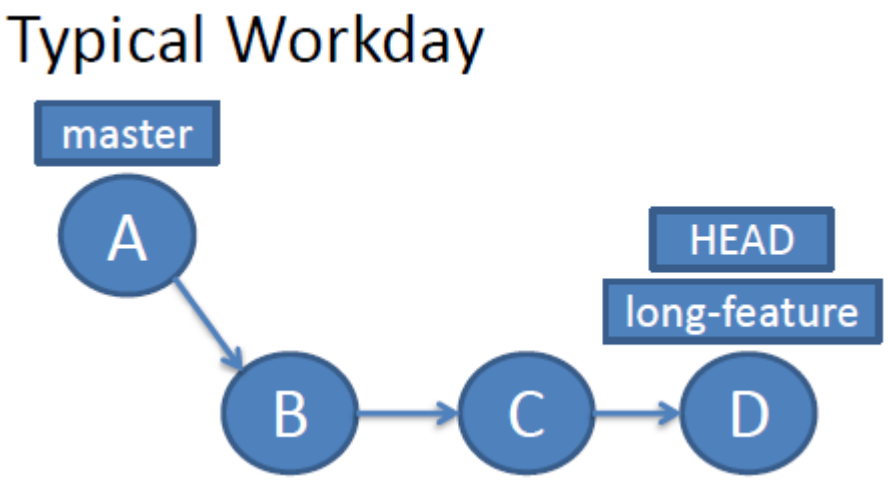
Merge Flow

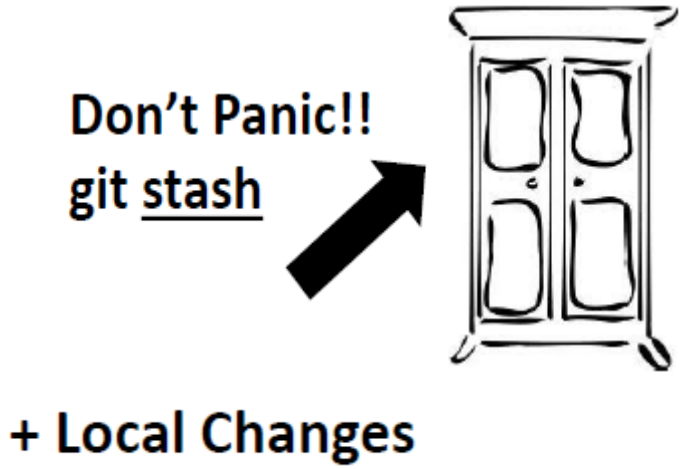
Other merge alternative: cherry-pick

"Given one or more existing commits, apply the change each one introduces, recording a new commit for each."

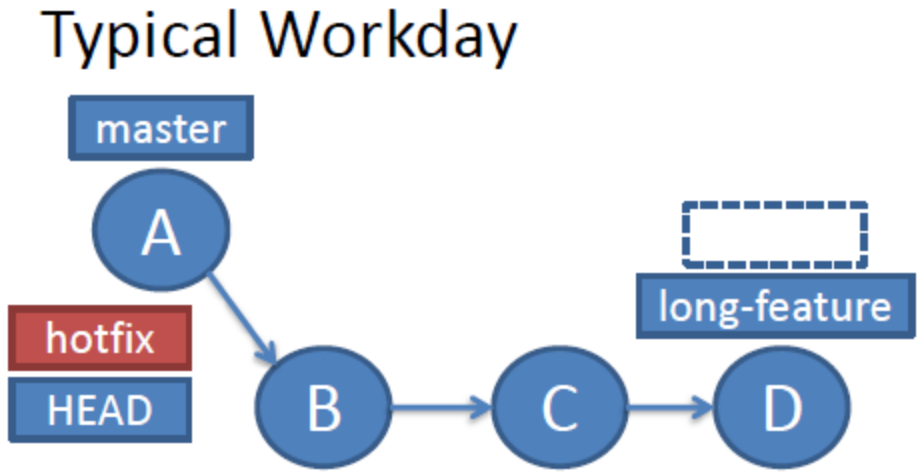


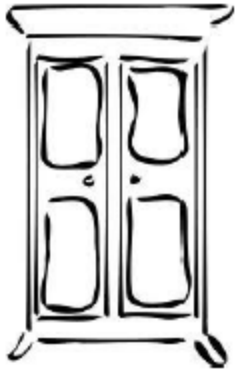
Using stash to improve your workflow





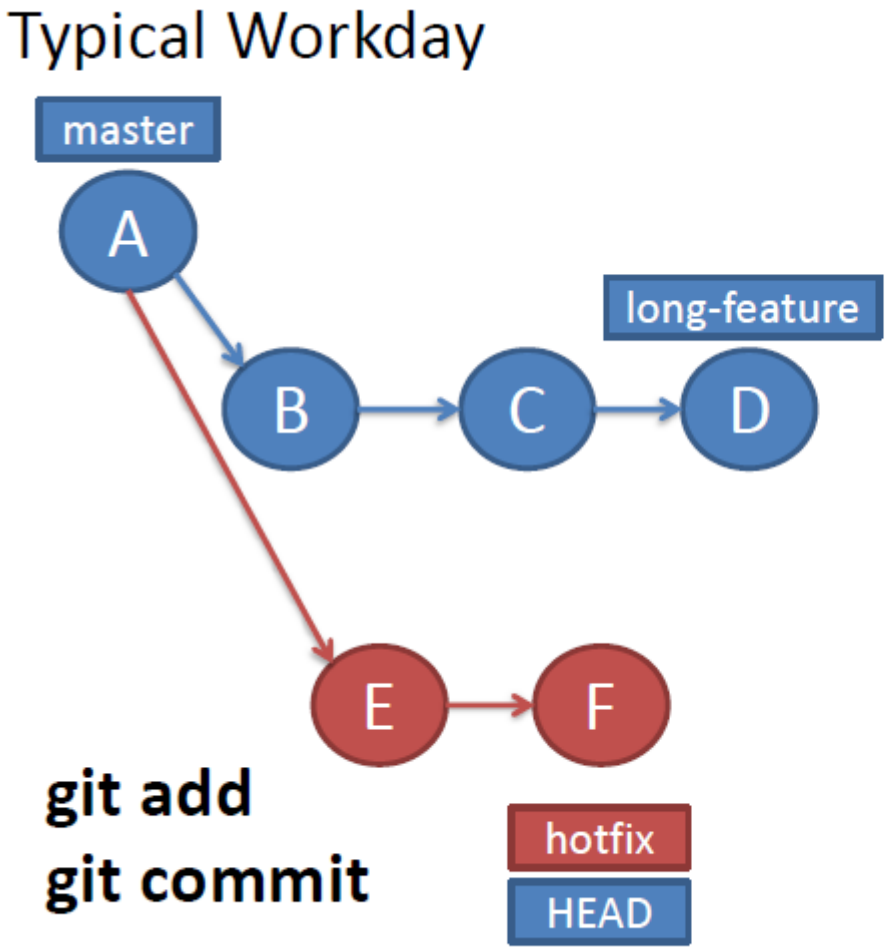
Using stash to improve your workflow

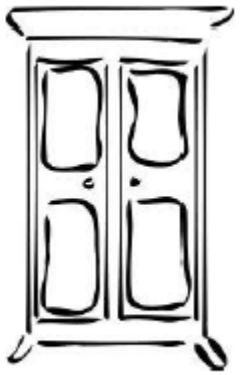




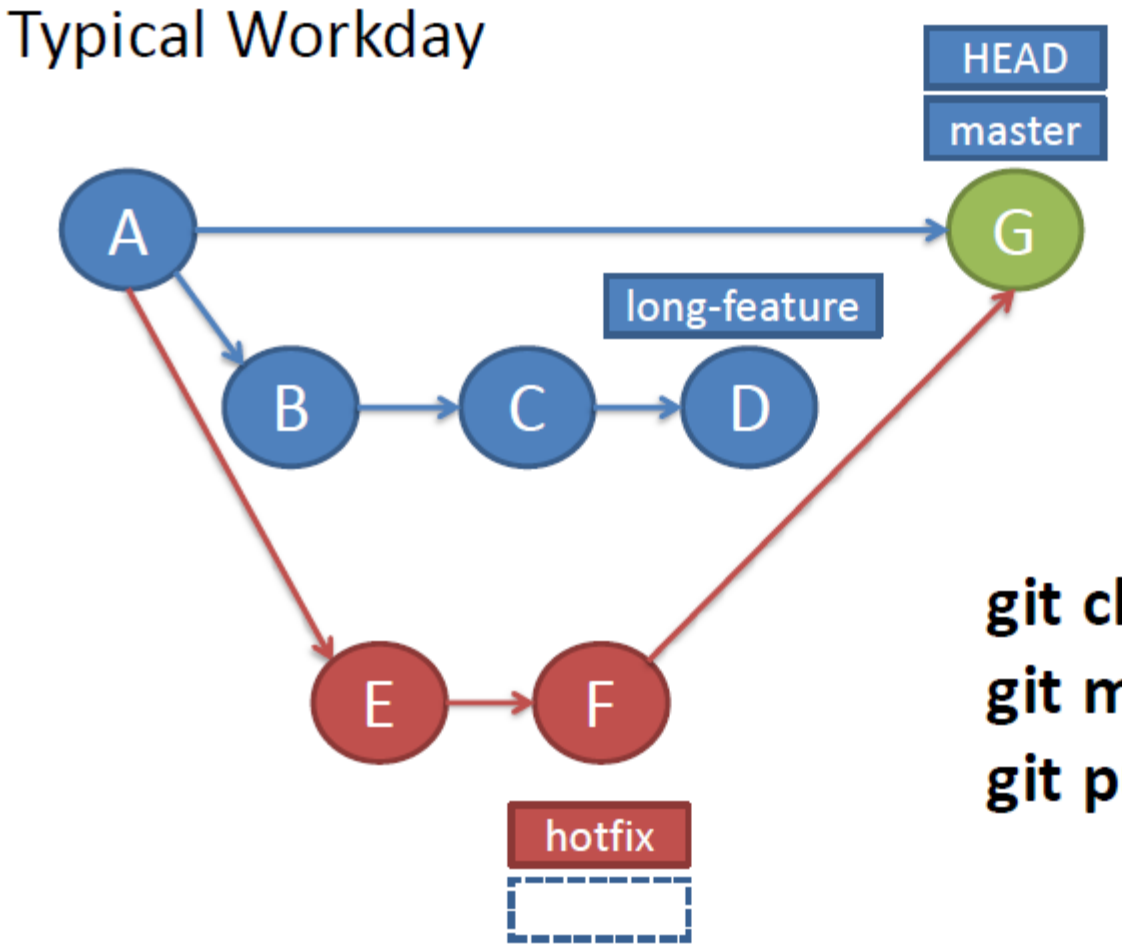
git branch hotfix
git checkout hotfix **==** **git checkout –b hotfix**

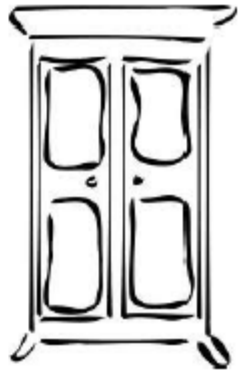
Using stash to improve your workflow





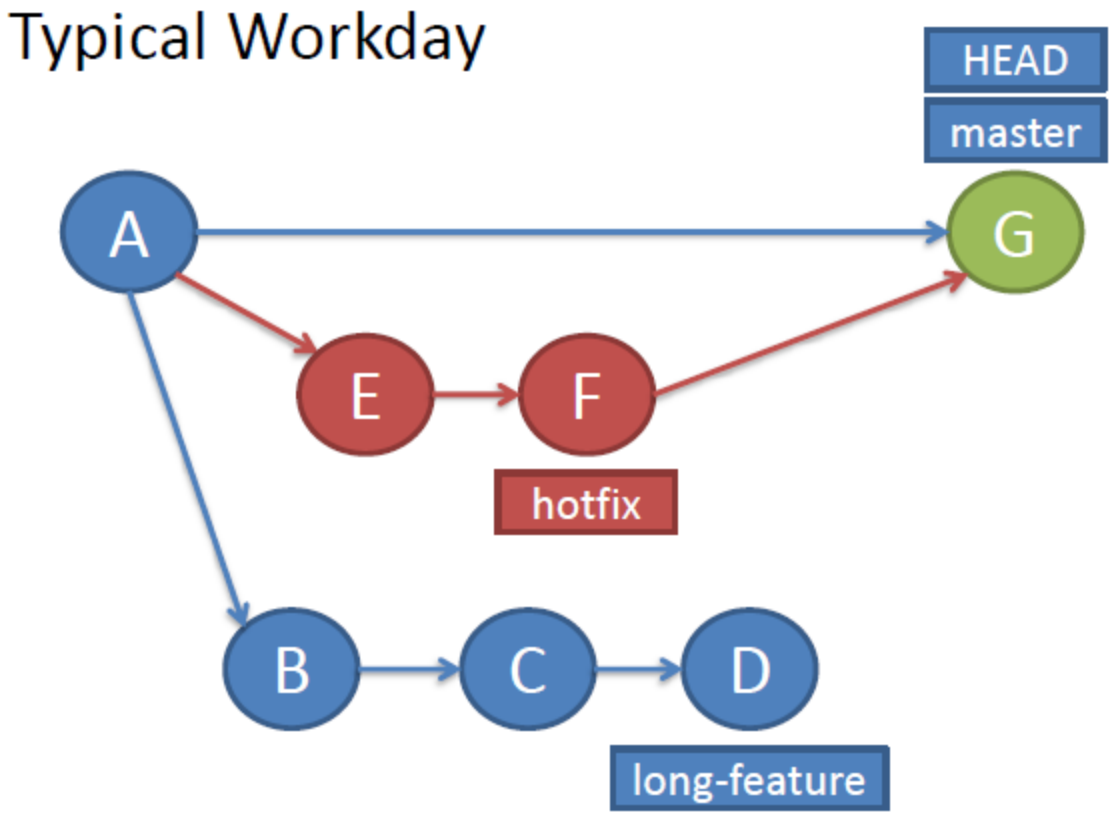
Using stash to improve your workflow

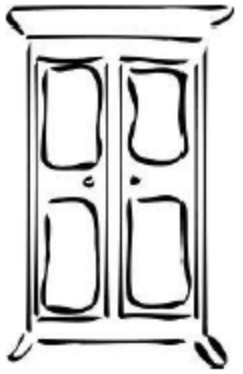




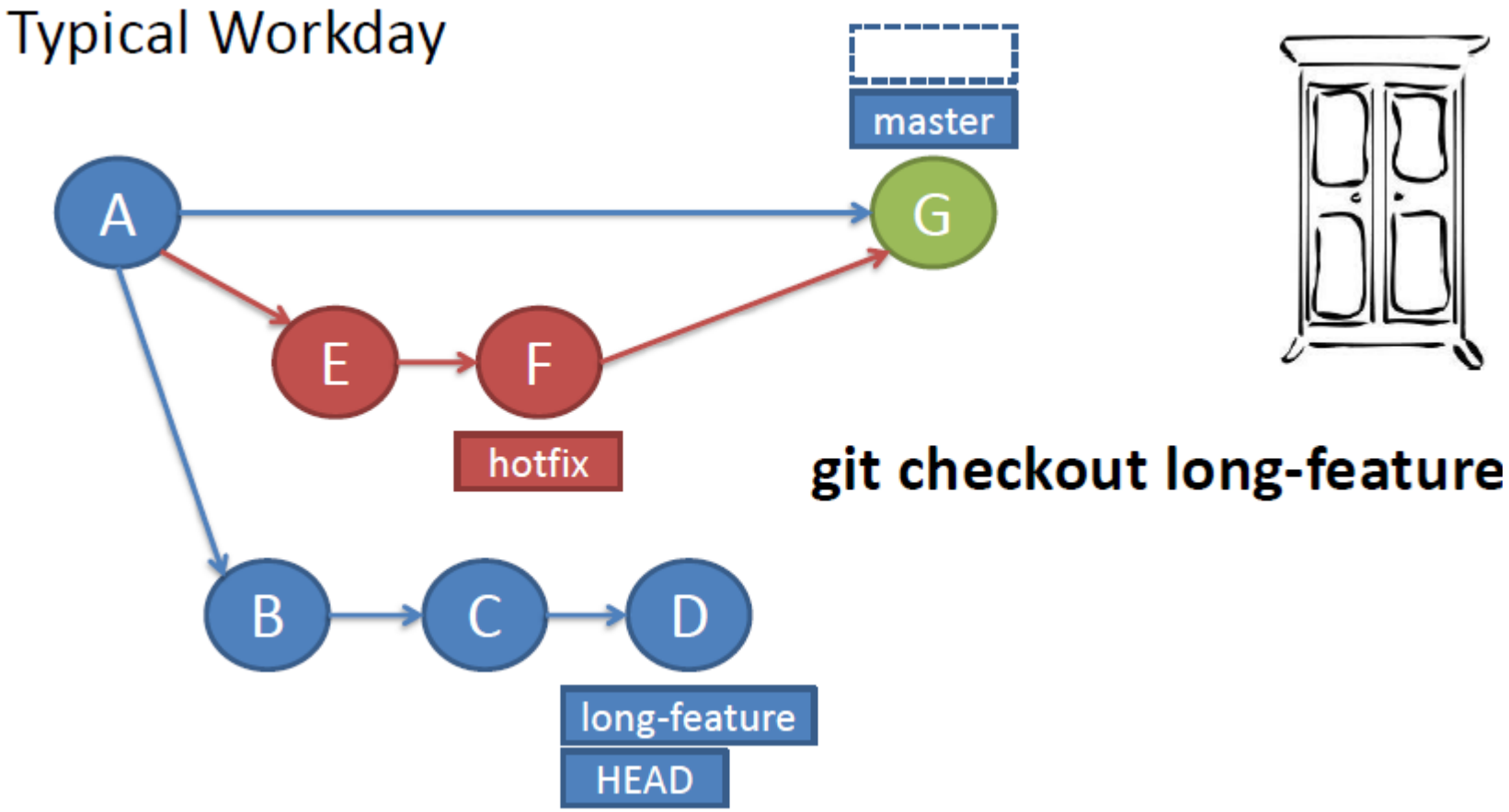
git checkout master
git merge hotfix
git push origin master

Using stash to improve your workflow

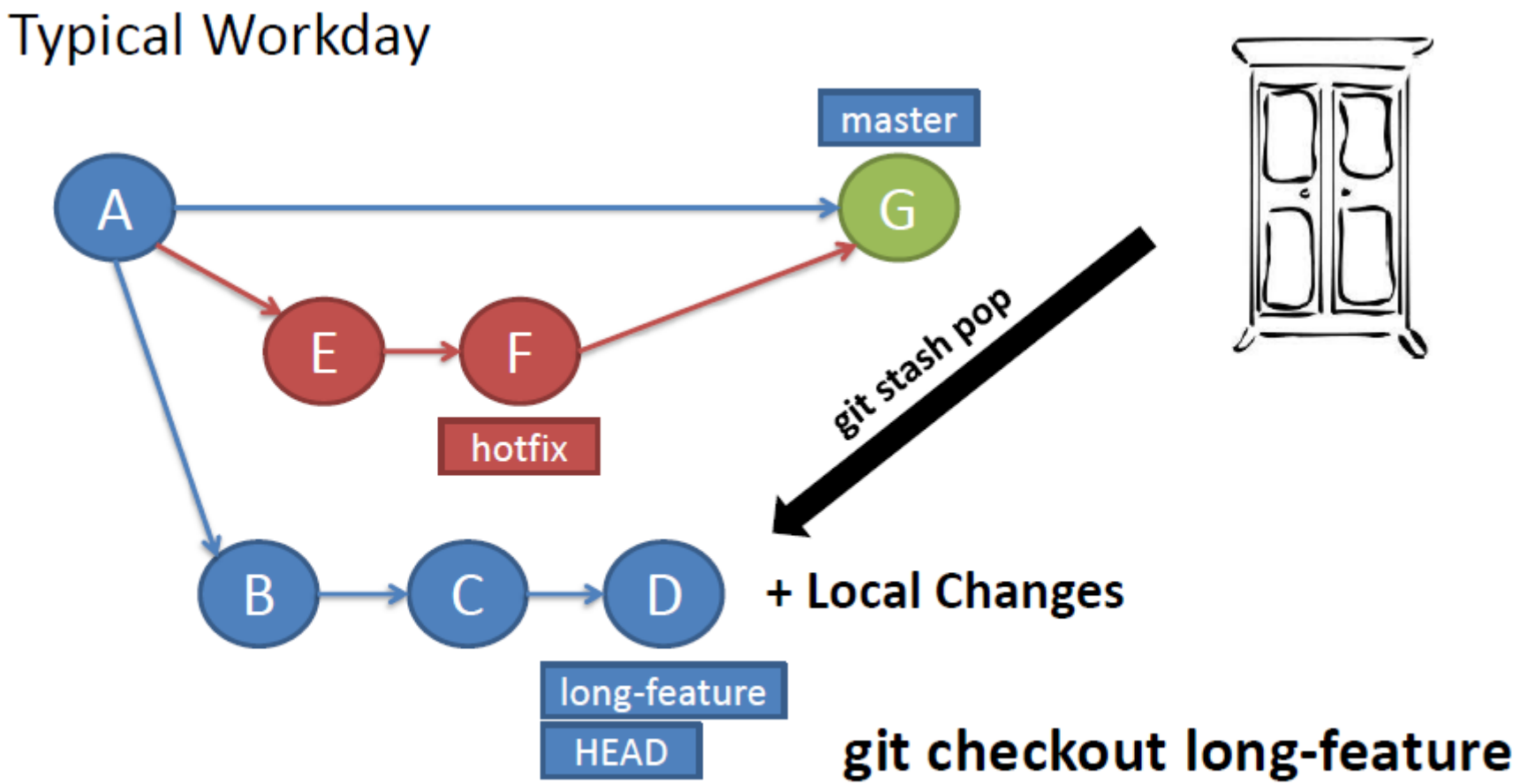




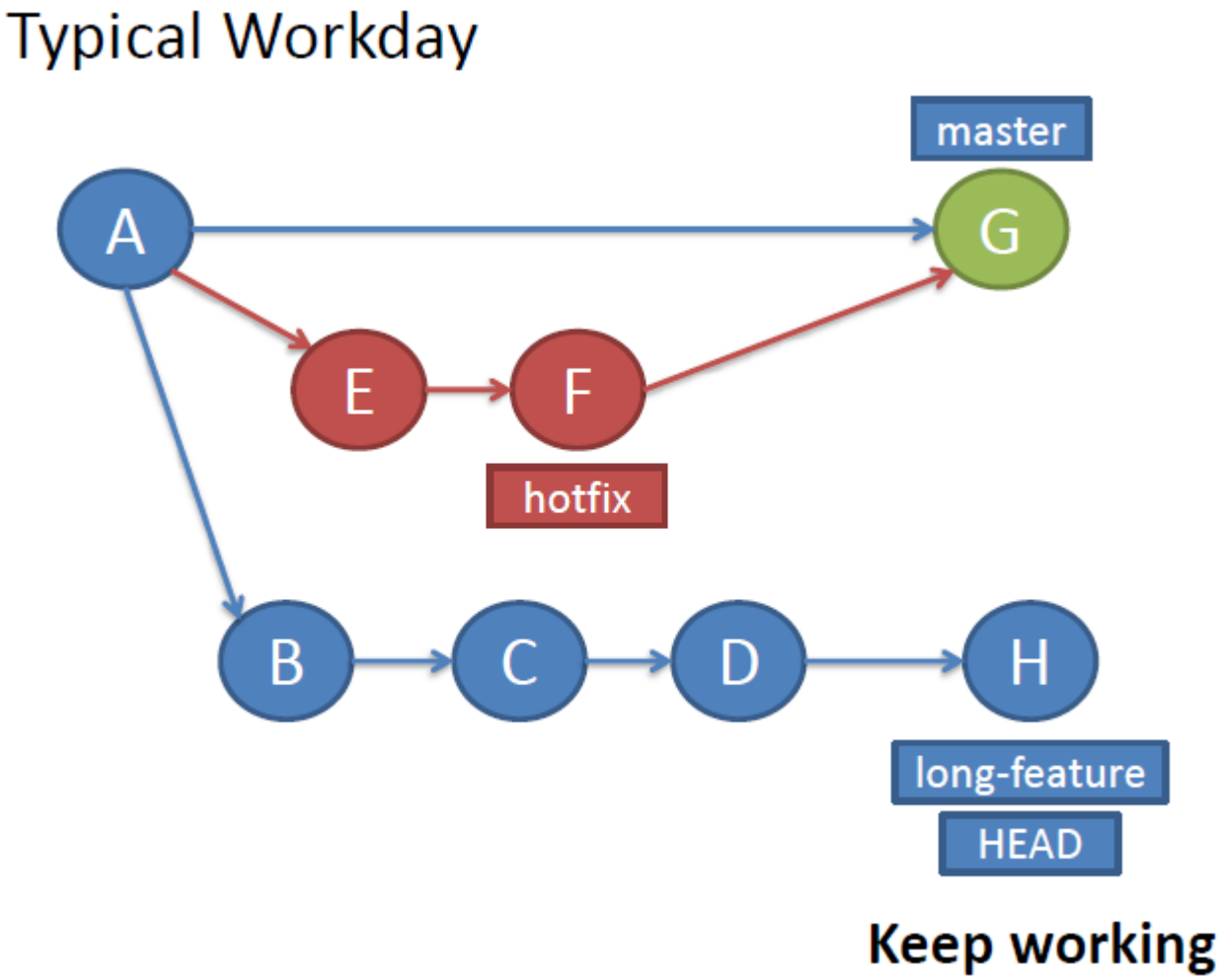
Using stash to improve your workflow



Using stash to improve your workflow

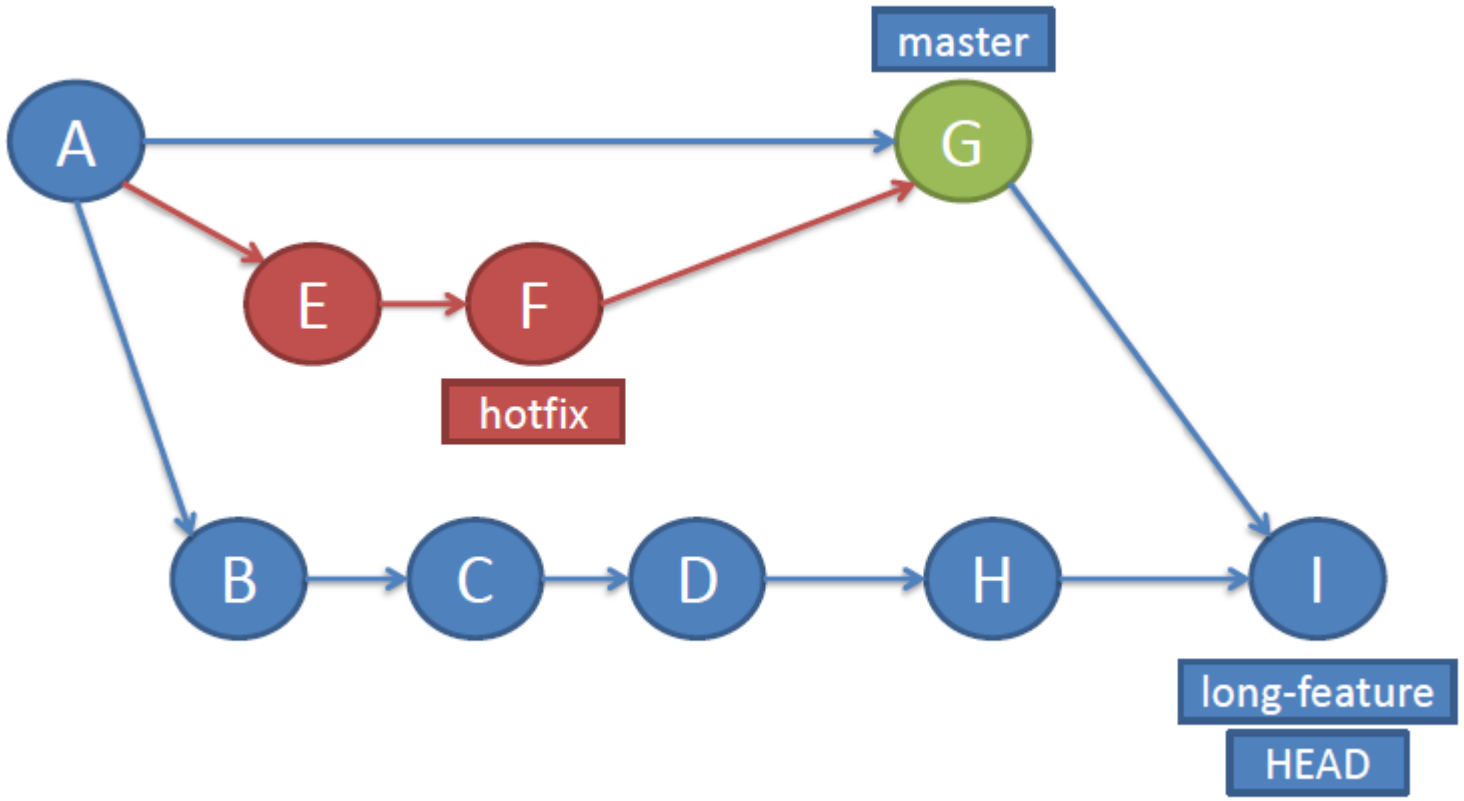


Using stash to improve your workflow



Using stash to improve your workflow

Typical Workday



Grab your fix: **git merge master**

Git reset: back to the past

- Git reset allows to:
  - Put committed data back to the working dir ➔ soft reset
  - Remove completely committed data ➔ hard reset
- Git reset is NOT revertible
- You want to reset ?
  - Do you really need it ?
  - Do you really want it ?
  - What do you want to achieve ?
  - … and then think again …

Getting remote Git repository updates

1. Fetching remote changes with `git fetch`

```
33degree (master) $ git fetch origin
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From ssh://gitent-scm.com/git/gitentdevelopment/33degree
   e3e5bd4..3d94a73  master     -> origin/master
33degree (master) $
```

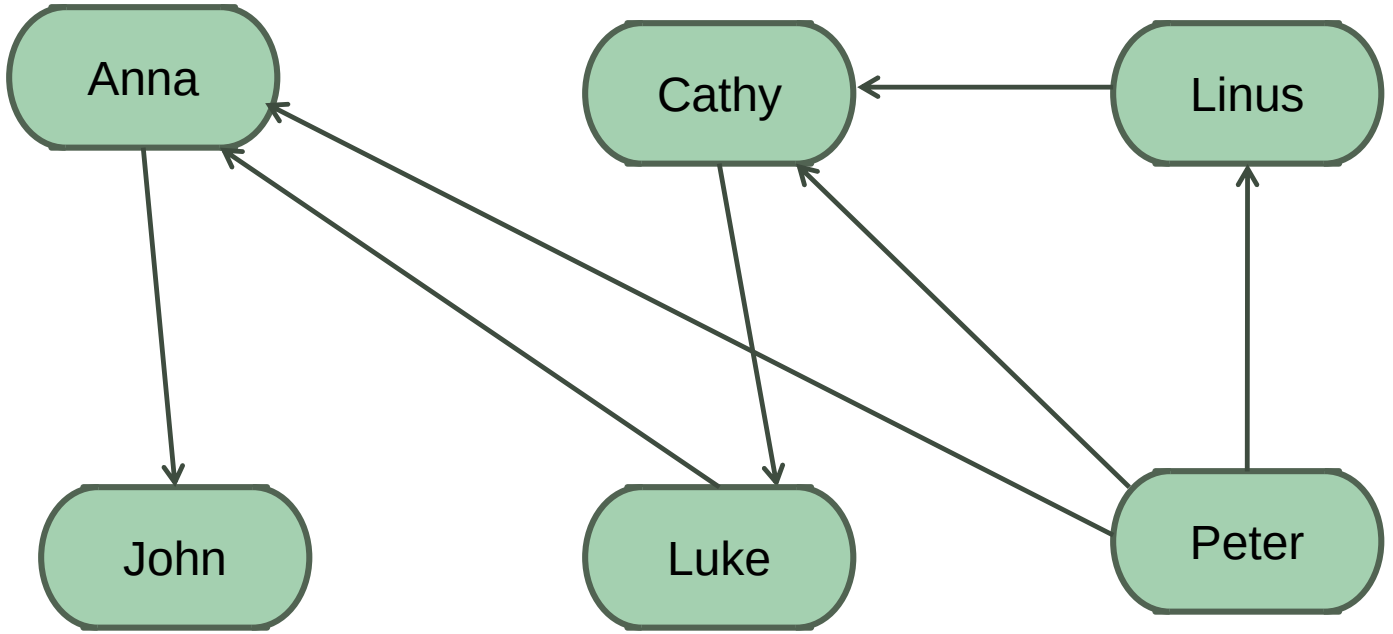
2. Merge (or rebase) changes

```
33degree (master) $ git merge origin/master
Updating e3e5bd4..3d94a73
Fast-forward
 INSTALL |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

= Pull

Git workflows

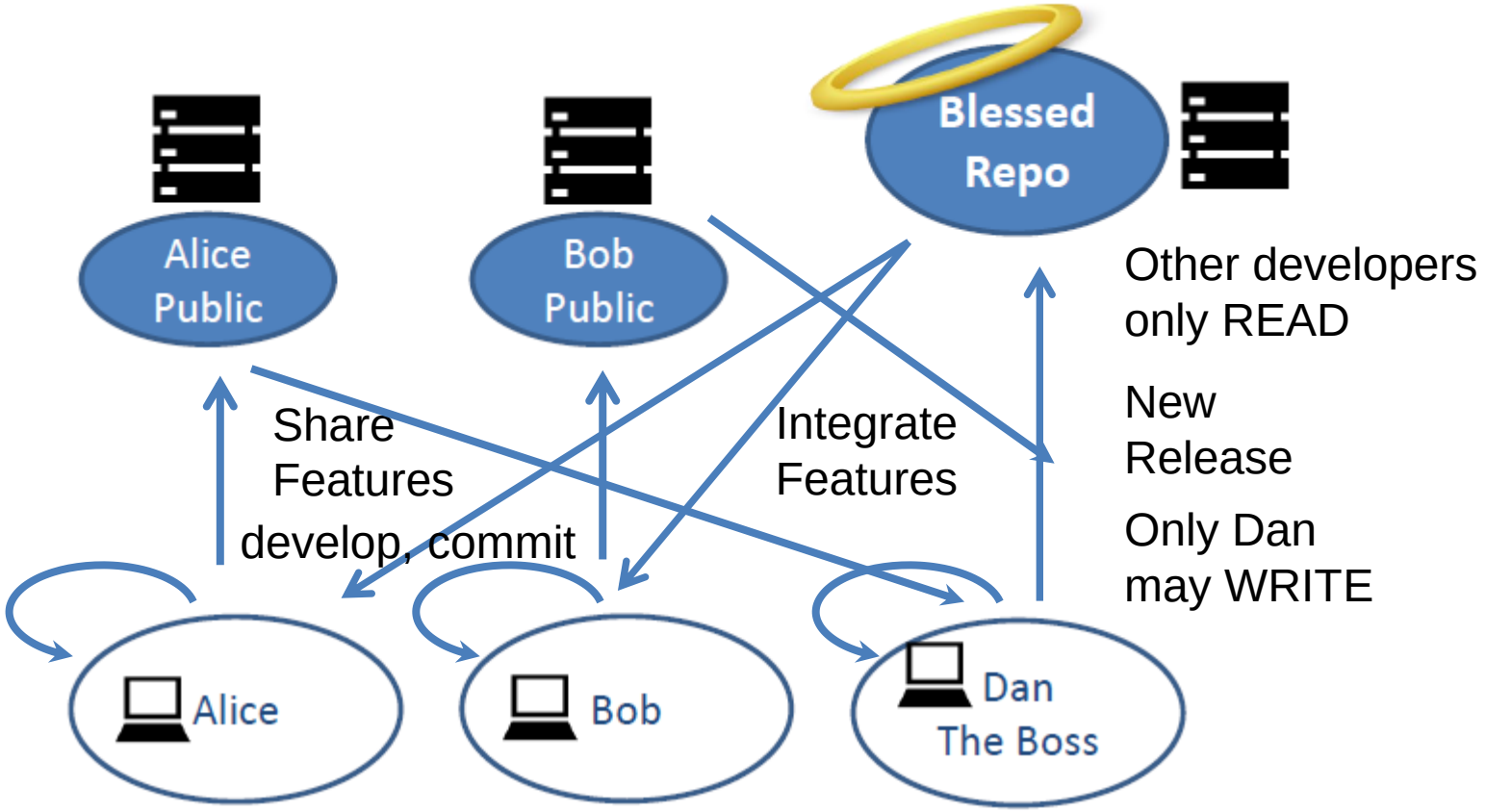
Everybody fetch/pull from each other



Nobody pushes: everybody fetch or pull

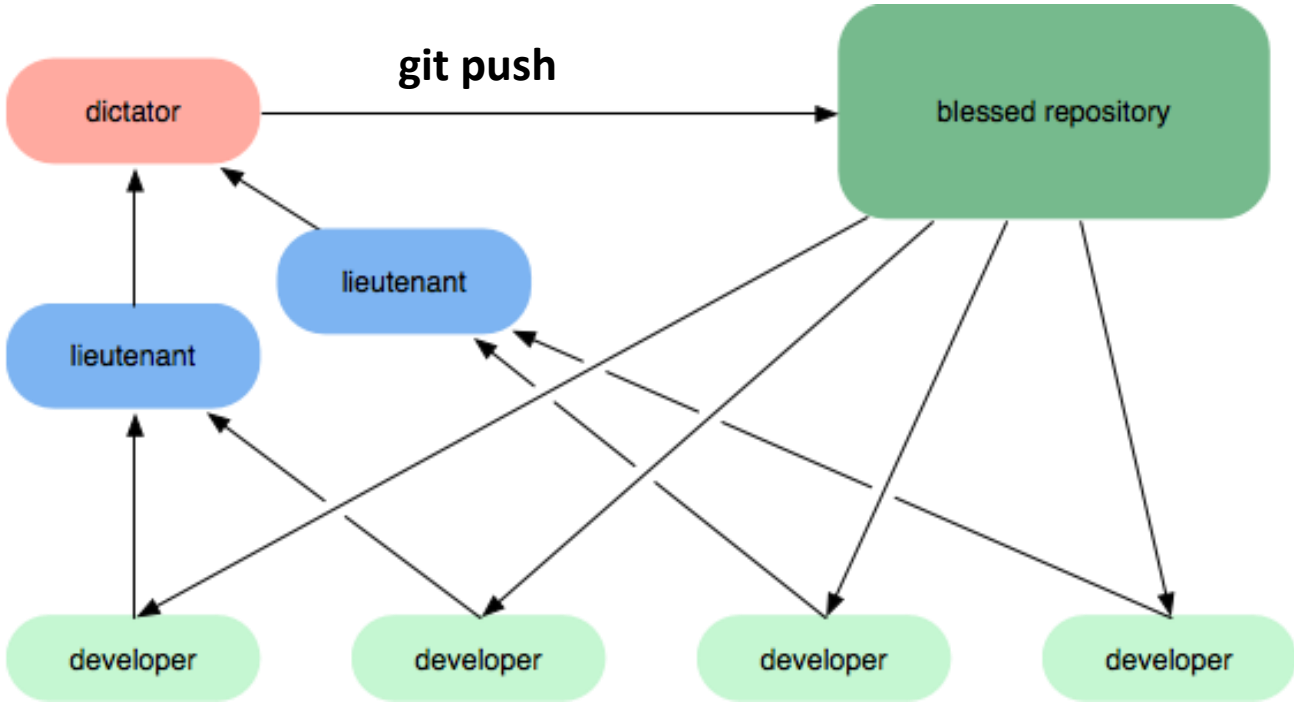
Every Git repository has the same importance

« Integration-Manager » (Dan)



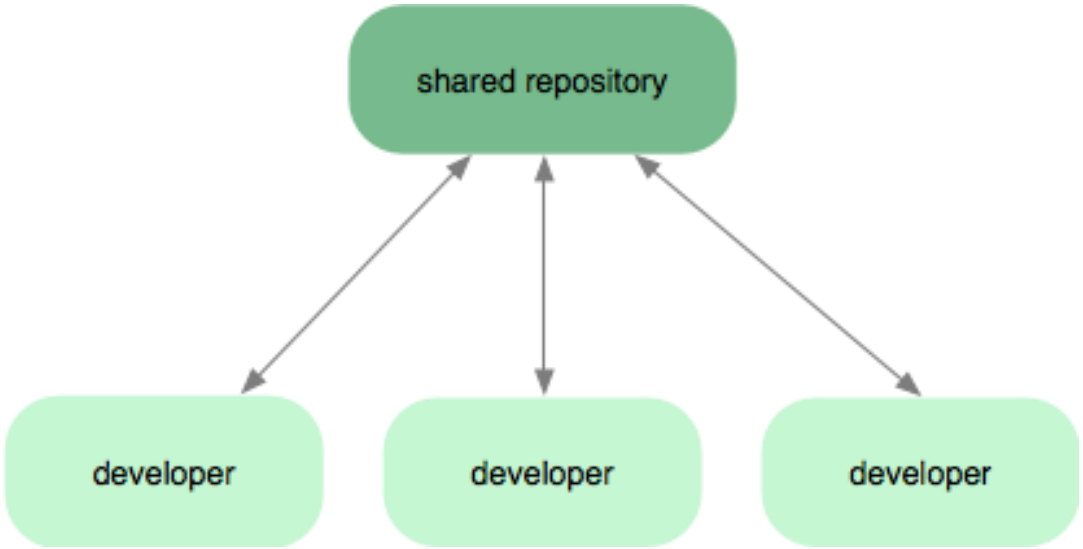
Github follows this approach, each public clone of the blessed repo is a fork in github terminology

Developers fetch/pull, dictator pushes



Developers pull, Lieutenants integrate
Dictator get integration branches together: he is the only one that PUSH to Git

Subversion-Style, centralised Workflow



Central repository dies ➔ elections of new repository

Back to centralisation ? How so?

- Git Democracy vs. SVN
  - Horizontal collaboration between developers (P2P)
  - Continuous branching / merging
  - “promotions” of changes through voting (Gerrit model)
  - Control over integration / release
- Does it seems like Git is mature for the Enterprise ?

Centralised Workflow – depicted in recap

- Git is a distributed version control system but you can use it like a centralized version control system (i.e. subversion).
- With the Centralized Workflow only the master branch is needed.
- Step-by-step:

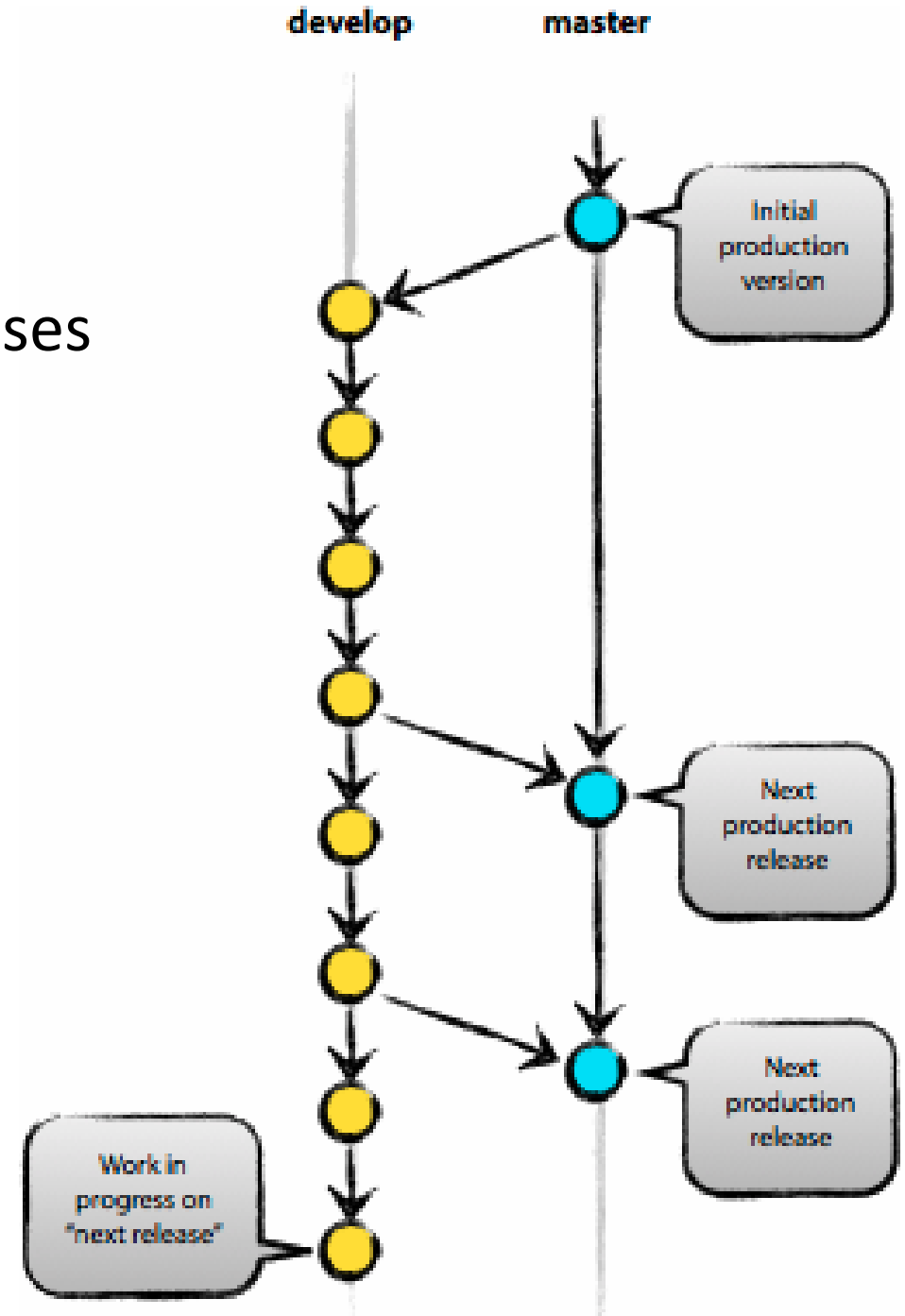
1. Create a centralized repository on github or other git hosting server.
2. Each developers clones the centralized repository.
3. Developers work locally committing changes ever so often.
4. The first developer, say John, does a git push origin master to merge his changes with the central repository. There are no conflicts because it is a simple fast-forward merge.
5. A second developer, say Mary, attempts a git push origin master, but she gets an error message because her local copy is behind the remote copy. She must do a git pull origin master (or git pull --rebase origin master) to merge John's changes before she can push her changes. This may require manual merge resolution. After merging, Mary can retry the git push origin master.

Feature Branch Workflow (extension of the centralised workflow)

- The core idea behind the Feature Branch Workflow is that all feature development should take place in a dedicated branch instead of the master branch.
- With feature branches you can also use pull requests to initiate discussions around a branch before it gets integrated.
- Developers create a new branch every time they start work on a new feature.
- Feature branches can (and should) be pushed to the central repository. Provides a mechanism for sharing work and also serves as a backup for local commits.

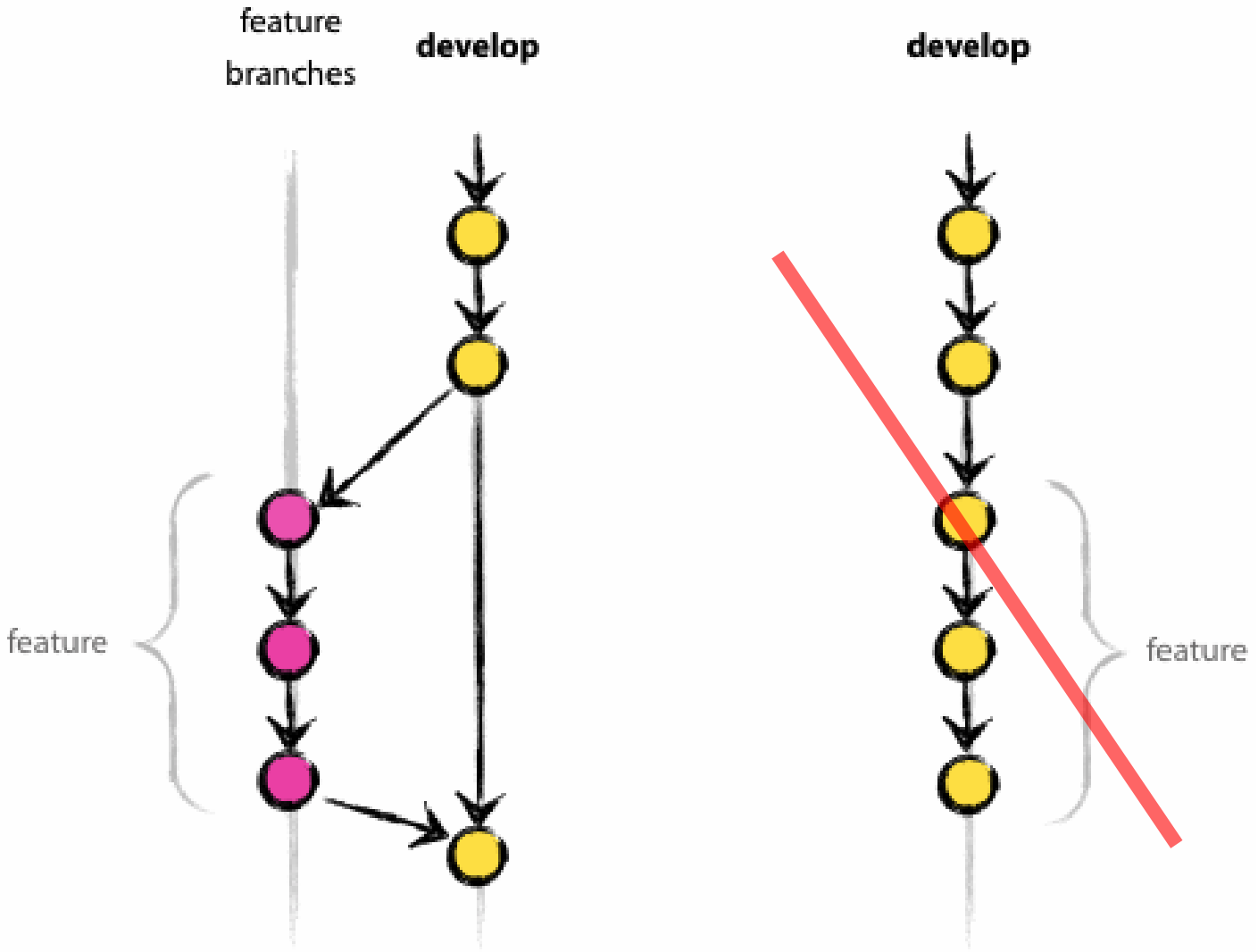
Main branches

- *master* used for production releases
- *develop* used for integration and regression testing



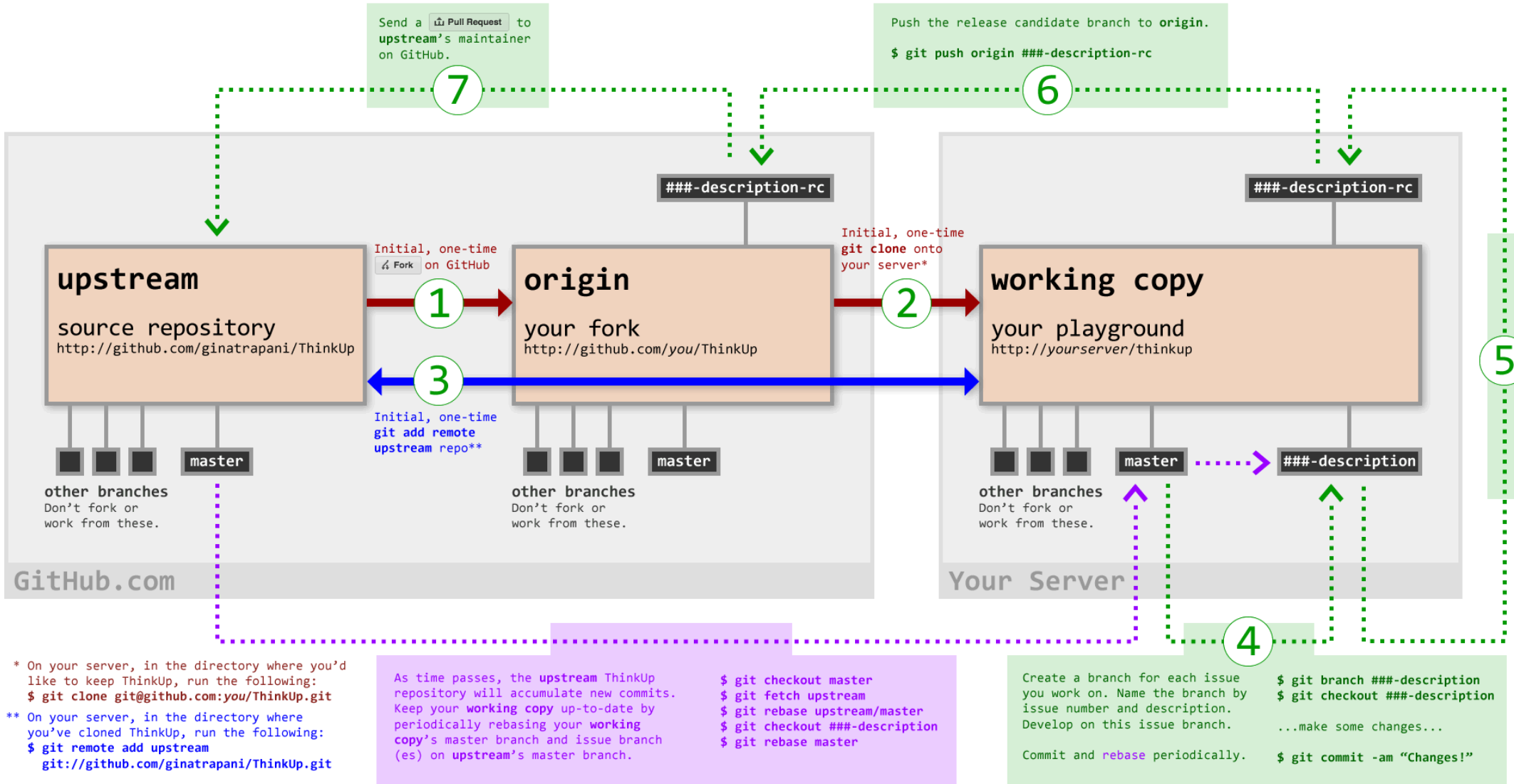
http://nvie.com/git-model

Feature branches



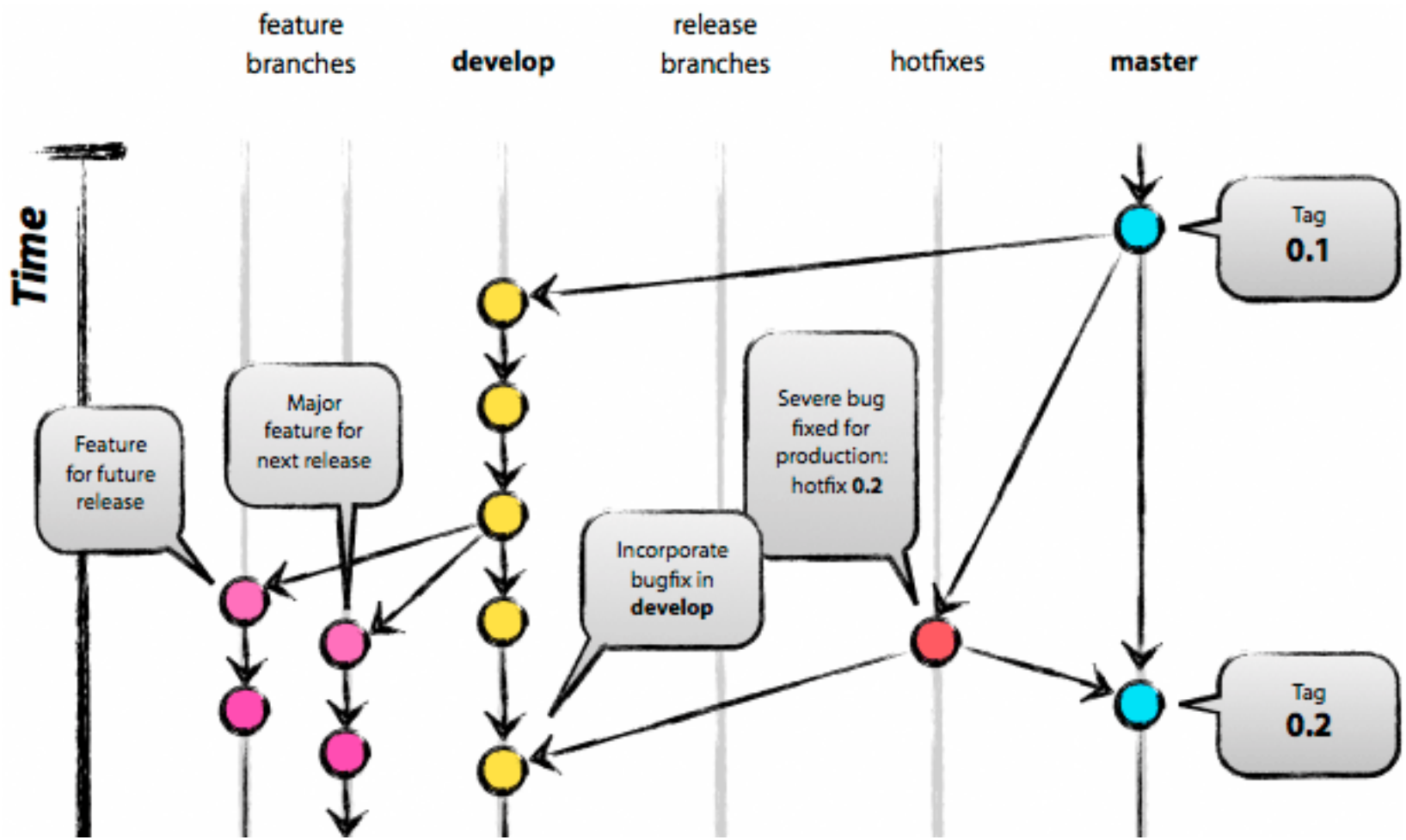
http://nvie.com/git-model

ThinkUp Contributor Workflow



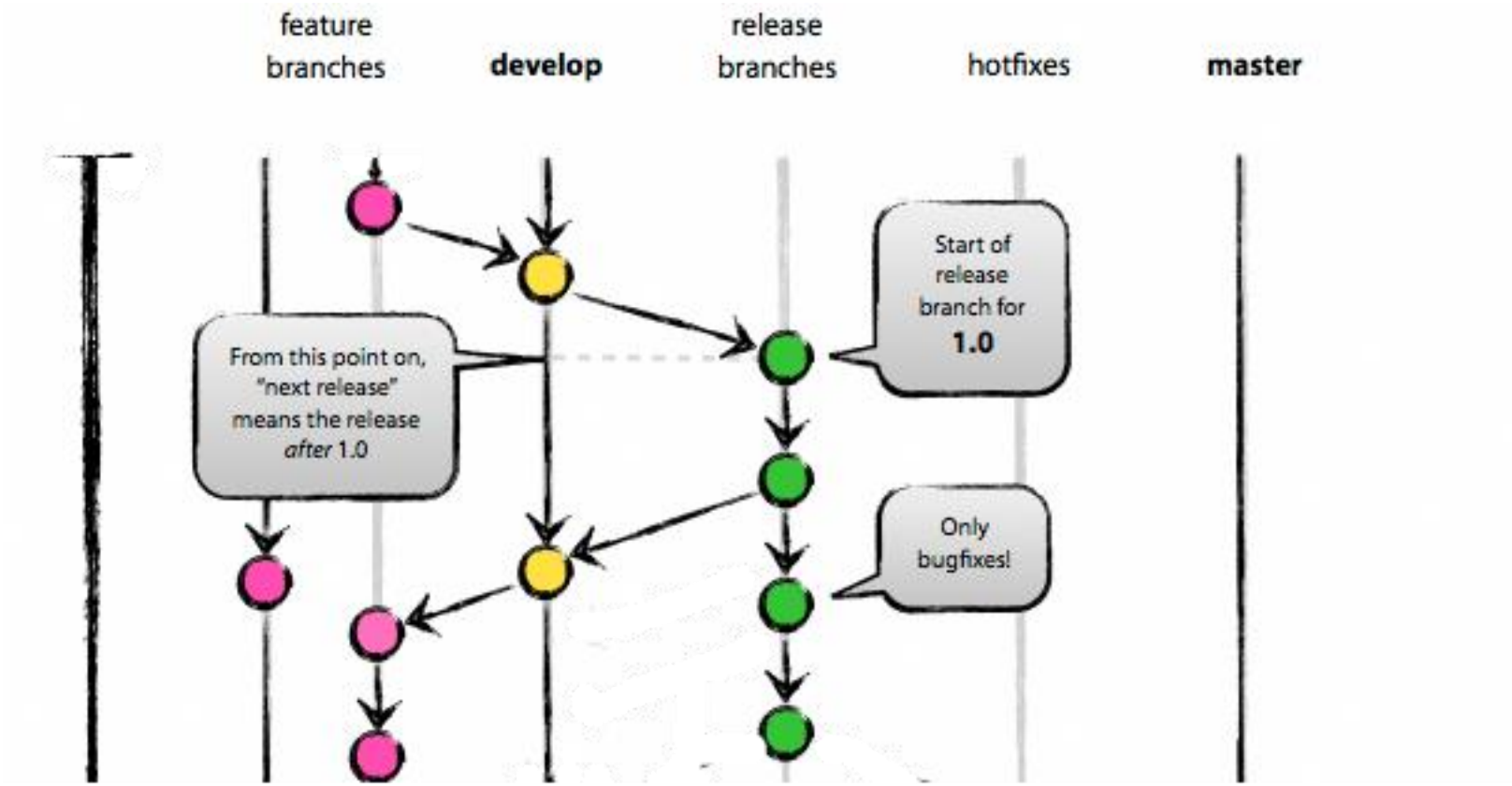
Your fork could be on other services as well, not necessarily on github

The big picture

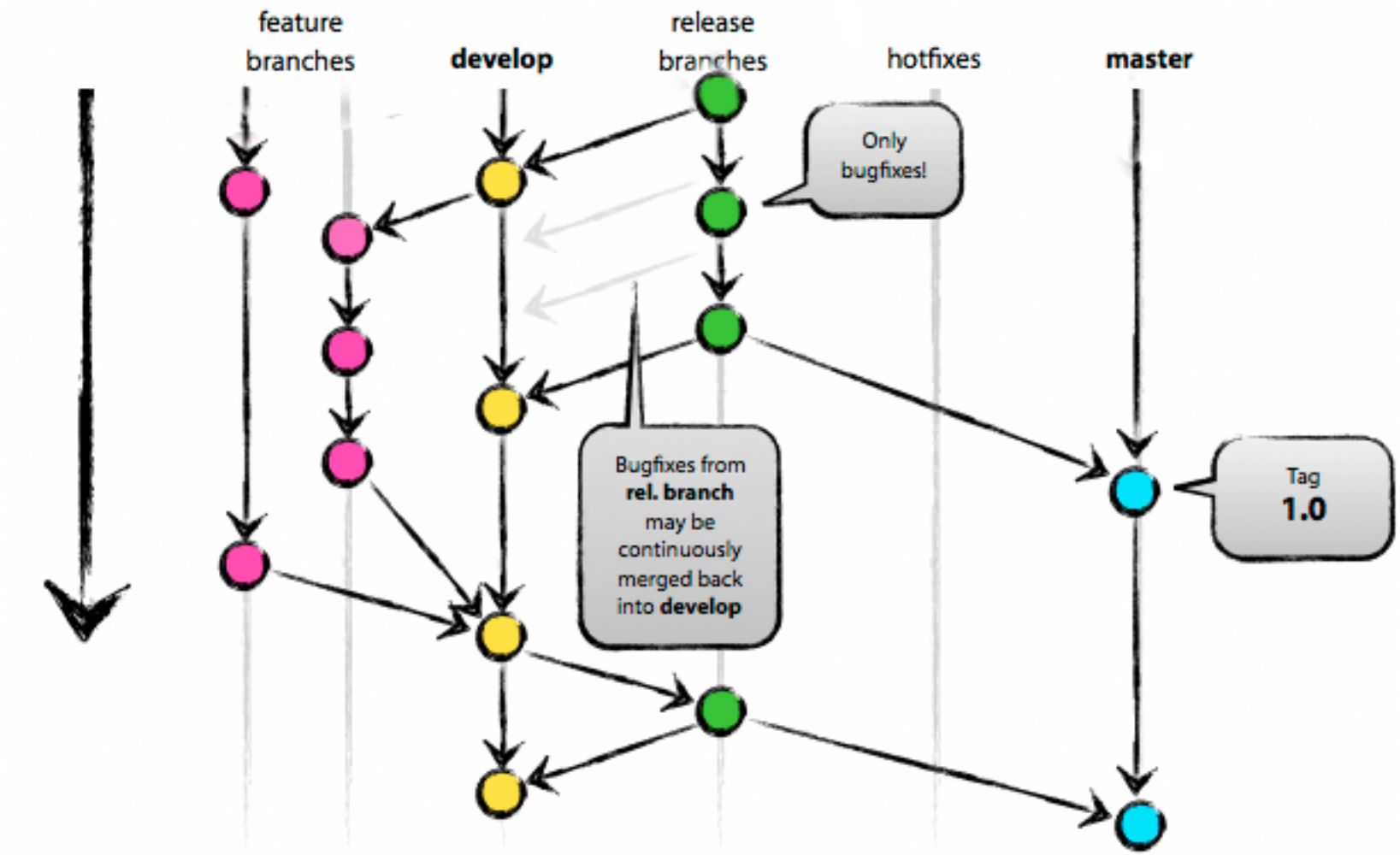


Original blog by Vincent Driessen

http://nvie.com/git-model



http://nvie.com/git-model



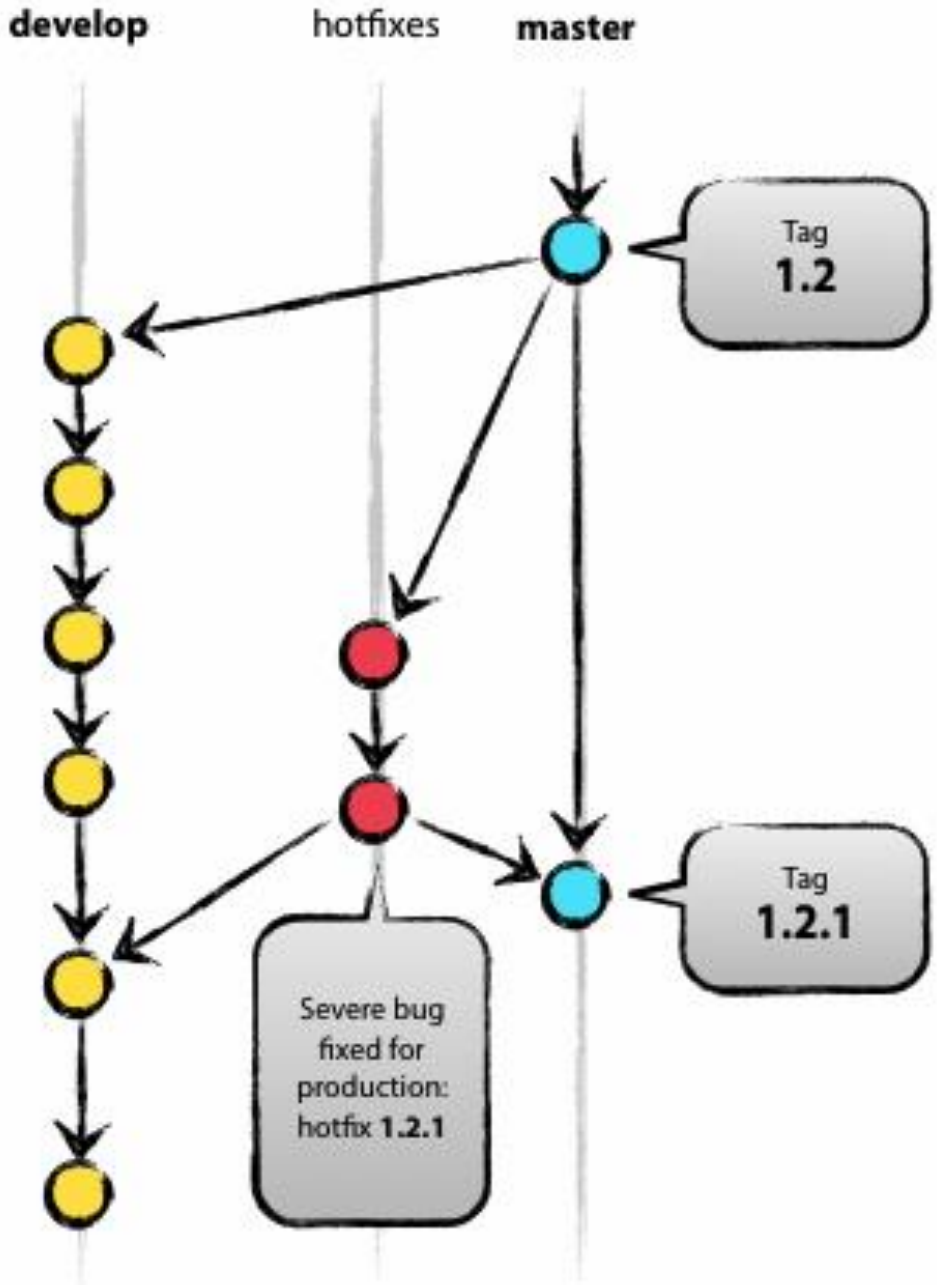
http://nvie.com/git-model

Release branches

- Branched from *develop* when it almost reflects the desired new release
- Bug fixes for the release are applied here
- No new features may be added to this branch
- New feature development can continue in parallel on the develop branch

Hotfix Branches

Are like release branches, but for bug fixes

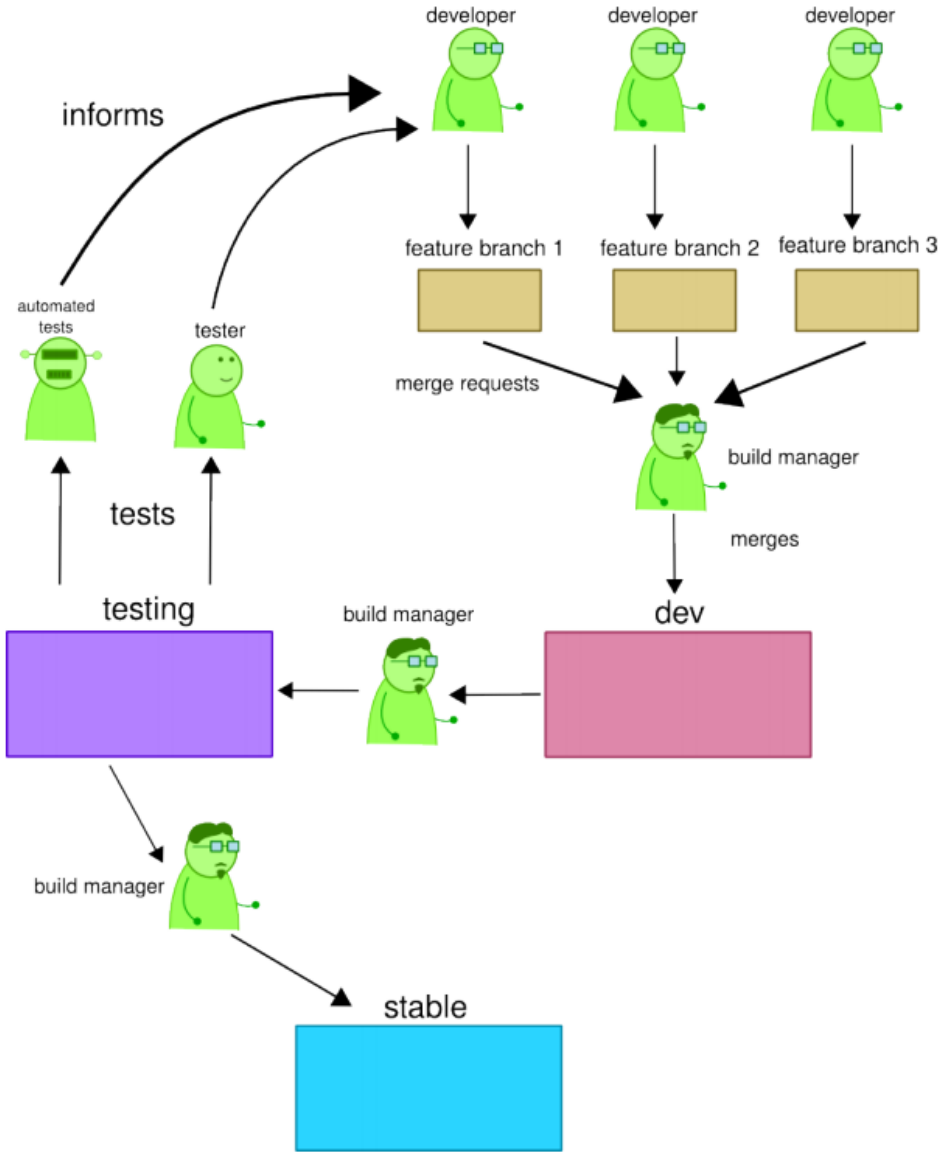


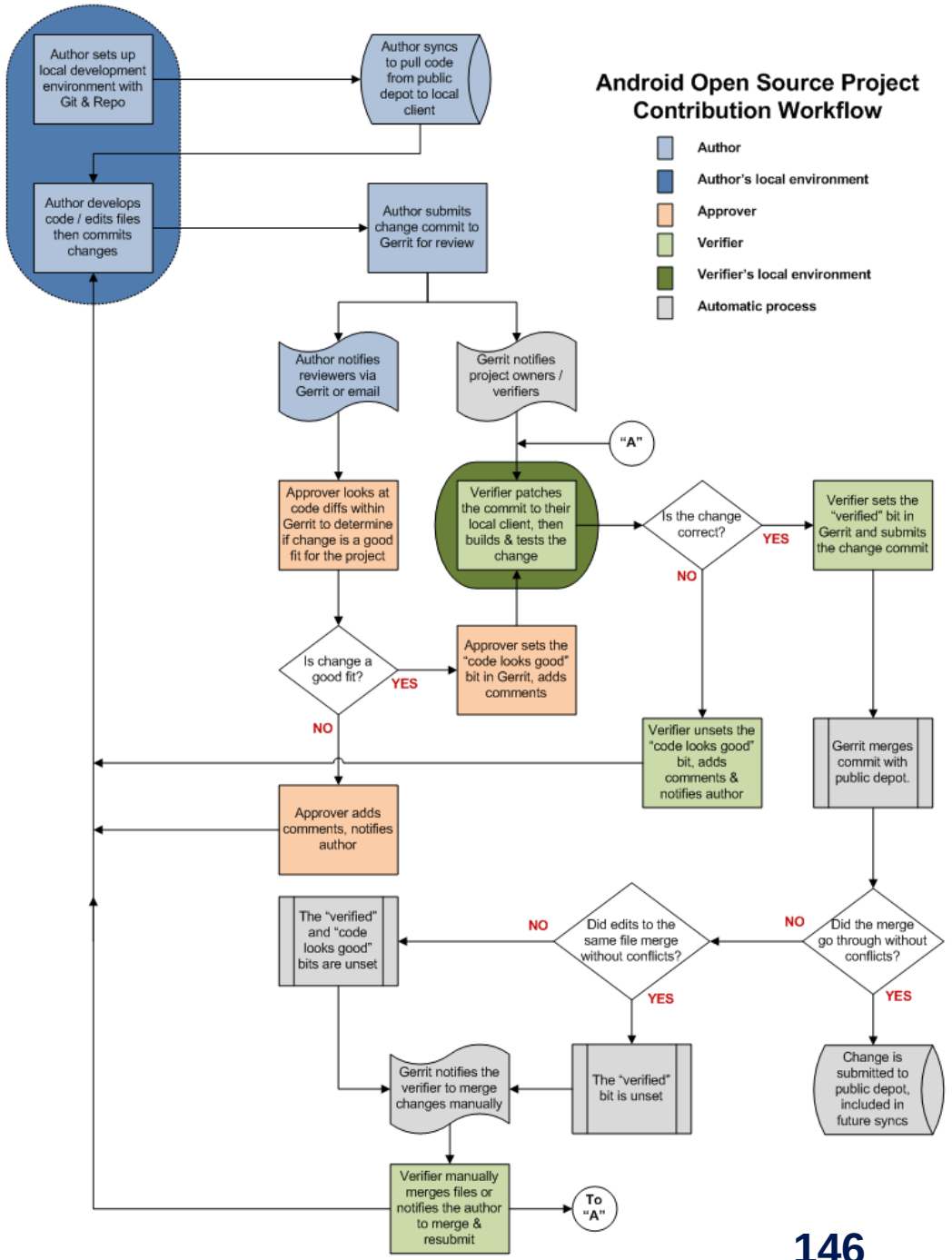
http://nvie.com/git-model

Support Branches

- If master has moved on to a new and a hotfix must be applied to a older version (e.g 1.x):
  - create a support-1.x branch (if none exists) based on the newest 1.x tag in master
  - create a branch (e.g. hotfix-1.1.1), based on support-1.x
  - fix the bug and merge hotfix-1.1.1 back into support-1.x
  - Do this for other older major releases as necessary
- The support branch effectively becomes a master branch for a past version.

Case studies (FreeNest, Android)





http://source.android.com/source/life-of-a-patch.html

Things to remember

Golden Rules of Git

- Always write a detailed commit message! (Anyone should know exactly what has changed in functionality just by viewing the message)
- Always specify what files you are going to work with during a coding session.
- Make sure you only add the files you want to add, especially if you are not using a git ignore file.
- It is ok to compromise your local branch, but never the remote!
- Add/commit/push your code EARLY and OFTEN!!!
  - You really, really, really don’t want to deal with merge conflicts
  - Keep your repository up-to-date all the time

git gotchas

- Do not forget to add/commit/push files or your group members will be very unhappy
- Keep in the repository *exactly* (and *only*) what you need to build the application!
  - Yes: foo.java pom.xml
  - No: foo.class, foo.jar, foo.java~, foo.java.bak
  - You don't want versions of .class files etc.:
    - Replaceable things have no value
    - They change a lot when .java files change a little
    - Developers on other machines can't use them
- A simple .gitignore file can be used to tell git which sorts of files should not be tracked (*.class, *~, .DS_Store (OS X))
  - Goes in top-level repo directory; useful to push to Bitbucket and share amongst group members

.gitignore

- You can make sure that no unnecessary files can be added to your repository by using a .gitignore file
- it contains filenames and paths that cannot be added to the repository and will not be shown in git status when they change
- it should be placed in your projects root directory
- it can be tracked by git just like any other file

Binaries and git

- Git stores copy of every file
  - => Large binaries bloat repositories
  - => But fast reverts are in order
- Merging binaries can be troublesome
  - => But some user interfaces offer custom merging tools
  - Git has heuristics to determine binary files and prevents normal merge tools from being used
  - Conflicts must be resolved by copying the correct version by hand
- Managing binaries is a case-by-case-affair
  - Submodules can be handy with this
  - Some extra tools also exist
    - Git-annex

Blame it on others

- blame allows you to see who has modified a file on a line by line basis
- blame prints out the entire file accompanied by the name of the user who made changes to that specific file
- usage example:
  $ git blame <filename>

git vs. github/bitbucket

git != github/bitbucket

git is a version control system.

github.com, bitbucket.org are web sites where you can publish git repositories.

Typical workflow: use git locally to manage a set of files. Push or publish to github/bitbucket to backup your work and share it with others.

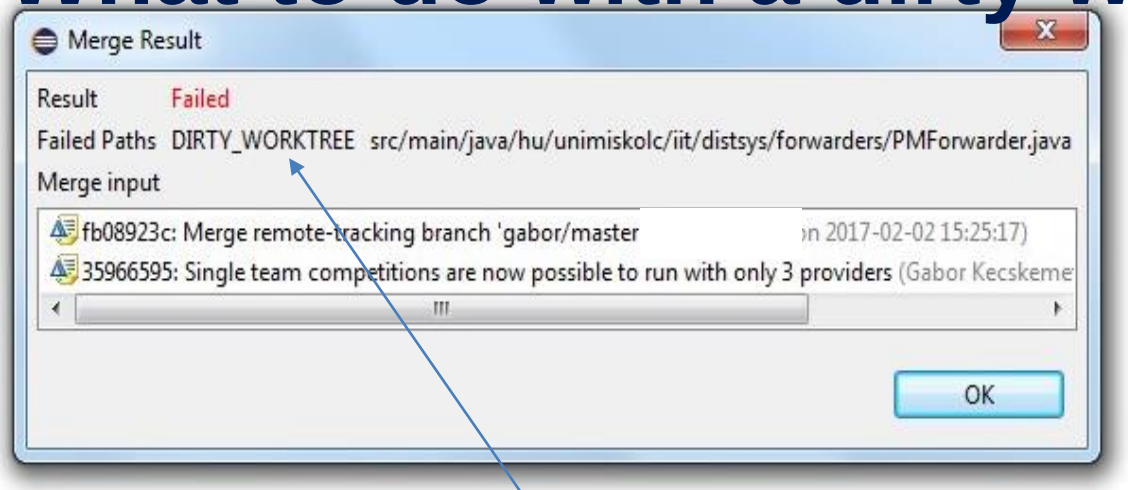
CSW and git/bitbucket

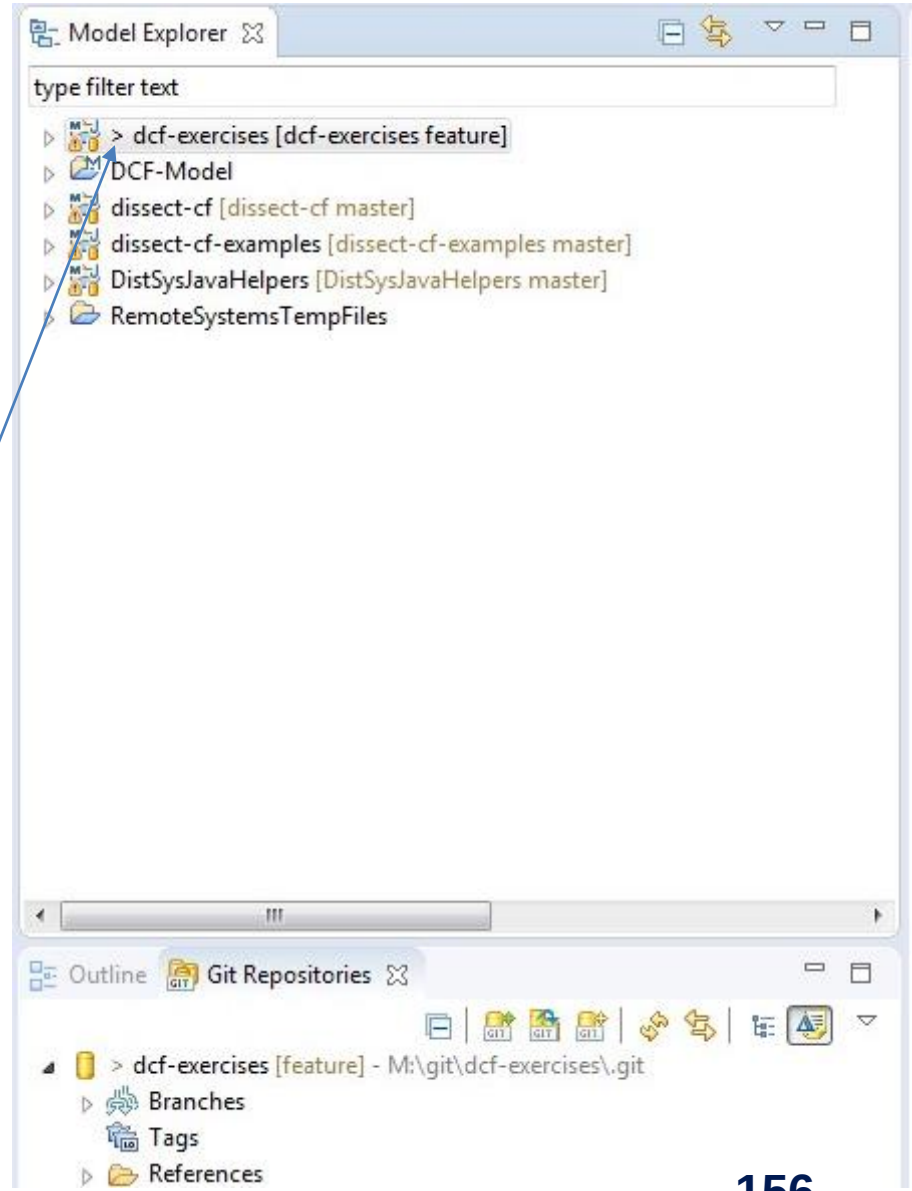
Covering some typical student issues from last year

When to use bitbucket and git

- Not sharing your bitbucket with Gabor is ok, not sharing with others in your group is bad
- Final submission of your coursework should have a compressed version of your git folder
  - equivalent to sharing the bitbucket at the last moment

What to do with a dirty working tree





- You will be unable to merge your repo
  - If you have uncommitted changes
- Fix:
  - Commit your work first before pulling from another repository

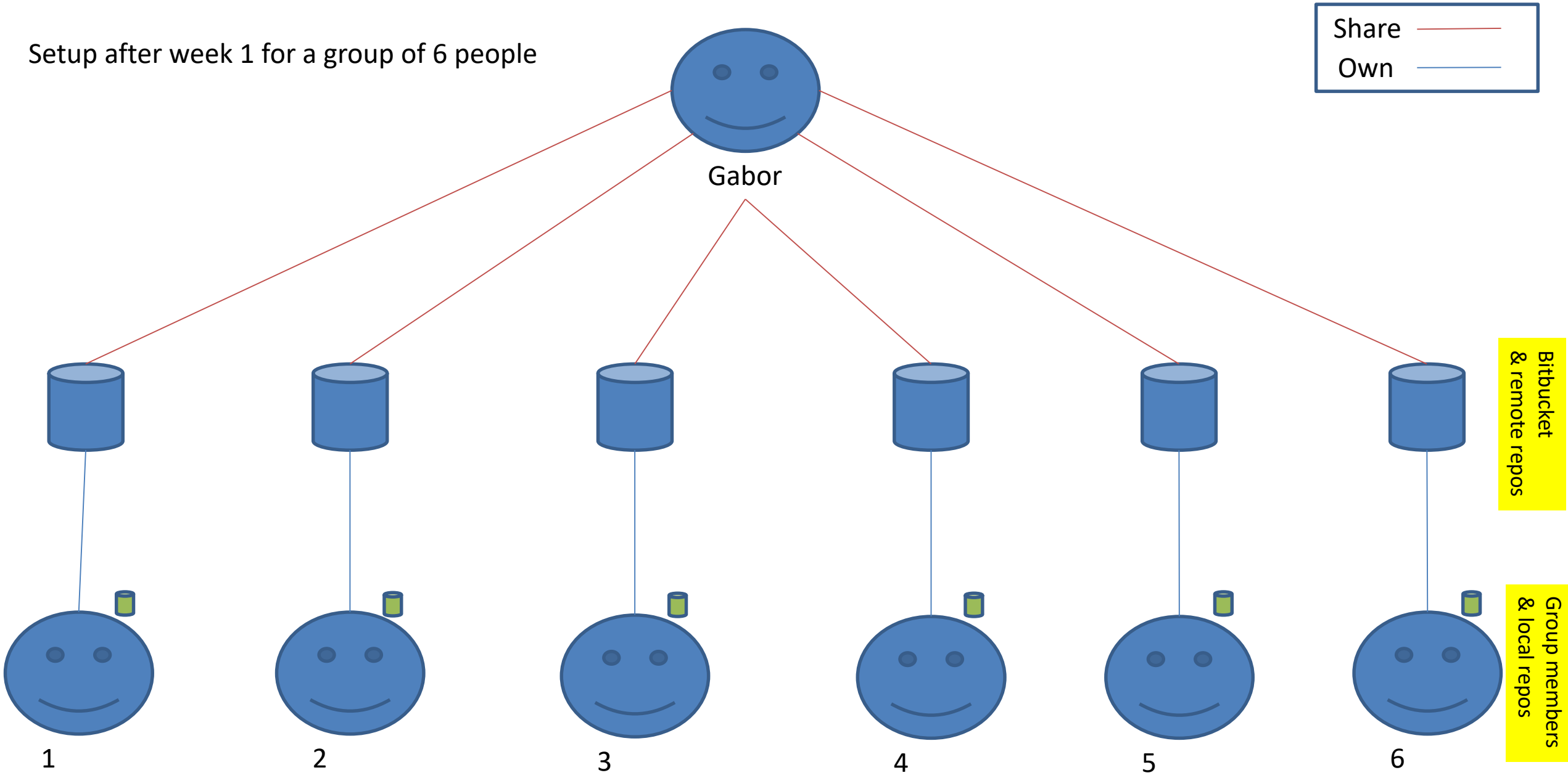
Sharing code in a group with more than 4 people

Assuming you have not registered with your university account

Situation

- To create a team on bitbucket, the maximum limit of adding other members to a team is 5
  - Note: this limitation is only a limitation if you missed one of these steps:
    - you have not set up your bitbucket account with your student email address
    - you have not upgraded your bitbucket account to a paid one
    - Your repository is not public (i.e., private as you were initially told to set it up in the first week)
- Sample problem:
  - My group has 6 members, I was wondering what alternative options you would recommend to merge everyone's project together.
- Solutions:
  - Do not follow a centralised git workflow.
  - Do a hierarchy – e.g. dictator and lieutenants workflow
  - Don’t forget to keep sharing everyone’s with me (ie., you have a max of 4 not a max of 5…)

Setup after week 1 for a group of 6 people

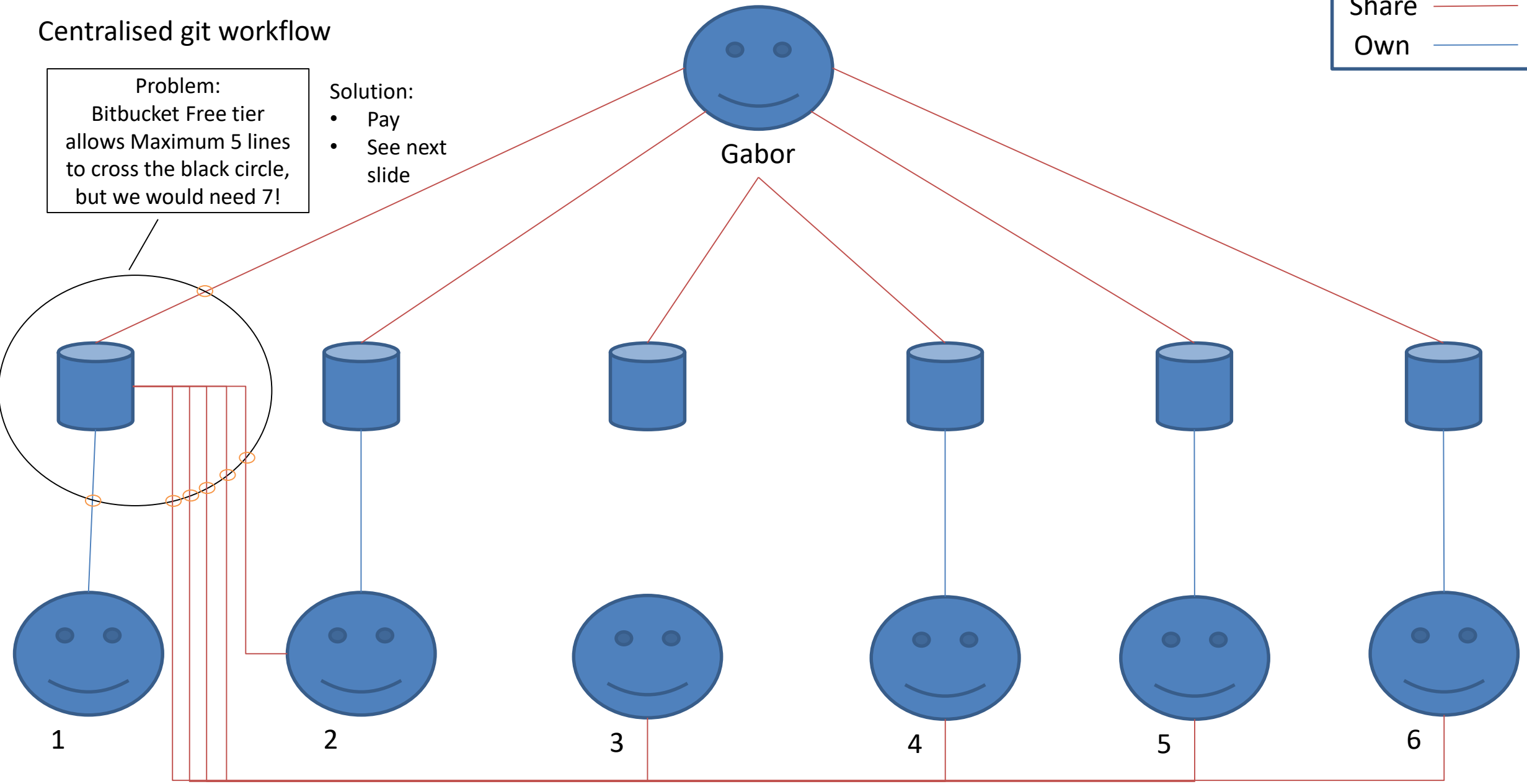


*Local repos are shown only on this figure, despite they are always there

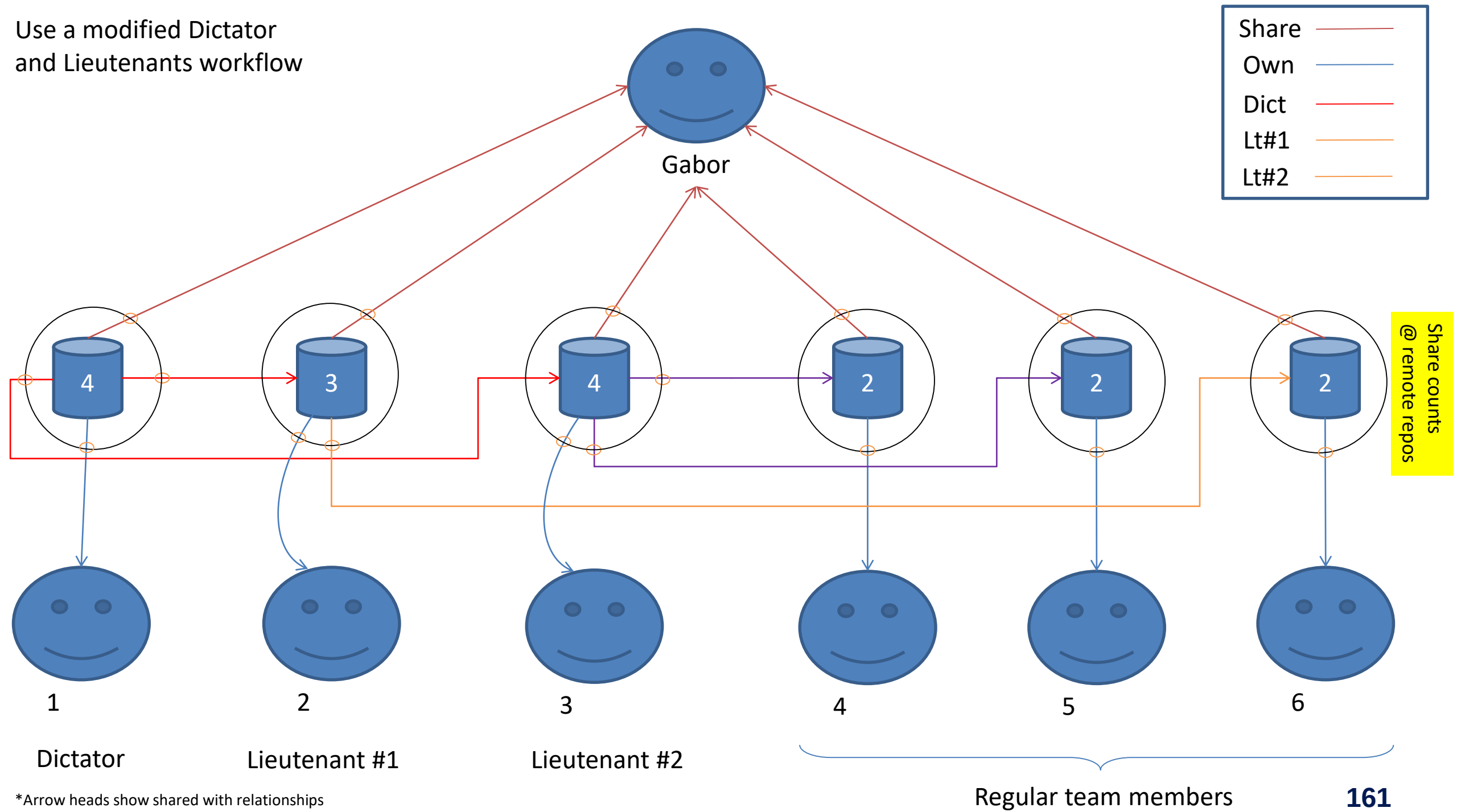
Centralised git workflow

Problem:
Bitbucket Free tier allows Maximum 5 lines to cross the black circle, but we would need 7!

Solution:
- Pay
- See next slide

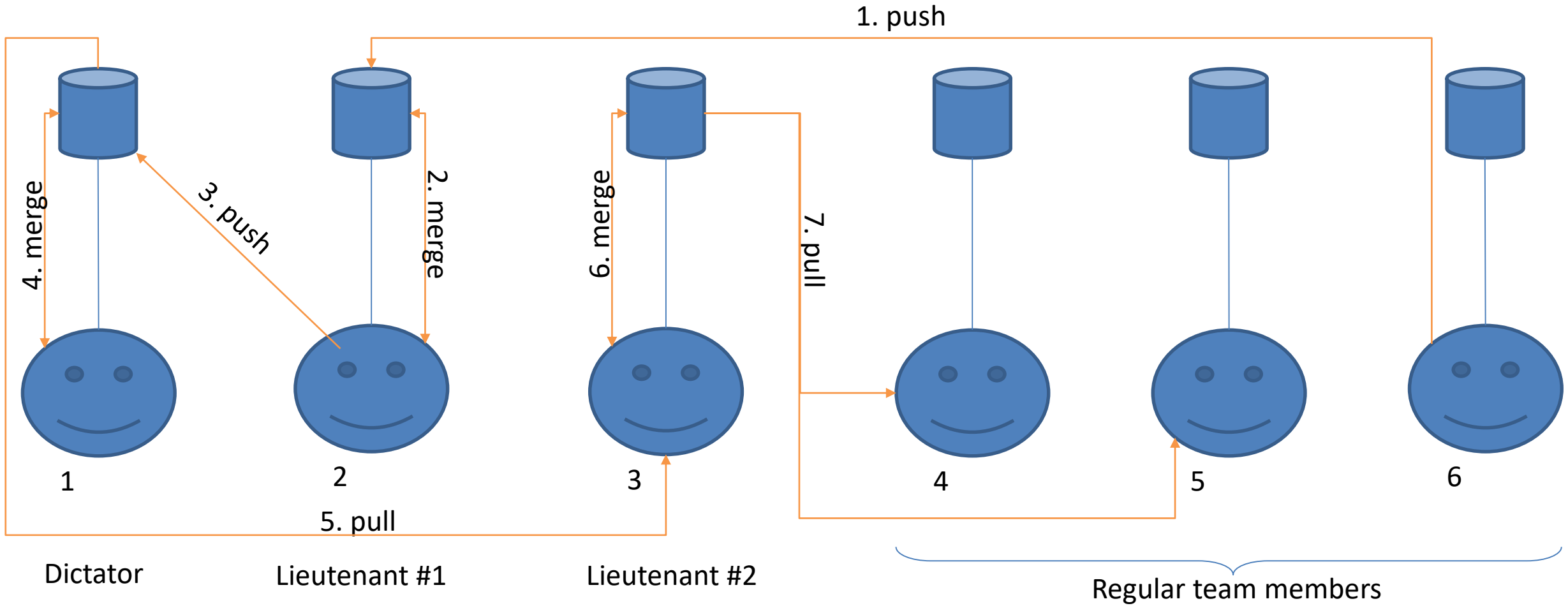


Use a modified Dictator and Lieutenants workflow



*Arrow heads show shared with relationships

Code propagation between regular team members



Summary

- We have seen why version control is useful during groupwork
- We investigated the various use cases and techniques a group can use git during collaborative software development

Next session

- We will discuss Maven, a tool for java software management