Version Control Systems
VCSs

About 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

- Even though software source code is often considered, in reality **any type of file** on a computer can be placed under version control

- If you are a graphic or web designer and want to keep every version of an image or layout (which you certainly would), it is very wise to use a Version Control System

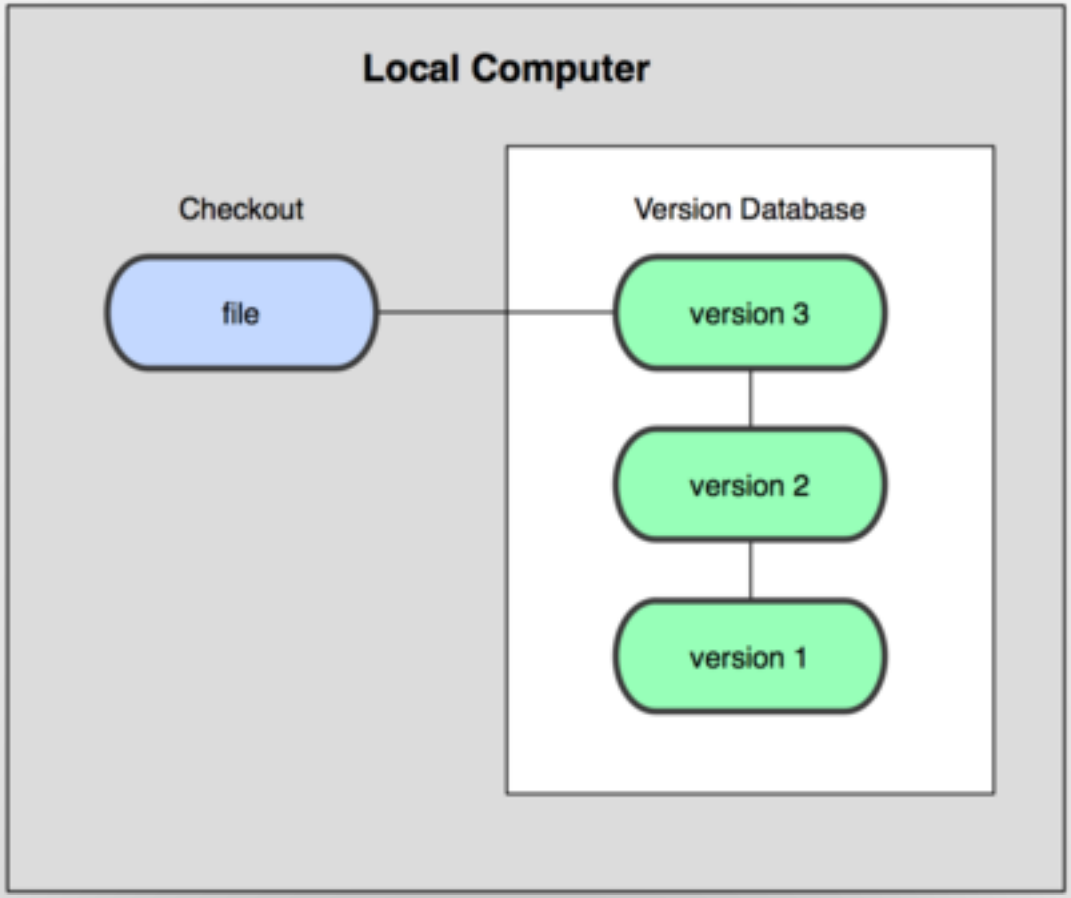
About Version Control

- A VCS allows you to:
  - revert files back to a previous state,
  - revert the entire project back to a previous state,
  - review changes made over time,
  - see who last modified something that might be causing a problem,
  - who introduced an issue and when,
  - …….
- Using a VCS also means that if you make a mistake or lose files, you can generally recover it easily.
- In addition, you get all this for very little overhead

Local Version Control Systems

- Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever)
- This approach is very common because it is so simple, but it is also incredibly error prone
- It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to
- To deal with this issue, programmers long ago developed **local VCSs** that had a **simple database** that kept all the changes to files under revision control

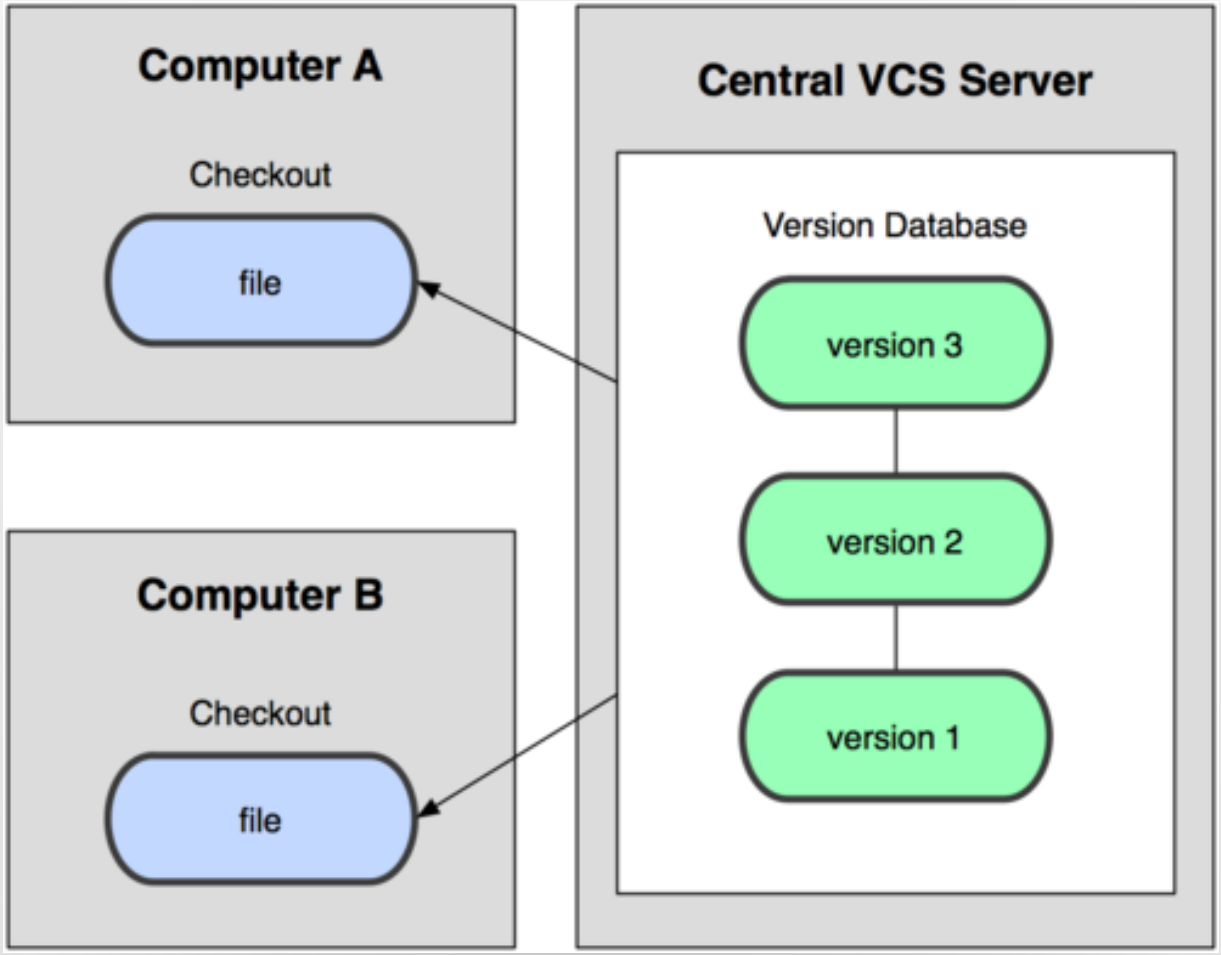
Local Version Control Systems



Centralized Version Control Systems

- The next major issue that people encounter is that they need to collaborate with developers on other systems
- To deal with this problem, **Centralized Version Control Systems (CVCSs)** were developed
- These systems, such as *CVS*, *Subversion*, and *Perforce*, have a single server that contains all the versioned files, and a number of clients that check out files from that central place
- For many years, this has been the standard for version control

Centralized Version Control Systems



Centralized Version Control Systems

- This setup offers many advantages, especially over local VCSs
- For example, everyone knows to a certain degree what everyone else on the project is doing
- Administrators have fine-grained control over who can do what;
- It’s far easier to administer a CVCS than it is to deal with local databases on every client.

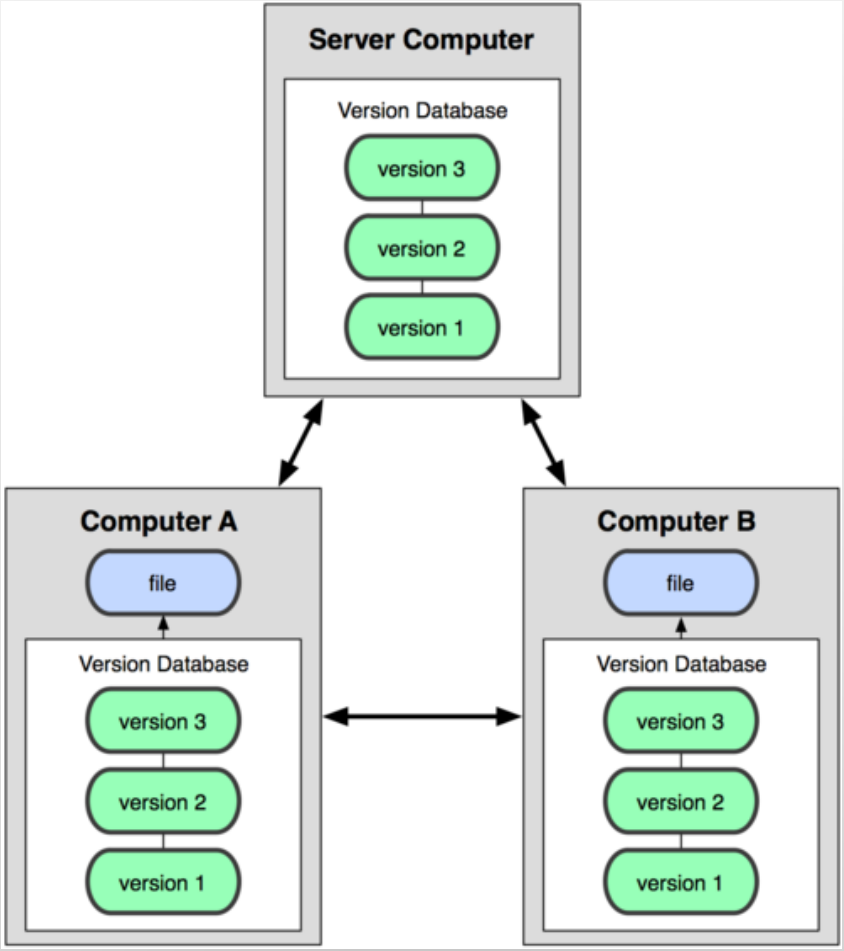
Centralized Version Control Systems (serious downside)

- The most obvious is the **single point of failure** that the centralized server represents
  - If that *server goes down* for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they're working on.
  - If the hard disk the central *database* is on *becomes corrupted*, and proper backups haven't been kept, you lose absolutely all the stuff
- Local VCS systems suffer from this same problem

Distributed Version Control Systems

- Distributed Version Control Systems (DVCSs) face to previously described issues
- In a DVCS (such as *Git*, *Mercurial*, *Bazaar* or *Darcs*), clients don't just check out the latest snapshot of the files: **they fully mirror the repository**
- Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it.
- Every checkout is really a full backup of all the data

Distributed Version Control Systems



GIT (http://www.git-scm.com)

git --everything-is-local

Search entire site...

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is easy to learn and has a tiny footprint with lightning fast performance. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like cheap local branching, convenient staging areas, and multiple workflows.

Learn Git in your browser for free with Try Git.

About
The advantages of Git compared to other source control systems.

Documentation
Command reference pages, Pro Git book content, videos and other material.

Downloads
GUI clients and binary releases for all major platforms.

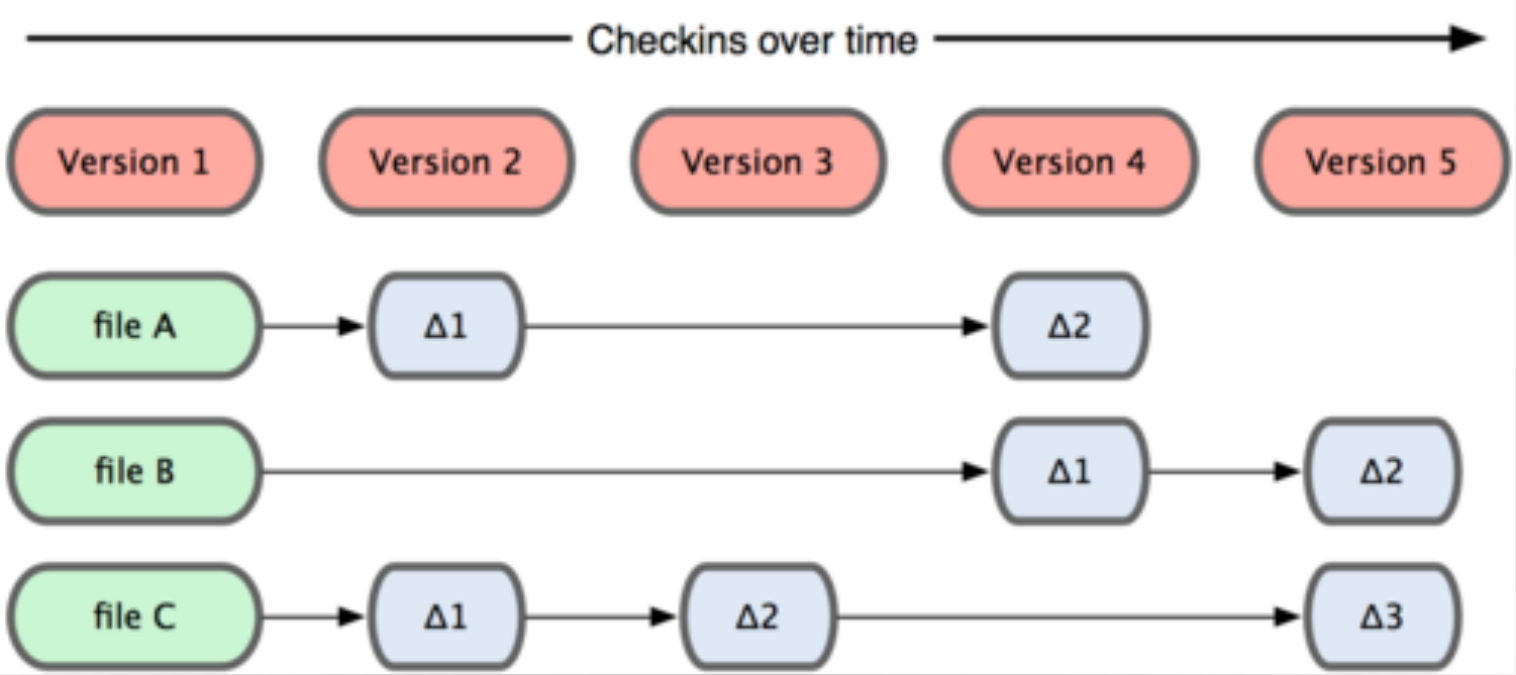
Community
Get involved! Mailing list, chat, development and more.

About GIT

- Git is released under the GPLv2 open source license. This means that you are free to inspect the source code at any time or contribute to the project yourself.
- Under Git's GPLv2 software license, you **may**:
- Use Git on open or proprietary projects for free, forever
- Download, inspect and modify the source code to Git
- Make proprietary changes to Git that you do not redistribute publicly
- Call Git binaries from your open or proprietary programs
- Publicly redistribute Git binaries with open or proprietary programs, given that they are unmodified or the modifications are public
- Under this license you **may not**:
- Make proprietary changes to Git and publicly redistribute it without sharing the changes
- Make and publicly distribute changes to Git under a different license
- Use source code from the Git repository in a project under a different license without permission

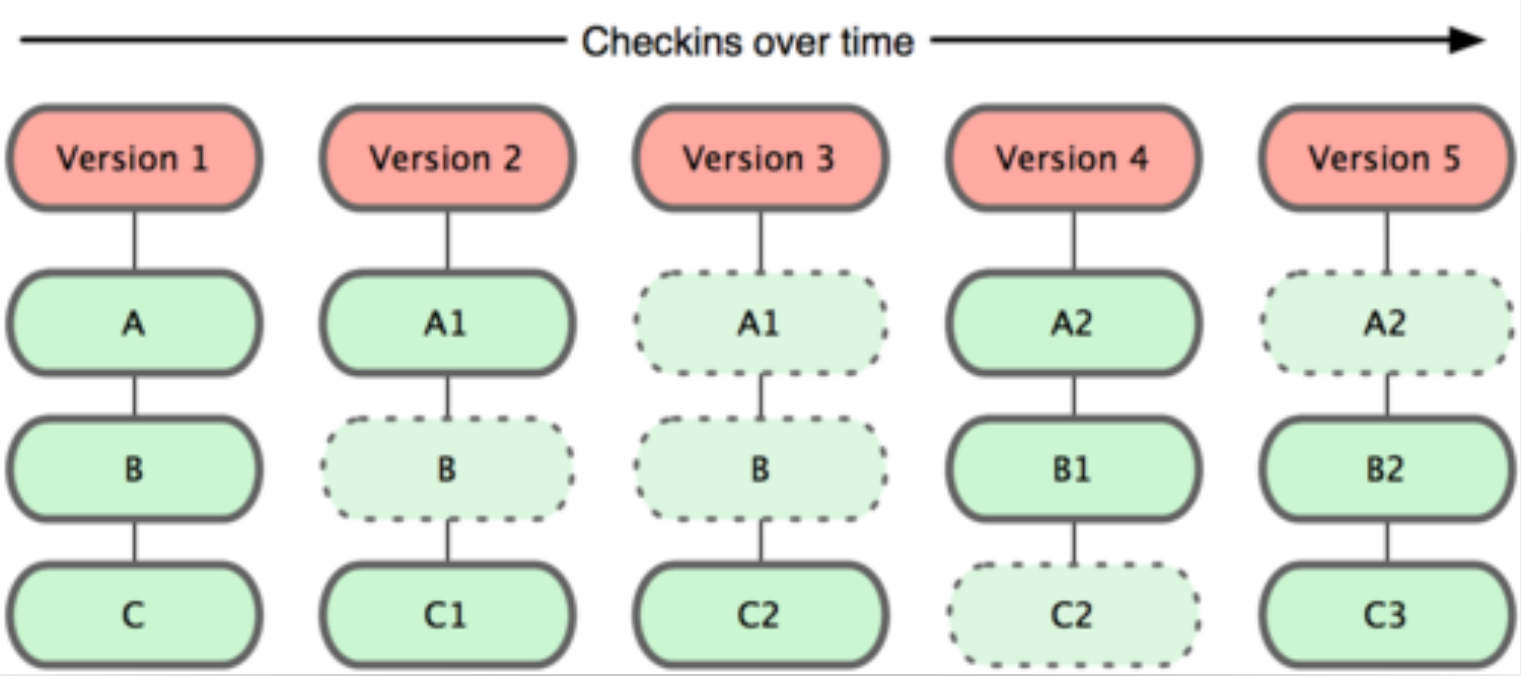
Git Basics: Snapshots, Not Differences

- The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data
- Conceptually, most other systems store information as a list of file-based changes



Git Basics: Snapshots, Not Differences

- Git doesn't think of or store its data this way.
- Instead, Git thinks of its data more like **a set of snapshots** of a mini filesystem. Git **doesn't store the file again—just a link to the previous identical file** it has already stored



Git Basics: Nearly Every Operation Is Local

- Most operations in Git **only need local files and resources** to operate — generally no information is needed from another computer on your network
- Because you have the entire history of the project right there on your local disk, **most operations seem almost instantaneous**.
- If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation, instead of having to either ask a remote server to do it or pull an older version of the file from the remote server to do it locally
- **This also means that there is very little you can't do if you're offline**

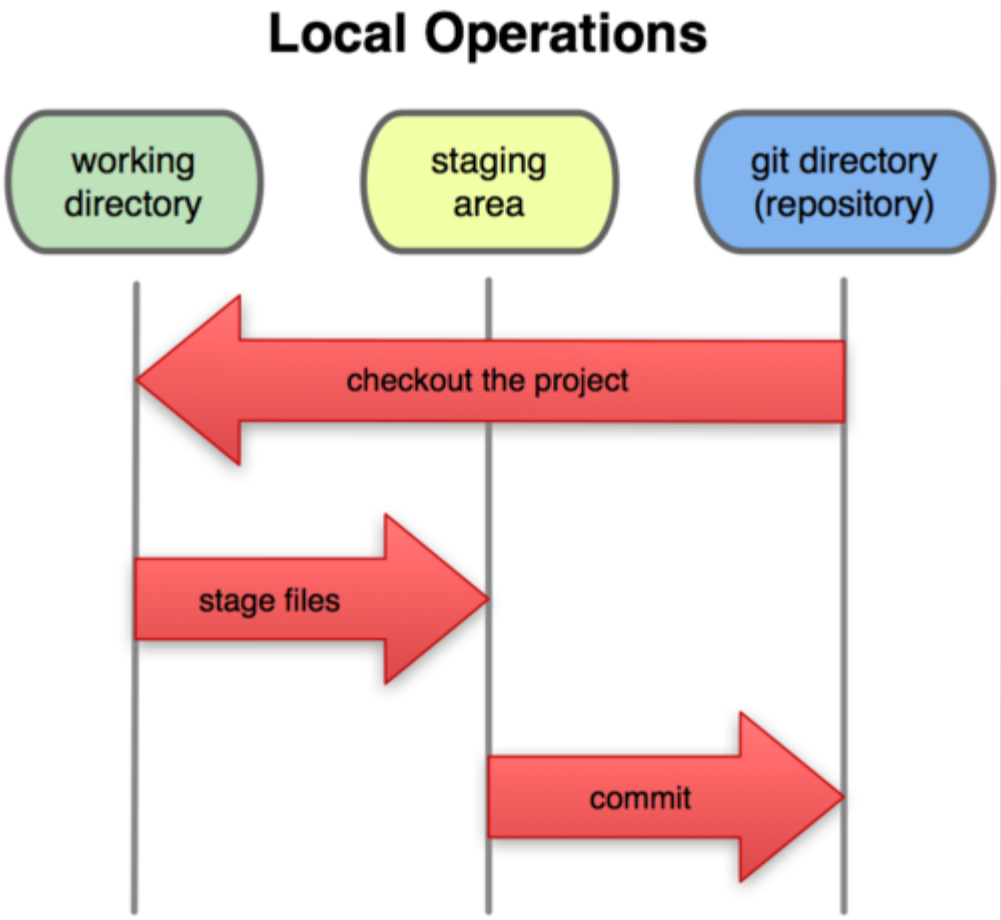
Git Basics: Git Has Integrity

- Everything in Git is check-summed before it is stored and is then referred to by that checksum
- This means **it's impossible to change the contents of any file or directory without Git knowing about it**
- This functionality is built into Git at the lowest levels and is integral to its philosophy
- **You can't lose information in transit or get file corruption without Git being able to detect it.**

Git Basics: The Three States

- Git has three main states that your files can reside in: **committed**, **modified**, and **staged**.
- Committed means that the data is safely stored in your local database
- Modified means that you have changed the file but have not committed it to your database yet
- Staged means that you have marked a modified file in its current version to go into your next commit snapshot
- This leads us to the three main sections of a Git project: the **Git directory**, the **working directory**, and the **staging area**

Git Basics: The Three States



The Basic Git workflow

- The basic Git workflow goes something like this:

1. You modify files in your working directory
2. You stage the files, adding snapshots of them to your staging area (the file containing staged items are sometimes referred to as the index)
3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

Some basic terms

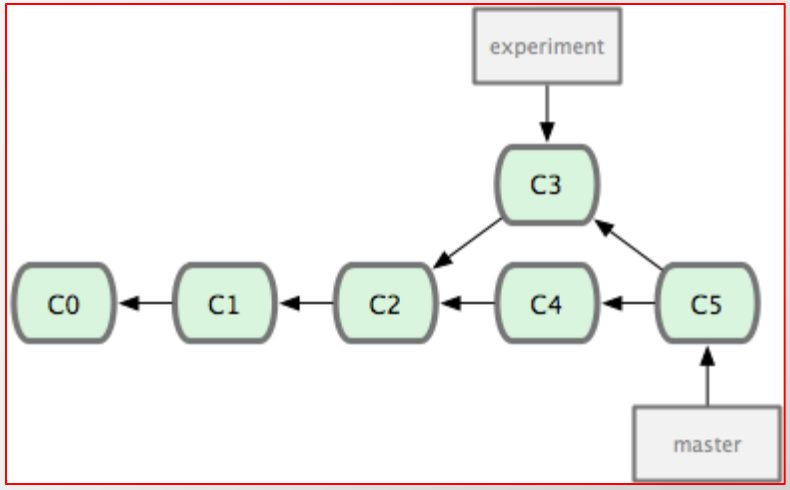
- **Clone**: If you want to get a copy of an existing Git repository — for example, a project you'd like to contribute to — the command you need is Git clone
- **Checkout**: The working directory is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify
- **Tracked vs. Untracked**: Each file in your working directory can be in one of two states tracked or untracked. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. Untracked files are everything else
- **Push and Pull**: Collaborating with others involves managing these remote repositories and pushing and pulling data to and from them when you need to share work.

Some basic terms

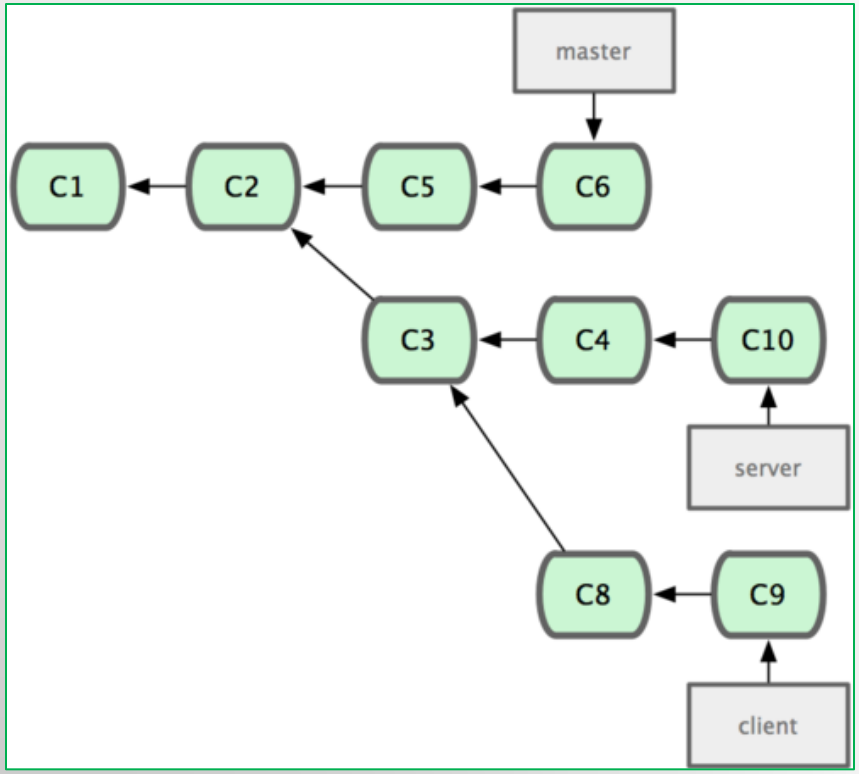
- **Branching**: Branching means you diverge from the main line of development and continue to do work without messing with that main line. The way Git branches is incredibly lightweight, making branching operations nearly instantaneous and switching back and forth between branches generally just as fast.

- **Merging**: To merge branches

Branching scheme



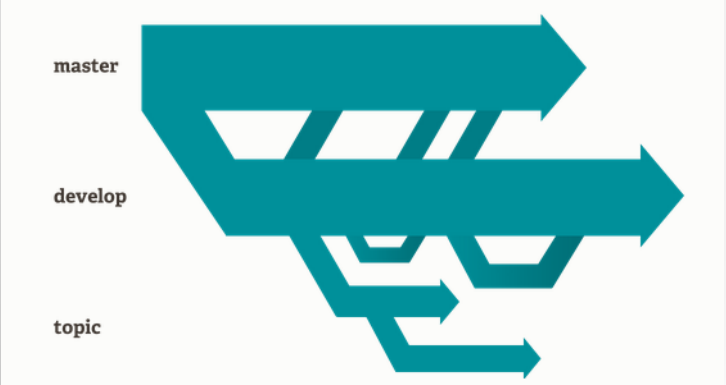
(a)



(b)

Branching and Merging

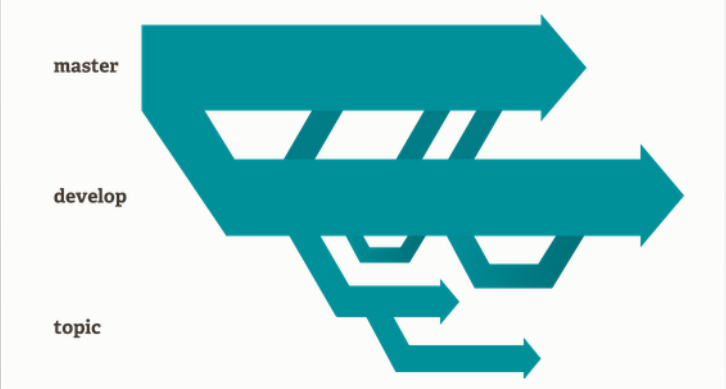
- Git allows and *encourages* you to have *multiple local branches* that can be entirely independent of each other
- You can do things like:
- **Frictionless Context Switching**. Create a branch to try out an idea, commit a few times, switch back to where you branched from, apply a patch, switch back to where you are experimenting, and merge it in.
- **Role-Based Codelines**. Have a branch that always contains only what goes to production, another for testing, and several smaller ones for day to day work.



Branching and Merging

(can do things like)

- **Feature Based Workflow**. Create new branches for each new feature you're working on so you can seamlessly switch back and forth between them, then delete each branch when that feature gets merged into your main line.
- **Disposable Experimentation.** Create a branch to experiment in, realize it's not going to work, and just delete it - abandoning the work—with nobody else ever seeing it (even if you've pushed other branches in the meantime).



Some Git resources:

- http://git-scm.com/ (Git site)
- http://git-scm.com/book/en (Book)
- http://git-scm.com/downloads (the Git software)
- http://code.google.com/p/gitextensions/ (a Git GUI)
- http://www.eclipse.org/egit/
  (**EGit** is an Eclipse Team provider for the Git version control system)

Setting your identity

- The first thing you should do when you install Git is to set your **user name** and **e-mail** address (Git commit uses this information, and it's immutably baked into the commits)
  - **$ git config --global user.name "John Doe"**
  - **$ git config --global user.email johndoe@example.com**

- You need to do this only once if you pass the *--global*. If you want to override this with a different name or e-mail address for specific projects, you can run the command without the --global option when you're in that project.

Checking your setting

- If you want to check your settings, you can use the **git config --list** command to list all the settings Git can find at that point:

```
$ git config –-list

user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Git basics: Getting a Git Repository

- You can get a Git project using **two main** approaches
  - by taking an existing directory and imports it into Git
  - by cloning an existing Git repository from another server.
- If you're starting to track an existing project in Git, you need to go to the project's directory and type:

$ git init

- This creates a new subdirectory named **.git** that contains all of your necessary repository files (Git repository skeleton). At this point, nothing in your project is tracked yet

Git basics: Getting a Git Repository

- If you want to start version-controlling on existing files (as opposed to an empty directory), you should probably begin tracking those files and do an initial commit
- You can accomplish that with a few git **add** commands that specify the files you want to track, followed by a **commit**:

```
$ git add *.c
$ git add README
$ git commit m 'initial project version'
```

Git basics: Cloning a Git Repository

- Every version of every file for the history of the project is pulled down when you run **git clone**
- You clone a repository with **git clone [url]**
- For example:

$ git clone git://github.com/schacon/grit.git

- This creates a directory named *"grit"*, initializes a *.git* directory inside it, *pulls* down all the data for that repository, and *checks out* the *latest version*
- To clone the repository into a directory named something other than grit, the command is:

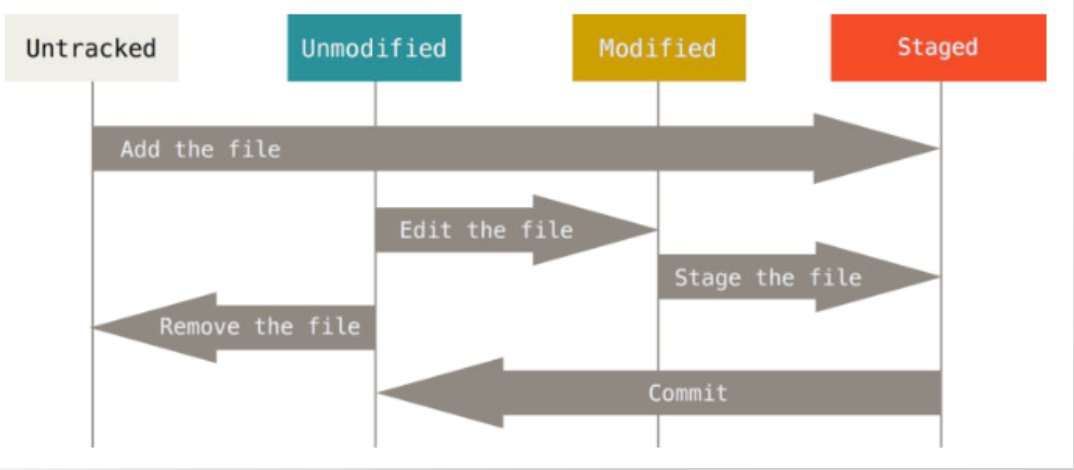
$ git clone git://github.com/schacon/grit.git mygrit

Git basics: file status lifecycle

$ git add <*files*> [tracking new files, stage files, marking merge-conflicted files as resolved]

$ git diff [To see what you've changed but not yet staged]

$ git diff -staged [To compares the staged changes with the last commit]



$ git status

$ git commit <*msg*>

$ git rm <*file*> [removes a file from the staging area and delete it]

Git basics: Checking the Status of Your Files

- The main tool you use to determine which files are in which state is the git **status** command. If you run this command directly after a clone, you should see something like this:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

Git basics: Checking the Status of Your Files

- Let's say you add a *new file to your project*, a simple README file. If the file didn't exist before, and you run git status, you see your untracked file like so:

```
$ git status
# On branch master
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
#
# README
nothing added to commit but untracked files present (use "git add" to track)
```

Git basics: Checking the Status of Your Files

- In order to begin tracking a new file, you use the command git **add**:

```
$ git add README
```

- If you run your status command again, you can see that your README file is now tracked and staged:

```
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# new file: README
#
```

Git basics: Staging a modified files

- Let's change a file that was already tracked, e.g. the file called benchmarks.rb. Running **status**, you get:

```
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# new file: README
#
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
#
# modified: benchmarks.rb
#
```

Git basics: Staging a modified files

- Let's run git **add** now to stage the benchmarks.rb file, and then run git **status** again:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# new file: README
# modified: benchmarks.rb
#
```

Git basics: Staging a modified files

- At this point, suppose you remember one little change that you want to make in benchmarks.rb before you commit it.

```
$ vim benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# new file: README
# modified: benchmarks.rb
#
# Changed but not updated:
# (use "git add <file>..." to update what will be committed)
#
# modified: benchmarks.rb
#
```

Git basics: Staging a modified files

- If you modify a file after you run git add, you have to run git add again to stage the latest version of the file:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
# (use "git reset HEAD <file>..." to unstage)
#
# new file: README
# modified: benchmarks.rb
#
```

Git basics: Viewing Your Changes

- If you want to know exactly what you changed you can use the git **diff** command
- To see what you've changed but not yet staged, type git **diff** with no other arguments

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
@commit.parents[0].parents[0].parents[0]
end
+ run_code(x, 'commits 1') do
+ git.commits.size
+ end
+
run_code(x, 'commits 2') do
log = git.commits('master', 15)
log.size
```

Git basics: Viewing the Commit History

- When you run git **log**, you want to look back to see what has happened in a project

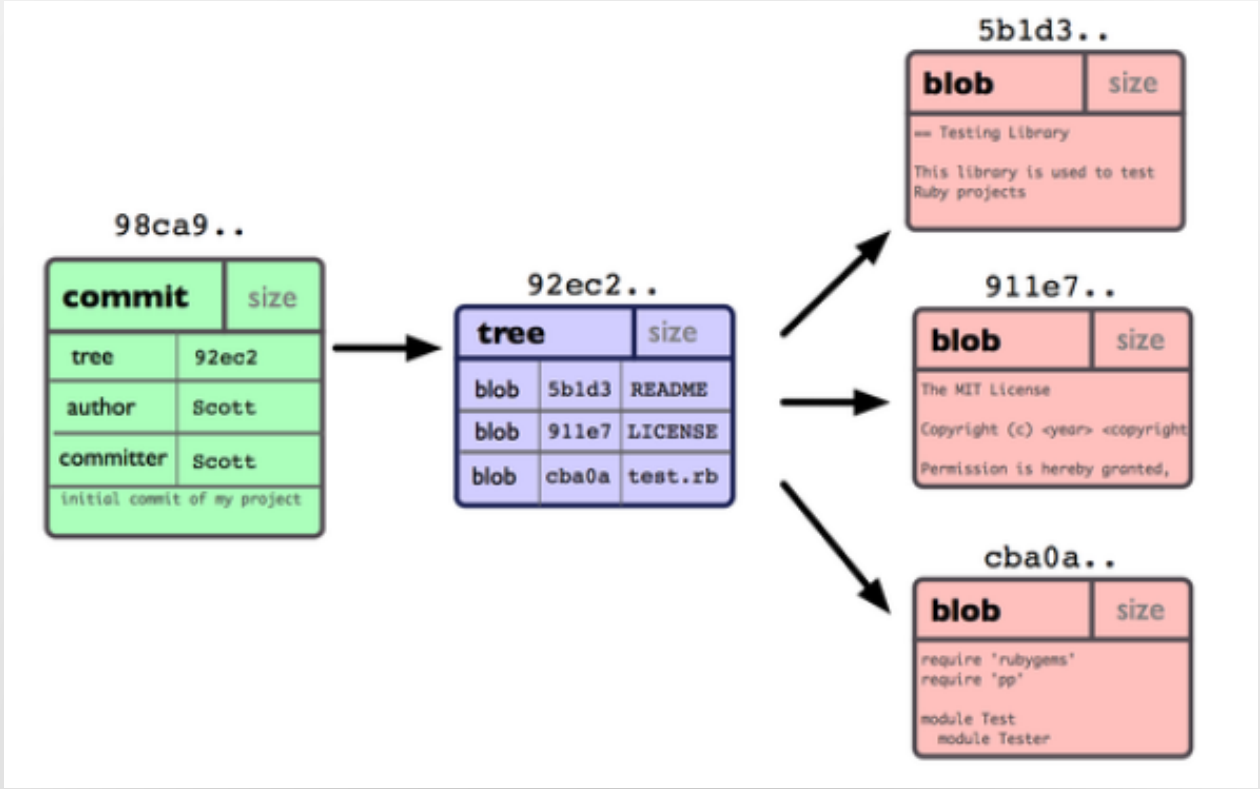
```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date: Mon Mar 17 21:52:11 2008 -0700
changed the verison number
commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sat Mar 15 16:40:33 2008 -0700
removed unnecessary test code
commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sat Mar 15 10:31:28 2008 -0700
first commit
```

Git branching

- Some people refer to the branching model in Git as its "killer feature," and it certainly sets Git apart in the VCS community
- Why is it so special? The way Git branches is incredibly lightweight, making branching operations nearly instantaneous and switching back and forth between branches generally just as fast
- Git encourages a workflow that branches and merges often, even multiple times in a day

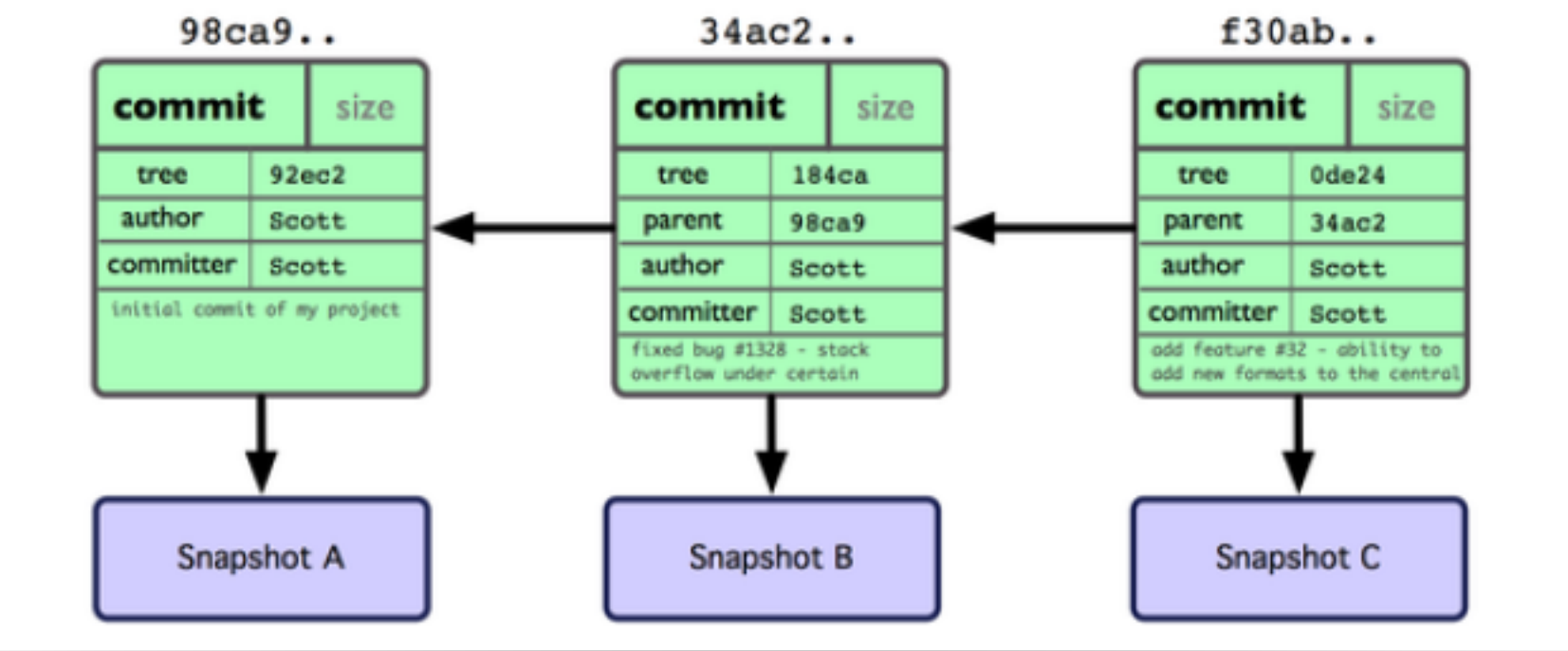
Git branching

$ git add README test.rb LICENSE2
$ git commit -m 'initial commit of my project'



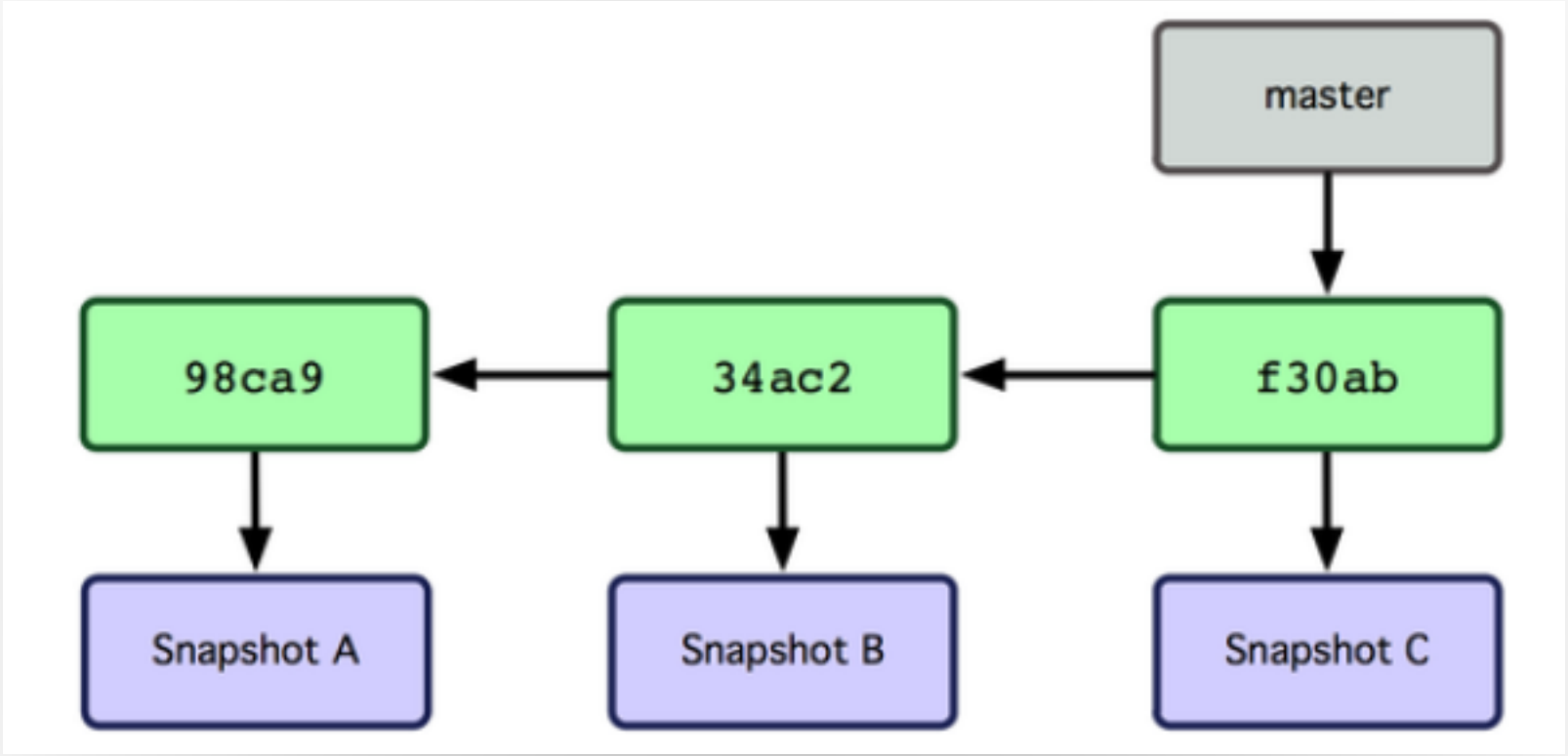
Git branching

If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it



Git branching

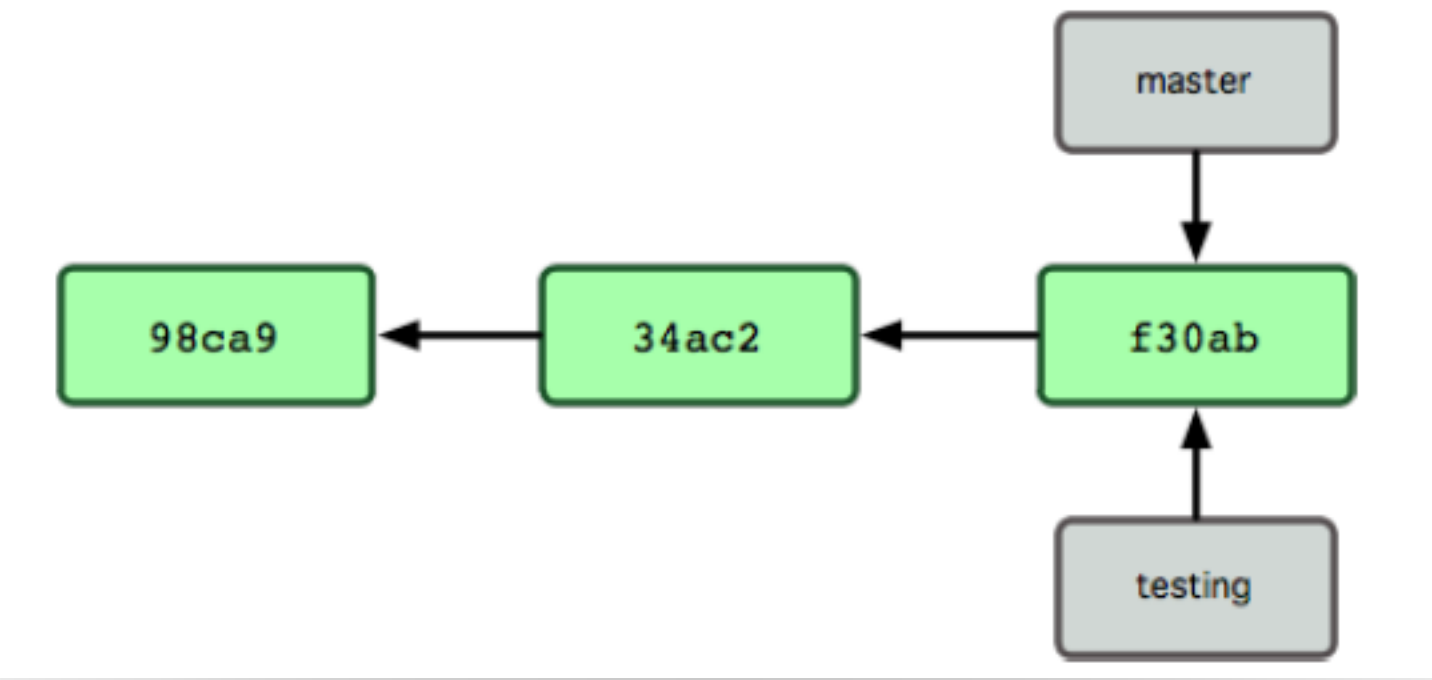
A branch in Git is simply a **lightweight movable pointer** to one of these commits. The default branch name in Git is **master**



Git branching

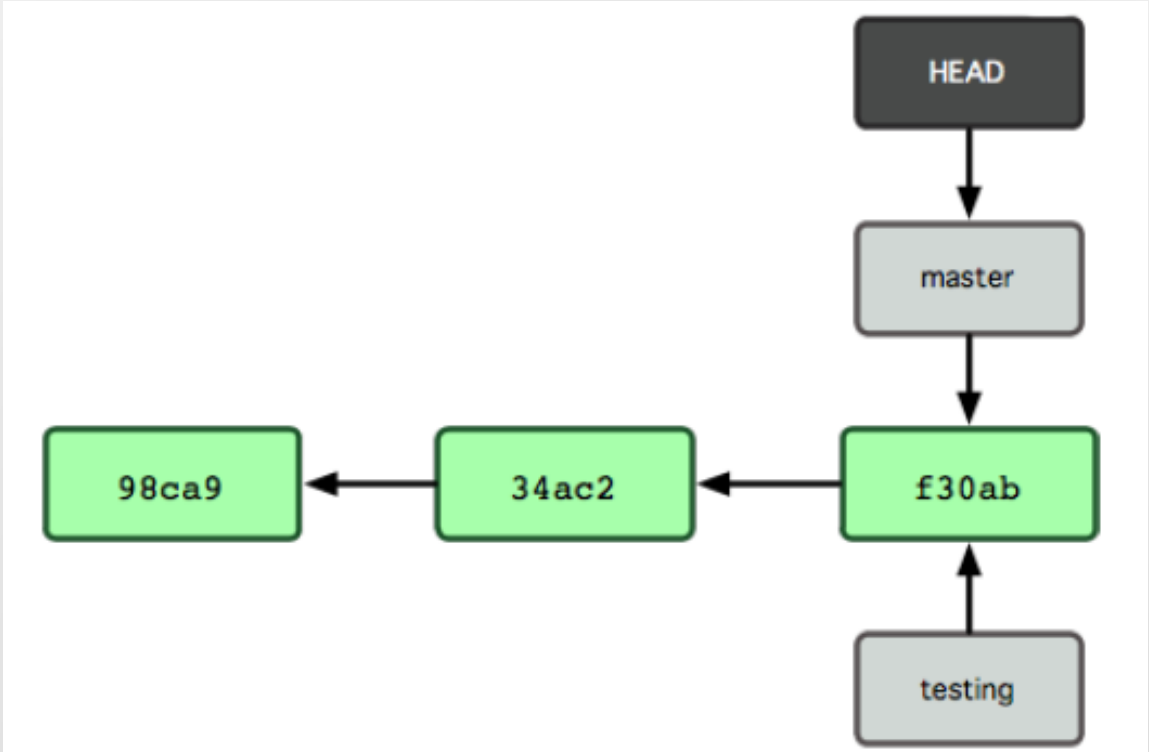
What happens if you create a new branch? Well, doing so creates a new pointer for you to move around. Let's say you create a new branch called **testing**. You do this with the git **branch** command:

$ git branch testing



Git branching

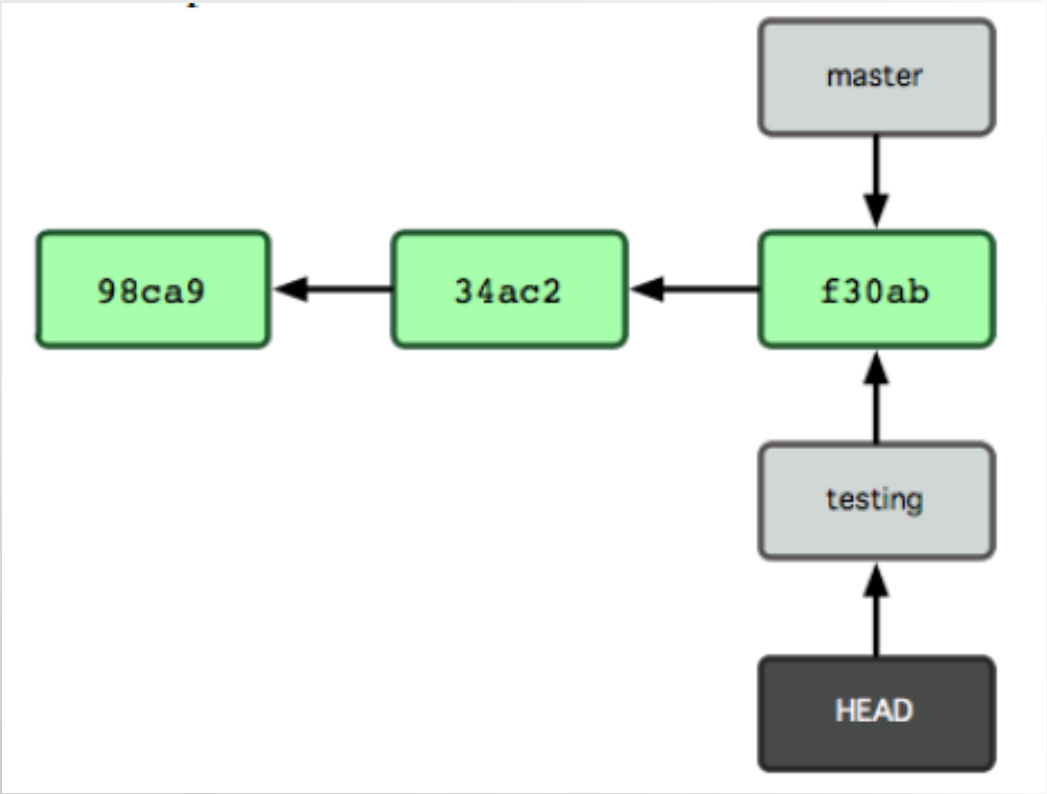
How does Git know what branch you're currently on? It keeps a special pointer called **HEAD**. This is a pointer to the local branch you're currently on. The git branch command only created a new branch— it didn't switch to that branch.



Git branching

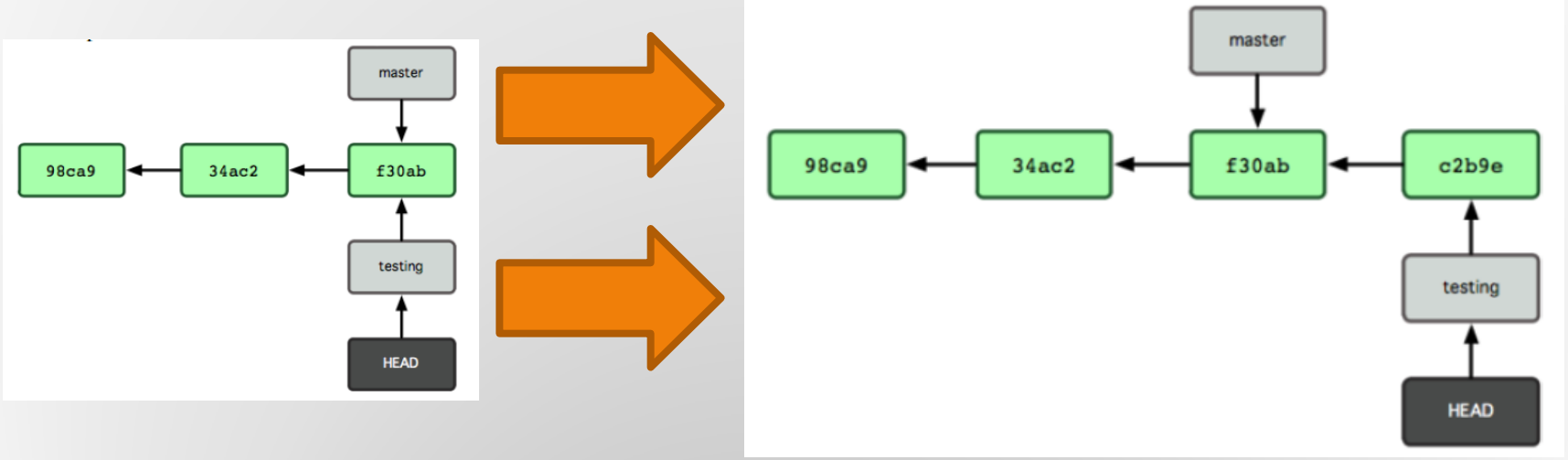
To switch to an existing branch, you run the git **checkout** command. That command did two things. It moved the HEAD pointer back to point to the new branch, and it reverted the files in your working directory back to the snapshot that pointer points to.

$ git checkout testing



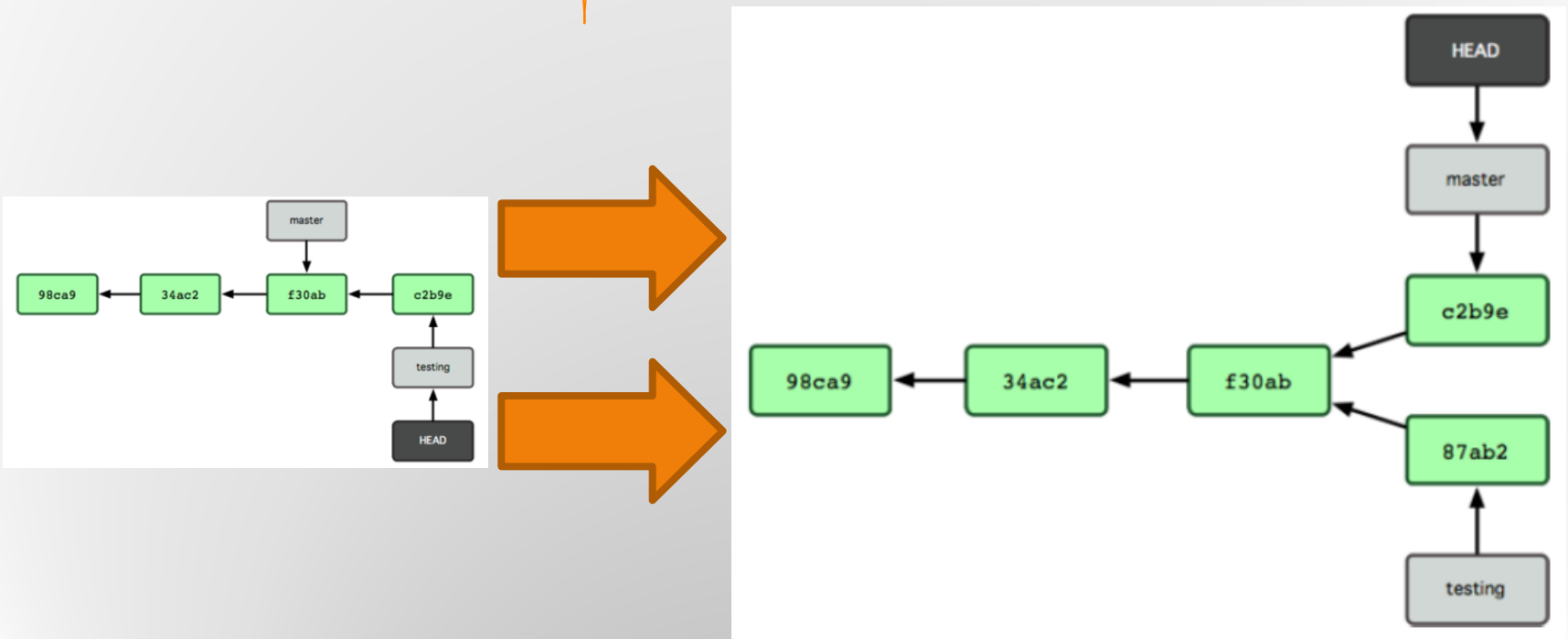
Git branching

```
$ vim test.rb
$ git commit -a -m 'made a change'
```



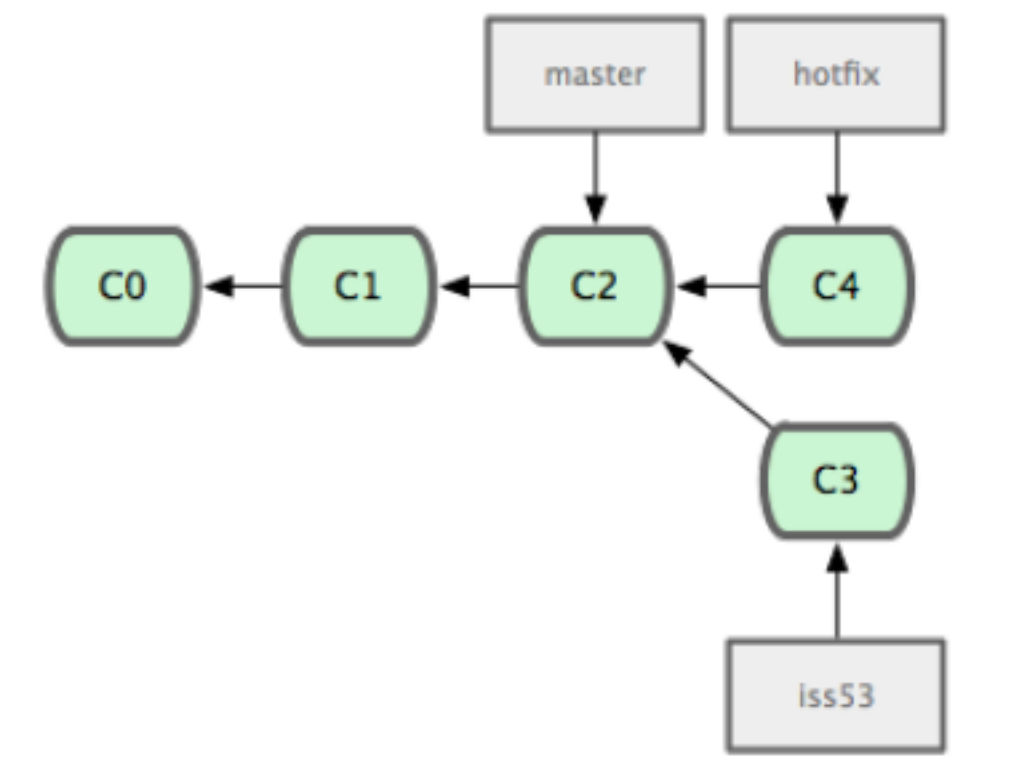
Git branching

```
$ git checkout master
$ vim test.rb
$ git commit -a -m 'made other changes'
```



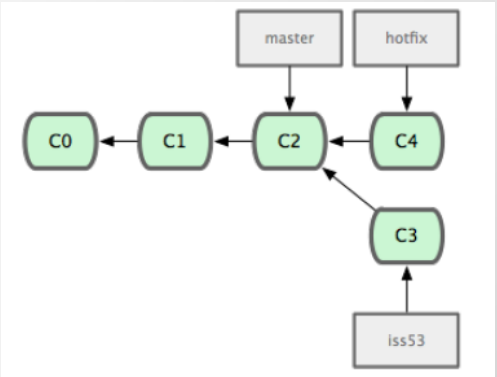
Git merging (1)

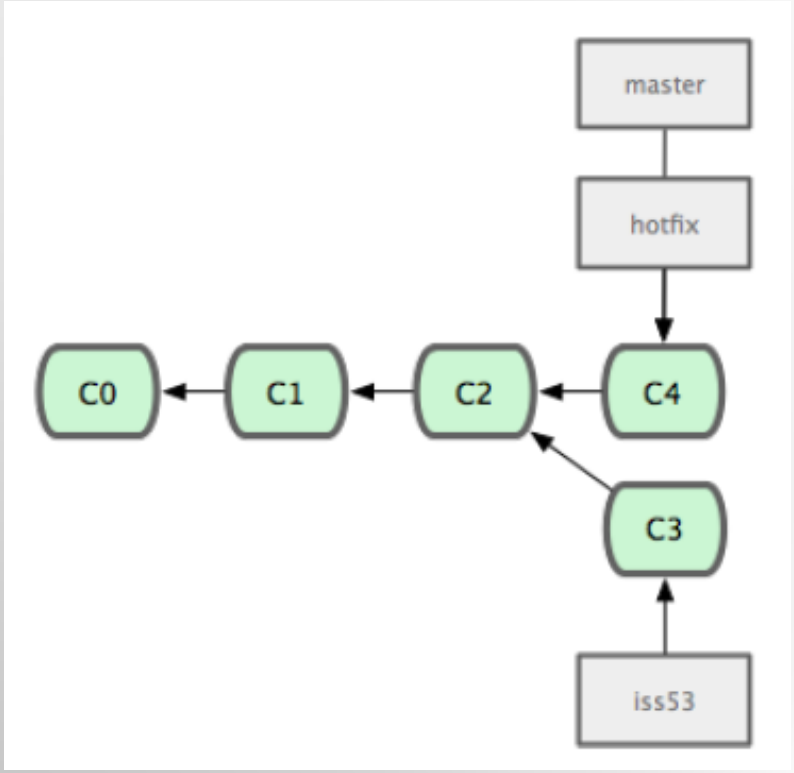
Let’s consider the following scenario:



Git merging (1)

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
README | 1 -
1 files changed, 0 insertions(+), 1
deletions(-)'
```

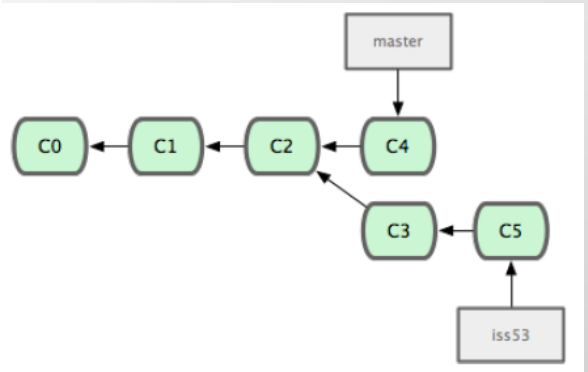


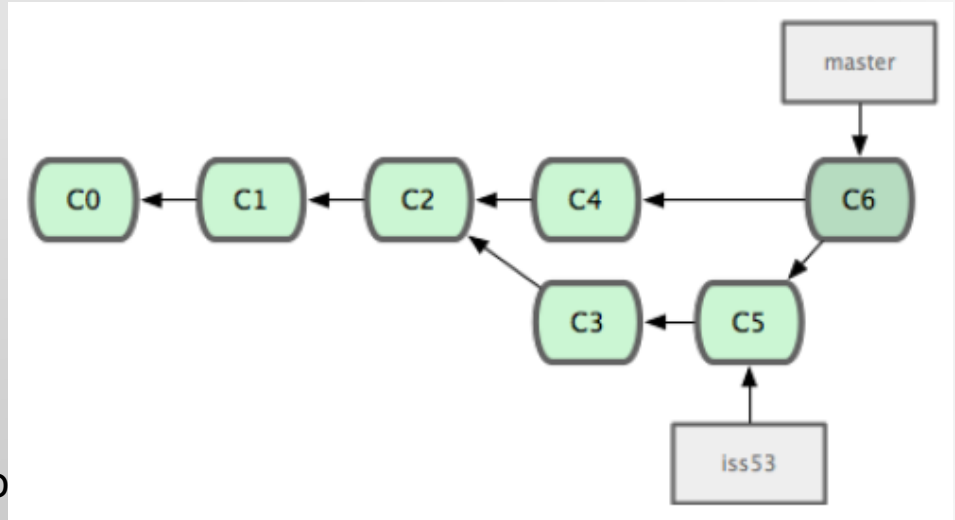


You can **delete** *hotfix* with the **-d** option to git branch.

Git merging (2)

```
$ git checkout master
$ git merge iss53
Merge made by recursive.
README | 1 +
1 files changed, 1 insertions(+), 0
deletions(-)
```



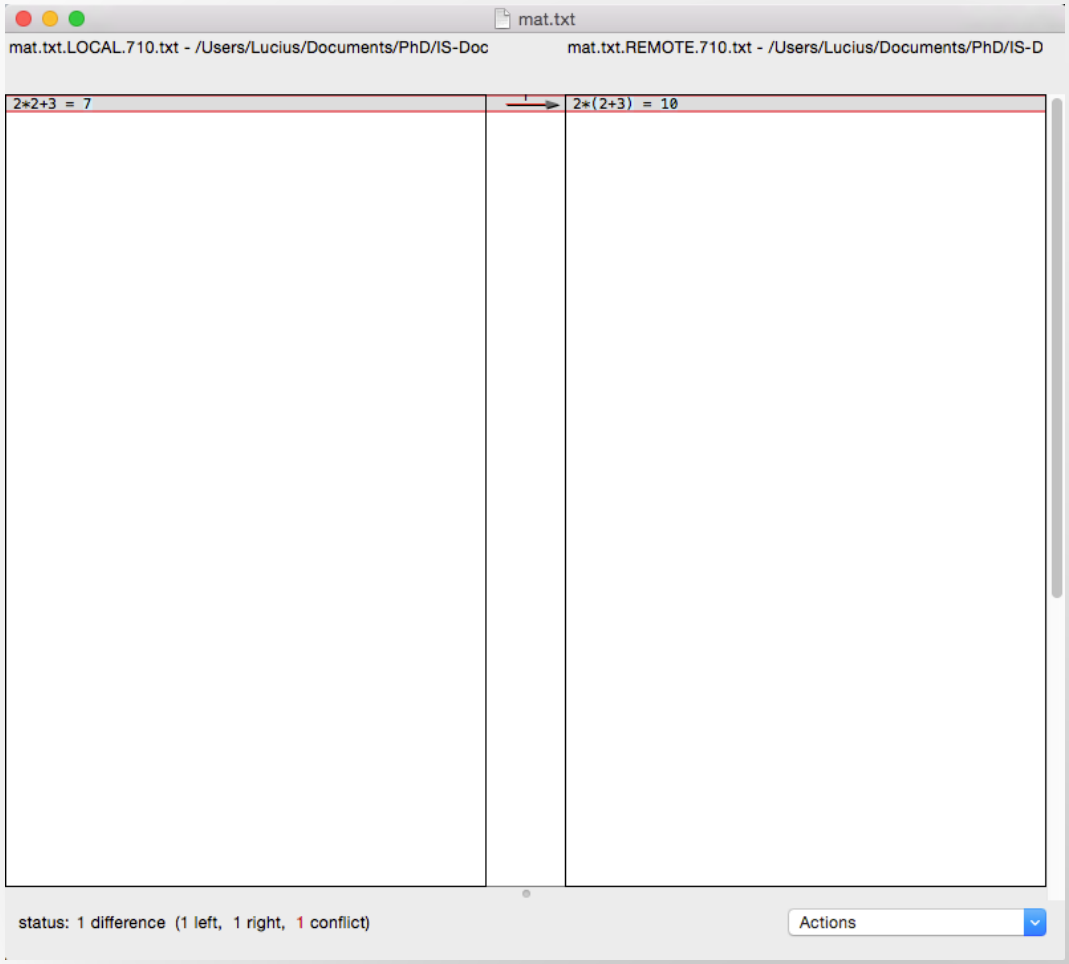


Git automatically creates a new commit o
that contains the merged work.

Git merging: basic conflicts

- Occasionally, the merge process doesn't go smoothly (e.g. if the same part of the same file is changed differently in the two branches you're merging together)
- Git don't automatically created a new merge commit. It has paused the process while you resolve the conflict (To see which files are unmerged you can run git status)
- Git adds standard conflict-resolution markers to the files that have conflicts, so you can open them manually and resolve those conflicts
- Anyway, the use of a mergetool is suggested (the command **$ git mergetool** allows to chose a mergetool to use). After you exit the merge tool, Git asks you if the merge was successful. If you tell the script that it was, it stages the file to mark it as resolved for you.

Git merging: git mergetool



Click on the menu **Actions** and choose how to resolve the conflict (for example, you can take the content on the left or the right box).

Remember: if you tell git that the merge was successful you have to use **git commit** in order to complete the merge process.

Getting Git on a Server:

- In order to do any collaboration in Git, you'll need to have a **remote Git repository**
- Although you can technically push changes to and pull changes from individuals' repositories, doing so is discouraged because you can fairly easily confuse what they're working on if you're not careful
- Furthermore, you want your collaborators to be able to access the repository even if your computer is offline
- *having a more reliable common repository is often useful*

Getting Git on a Server:

- We'll refer to this repository as a "Git server" (rarely need to use an entire server for it)
- A remote repository is generally a **bare repository** (because the repository is only used as a collaboration point, there is no reason to have a snapshot checked out on disk)
- A bare repository is the contents of your project's **.git directory** and nothing else
- Git can use four major network protocols to transfer data: Local, **Secure Shell (SSH)**, Git, and HTTP
- Obviously Git requires to be installed on the server

Getting Git on a Server: creating an empty bare repository

- An empty bare repository on the (ssh) server can be initialized by issuing the following commands:

```
$ ssh user@git.example.com
$ cd /opt/git
$ mkdir my_project.git
$ cd my_project.git
$ git init --bare --shared
```

Getting Git on a Server: creating a bare repository (clone)

- In order to initially set up any Git server, you have to export an existing repository into a new **bare repository** (a repository that doesn't contain a working directory)
- In order to clone your repository to create a new bare repository, you run the **clone** command with the **--bare** option:

  $ git clone --bare my_project my_project.git

- By convention, bare repository directories end in .git

Getting Git on a Server: Putting the Repository on a Server

- Now that you have a bare copy of your repository, all you need to do is put it on a server and set up your protocols
- Let's say:
  - you've set up a server called **git.example.com**
  - that you have **SSH** access to it
  - you want to store all your Git repositories under the **/opt/git** directory
- You can set up your new repository by copying your bare repository over:

$ scp -r my_project.git user@git.example.com:/opt/git

(scp means secure copy)

Getting Git on a Server: Putting the Repository on a Server

- At this point, other users who have SSH access to the same server which has read access to the /opt/git directory **can clone** your repository by running:

$ git clone user@git.example.com:/opt/git/my_project.git

Graphical Interfaces

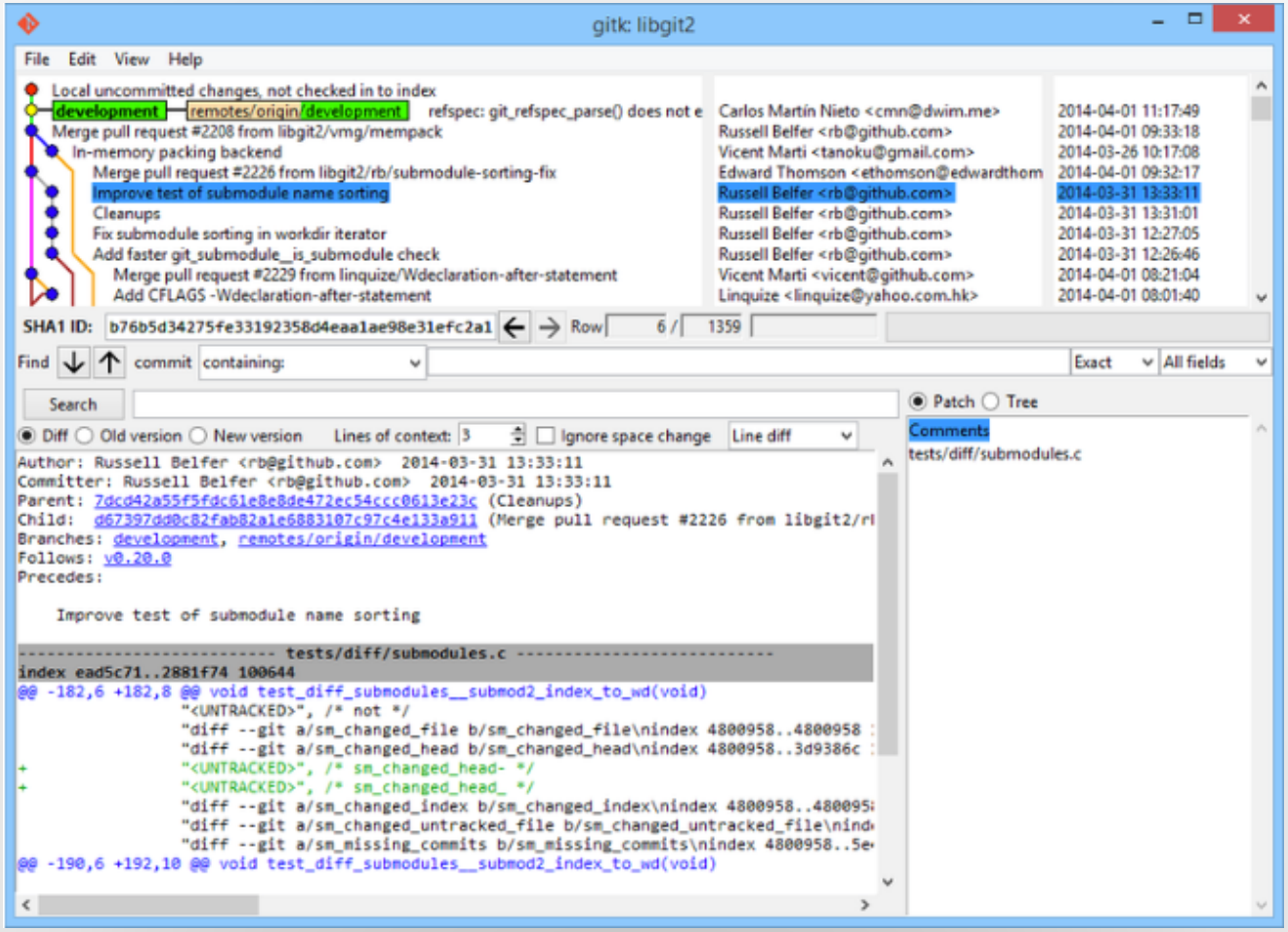
- Git's native environment is in the terminal. New features show up there first, and only at the command line is the full power of Git completely at your disposal.
- But plain text isn't the best choice for all tasks; sometimes a visual representation is what you need.
- Note that there's nothing a graphical client can do that the command-line client can't; the command-line is still where you'll have the most power and control when working with your repositories.

Graphical Interfaces: gitk

- gitk is a graphical history viewer. Think of it like a powerful GUI shell over **git log** and **git grep**.
- This is the tool to use when you're trying to find something that happened in the past, or visualize your project's history.
- To invoke Gitk from the command-line just cd into a Git repository, and type:

$ gitk [git log options]

Graphical Interfaces: gitk



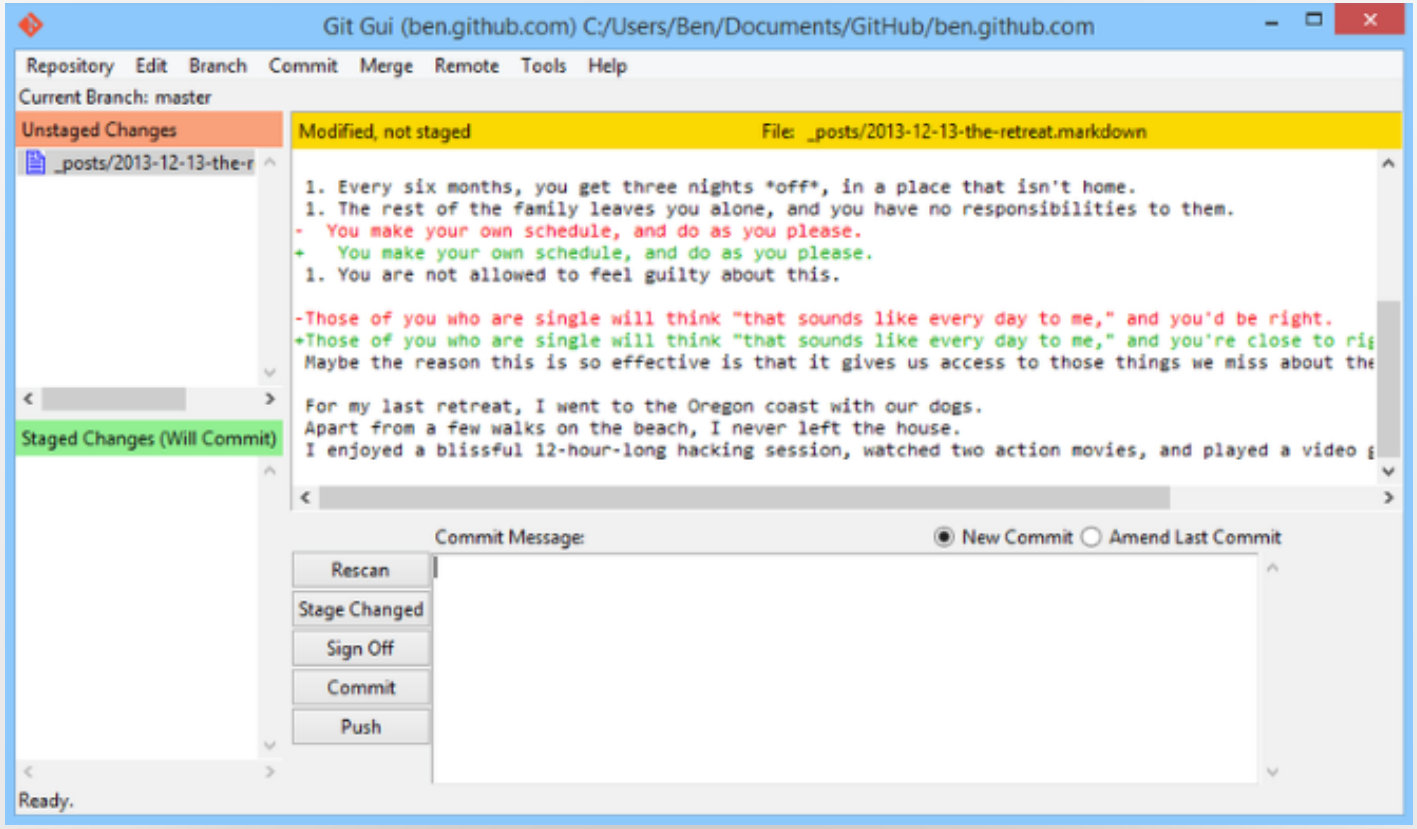
Graphical Interfaces: git gui

- git-gui is primarily a tool for crafting commits. It, too, is easiest to invoke from the command line:

$ git gui

- git-gui focuses on allowing users to make changes to their repository by making new commits, amending existing ones, creating branches, performing local merges, and fetching/pushing to remote repositories.

Graphical Interfaces: git gui



Graphical Interfaces: EGit

- EGit is the Git integration for the Eclipse IDE, see http://eclipse.org/egit.
- EGit is already included in the Eclipse Juno Release, so you do not need to install it. If you use an older version of Eclipse, open the Eclipse Wizard to install new software Help => Install New Software (insert *http://download.eclipse.org/egit/*updates after Work with:).

Graphical Interfaces: EGit - Configuration

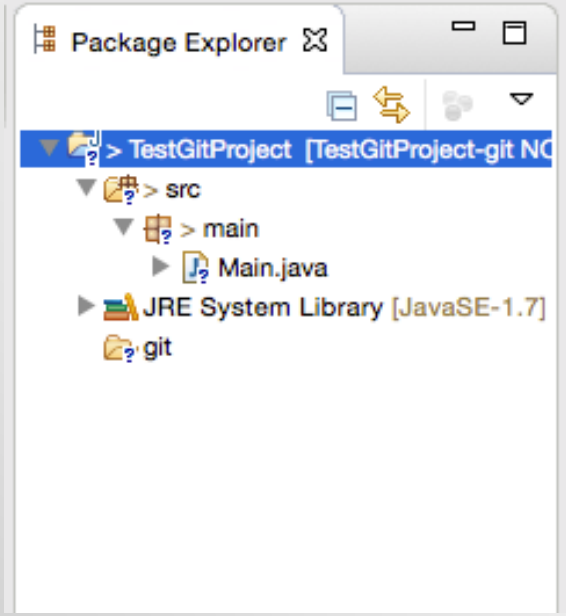
- Every commit in EGit will include the user's name and his email-address.
- These attributes can be set in the *Preferences-window Window => Preferences.* Navigate to *Team => Git => Configuration* and hit the *New Entry…* button. Enter *user.name* as Key and your name as value and confirm. Repeat this procedure with *user.email* and your email address and click *OK* in the Preferences window.

Graphical Interfaces: EGit - Creating Local Repositories

- First, you have to create a project in Eclipse that you want to share via your local repository.
- After you have created your project, select the context menu by right clicking it and navigate to *Team => Share Project…* . Select Git as the repository type and hit *Next*. In the following window select your project, hit the *Create*-button and click *Finish*.

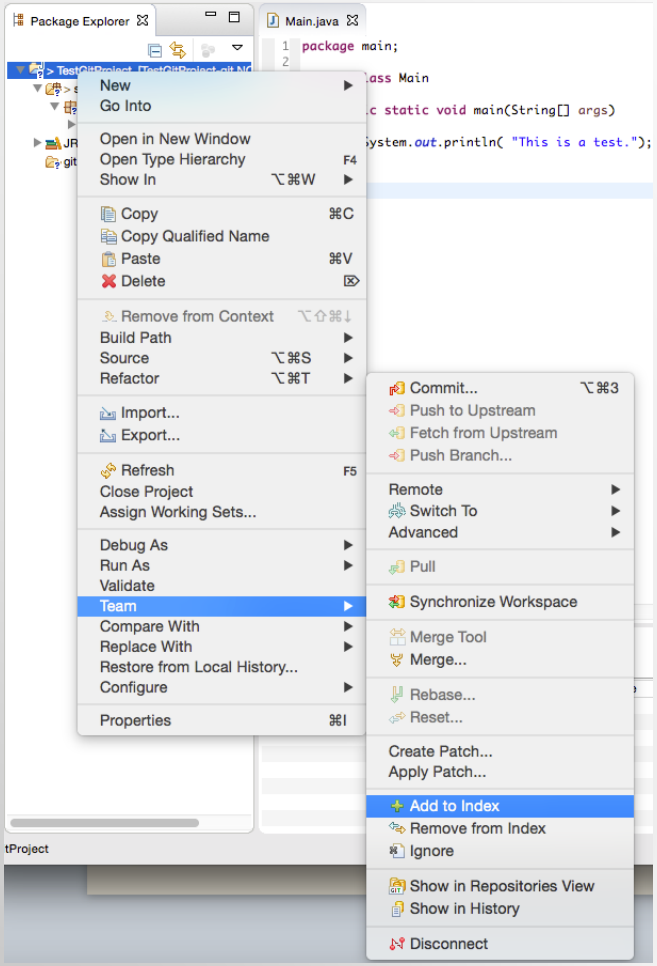
Graphical Interfaces: EGit - Creating Local Repositories

- The newly created repository will be empty, although the project is assigned to it.
- The project node will have a repository icon, the child nodes will have an icon with a question mark, ignored files, e.g. the bin directory, won't have any icons at all.



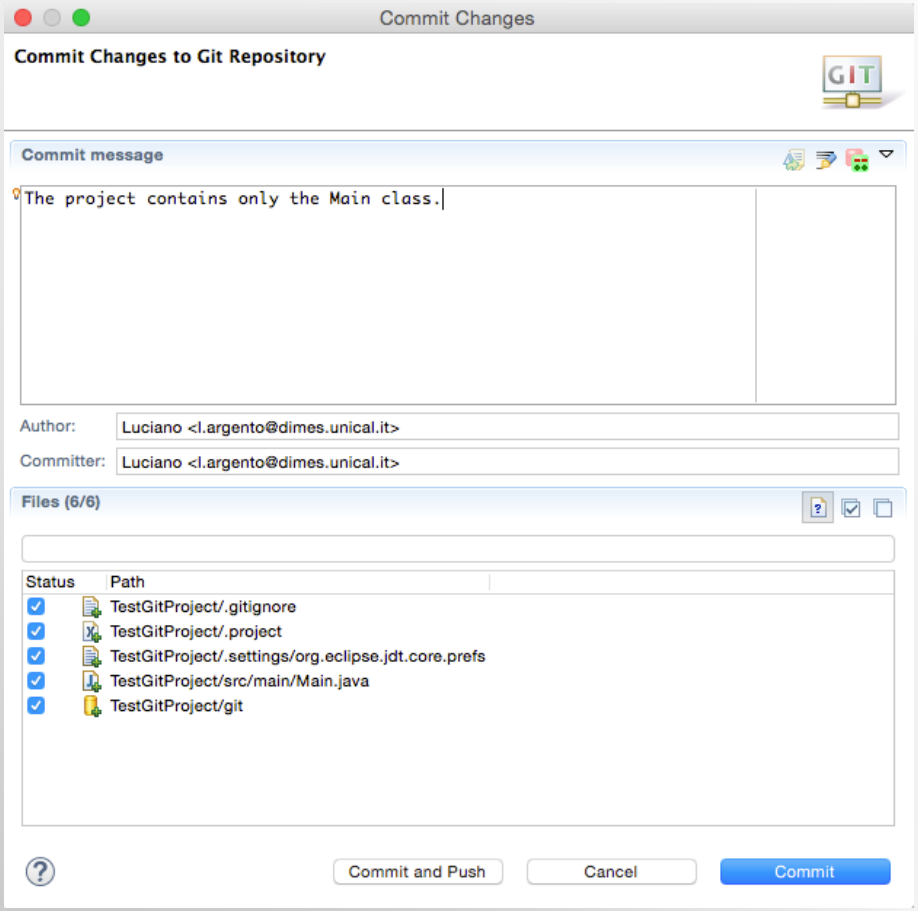
Graphical Interfaces: EGit - Creating Local Repositories

- Before you can commit the files to your repository, you need to add them. Simply right click the shared project's node and navigate to *Team => Add*. After this operation, the question mark should change to a plus symbol.



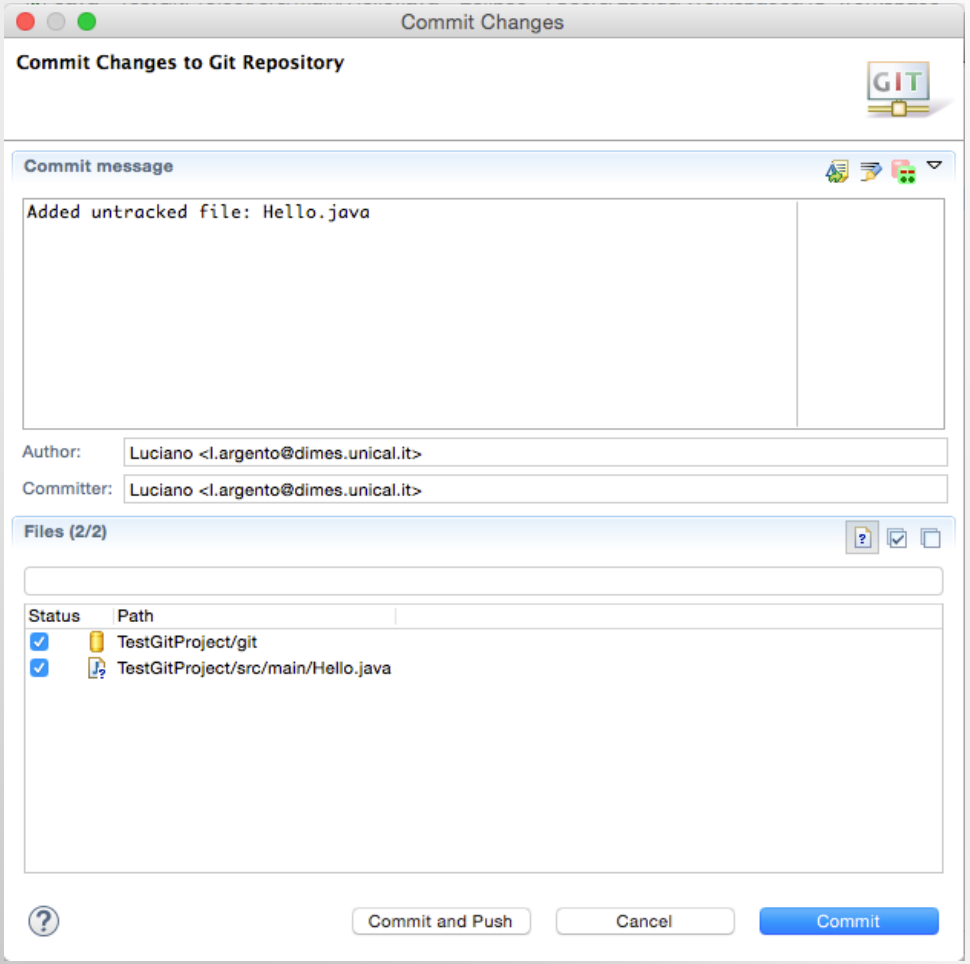
Graphical Interfaces: EGit - Creating Local Repositories

- The last thing to do is commit the project by right clicking the project node and selecting *Team => Commit…* from the context menu. In the Commit wizard, all files should be selected automatically. Enter a commit message (the first line should be headline-like, as it will appear in the history view) and hit the *Commit* button.
- If the commit was successful, the plus symbols will have turned into repository icons.



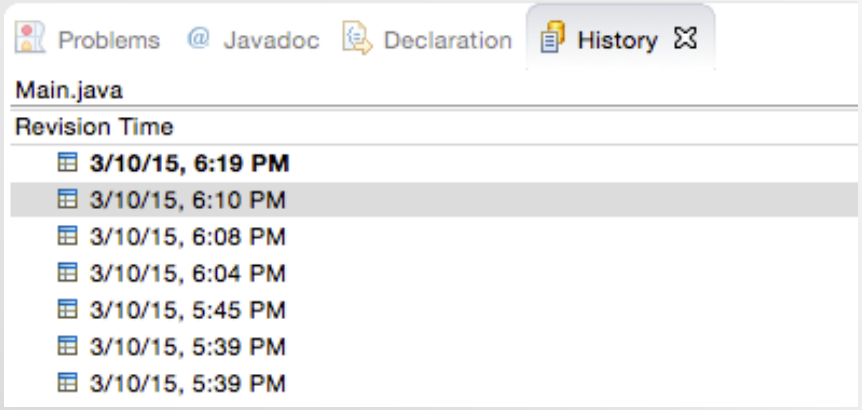
Graphical Interfaces: EGit - Commit untracked files

- EGit also allows selecting untracked files to be added in the commit dialog if you turn on the option "Show untracked files". In this case, they will be added and committed at the same time.

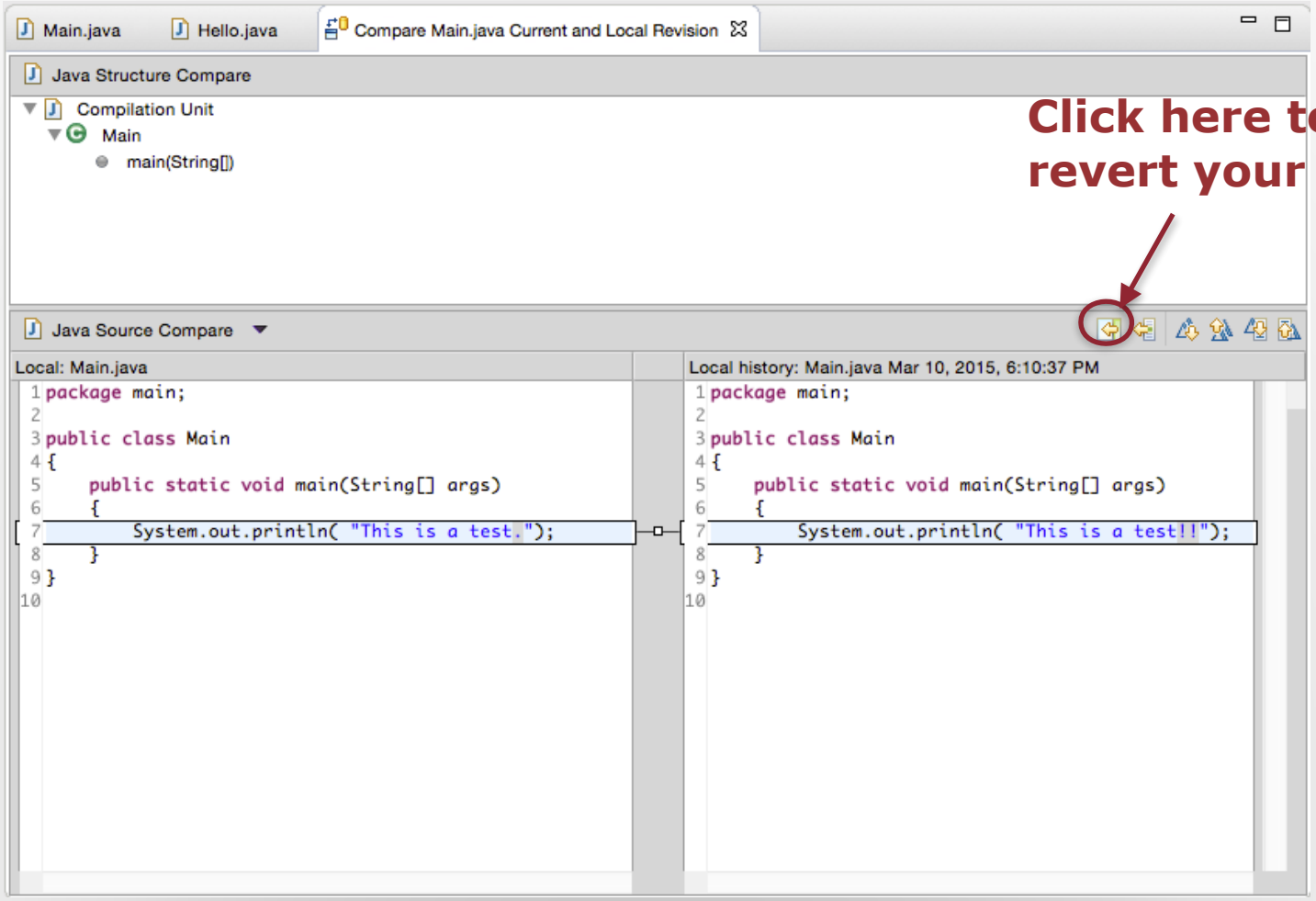


Graphical Interfaces: EGit - Reverting Changes

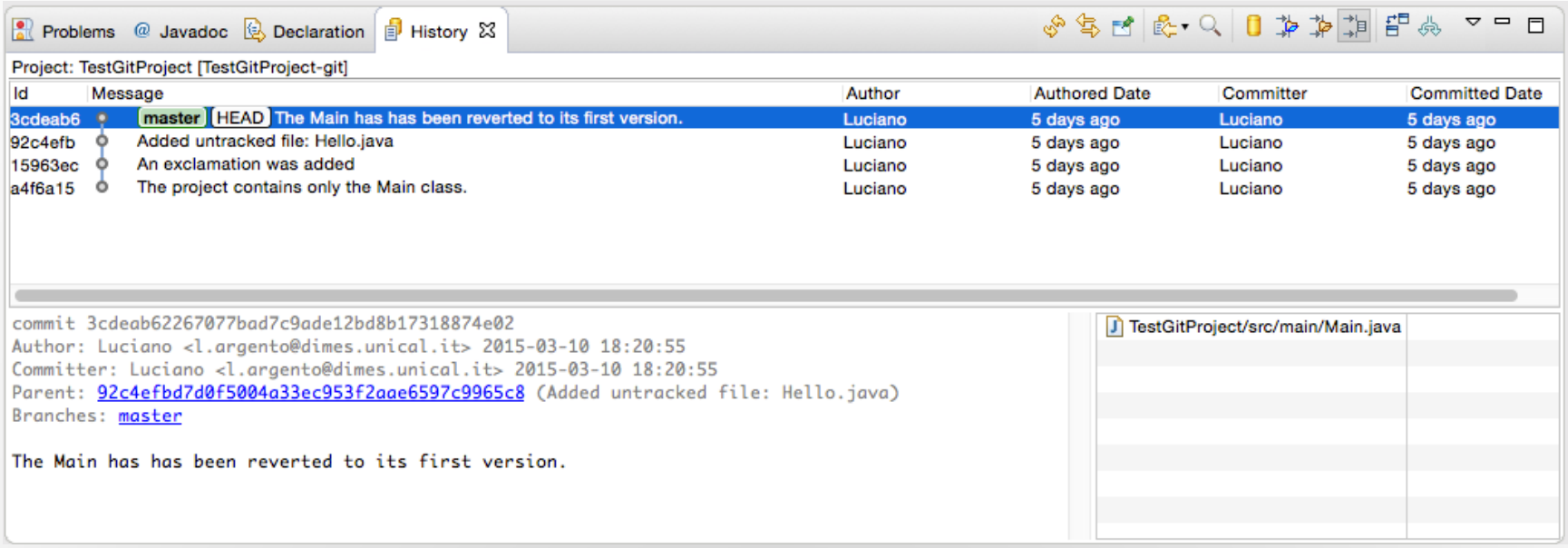
- By right clicking on a file and selecting *Compared with =>local history…* it is possible to show its entire local history.
- If you want to see the differences between the current and a specific previous version of the file you have to double-click on the appropriate entry.



Graphical Interfaces: EGit - Reverting Changes

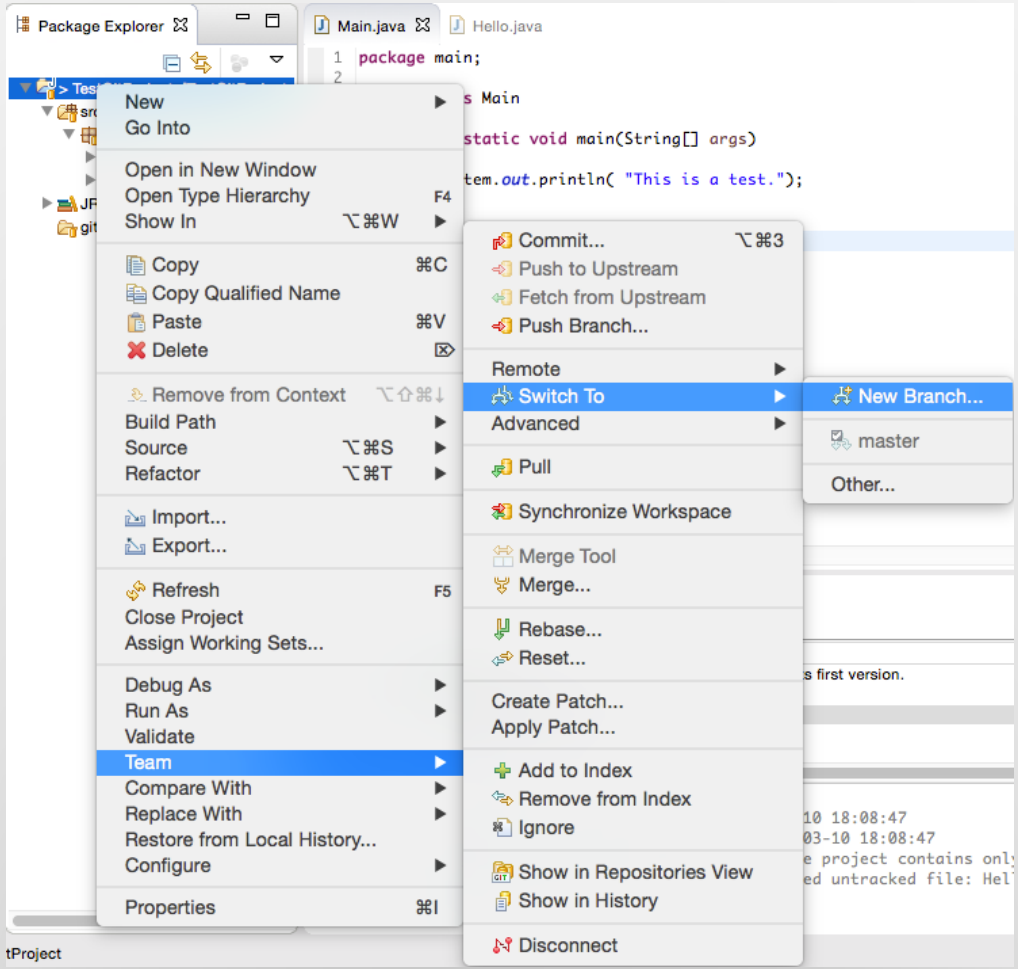


Graphical Interfaces: EGit - History of the project

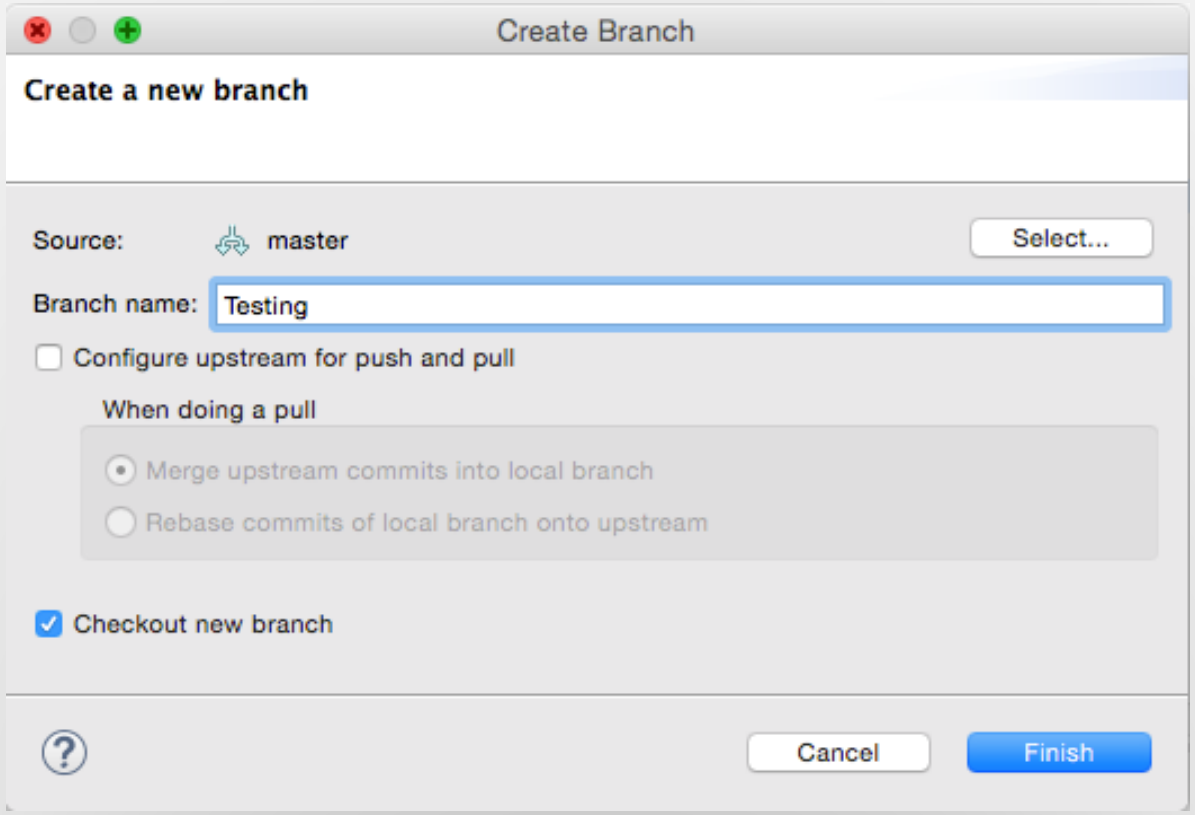


- By right clicking on the project icon and selecting *Team => Show in history* you can see all commits.

Graphical Interfaces: EGit - Create a new branch

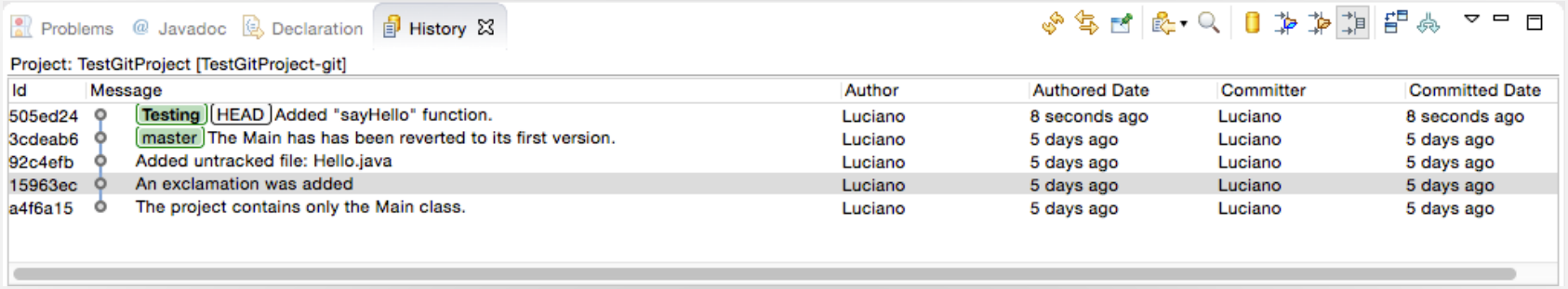


Graphical Interfaces: EGit - Create a new branch



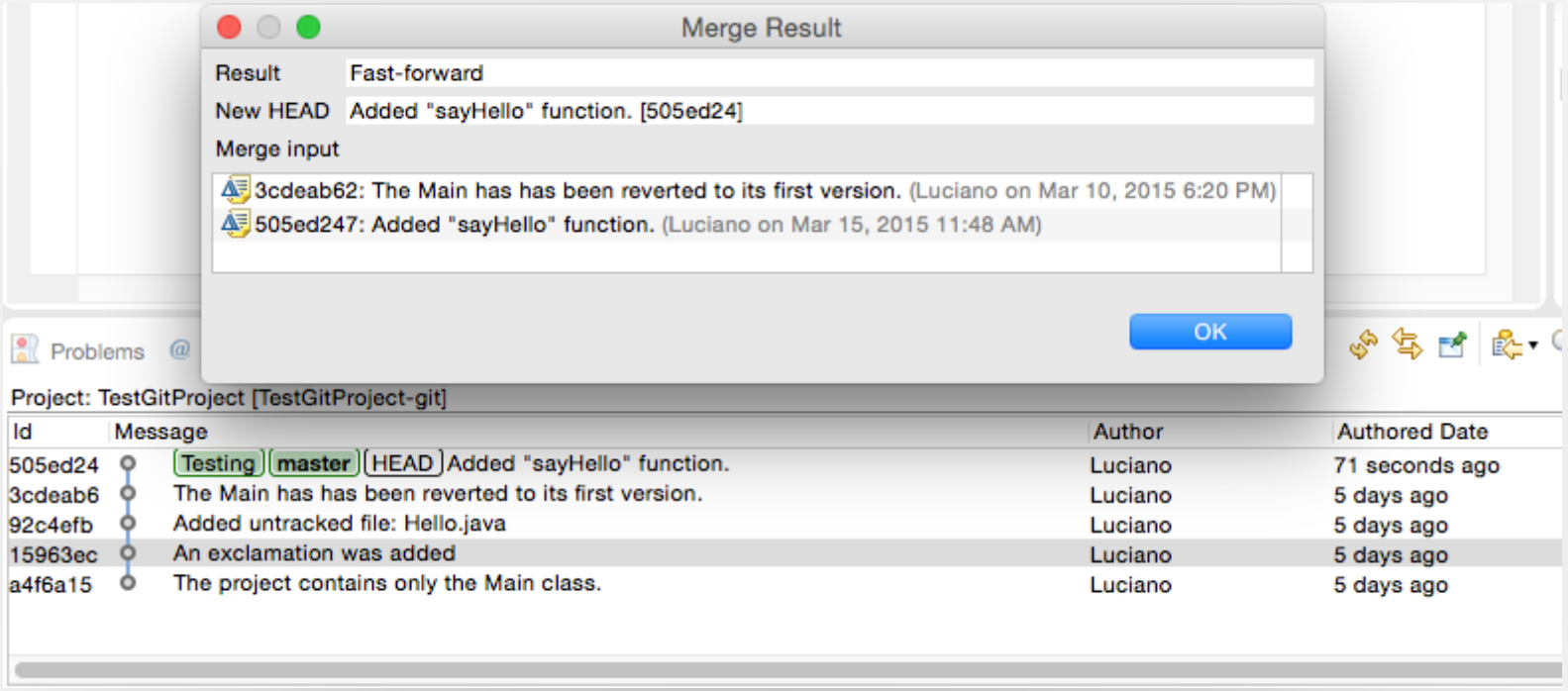
Graphical Interfaces: EGit - Create a new branch

- The "Hello" class had been modified in the branch testing, subsequently the change was committed.
- The figure below shows the result of the previous operation.



Graphical Interfaces: EGit - Merge

- To merge the master and testing branch, right click on the project and select *Team => Merge*.



References

- http://git-scm.com/book/it/v2
- http://eclipsesource.com/blogs/tutorials/egit-tutorial/