Git:
Time travel & other stories

Graeme West

What is Git?

- Modern revision management software for file structures
- Managing collections of digital ‘stuff’ in a sensible way, across ***TIME*** and ***SPACE***

Summary

1. Git is flexible and facilitative source control
2. Rebasing allows flexible workflows
3. Merging in Git is intelligent & straightforward
4. Remotes allow code sharing
5. There's more nifty stuff too

But I don't need revision control!

Gitvantages (part 1)

- Modern
  - fast, lean, low overhead (no servers)
- Flexible
  - Great for single or multi-user setups
  - Versatile branching, merging and stashing
- Plays well with others
  - Amazing SVN migration/parallel working
  - Multi-platform

Gitvantages (part 2)

- Strong integrity - everything is hashed
- Offline working - most operations are local
  - Commit code on the plane/boat/your secret underground lair
- Not scary
  - (almost) everything is undo-able
- Great documentation and large community of users

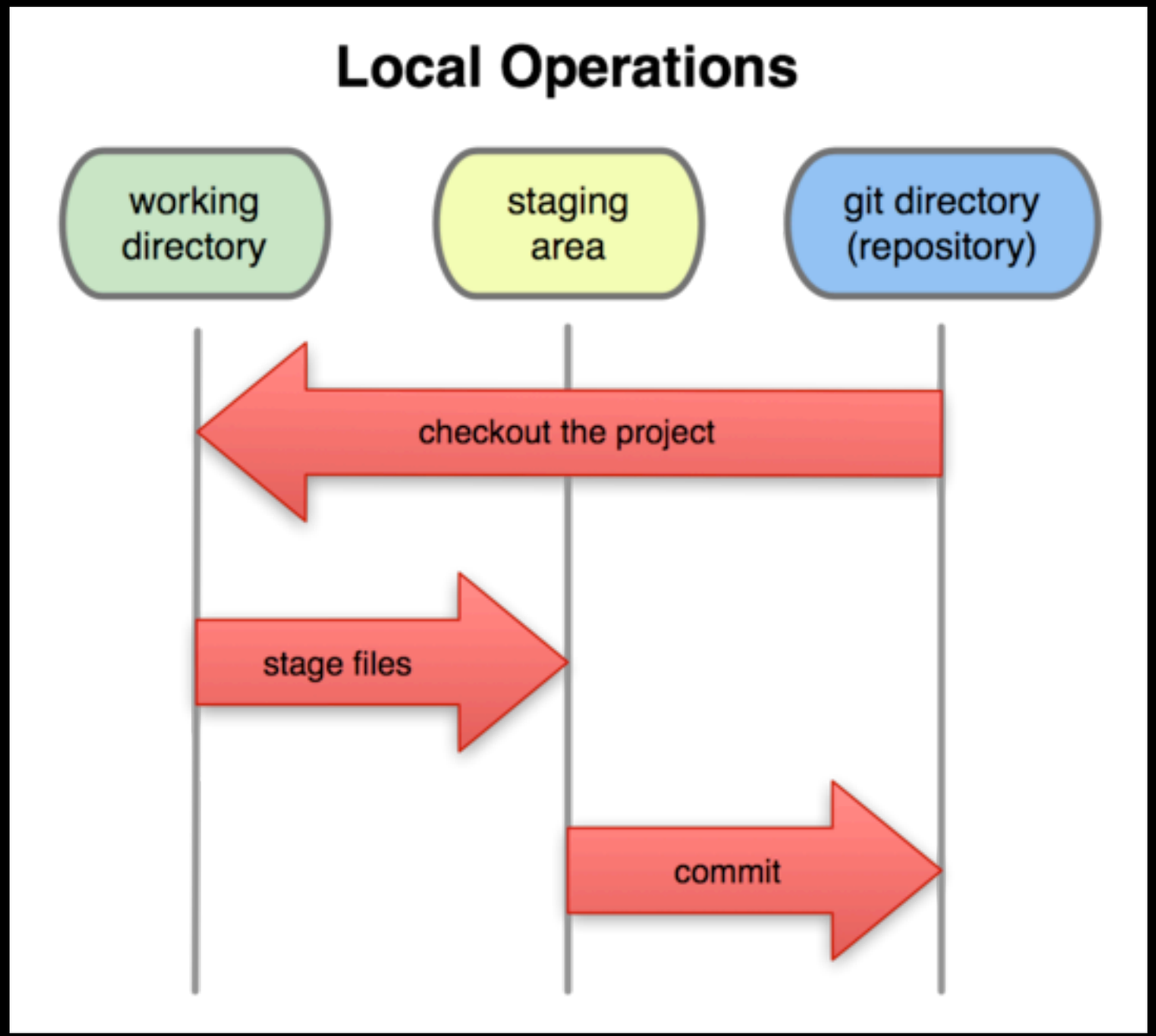
Disadvantages

- Memory hungry (clone operation)
- Fewer tools with built-in support than SVN
- Flexibility means multi-person workflow can present learning curve

Design

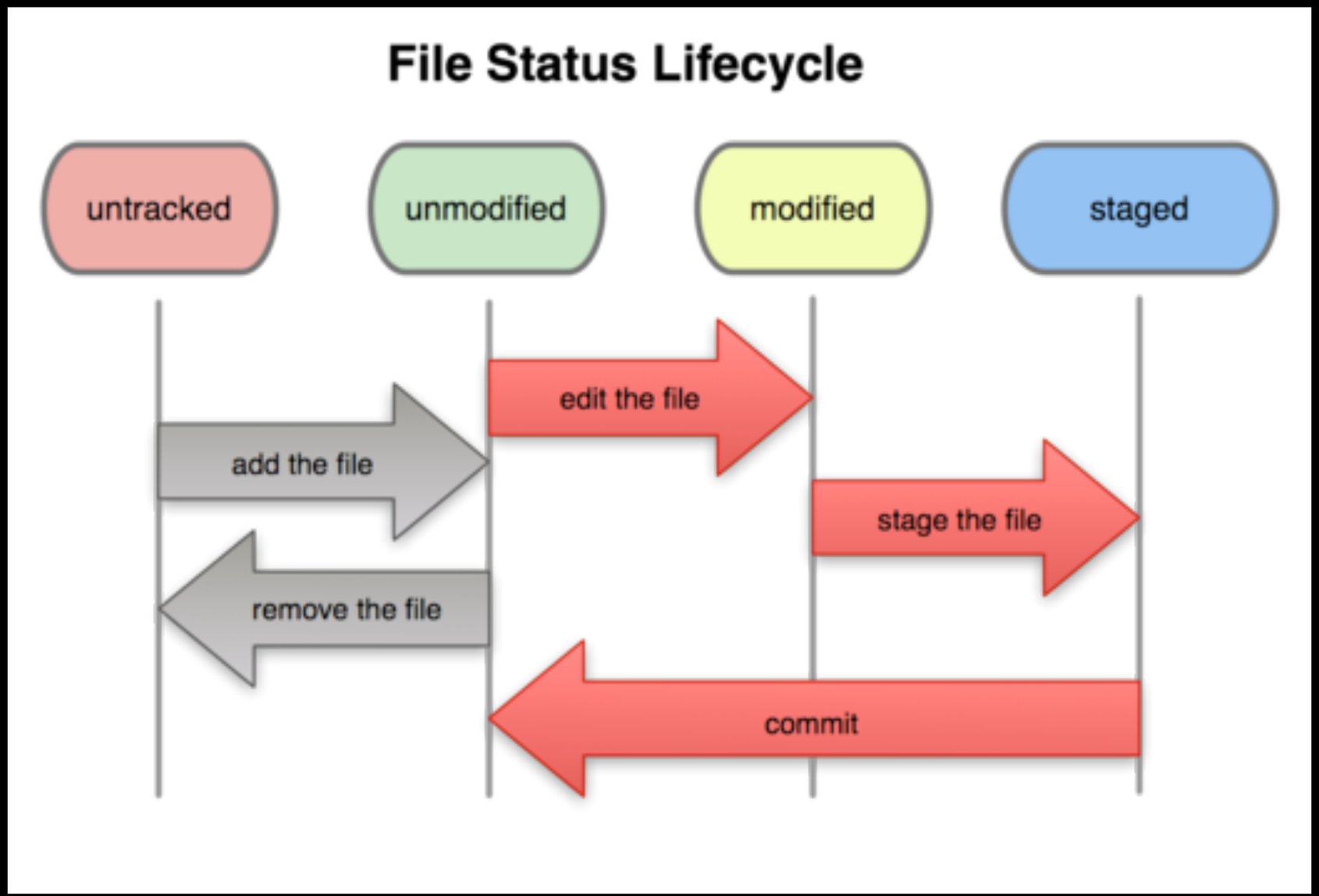
- Everyone's 'copy' is a complete repository
- Git stores snapshots of entire repo, not just file changes
- Every change (commit) is based on the entire history up to that point.
- Works with standard directories
- Add, stage, commit then push.

Commit model



Credit: Scott Chacon, Pro Git, http://progit.org/book/ch1-3.html

Commit model



Credit: Scott Chacon, Pro Git, http://progit.org/book/ch2-2.html

Basic workflow

Create a repository

```
$ git init
```

...then create a file...

```
$ git add test.txt
$ git commit -m “Added a test file with one line"
```

Here’s the output:

```
[master (root-commit) 48474ae] Added a test file
with one line
 1 files changed, 1 insertions(+), 0 deletions(-)
 create mode 100644 test.txt
```

Basic workflow

Let's make another commit. Make another change and then:

```
$ git add test.txt
$ git commit -m "Fixed a bug"
```

Or for short, let's do it in one line:

```
$ git commit -am "Fixed a bug"
```

Here's the output:

```
$ [master 95da18b] Fixed a bug
 1 files changed, 2 insertions(+), 1 deletions(-)
```

Basic workflow

Let's have a look at the log:

```
$ git log
```

Here's the output:

```
commit 95da18b765dae4b9e603d313302ad9e5cd42ede2
Author: Graeme West <graeme@domain.net>
Date:   Wed Aug 31 23:08:08 2011 +0100

    Fixed a bug

commit 48474ae8e62b0c8c02da925dd3c978b911828bc9
Author: Graeme West <graeme@domain.net>
Date:   Wed Aug 31 23:03:48 2011 +0100

    Added a test file with one line
```

About commits

- The term ‘commit’ refers to not just a change but to an entire history *and* the change
- SHA-1 checksum forms unique identifier (95da18b765dae4b9e603d313302ad9e5cd42ede2)
- Commits have ‘parents’
  - Usually one - the predecessor commit
  - Sometimes two - e.g. merge commits

Branching

Let's see where we are first:

```
$ git status
```

Here's the output:

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

Let's list all branches:

```
$ git branch
```

Here's the output:

```
* master
```

Branching

Summary:

- Our working copy is ‘clean’
- We’ve committed all changes
- There is only one branch - the default ‘master’ branch

Branching

Create a branch

```
$ git checkout -b “experiment”
```

Here’s the output:

```
Switched to a new branch 'experiment'
```

Let’s take a look at the branch:

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

Branching

Let's list all branches again:

```
$ git branch
```

Here's the output:

```
* experiment
  master
```

Make a change to this branch then commit:

```
$ git commit -m "added a line"
```

Branching

Summary:

- We've made a change on a branch called 'experiment' that does not exist in the 'master' branch
- But what if the 'master' branch changes while we're working?
- Answer: 'rebase'.

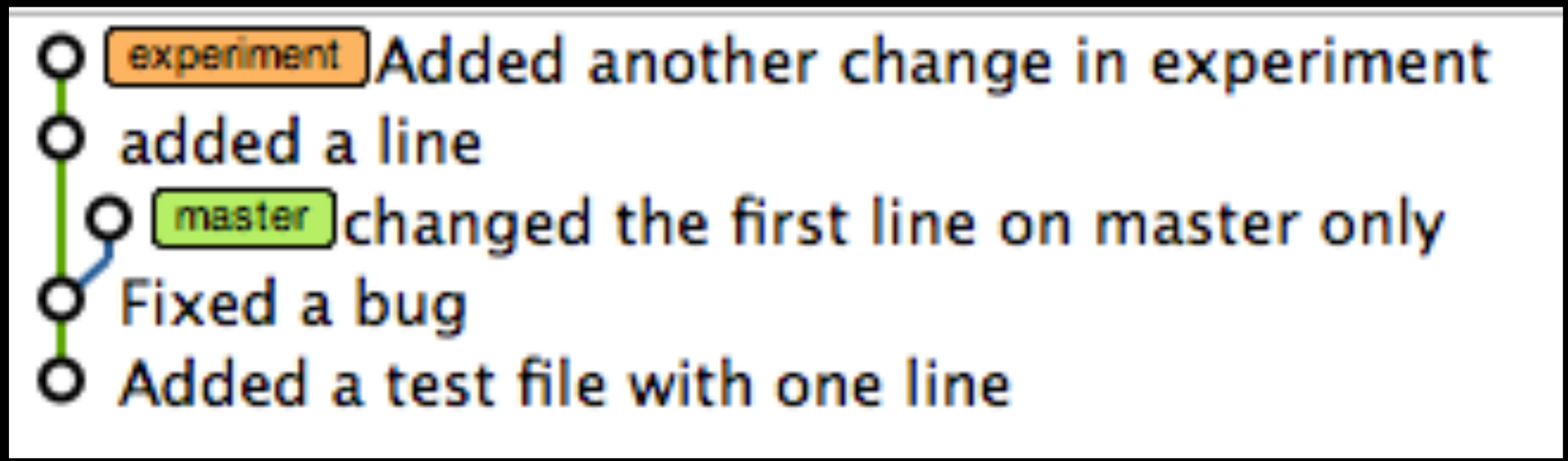
Branching

Make a change on master, then switch to 'experiment':

```
$ git checkout master
```
(change our file)

```
$ git commit -am "changed the first line on
master only"

$ git checkout experiment
```

Branching

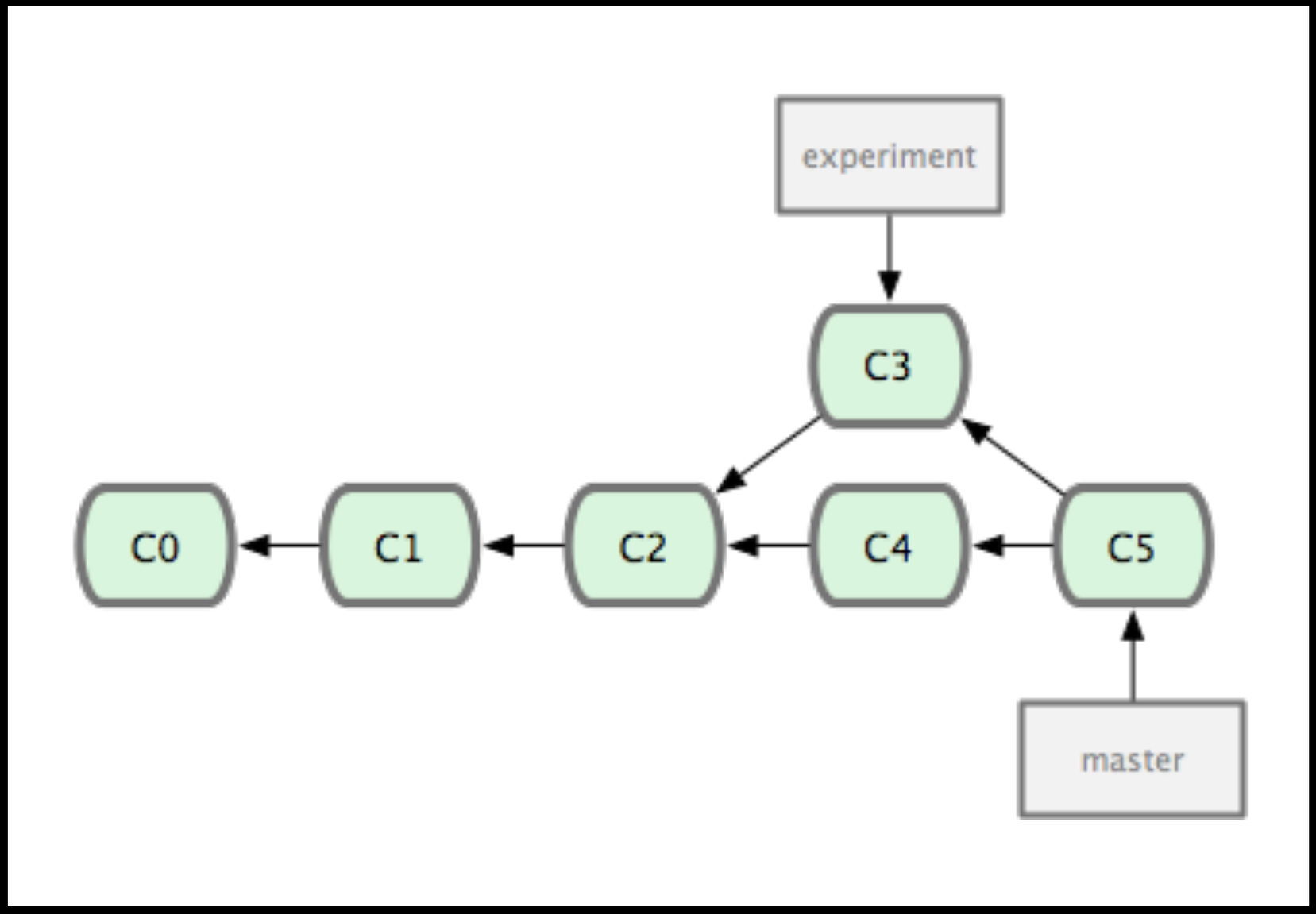


- Diverging histories in each branch
- We'll bring in the changes from 'master' into 'experiment' using 'git rebase'
- We could just 'merge'. Git deals with this gracefully, detecting potential conflicts

Rebasing

- Instead, let's bring in the changes from 'master' and 'rebase' them into 'experiment'.
- This means
  - rolling back the repo to the point of divergence
  - Then 'playing back' the commits from 'master'
  - Then 'playing back' the commits from 'experiment' on top

Rebasing

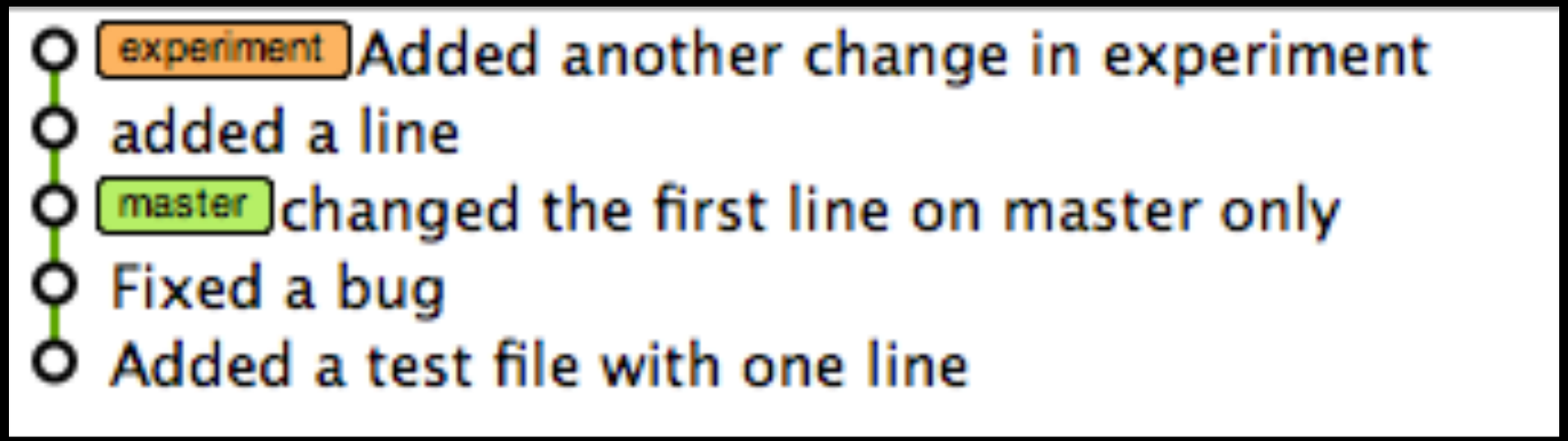


Credit: Scott Chacon, Pro Git, http://progit.org/book/ch3-6.html

Rebasing

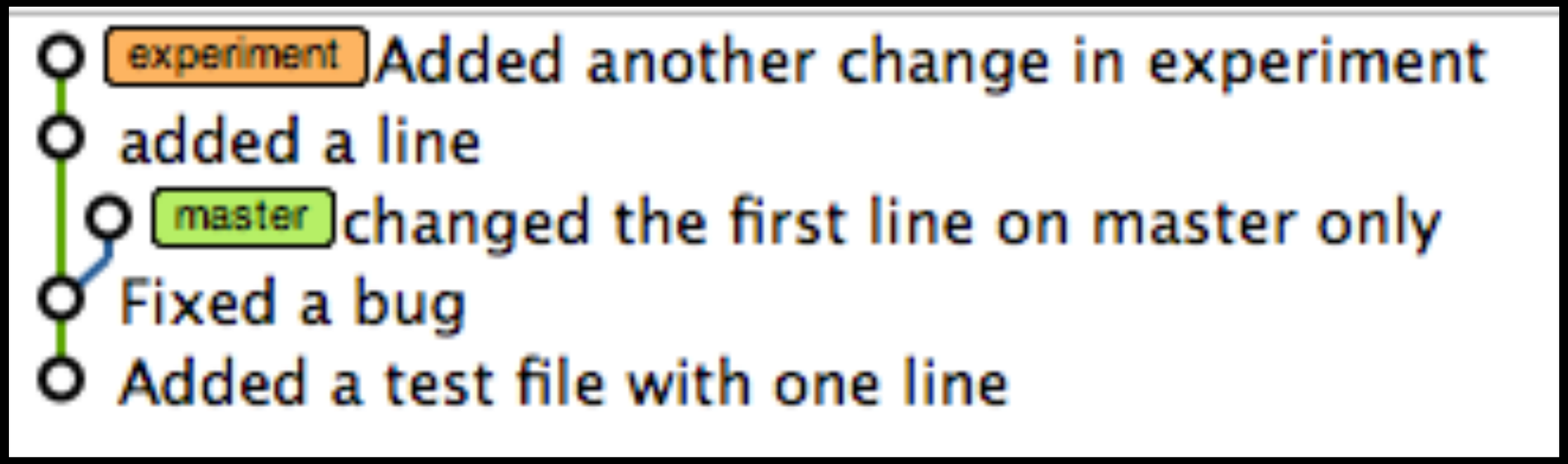
Let's do it!:

```
$ git checkout experiment

$ git rebase master
```

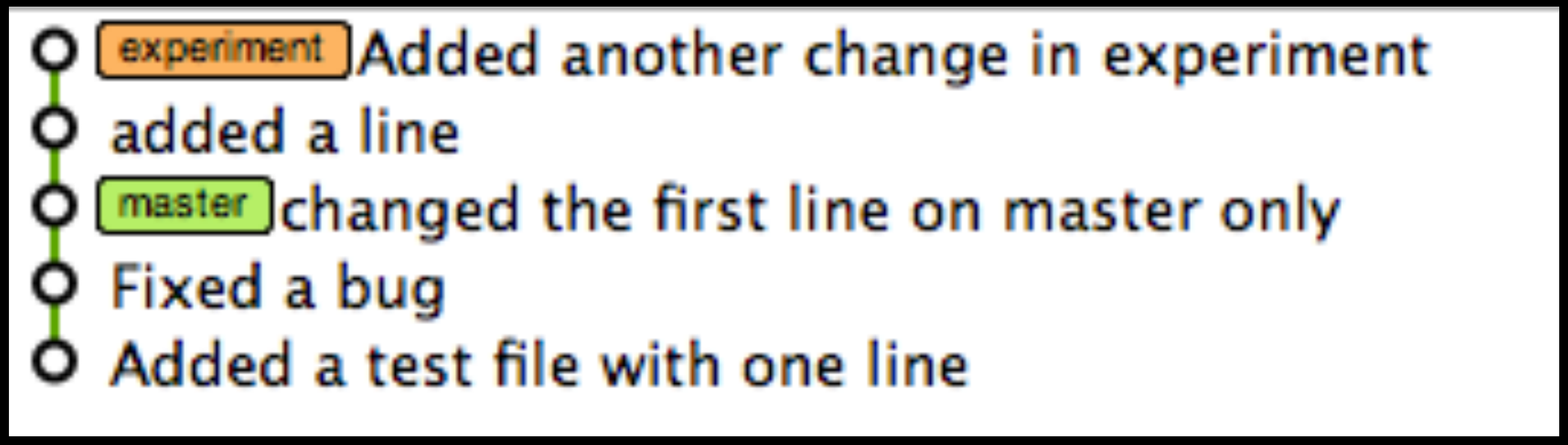


- It's as if we did everything linearly
- Enables asynchronous working

Before rebase: divergent histories



After rebase: changes from ‘experiment’ inline



Merging

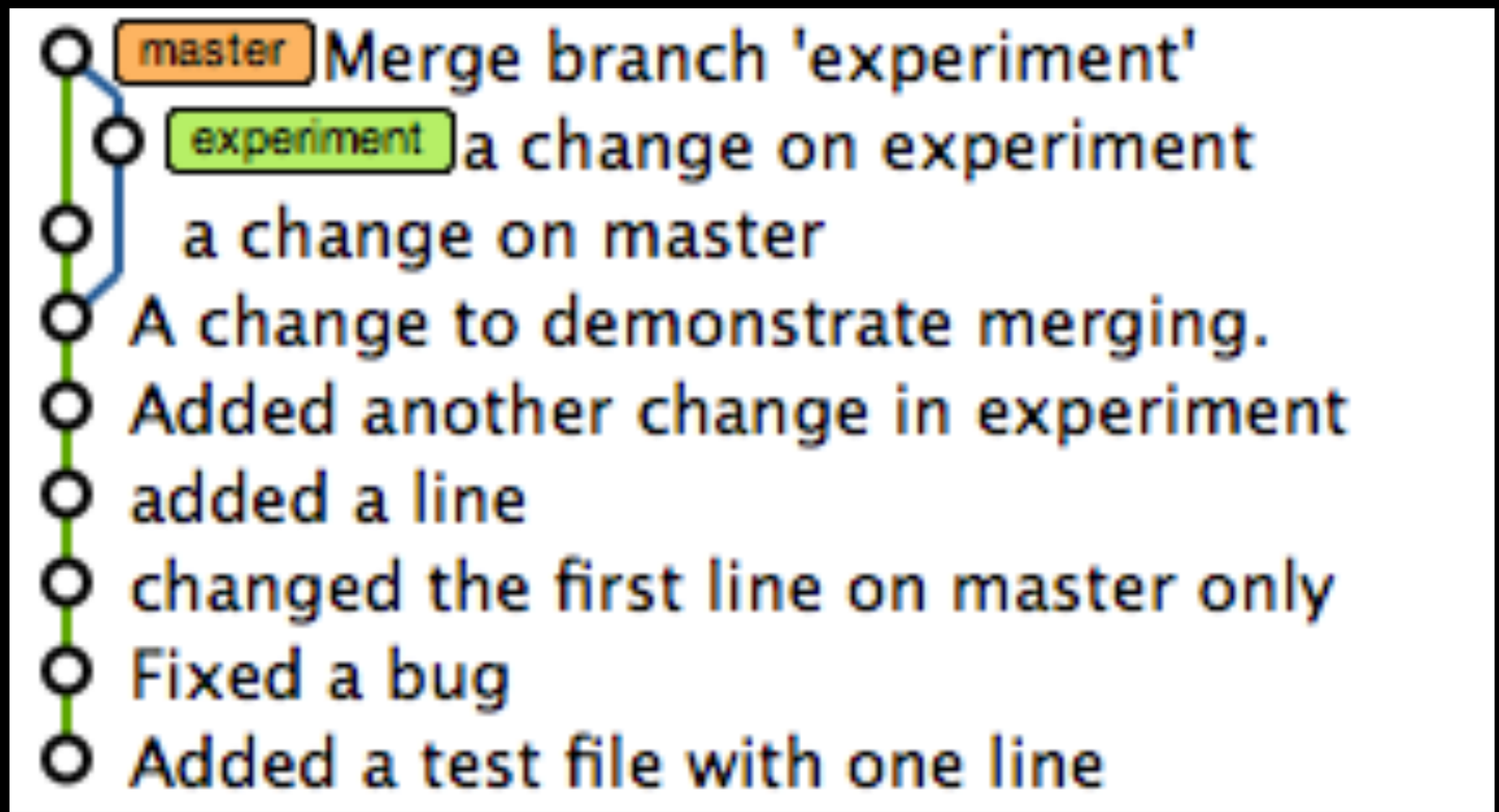
Let's do it!:

```
$ git checkout master
```
(change our file)
```
$ git commit -am "a change on master"

$ git checkout experiment
```
(change our file)
```
$ git commit -am "a change on experiment"

$ git checkout master

$ git merge experiment
```

Merging

Here's the output:

```
Auto-merging test.txt
Merge made by recursive.
 test.txt |    4 +++-
 1 files changed, 3 insertions(+), 1 deletions(-)
```



Other cool features

- Stashing: an easy way to get a clean tree
  - Or, how to drop what you're working on to fix a critical bug, and pick up right where you left off, without committing
- Partial staging: committing only part of the changes to a file
- Track a change between files
- Tagging: optionally mark releases/changes
- The reflog: your ultimate back-up

Remotes and pushing

- A ‘remote’ is another repository which you want to share code with.
- As long as they have a shared history, you can ‘push’ (send) commits
- You can have multiple remotes
  - E.g. collaboration server; Jim’s Mac; testing server; GitHub; backup server

Remotes and pushing

- ‘Pulling’ means getting and applying changes from a remote to your repo.
- ‘Pushing’ means sending a commit and its history to a remote, and fast-forwarding the remote to match that state
- Basic workflow:
  - ‘git pull‘ - to apply others’ recent changes. Solve any issues then...
  - ‘git push’ - to push your changes back

Workflow models

- CVS/SVN-style ('hub-spoke')
  - Single collab server; everyone pushes/pulls from there
- Release manager (BDFL - Linux kernel model)
  - Everyone prepares repos for manager to pull from
- Personal repos - P2P pull and pull request (GitHub model)
- ...Or any mixture of the above

Workflow models

- CVS/SVN-style with ‘push-and-push’ automatic replication
- Benefits: simple workflow with continuous integration and/or backup
- Implementation: post-receive hook with ‘git push deployserver master’
- Beware permissions

Workflow models - how we do it

- CVS/SVN-style central server with:
  - pushing primarily over SSH
  - ‘push-and-push’ automatic off-site replication, plus:
  - Daily deploy script which tags releases back to central Git (e.g. ‘daily_2011-10-04’)
  - Production servers get clones
  - Redmine for issue management
  - ### Email notifications for pushes

Git in the real world

- GitHub
  - Great interface; issues system, pull requests and OCTOCATS!!!
  - Huge community

- BitBucket
  - Unlimited private repos; Git & Mercurial support

- Gitorious
  - AGPL licensed; clean & simple

GITORIOUS

Nyancat Octocat by Cameron McEfee. The BitBucket logo is a trademark of Atlassian Pty Ltd. The Gitorious logo is a trademark of Gitorious AS.

Workflow models - how we do it

- CVS/SVN-style central server with:
  - pushing primarily over SSH
  - ‘push-and-push’ automatic off-site replication, plus:
  - Daily deploy script which tags releases back to central Git (e.g. ‘daily_2011-10-04’)
  - Production servers get clones
  - Redmine for issue management
  - ### Email notifications for pushes

Problems encountered - technical

- Lack of RAM for initial clone operations
  - No easy solution (either assign more swap, buy more RAM or delete stuff)
- Lack of partial checkout means penalty on monolithic repos
- Permissions can be fiddly (Gitosis worth a look)
- Tags don't offer metadata fields - just a name and a comment string
- Stashes cannot be pushed. Patches instead.

Problems encountered - meat-based

- Some commands have non-intuitive names
- Developer trepidation: holdover from irreversible SVN operations
- Getting people to see ‘add, stage, commit, push’ workflow as an enabler/time-saver
- Peer-to-peer pushing requires machines to be available; presents code visibility issues
- No more going home after the broadband goes down :(

What we'd do differently + future projects

- Migrate sooner!
- Move production deployment system entirely to Git
- Check production deployments back to a repo for debugging purposes
- Start retrospectively tagging releases
- GPG tag releases for authenticity & security
- Developer VMs set up as remotes to their own repos - local testing with consistent environment

Migration - from SVN

1. Move clients/developers to Git first
2. Shut down writes to SVN except for one Git collaboration server
3. Prepare an .svnauthors file
4. SVN becomes a remote like any other.
5. Change remotes on clients
6. ‘dcommit’ any remaining changes back to SVN from collaboration server
7. Remove collab server’s Git remote

Migration - from CVS

- WHY ARE YOU STILL USING CVS?
- SEE A PSYCHIATRIST
- Use ‘cvs2git’ - YMMV

Further reading

- GitReady.com
  - Tutorials at every level
- Git book - clone it!
  - http://book.git-scm.com/
- GitHub help
  - http://help.github.com/
- SVN --> Git Crash Course
  - http://git.or.cz/course/svn.html

```
$ git commit -m “thanks for listening”
```

To get this presentation, run:

```
$ git clone git://github.com/
capncodewash/Git-presentation.git
```