Git: How to Clean (and Rewrite) History

Sylvain Bouveret, Grégory Mounié, Matthieu Moy

2017

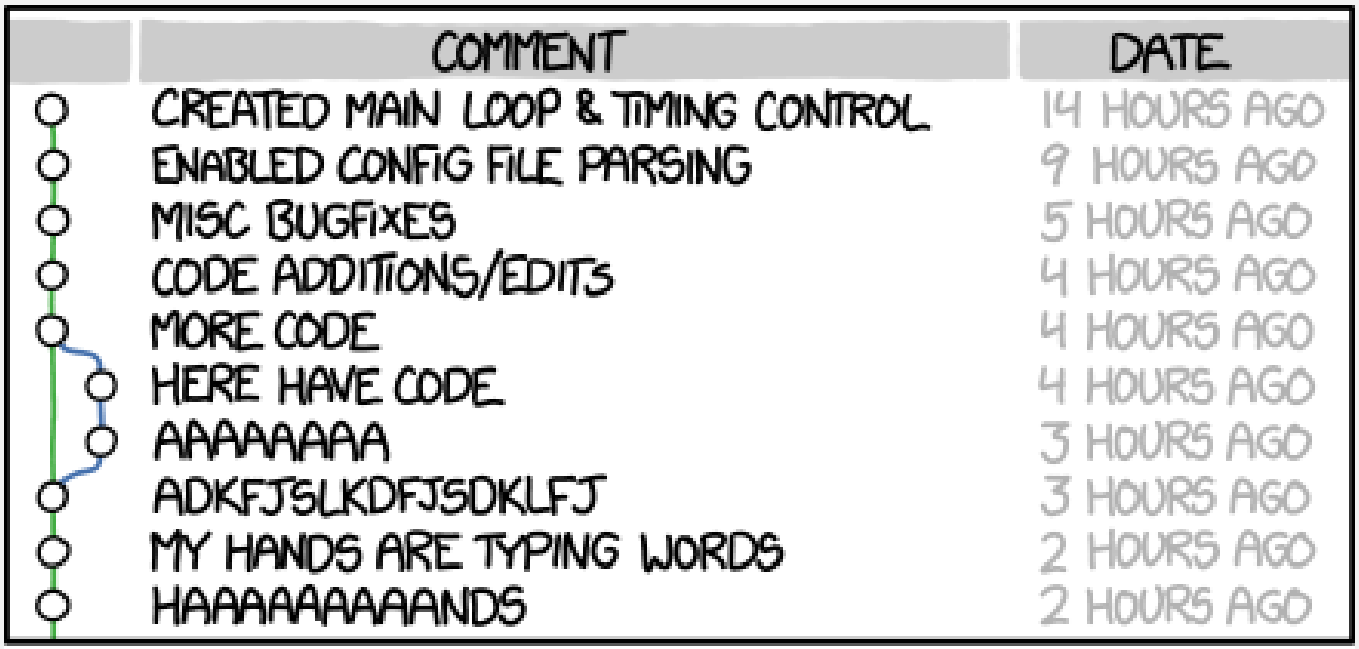
[first].[last]@imag.fr

http://recherche.noiraudes.net/resources/git/Slides/git-history-slides.pdf

Goals of the presentation

- Explaining why having a clean history is important
- Showing how to use the index
- Introducing the mechanisms that git provides to deal with history

Clean History: Why?


COMMENT
DATE
CREATED MAIN LOOP & TIMING CONTROL
14 HOURS AGO
ENABLED CONFIG FILE PARSING
9 HOURS AGO
MISC BUGFIXES
5 HOURS AGO
CODE ADDITIONS/EDITS
4 HOURS AGO
MORE CODE
4 HOURS AGO
HERE HAVE CODE
4 HOURS AGO
AAAAAAAA
3 HOURS AGO
ADKFJSLKDFJSDKLFJ
3 HOURS AGO
MY HANDS ARE TYPING WORDS
2 HOURS AGO
HAAAAAAAAANDS
2 HOURS AGO
AS A PROJECT DRAGS ON, MY GIT COMMIT MESSAGES GET LESS AND LESS INFORMATIVE.

Git blame: Who did that?

Grenoble INP ensimag

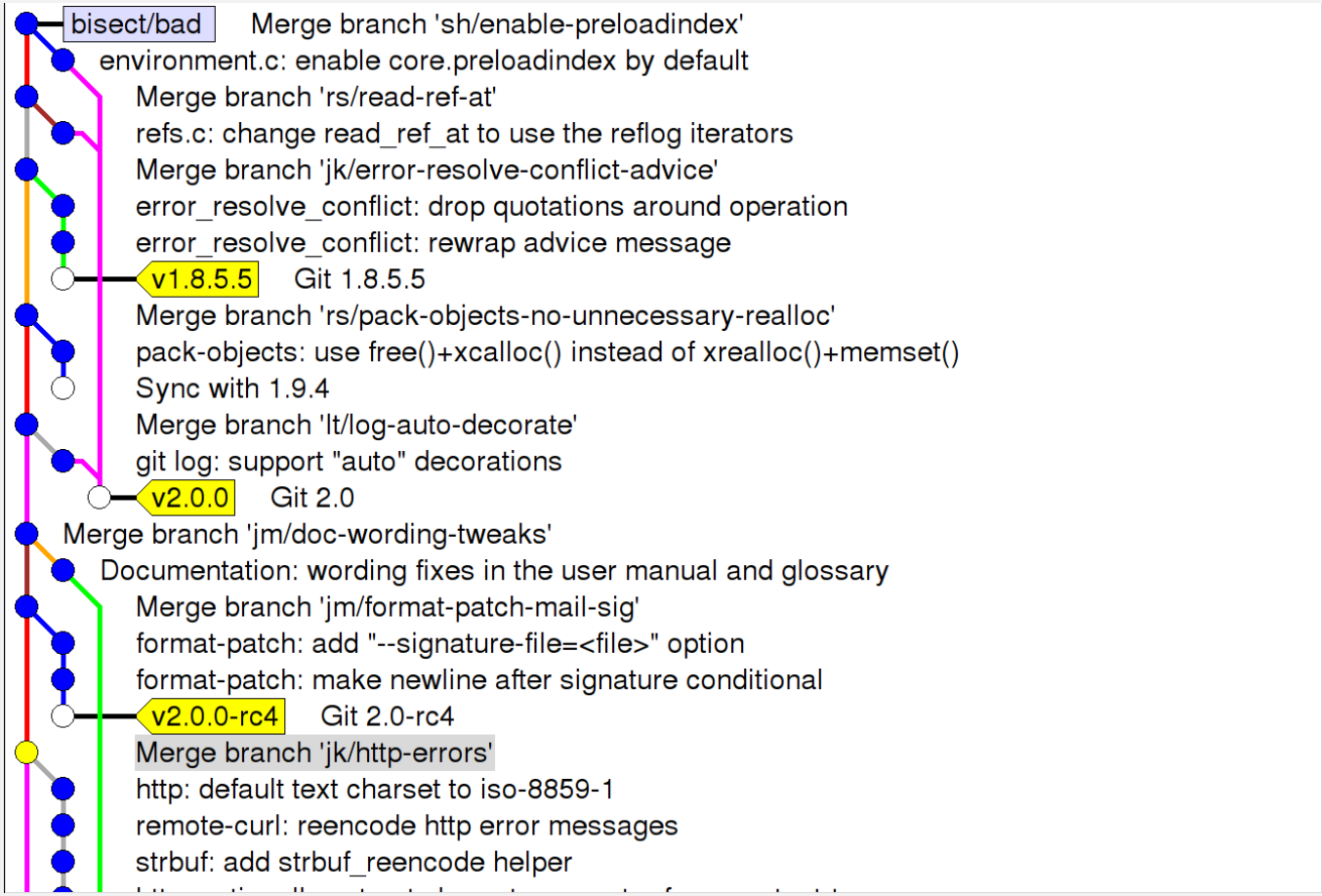
`git gui blame` *`file`*

Bisect: Find regressions

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.9.0
Bisecting: 607 revisions left to test after this (roughly 9 steps)
[8fe3ee67adcd2ee9372c7044fa311ce55eb285b4] Merge branch 'jx/i18n'
$ git bisect good
Bisecting: 299 revisions left to test after this (roughly 8 steps)
[aa4bffa23599e0c2e611be7012ecb5f596ef88b5] Merge branch 'jc/cod[...]
$ git bisect good
Bisecting: 150 revisions left to test after this (roughly 7 steps)
[96b29bde9194f96cb711a00876700ea8dd9c0727] Merge branch 'sh/ena[...]
$ git bisect bad
Bisecting: 72 revisions left to test after this (roughly 6 steps)
[09e13ad5b0f0689418a723289dca7b3c72d538c4] Merge branch 'as/pre[...]
...
$ git bisect good
60ed26438c909fd273528e67 is the first bad commit
commit 60ed26438c909fd273528e67b399ee6ca4028e1e
```

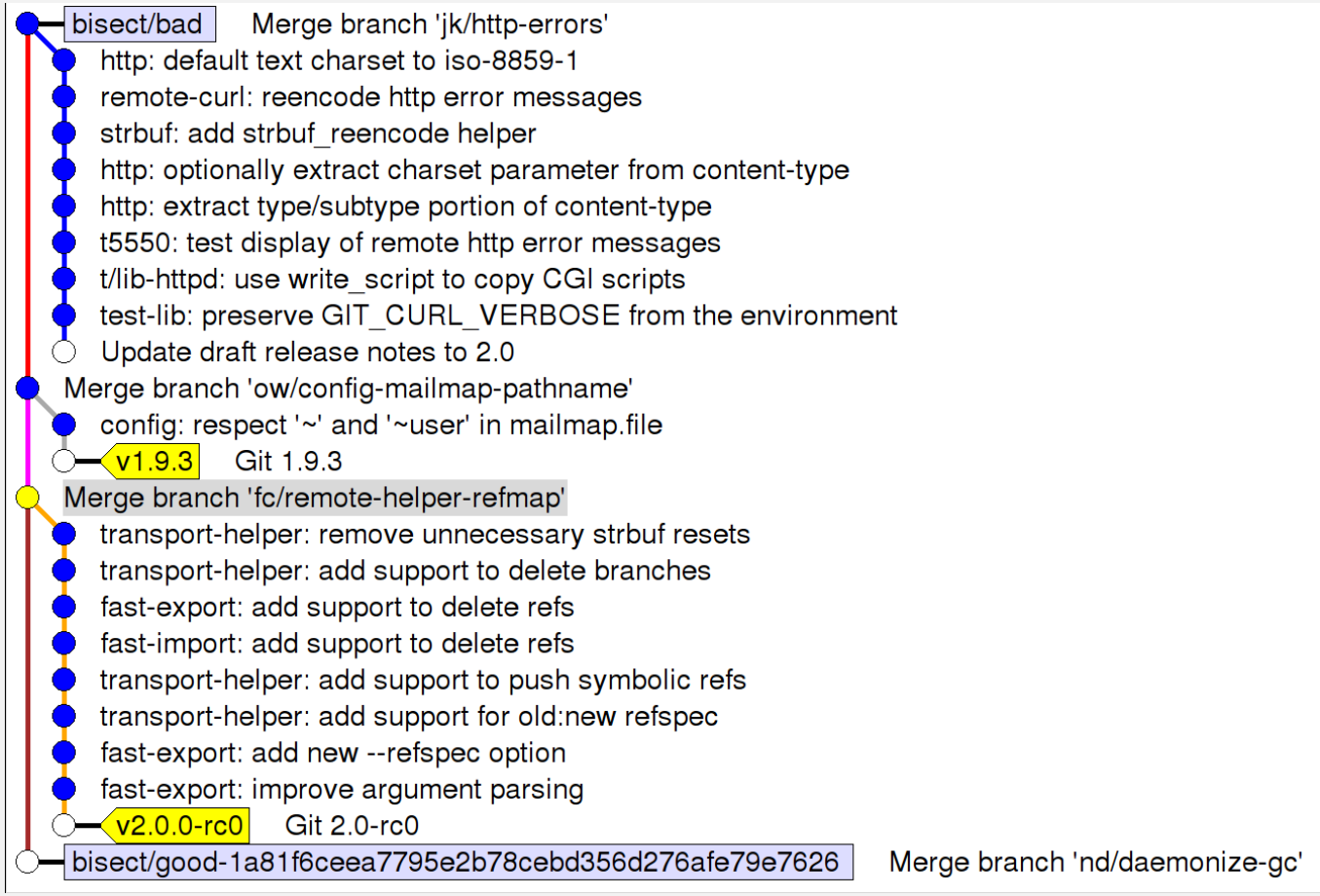
Bisect: Binary search

`git bisect visualize`



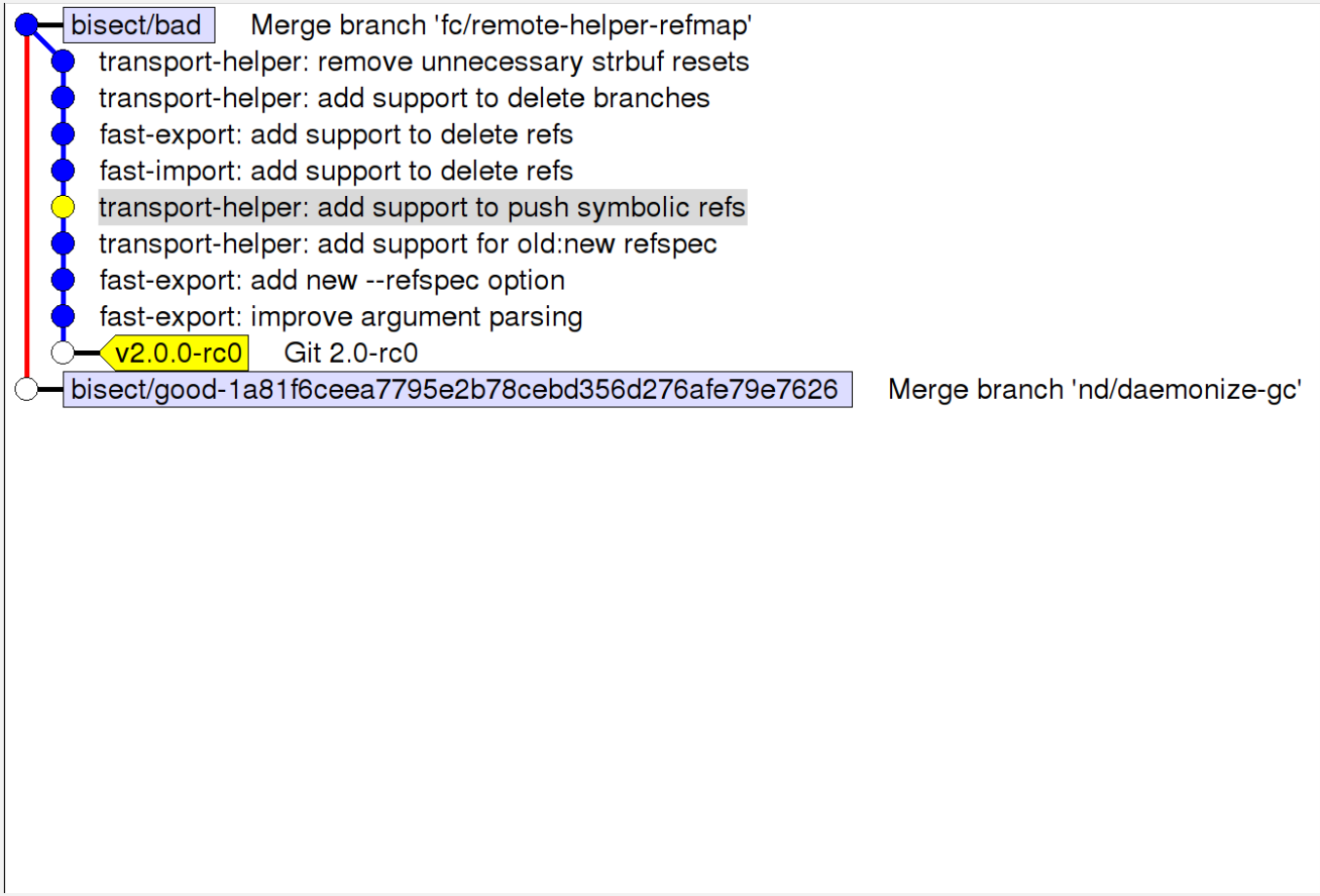
Bisect: Binary search

`git bisect visualize`



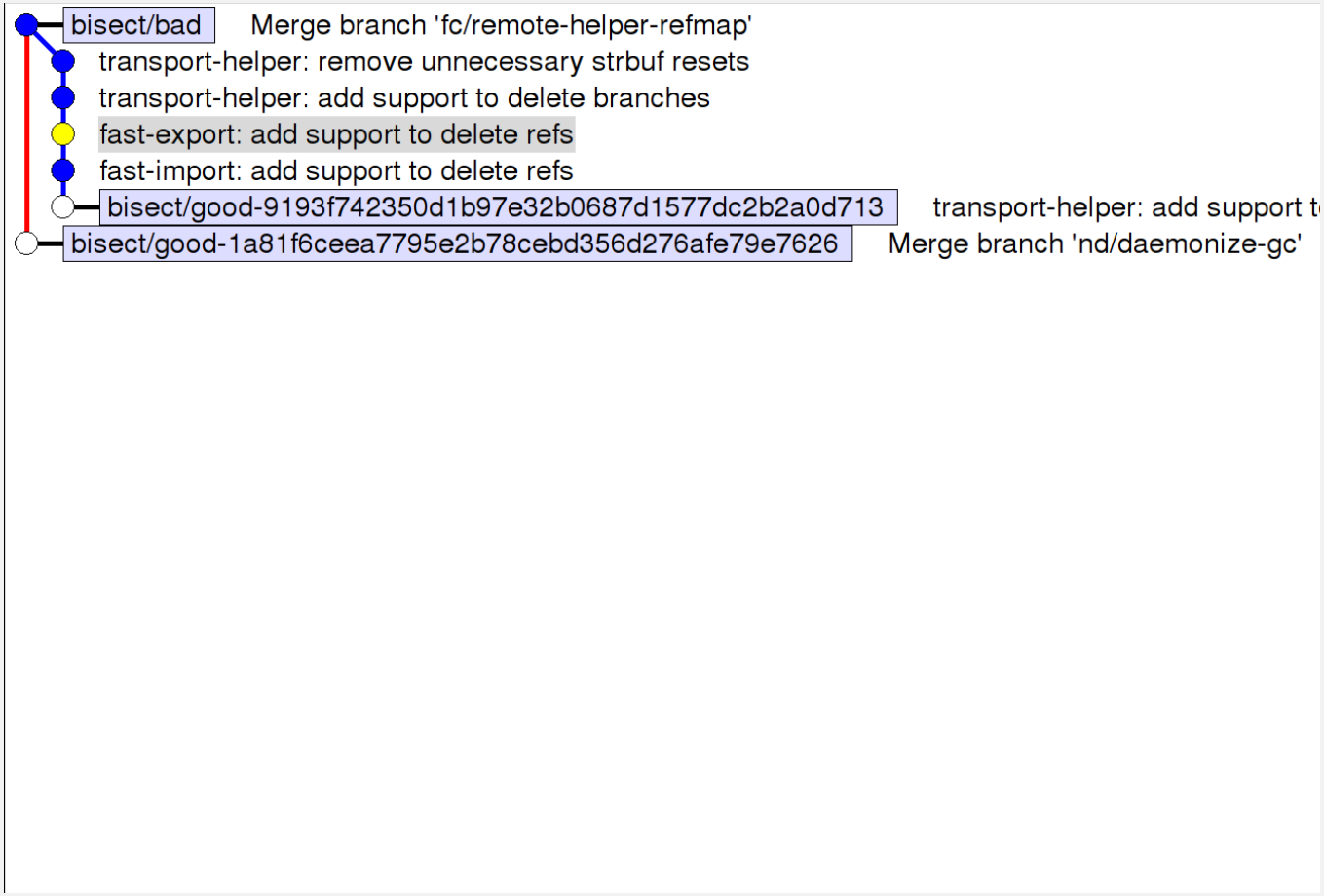
Bisect: Binary search

`git bisect visualize`



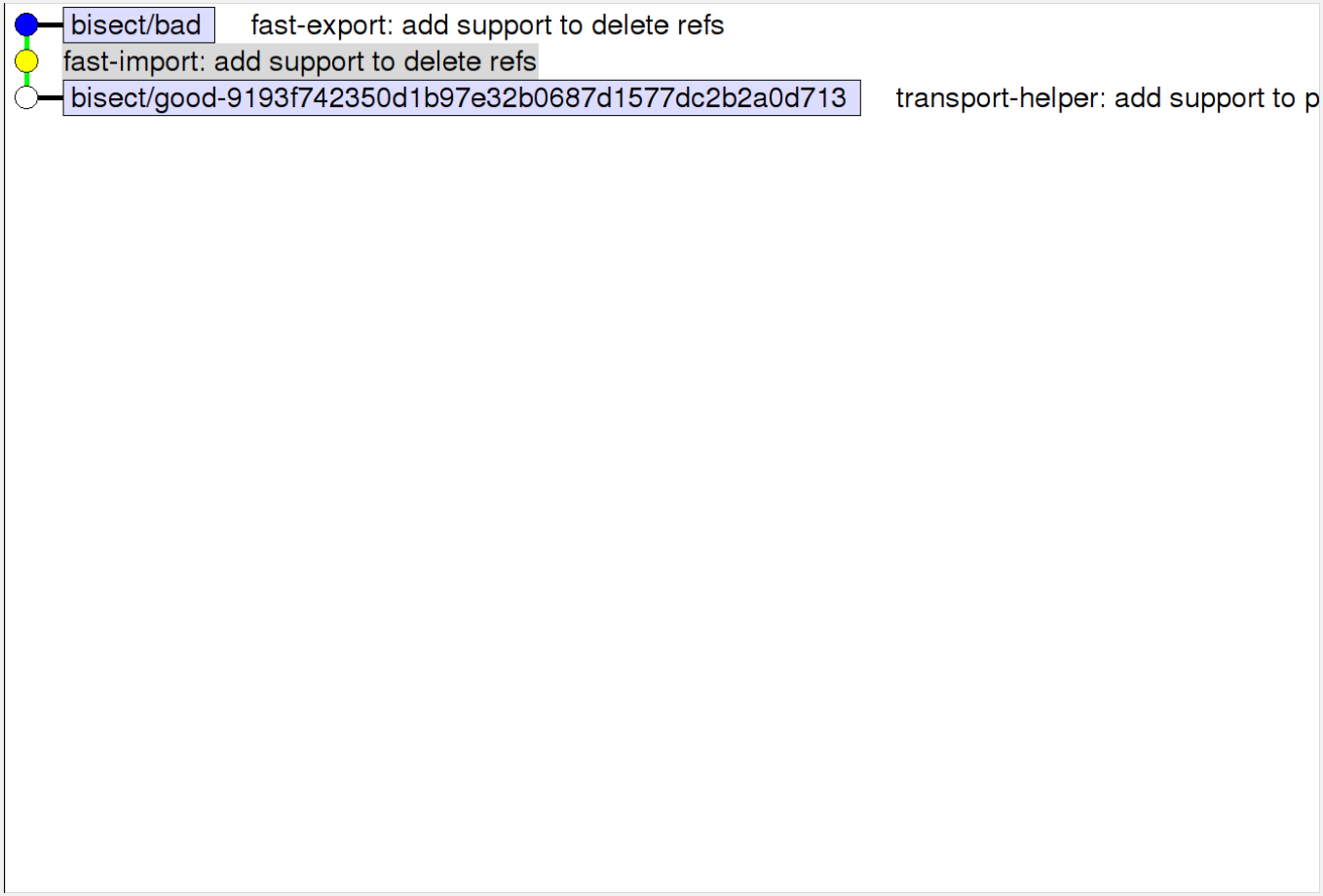
Bisect: Binary search

`git bisect visualize`



Bisect: Binary search

`git bisect visualize`



Then what?

`git blame` and `git bisect` point you to a commit, then ...

- Dream:
  - Commit is a 50-lines long patch
  - Commit message explains the intent of the programmer
- Nightmare 1:
  - Commit mixes a large reindentation, a bugfix and a real feature
  - Message says “I reindented, fixed a bug and added a feature”
- Nightmare 2:
  - Commit is a trivial fix for the previous commit
  - Message says “Oops, previous commit was stupid”
- Nightmare 3:
  - Bisect not even applicable because most commits aren't compilable.

Then what?

Which one do you prefer?

Then what?

Clean history is important for software maintainability

Then what?

Clean history is as important as comments for software maintainability

Two Approaches To Deal With History

Approach 1

"Mistakes are part of history."

Approach 2

"History is a set of lies agreed upon."[1]

[1] Napoleon Bonaparte

Approach 1: Mistakes are part of history

- $\approx$ the only option with Subversion/CVS/...
- History reflects the chronological order of events
- Pros:
  - Easy: just work and commit from time to time
  - Traceability
- But ...
  - Is the actual order of event what you want to remember?
  - When you write a draft of a document, and then a final version, does the final version reflect the mistakes you did in the draft?

Approach 2: History is a set of lies agreed upon

- Popular approach with modern VCS (Git, Mercurial. . .)
- History tries to show the best logical path from one point to another
- Pros:
  - See above: blame, bisect, ...
  - Code review
  - Claim that you are a better programmer than you really are!

Another View About Version Control

- 2 roles of version control:
  - For beginners: help the code reach upstream.
  - For advanced users: prevent bad code from reaching upstream.
- Several opportunities to reject bad code:
  - Before/during commit
  - Before push
  - Before merge

What is a clean history

- Each commit introduce small group of related changes ($\approx$ 100 lines changed max, no minimum!)
- Each commit is compilable and passes all tests ("bisectable history")
- "Good" commit messages

Writing good commit messages

Reminder: good comments

- Bad:

```
1  int i; // Declare i of type int
2  for (i = 0; i < 10; i++) { ... }
3  f(i)
```

- Possibly good:

```
1  int i; // We need to declare i outside the for
2         // loop because we'll use it after.
3  for (i = 0; i < 10; i++) { ... }
4  f(i)
```

Reminder: good comments

- Bad: What? The code already tells

```
1  int i; // Declare i of type int
2  for (i = 0; i < 10; i++) { ... }
3  f(i)
```

- Possibly good: Why? Usually the relevant question

```
1  int i; // We need to declare i outside the for
2         // loop because we'll use it after.
3  for (i = 0; i < 10; i++) { ... }
4  f(i)
```

Reminder: good comments

- Bad: What? The code already tells

```
1  int i; // Declare i of type int
2  for (i = 0; i < 10; i++) { ... }
3  f(i)
```

- Possibly good: Why? Usually the relevant question

```
1  int i; // We need to declare i outside the for
2         // loop because we'll use it after.
3  for (i = 0; i < 10; i++) { ... }
4  f(i)
```

Common rule: if your code isn't clear enough,
rewrite it to make it clearer
instead of adding comments.

Good commit messages

- Recommended format:

```
One-line description (< 50 characters)

Explain here why your change is good.
```

- Write your commit messages like an email: subject and body
- Imagine your commit message is an email sent to the maintainer, trying to convince him to merge your code[2]
- Don't use `git commit -m`

[2] Not just imagination, see `git send-email`

Good commit messages: examples

https://github.com/git/git/commit/b1b49ff8d42a21ade6a72b40a147fd3eaff3db8d

daemon: plug memory leak

Call child_process_clear() when a child ends to release the memory allocated for its environment. This is necessary because unlike all other users of start_command() we don't call finish_command(), which would have taken care of that for us.

This leak was introduced by f063d38 (daemon: use cld->env_array when re-spawning).

Signed-off-by: Rene Scharfe <l.s.r@web.de>
Signed-off-by: Junio C Hamano <gitster@pobox.com>

Good commit messages: counter-example

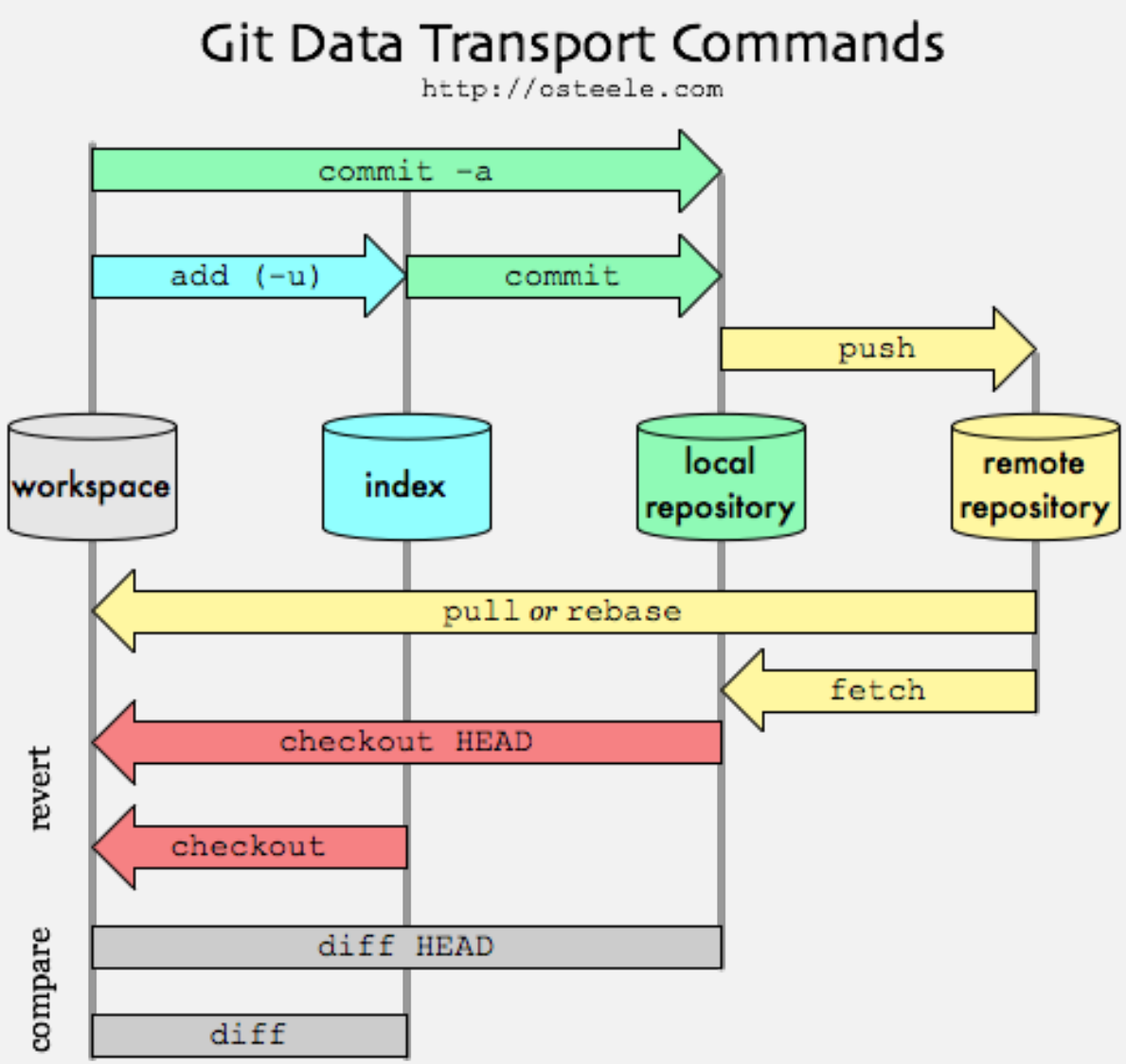
http://git.savannah.gnu.org/cgit/emacs.git/commit/?id=19e09cfab61436cb4590303871a31ee07624f5ab

```
Ensure redisplay after evaluation

* lisp/progmodes/elisp-mode.el (elisp-eval-last-sexp): Revert
last change.
* lisp/frame.el (redisplay-variables): Populate the
redisplay-variables list.
* src/xdisp.c (maybe_set_redisplay): New function.
(syms_of_xdisp) <redisplay-variables>: New variable.
* src/window.h (maybe_set_redisplay): Declare prototype.
* src/data.c (set_internal): Call maybe_set_redisplay. (Bug#21835)
```

Not much the patch didn't already say ... (do you understand the problem the commit is trying to solve?)

Partial commits, the index


Git Data Transport Commands
http://osteele.com
commit -a
add (-u)
commit
push
workspace
index
local repository
remote repository
pull or rebase
fetch
checkout HEAD
checkout
diff HEAD
diff
revert
compare

The index, or “Staging Area”

- “the index” is where the next commit is prepared
- Contains the list of files and their content
- `git commit` transforms the index into a commit
- `git commit -a` stages all changes in the worktree in the index before committing. You’ll find it sloppy soon.

Dealing with the index

- Commit only 2 files:

```
git add file1.txt
git add file2.txt
git commit
```

- Commit only some patch hunks:

```
git add -p
(answer yes or no for each hunk)
git commit
```

git add -p: example

```
$ git add -p
@@ -1,7 +1,7 @@
 int main()
-       int i;
+       int i = 0;
        printf("Hello, ");
        i++;
Stage this hunk [y,n,q,a,d,/,K,g,e,?]? y
```

git add -p: example

```
$ git add -p
@@ -1,7 +1,7 @@
 int main()
-       int i;
+       int i = 0;
        printf("Hello, ");
        i++;
Stage this hunk [y,n,q,a,d,/,K,g,e,?]? y
@@ -5,6 +5,6 @@

-       printf("i is %s\n", i);
+       printf("i is %d\n", i);

Stage this hunk [y,n,q,a,d,/,K,g,e,?]? n
```

git add -p: example

```
$ git add -p
@@ -1,7 +1,7 @@
 int main()
-       int i;
+       int i = 0;
        printf("Hello, ");
        i++;
Stage this hunk [y,n,q,a,d,/,K,g,e,?]? y
@@ -5,6 +5,6 @@

-       printf("i is %s\n", i);
+       printf("i is %d\n", i);

Stage this hunk [y,n,q,a,d,/,K,g,e,?]? n
$ git commit -m "Initialize i properly"
[master c4ba68b] Initialize i properly
 1 file changed, 1 insertion(+), 1 deletion(-)
```

`git add -p`: dangers

- Commits created with `git add -p` do not correspond to what you have on disk
- You probably never tested this commit ...
- Solutions:
  - `git stash -k`: stash what's not in the index
  - `git rebase --exec`: see later
  - (and code review)

Clean local history

Example

Implement `git clone -c var=value` : 9 preparation patches, 1 real (trivial) patch at the end!

`https://github.com/git/git/commits/84054f79de35015fc92f73ec4780102dd820e452`

Did the author actually write this in this order?

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 1: merge (default with `git pull`)

Merging With Upstream

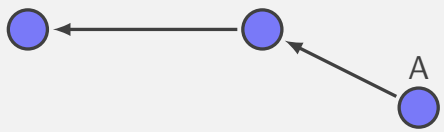
Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 1: merge (default with `git pull`)

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

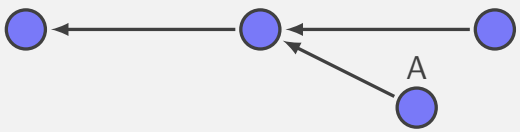
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

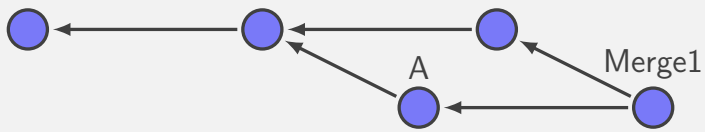
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

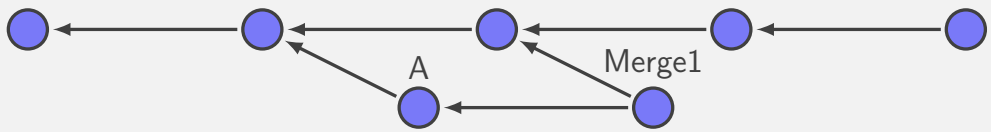
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

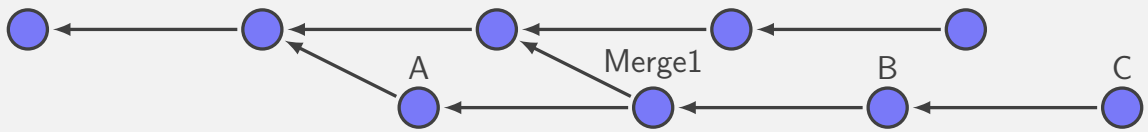
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

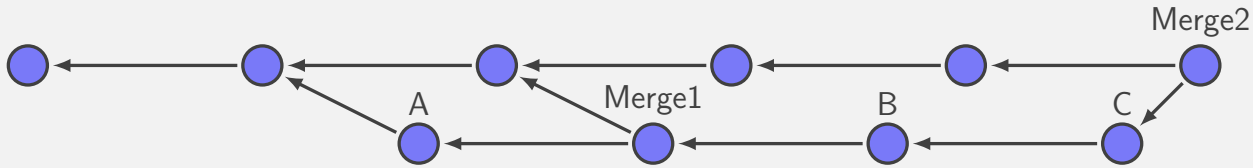
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

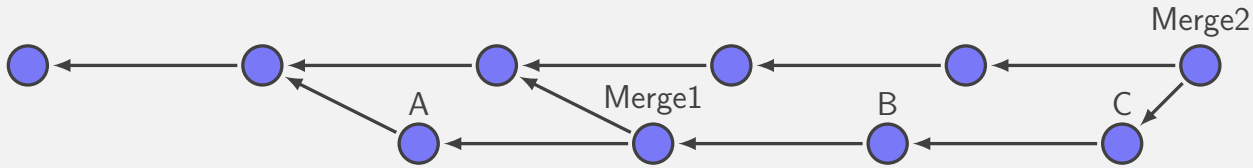
- Approach 1: merge (default with `git pull`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 1: merge (default with `git pull`)



- Drawbacks:
  - Merge1 is not relevant, distracts reviewers (unlike Merge2).

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 2: no merge

Merging With Upstream

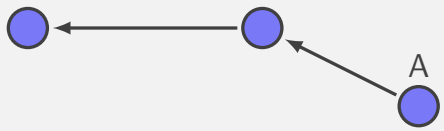
Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 2: no merge

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

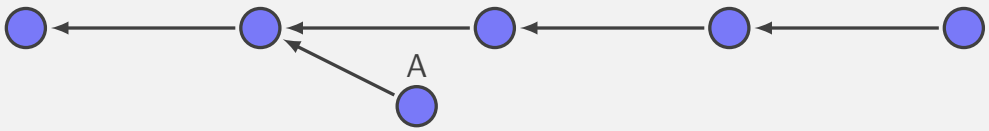
- Approach 2: no merge



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

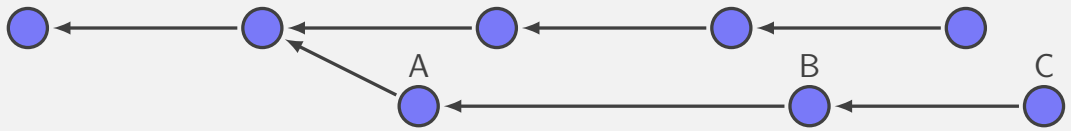
- Approach 2: no merge



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 2: no merge



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

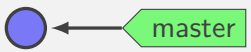
- Approach 2: no merge

- Drawbacks:
  - In case of conflict, they have to be resolved by the developer merging into upstream (possibly after code review)
  - Not always applicable (e.g. “I need this new upstream feature to continue working”)

Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

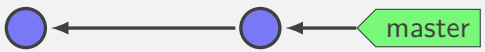
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

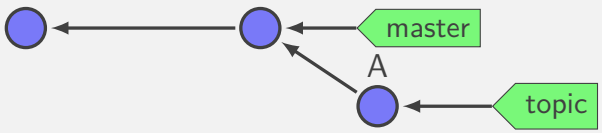
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

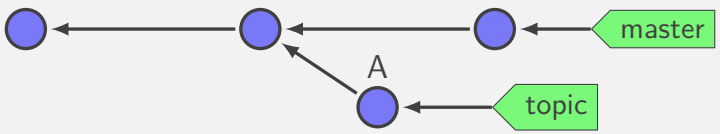
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

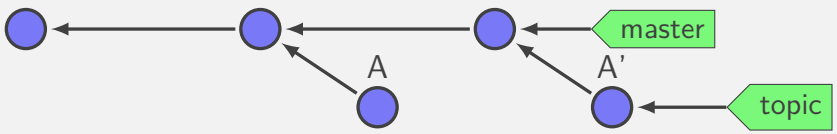
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

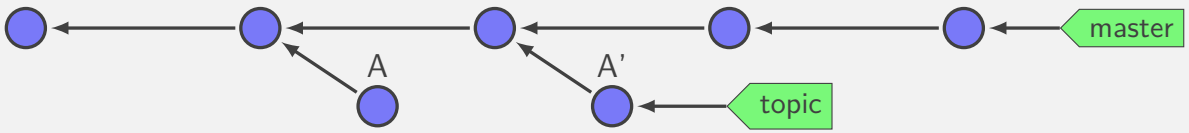
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

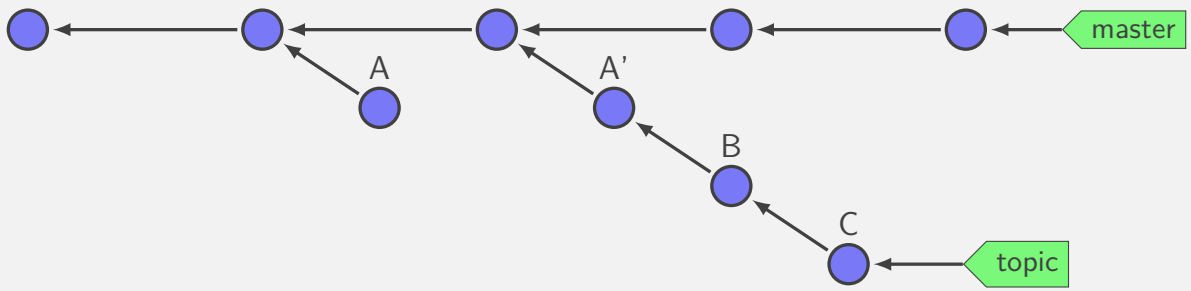
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

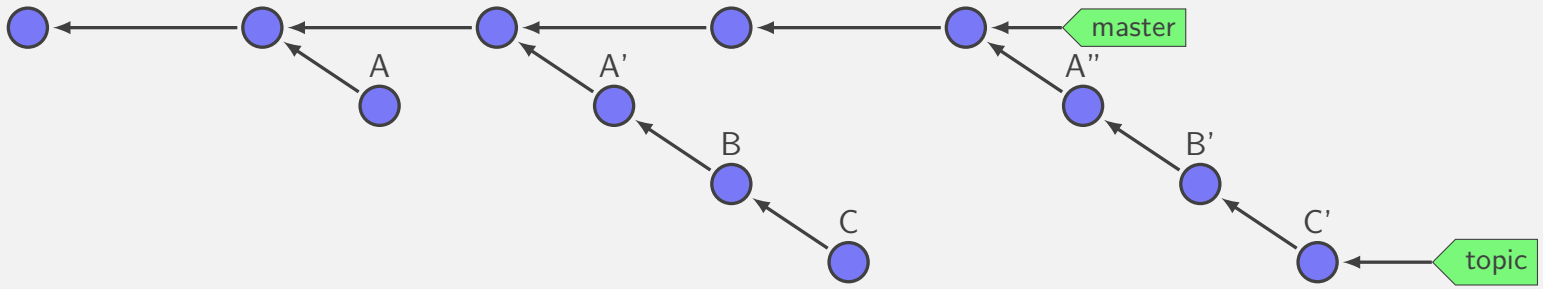
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

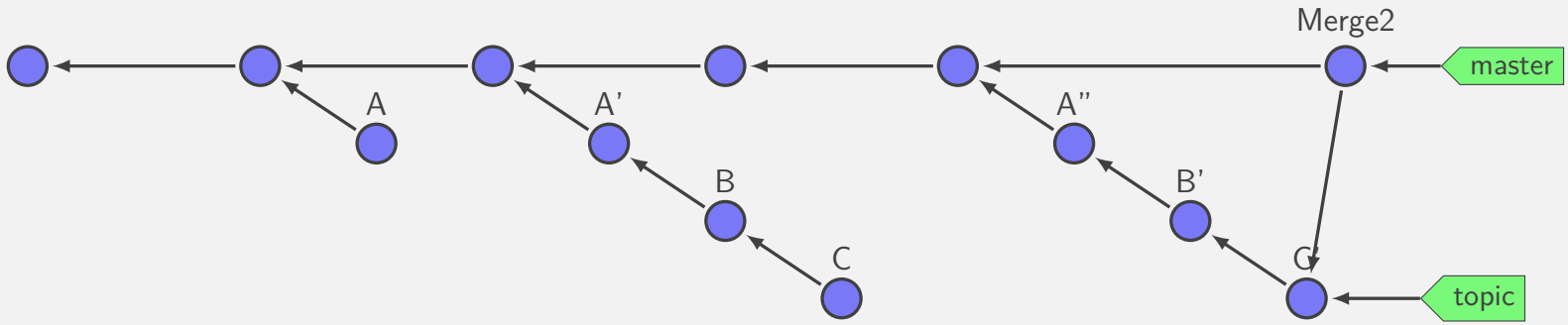
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

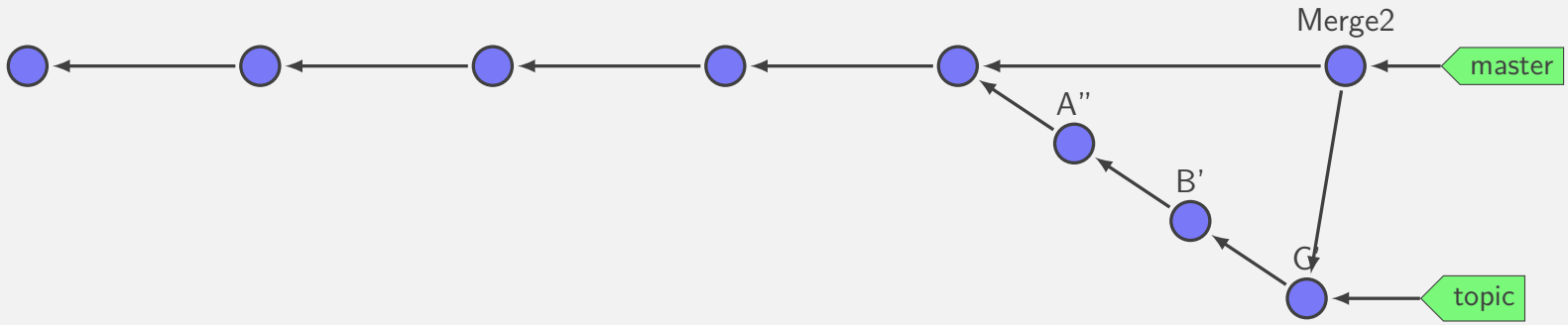
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

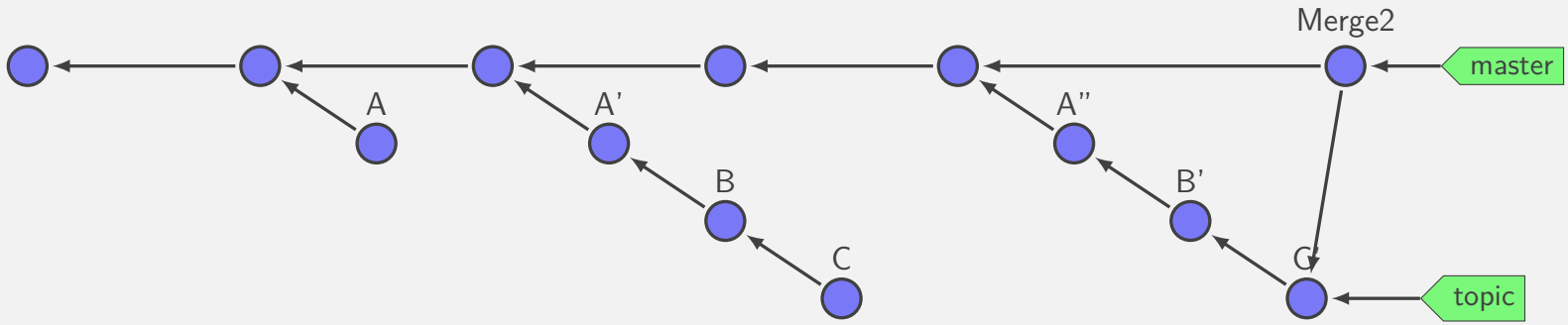
- Approach 3: rebase (`git rebase` or `git pull --rebase`)



Merging With Upstream

Question: upstream (where my code should eventually end up) has new code, how do I get it in my repo?

- Approach 3: rebase (`git rebase` or `git pull --rebase`)



- Drawbacks: rewriting history implies:
  - A', A'', B', C' probably haven't been tested (never existed on disk)
  - What if someone branched from A, A', B or C?
  - Basic rule: don't rewrite published history

Rewriting history with `rebase -i`

- `git rebase`: take all your commits, and re-apply them onto upstream
- `git rebase -i`: show all your commits, and asks you what to do when applying them onto upstream.

Rewriting history with `rebase -i`

```
pick ca6ed7a Start feature A
pick e345d54 Bugfix found when implementing A
pick c03fffc Continue feature A
pick 5bdb132 Oops, previous commit was totally buggy

# Rebase 9f58864..5bdb132 onto 9f58864
#
# Commands:
#  p, pick = use commit
#  r, reword = use commit, but edit the commit message
#  e, edit = use commit, but stop for amending
#  s, squash = use commit, but meld into previous commit
#  f, fixup = like "squash", but discard this commit's log message
#  x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#

# Note that empty commits are commented out
```

`git rebase -i` commands (1/2)

p, pick use commit (by default)

r, reword use commit, but edit the commit message

Fix a typo in a commit message

e, edit use commit, but stop for amending

⇝ Once stopped, use `git add -p`, `git commit -amend`,...

s, squash use commit, but meld into previous commit

f, fixup like "squash", but discard this commit's log message

⇝ Very useful when polishing a set of commits (before or after review): make a bunch of short fixup patches, and squash them into the real commits. No one will know you did this mistake ;-).

`git rebase -i` commands (2/2)

x, exec run command (the rest of the line) using shell

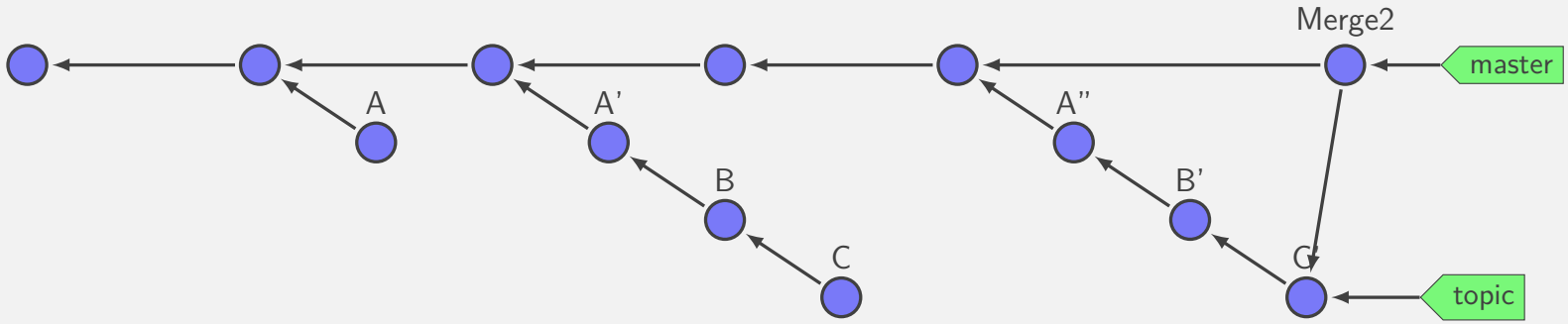
- Example: `exec make check`. Run tests for this commit, stop if test fail.
- Use `git rebase -i --exec 'make check'`[3] to run `make check` for each rebased commit.

[3] Implemented by Ensimag students!

Repairing mistakes: the reflog

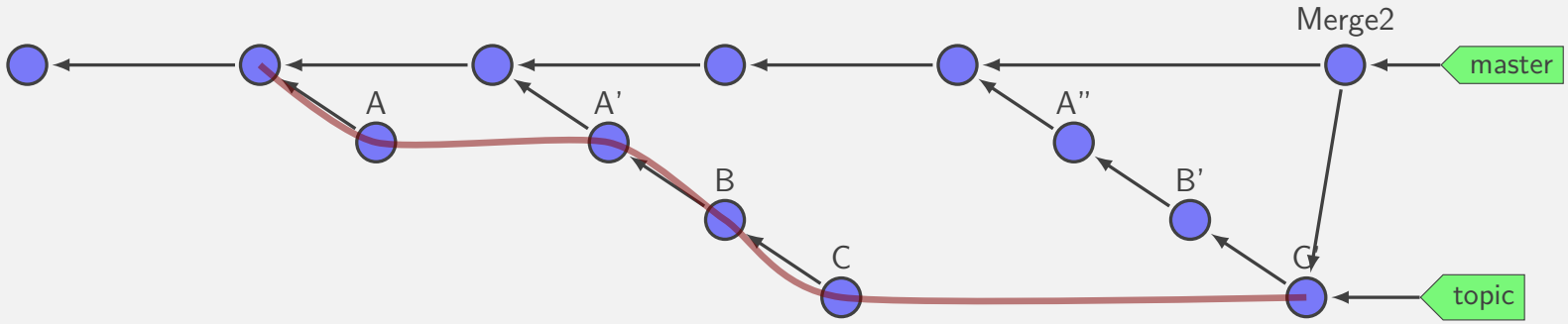
Git's reference journal: the reflog

- Remember the history of local refs.
- $\neq$ ancestry relation.



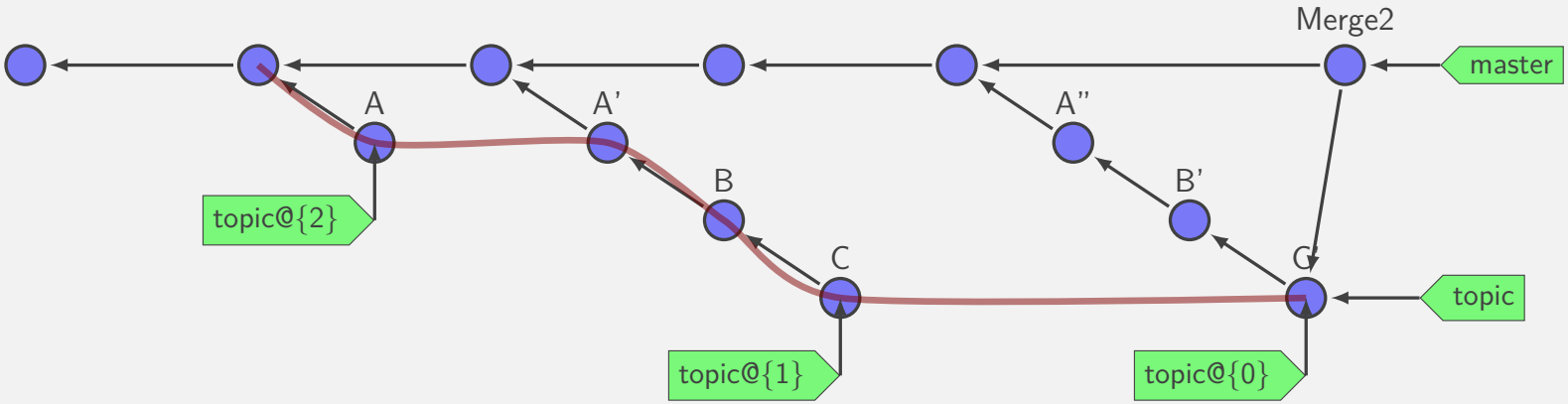
Git's reference journal: the reflog

- Remember the history of local refs.
- $\neq$ ancestry relation.



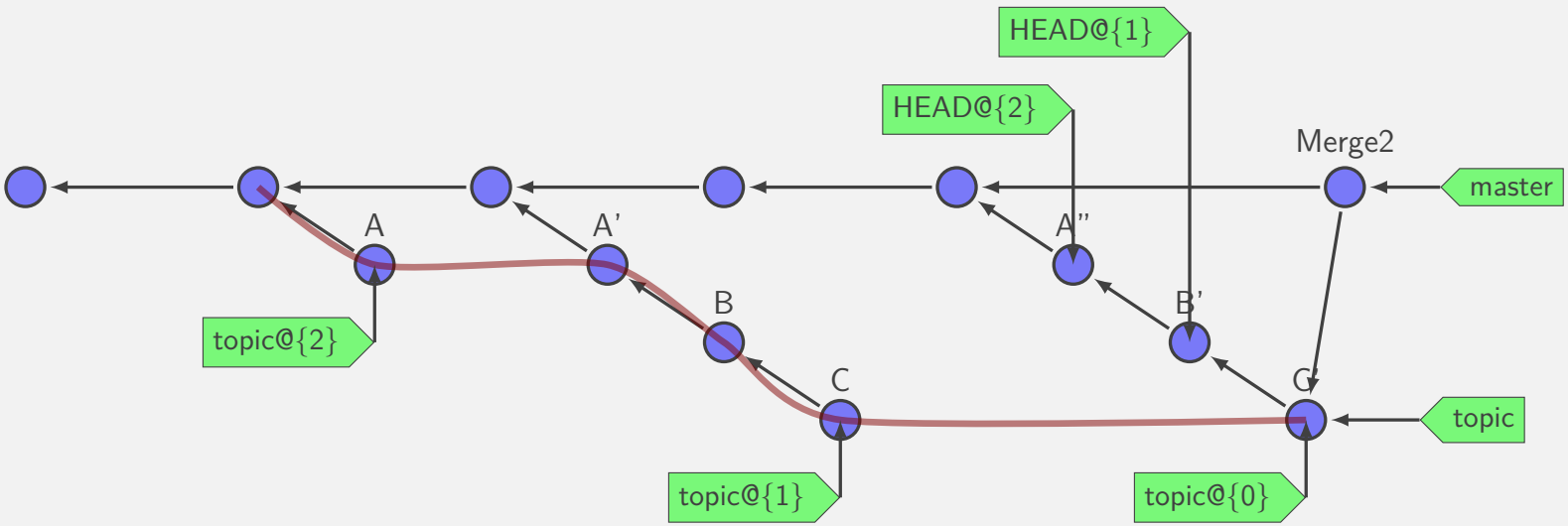
Git's reference journal: the reflog

- Remember the history of local refs.
- $\neq$ ancestry relation.



Git's reference journal: the reflog

- Remember the history of local refs.
- $\neq$ ancestry relation.



- *ref*`@{`*n*`}`: where *ref* was before the *n* last ref update.
- *ref*`~`*n*: the *n*-th generation ancestor of *ref*
- *ref*`^`: first parent of *ref*
- `git help revisions` for more

More Documentation

More Documentation

- http://ensiwiki.ensimag.fr/index.php/Maintenir_un_historique_propre_avec_Git
- http://ensiwiki.ensimag.fr/index.php/Ecrire_de_bons_messages_de_commit_avec_Git