git is the better quilt

Managing Debian packages in git with git-dpm

Bernhard R. Link
F8AC 04D5 0B9B 064B 3383 C3DA AFFC 96D1 151D FFDC

DebConf 2013

Setting

This presentation wants to give a short introduction how Debian packages can be kept in git using git-dpm. It will start with a short introduction to git and proceed with how git-dpm fills the gaps. Audience is assumed to know the basics of Debian packaging. Git knowledge is not assumed.

Git

- What is git?
- quick and short introduction to git
- Why git is so good for managing patches

Git-dpm

- general idea
- branches
- some commands

Git Basics

Git is a version control system (VCS). A VCS is something you commit information about different versions of your code. That includes:

- States your project tree was in,
- who did that commit and when,
- a description of that commit,
- the previous state this state builds upon,
- ...

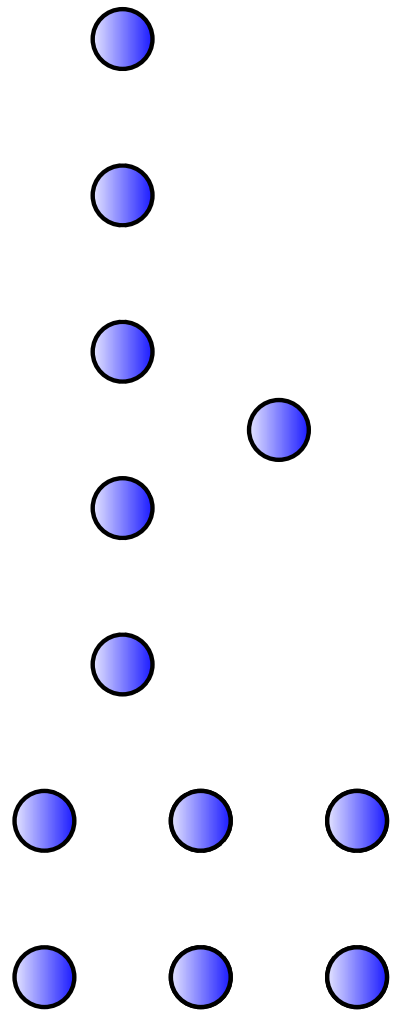
Git compared to other VCSes

Git

- only cares about files. Directories are created/removed as needed. $\Longrightarrow$ You cannot store empty directories in git.
- does not store diffs but snapshots $\Longrightarrow$ Removing and re-adding files is cheap. Storing both a classic development branch without autogenerated files and some commits containing full .orig.tar.gz contents is no problem.
- keeps commits in a graph, not a tree. $\Longrightarrow$ Merges are just joining two threads again. There is no real difference between merging branch A into B and branch B into A. Multiple merges are no problem at all.
- does not have a notion what branch a commit belongs to or any notion of persistent or global branches. Branches are just repository-local pointers to the commit that is the current head of a branch.

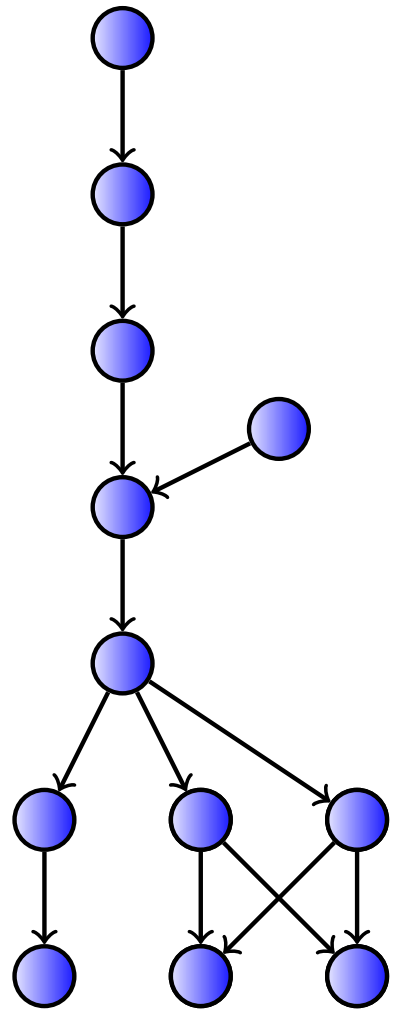
representation of history in git

- Git history consists out of commits. Each commit has author, date, a tree of files and



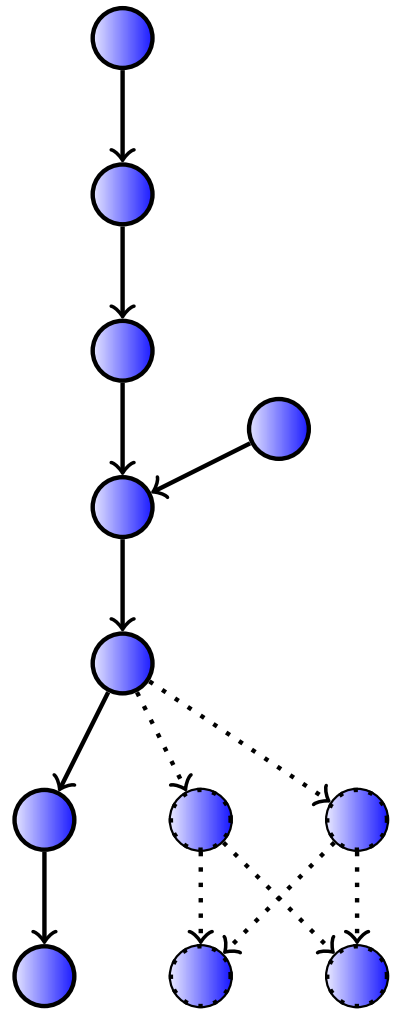
representation of history in git

- Git history consists out of commits. Each commit has author, date, a tree of files and
- a number of parents, forming a cycle free directed graph.



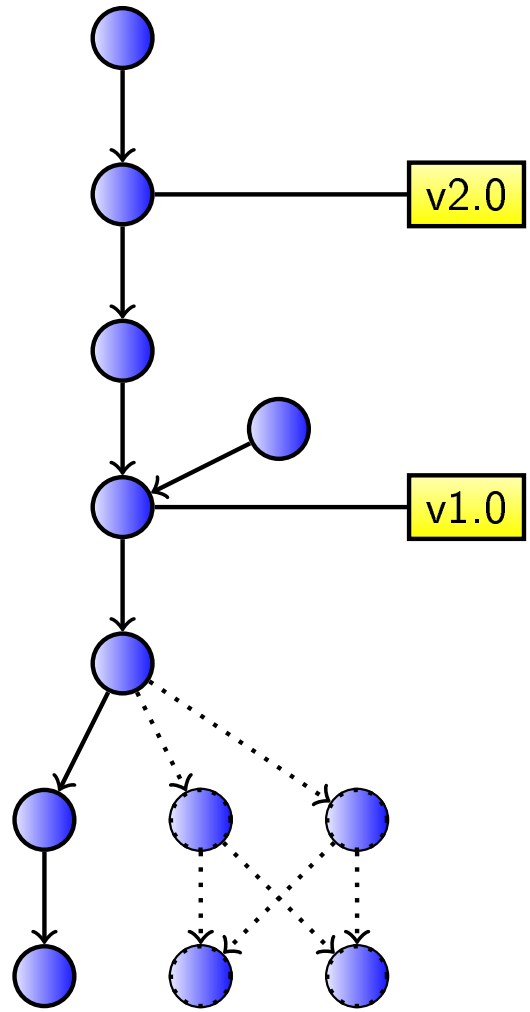
representation of history in git

- Git history consists out of commits. Each commit has author, date, a tree of files and
- a number of parents, forming a cycle free directed graph.



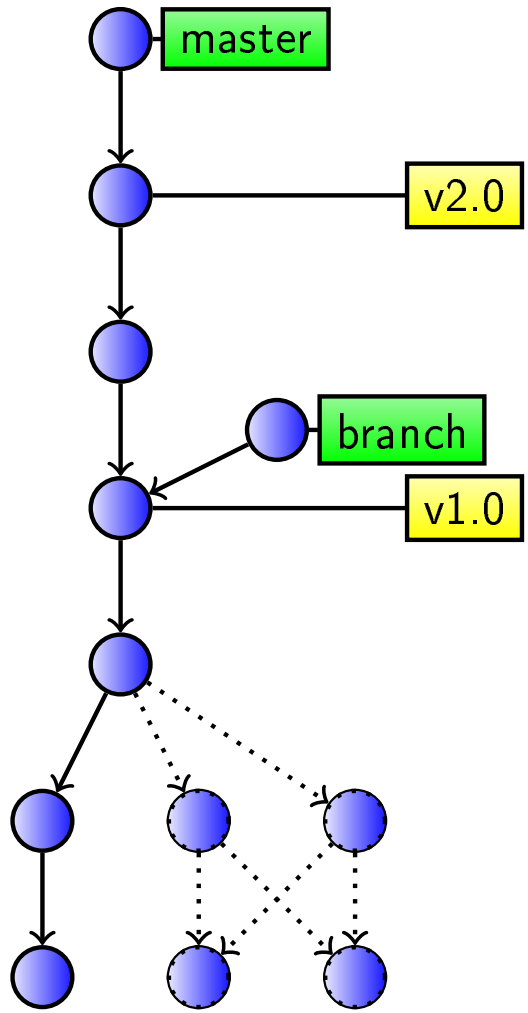
representation of history in git

- Git history consists out of commits. Each commit has author, date, a tree of files and
- a number of parents, forming a cycle free directed graph.
- A tag is a name for one commit.



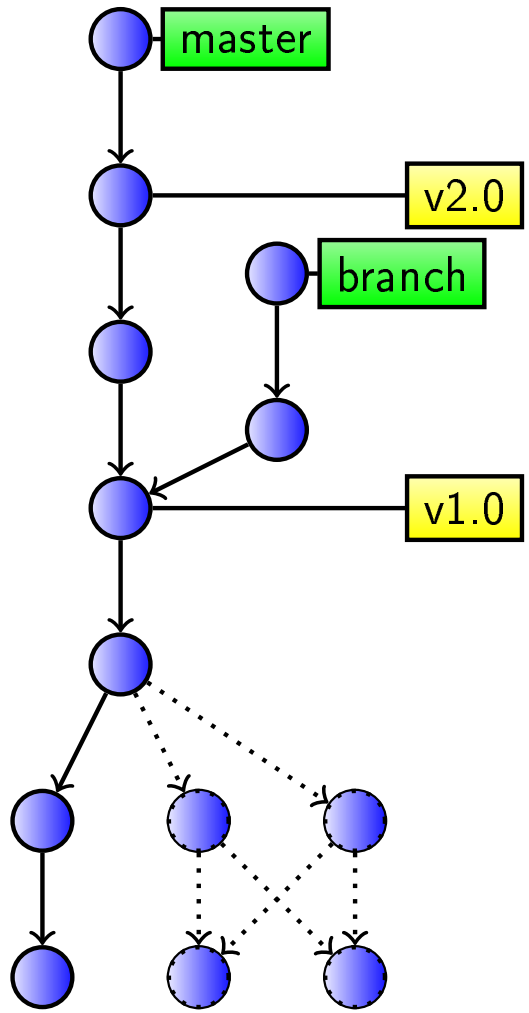
representation of history in git

- Git history consists out of commits. Each commit has author, date, a tree of files and
- a number of parents, forming a cycle free directed graph.
- A tag is a name for one commit.
- A branch is a name for one commit.



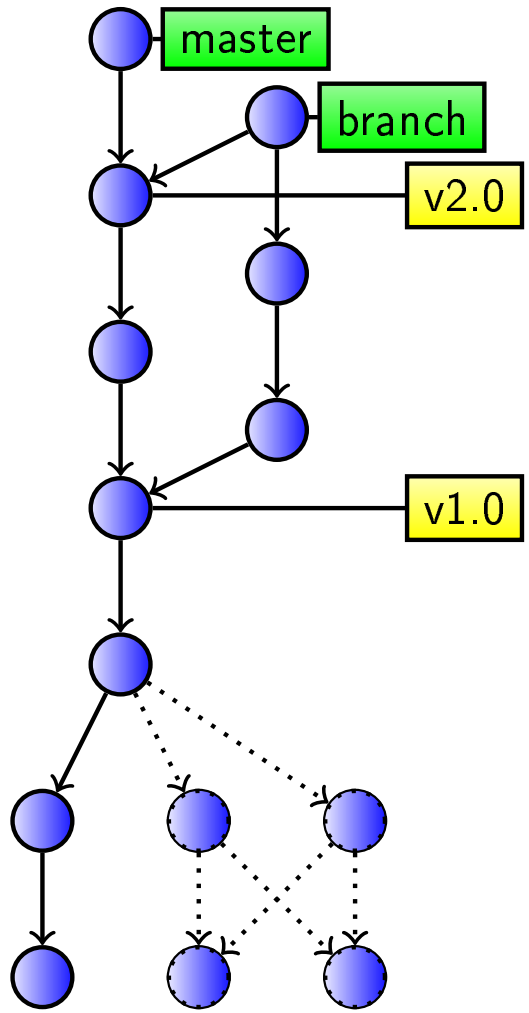
representation of history in git

- Git history consists out of commits. Each commit has author, date, a tree of files and
- a number of parents, forming a cycle free directed graph.
- A tag is a name for one commit.
- A branch is a name for one commit, which can move.



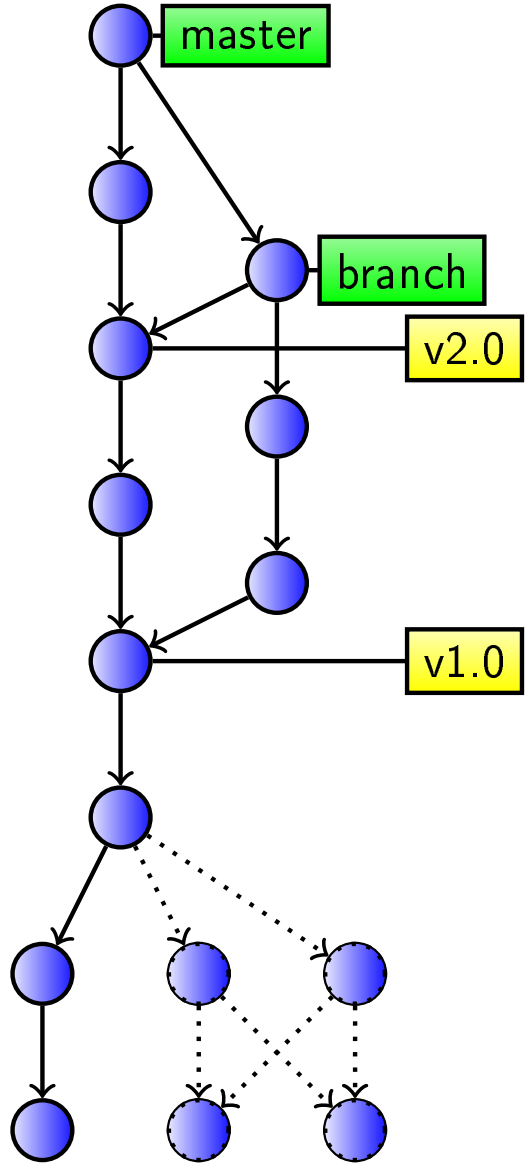
representation of history in git

- Git history consists out of commits. Each commit has author, date, a tree of files and
- a number of parents, forming a cycle free directed graph.
- A tag is a name for one commit.
- A branch is a name for one commit, which can move.
- Merging means creating a new commit with the merged commits as parents.



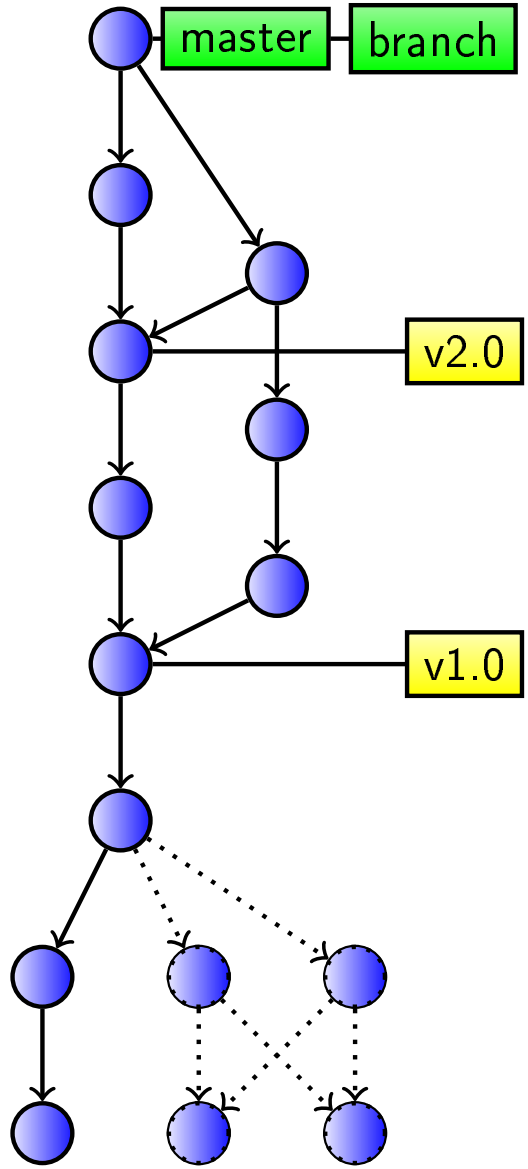
representation of history in git

- Git history consists out of commits. Each commit has author, date, a tree of files and
- a number of parents, forming a cycle free directed graph.
- A tag is a name for one commit.
- A branch is a name for one commit, which can move.
- Merging means creating a new commit with the merged commits as parents.



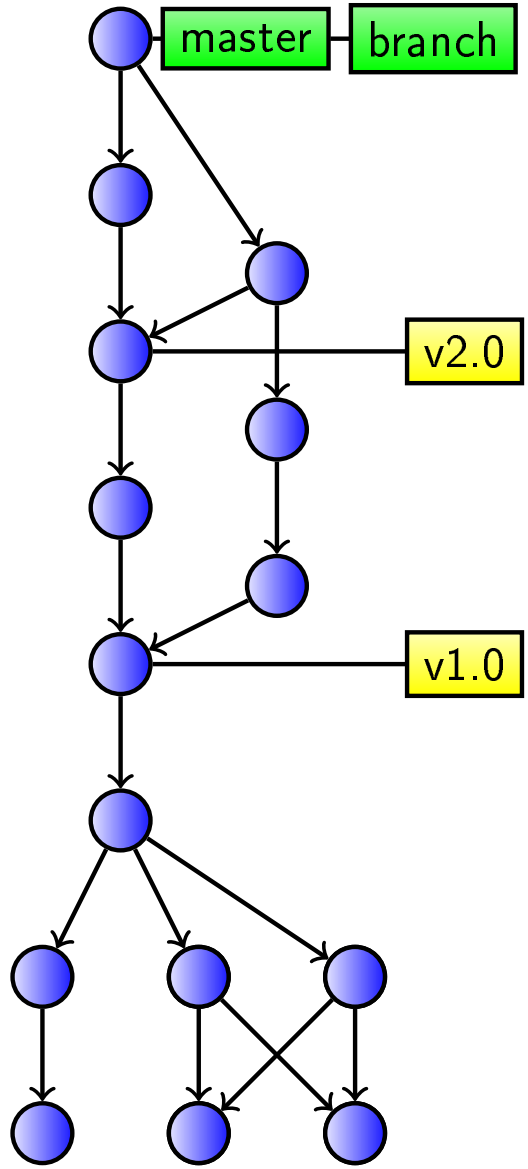
representation of history in git

- Git history consists out of commits. Each commit has author, date, a tree of files and
- a number of parents, forming a cycle free directed graph.
- A tag is a name for one commit.
- A branch is a name for one commit, which can move.
- Merging means creating a new commit with the merged commits as parents. Unless you merge something into an ancestor, then you just update the branch. This is called fast-forward.



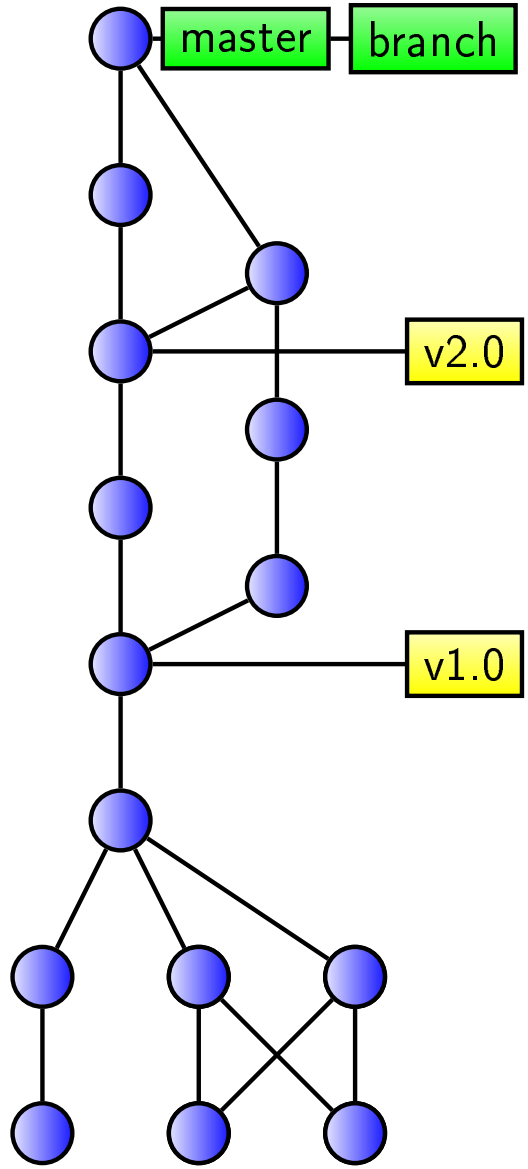
representation of history in git

- Git history consists out of commits. Each commit has author, date, a tree of files and
- a number of parents, forming a cycle free directed graph.
- A tag is a name for one commit.
- A branch is a name for one commit, which can move.
- Merging means creating a new commit with the merged commits as parents. Unless you merge something into an ancestor, then you just update the branch. This is called fast-forward.



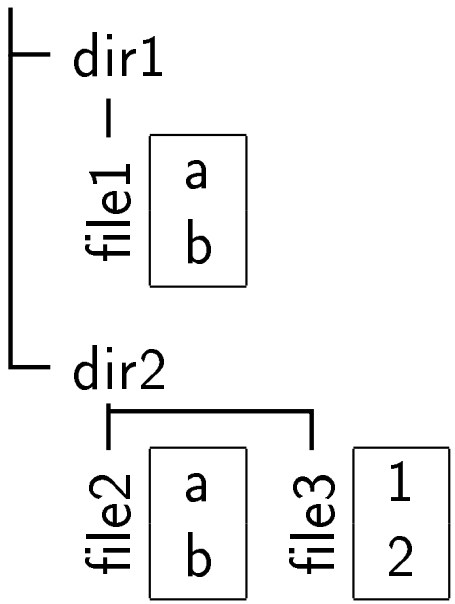
representation of history in git

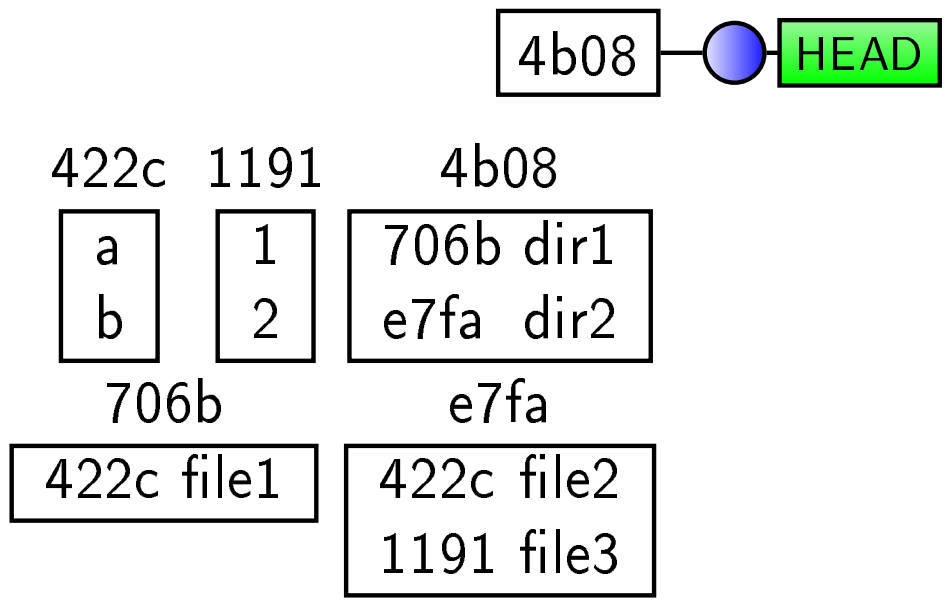
- Git history consists out of commits. Each commit has author, date, a tree of files and
- a number of parents, forming a cycle free directed graph.
- A tag is a name for one commit.
- A branch is a name for one commit, which can move.
- Merging means creating a new commit with the merged commits as parents. Unless you merge something into an ancestor, then you just update the branch. This is called fast-forward.



object storage and the staging area

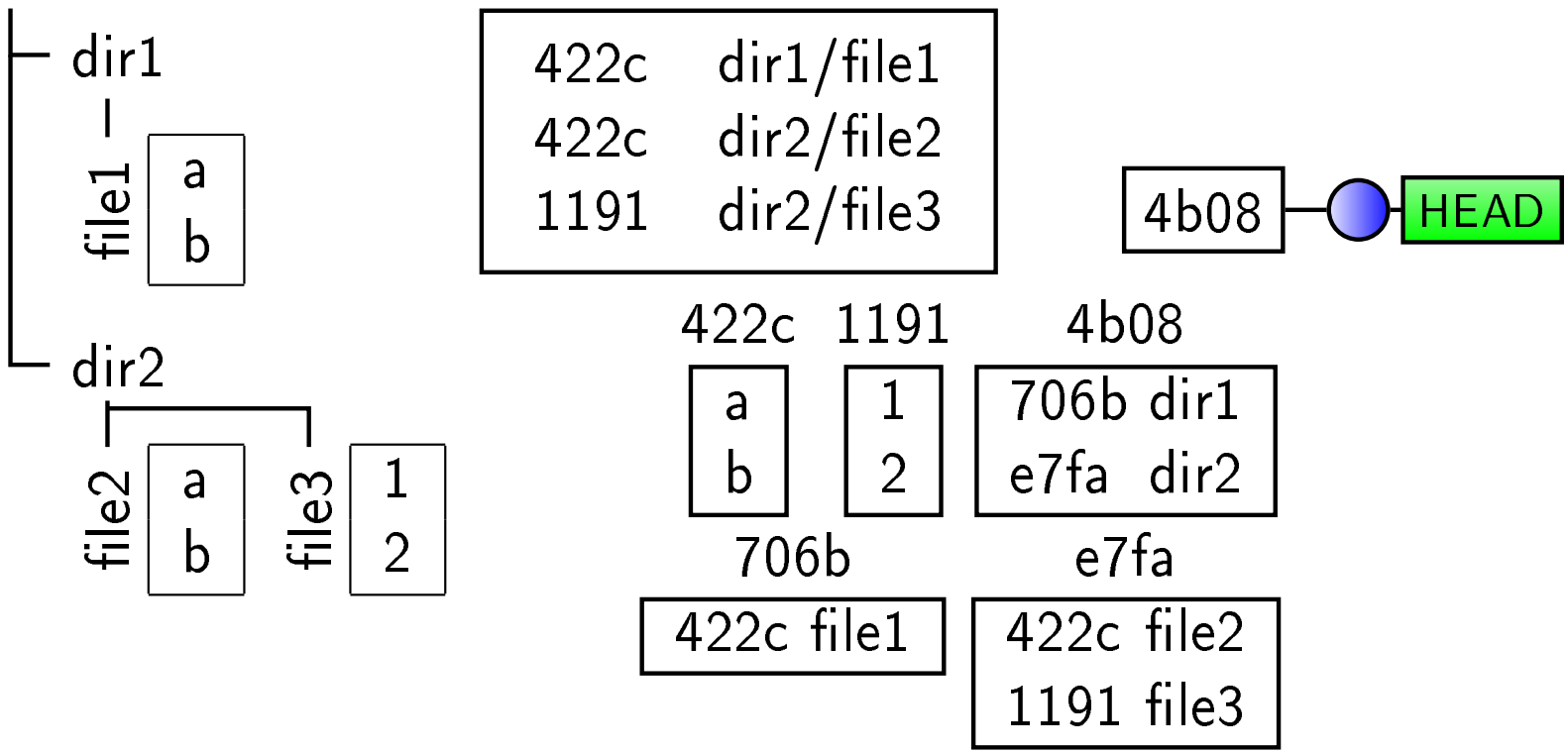
- Git references file contents, trees and commits via a sha1sum.





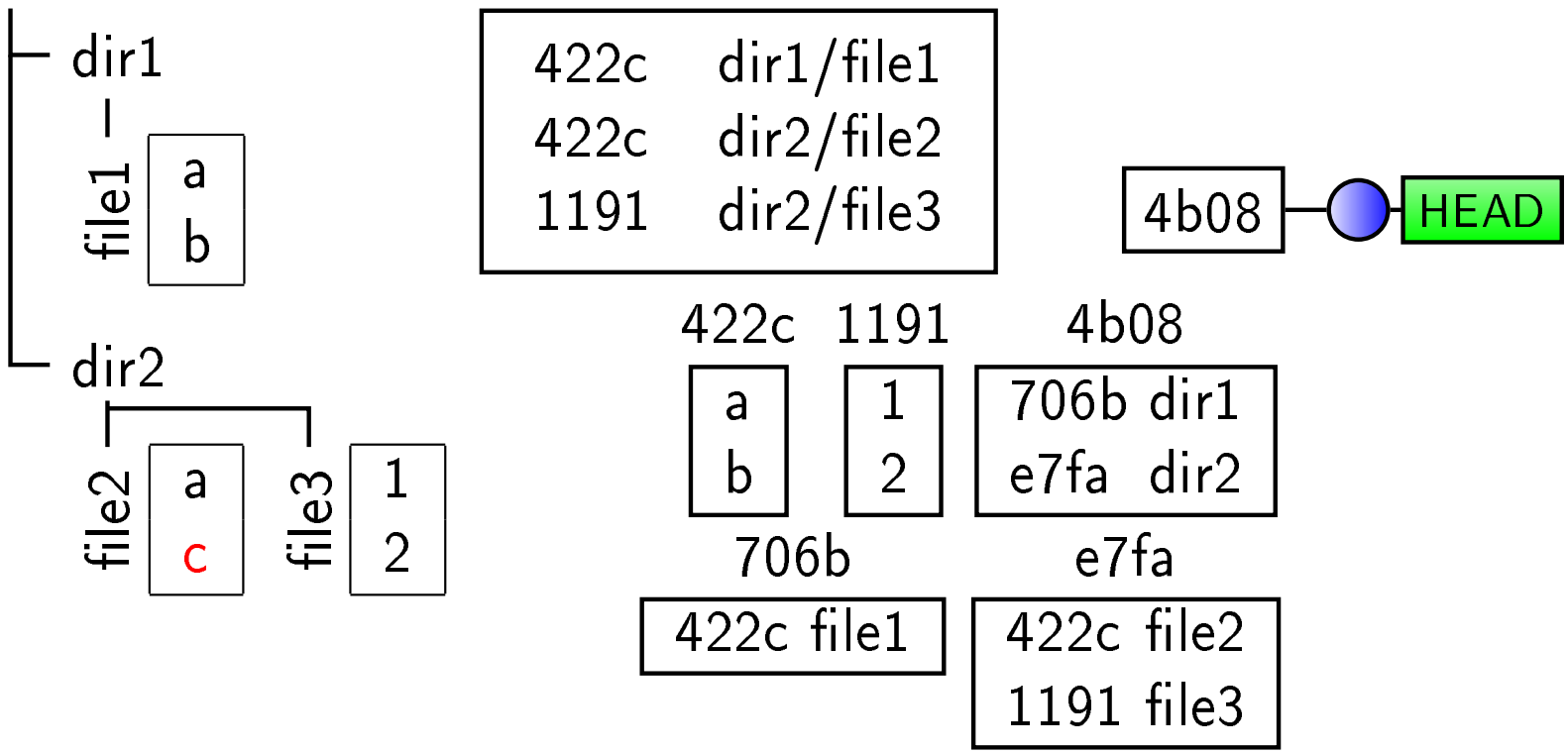
object storage and the staging area

- Git references file contents, trees and commits via a sha1sum.
- Git has a staging area, the *index*.



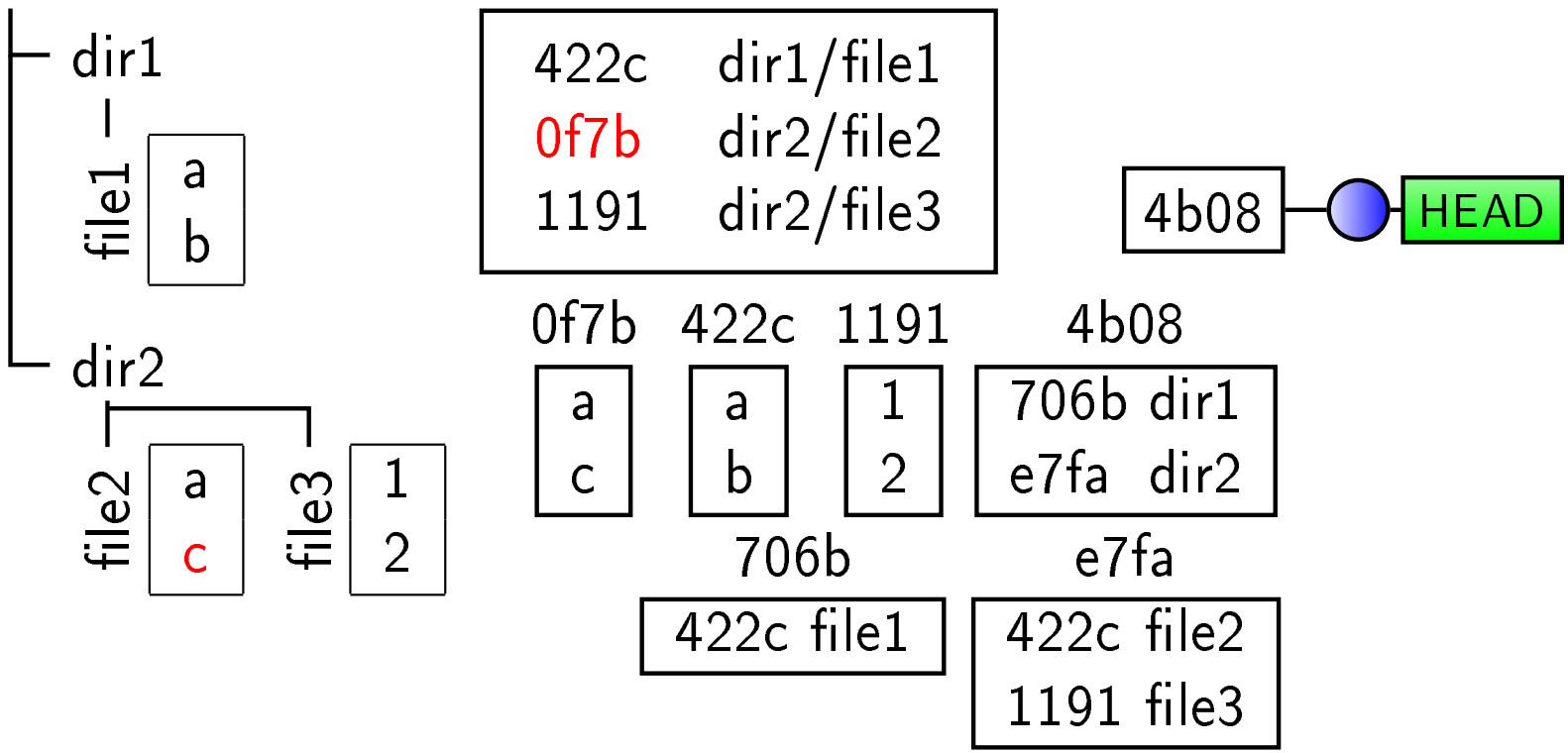
object storage and the staging area

- Git references file contents, trees and commits via a sha1sum.
- Git has a staging area, the *index*.
- Changes: working directory.



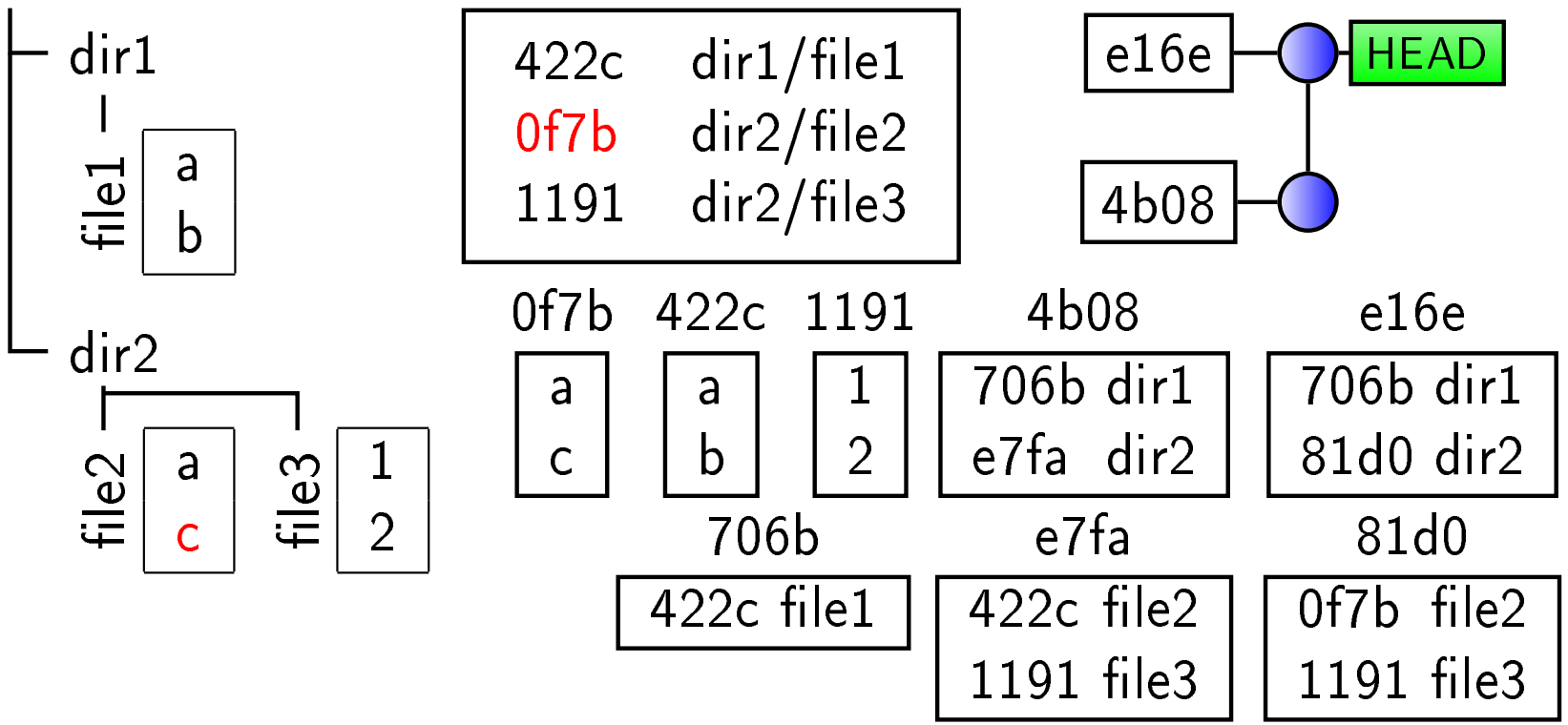
object storage and the staging area

- Git references file contents, trees and commits via a sha1sum.
- Git has a staging area, the *index*.
- Changes: working directory $\xrightarrow{\text{git add}}$ index.



object storage and the staging area

- Git references file contents, trees and commits via a sha1sum.
- Git has a staging area, the *index*.
- Changes: working directory $\xrightarrow{\text{git add}}$ index $\xrightarrow{\text{git commit}}$ commit.



staging area and important commands

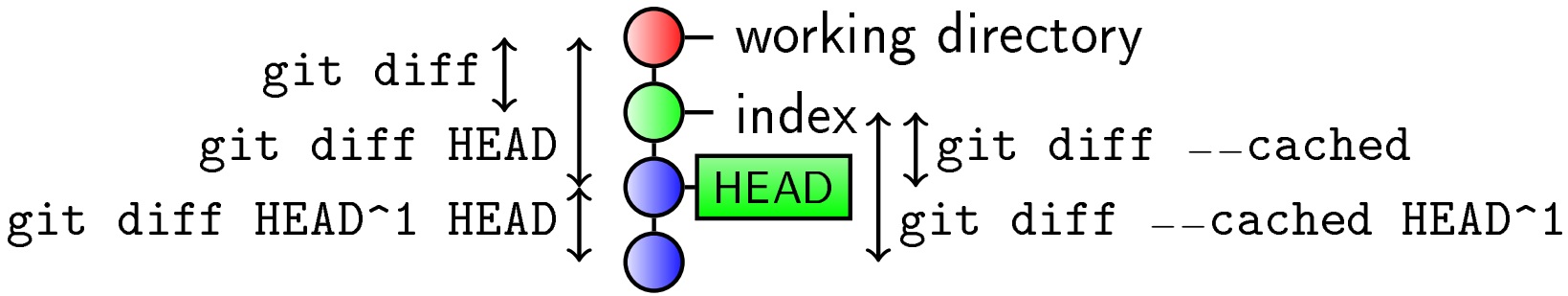
- Changes: working directory $\xrightarrow{\text{git add}}$ index $\xrightarrow{\text{git commit}}$ commit.

staging area and important commands

- Changes: working directory $\xrightarrow{\text{git add}}$ index $\xrightarrow{\text{git commit}}$ commit.
- Giving `git commit -a` automatically adds all modified files to the index before the commit.

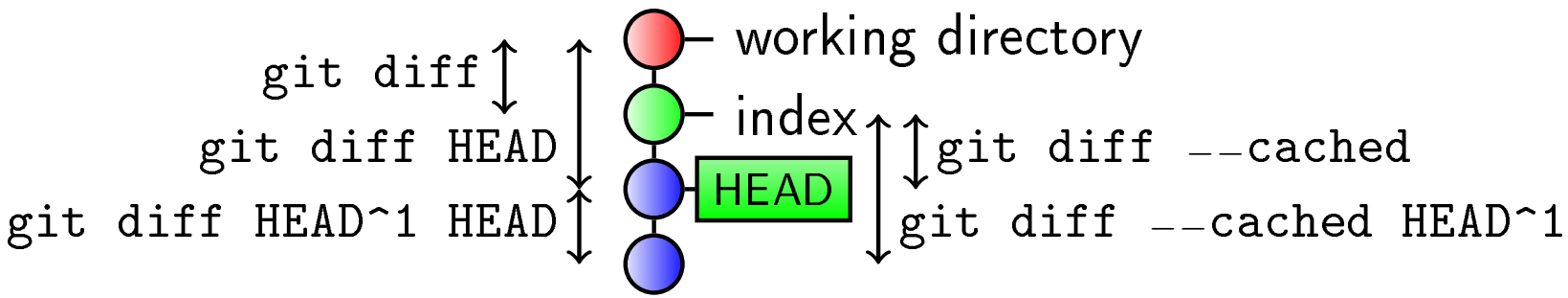
staging area and important commands

- Changes: working directory $\xrightarrow{\text{git add}}$ index $\xrightarrow{\text{git commit}}$ commit.
- Giving `git commit -a` automatically adds all modified files to the index before the commit.
- `git status` gives status, `git diff` shows differences.



staging area and important commands

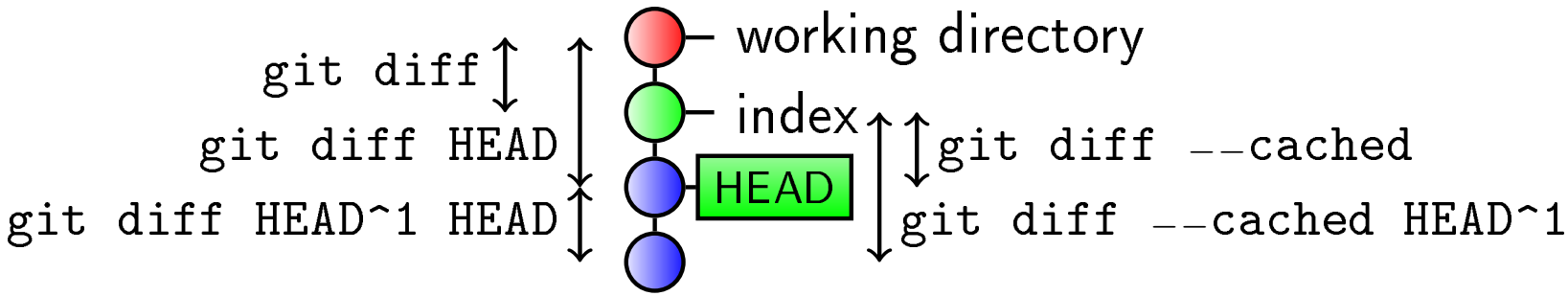
- Changes: working directory $\xrightarrow{\text{git add}}$ index $\xrightarrow{\text{git commit}}$ commit.
- Giving `git commit -a` automatically adds all modified files to the index before the commit.
- `git status` gives status, `git diff` shows differences.



- `git reset --soft` *name* changes the head commit of your current branch (usually to a parent or ancestor), `git reset --mixed` *name* also resets the index, `git reset --hard` *name* also the working directory.

staging area and important commands

- Changes: working directory $\xrightarrow{\text{git add}}$ index $\xrightarrow{\text{git commit}}$ commit.
- Giving `git commit -a` automatically adds all modified files to the index before the commit.
- `git status` gives status, `git diff` shows differences.



- `git reset --soft` *name* changes the head commit of your current branch (usually to a parent or ancestor), `git reset --mixed` *name* also resets the index, `git reset --hard` *name* also the working directory.
- You can move partial content using `git add -p`.

Managing patches in git

Why modifying history?

- Most upstreams will usually require some properties for patches they include:
  - Every commit should be in a working state (important for bisect).
  - Commits should not mix unrelated things (one issue at a time).
  - Commits should have proper description.
- Getting everything good enough the first time in practically impossible in most cases.
- Replaying all your modifications and doing the correct commits is both too much work and too error prone.

Managing patches in git

Why modifying history?

- Most upstreams will usually require some properties for patches they include:
  - Every commit should be in a working state (important for bisect).
  - Commits should not mix unrelated things (one issue at a time).
  - Commits should have proper description.
- Getting everything good enough the first time in practically impossible in most cases.
- Replaying all your modifications and doing the correct commits is both too much work and too error prone.
- Git to the rescue: modifying commits in git.

Amending a commit

Amending a commit with `git commit --amend` does not create a new commit on top of the current one, but creates a commit supposed to replace the current one.
The new commit will have the parents of the current HEAD, but can have a new tree and a new description and if you use `--reset-author` also a new author and timestamp.

Splitting a patch

Let's assume your last commit contains unrelated and you want to split it into two commits.

- `git branch old`
  First remember the current state somehow
  (I like temporary branches).

Splitting a patch

Let's assume your last commit contains unrelated and you want to split it into two commits.

- `git branch old`
- `git reset --mixed HEAD^1`
  Switch back to the previous state

Splitting a patch

Let's assume your last commit contains unrelated and you want to split it into two commits.

- `git branch old`
- `git reset --mixed HEAD^1`
- `git add file1 file2`
  Mark all files where you want all changes in the first commit.

Splitting a patch

Let's assume your last commit contains unrelated and you want to split it into two commits.

- `git branch old`
- `git reset --mixed HEAD^1`
- `git add file1 file2`
- `git add -p file3 file4`
  Add parts of the modifications of other files (give no names to be asked about all files)
  For every chunk in them answer whether to include it or not.

Splitting a patch

Let's assume your last commit contains unrelated and you want to split it into two commits.

- `git branch old`
- `git reset --mixed HEAD^1`
- `git add file1 file2`
- `git add -p file3 file4`
- `git commit -c old`
  Do the new commit (reusing the commit message where applicable using -c).

Splitting a patch

Let's assume your last commit contains unrelated and you want to split it into two commits.

- `git branch old`
- `git reset --mixed HEAD^1`
- `git add file1 file2`
- `git add -p file3 file4`
- `git commit -c old`
- `git add file5`
  Add files newly added in the second patch (if any).

Splitting a patch

Let's assume your last commit contains unrelated and you want to split it into two commits.

- `git branch old`
- `git reset --mixed HEAD^1`
- `git add file1 file2`
- `git add -p file3 file4`
- `git commit -c old`
- `git add file5`
- `git commit -c old -a`
  Do the second commit with all the other modifications (-a) to already tracked files.

Splitting a patch

Let's assume your last commit contains unrelated and you want to split it into two commits.

- `git branch old`
- `git reset --mixed HEAD^1`
- `git add file1 file2`
- `git add -p file3 file4`
- `git commit -c old`
- `git add file5`
- `git commit -c old -a`
- `git diff old HEAD`
  Compare the resulting trees.

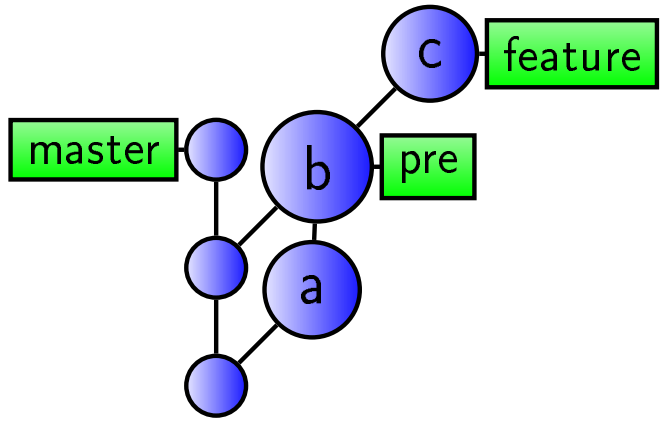
Splitting a patch

Let's assume your last commit contains unrelated and you want to split it into two commits.

- `git branch old`
- `git reset --mixed HEAD^1`
- `git add file1 file2`
- `git add -p file3 file4`
- `git commit -c old`
- `git add file5`
- `git commit -c old -a`
- `git diff old HEAD`
- `git branch -D old`
  Clean up. Or if you goofed up revert using:
  `git reset --hard old`

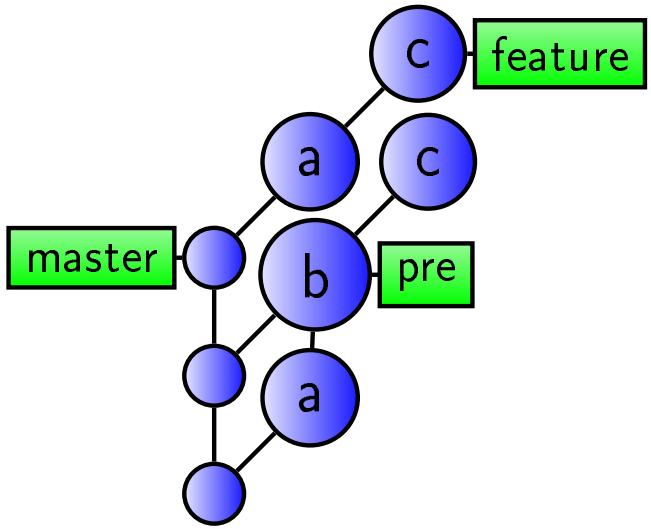
Git rebase

- Rebasing means translating commits to patches and applying those at another part.



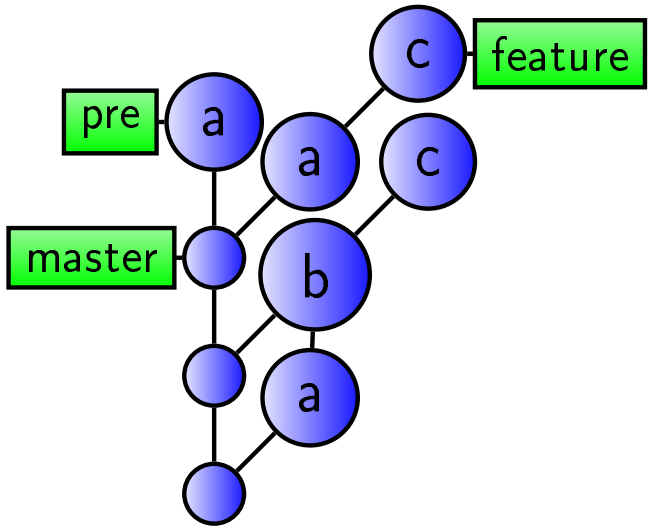
Git rebase

- Rebasing means translating commits to patches and applying those at another part.
- New commits are created, no other branches than the current are moved. Merges are not retained (but the history is linearized).



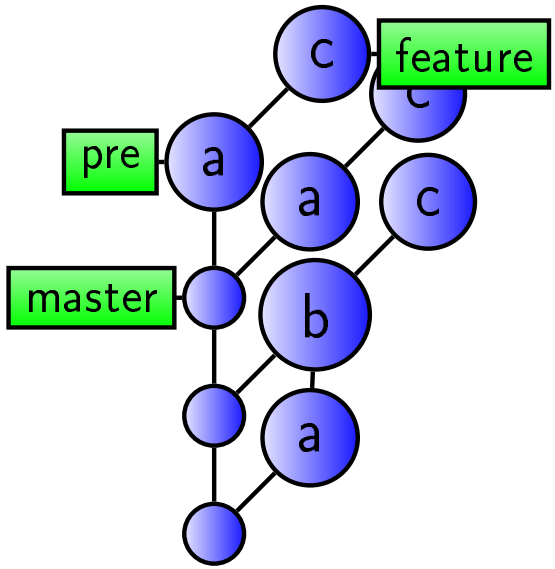
Git rebase

- Rebasing means translating commits to patches and applying those at another part.
- New commits are created, no other branches than the current are moved. Merges are not retained (but the history is linearized).
- No commits are reused (unless they stay exactly where they are).



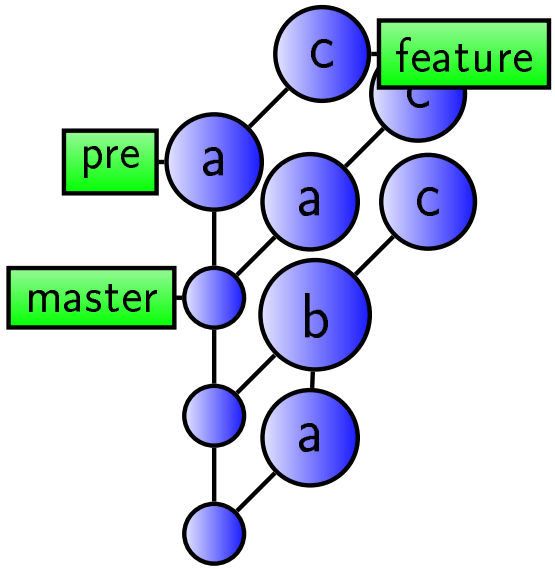
Git rebase

- Rebasing means translating commits to patches and applying those at another part.
- New commits are created, no other branches than the current are moved. Merges are not retained (but the history is linearized).
- No commits are reused (unless they stay exactly where they are).
- But the usual git magic will realize a patch is already applied and drop it.



Git rebase

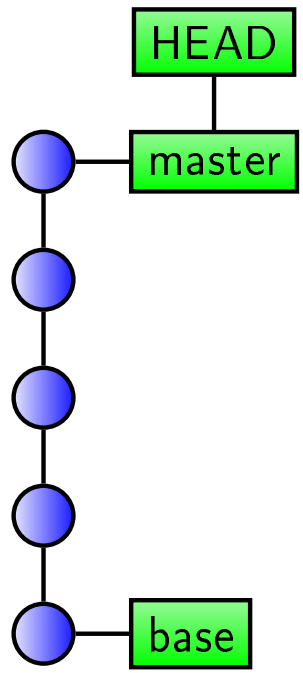
- Rebasing means translating commits to patches and applying those at another part.
- New commits are created, no other branches than the current are moved. Merges are not retained (but the history is linearized).
- No commits are reused (unless they stay exactly where they are).
- But the usual git magic will realize a patch is already applied and drop it.
- The operation changes the history, the history of this operation is not retained.



Git rebase –i

```
pick 5d0fb09 Some modification
pick 59853f1 Some other modfification
pick 4354556 Even more stuff
pick 3484448 belongs to the first commit
```

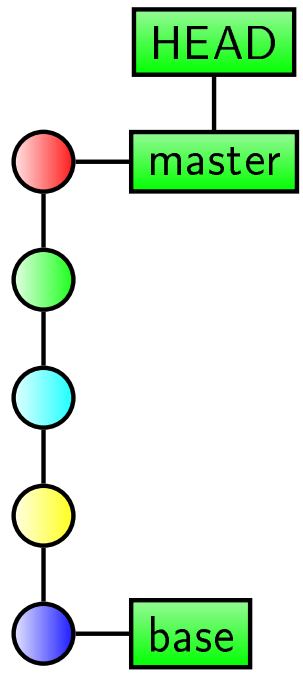
```
pick 5d0fb09 Some modification
pick 59853f1 Some other modfification
pick 4354556 Even more stuff
pick 3484448 belongs to the first commit

# Rebase 94c2c02..3484448 onto 94c2c02
#
# 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

```
pick 5d0fb09 Some modification
pick 59853f1 Some other modfification
pick 4354556 Even more stuff
pick 3484448 belongs to the first commit
```

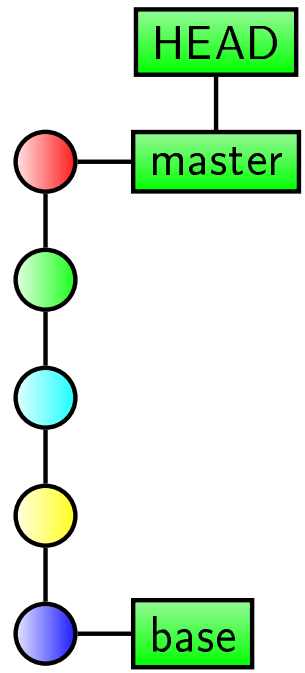
```
pick 5d0fb09 Some modification
pick 59853f1 Some other modfification
pick 4354556 Even more stuff
pick 3484448 belongs to the first commit

# Rebase 94c2c02..3484448 onto 94c2c02
#
# 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

```
pick 5d0fb09 Some modification
pick 59853f1 Some other modfification
pick 4354556 Even more stuff
pick 3484448 belongs to the first commit
```

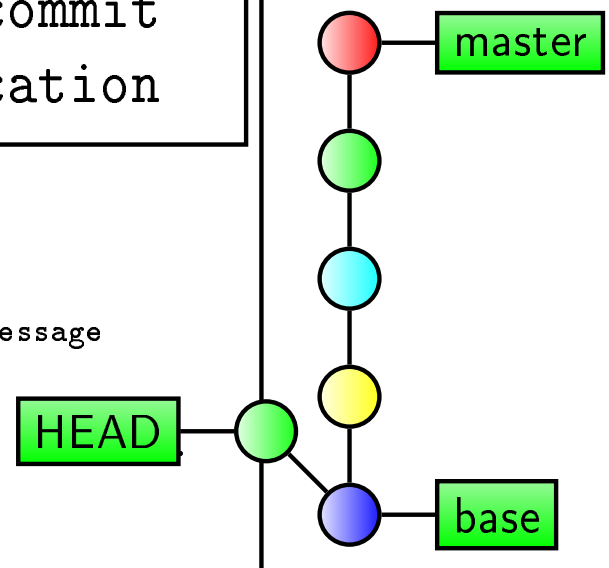
```
e 4354556 Even more stuff
pick 5d0fb09 Some modification
s 3484448 belongs to the first commit
pick 59853f1 Some other modfification
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous 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

```
pick 5d0fb09 Some modification
pick 59853f1 Some other modfification
pick 4354556 Even more stuff
pick 3484448 belongs to the first commit
```

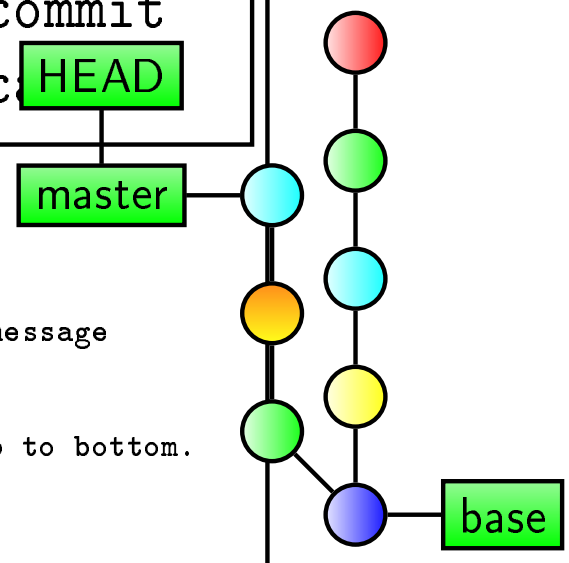
```
e 4354556 Even more stuff
pick 5d0fb09 Some modification
s 3484448 belongs to the first commit
pick 59853f1 Some other modfification
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous 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
#
# 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

```
pick 5d0fb09 Some modification
pick 59853f1 Some other modfification
pick 4354556 Even more stuff
pick 3484448 belongs to the first commit
```

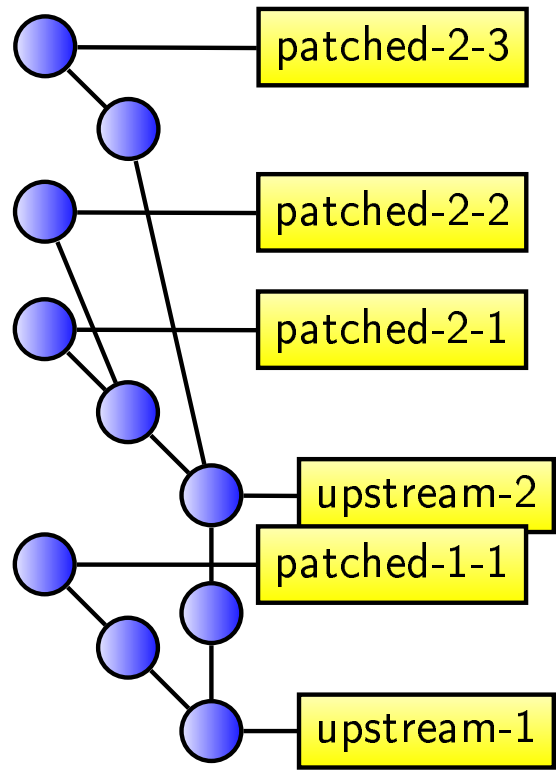
```
e 4354556 Even more stuff
pick 5d0fb09 Some modification
s 3484448 belongs to the first commit
pick 59853f1 Some other modfific
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous 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 for Debian packaging

So git is great for managing patches by representing them with commits. But there two problems:

- Git should be used for storying history, too.
- Git's support for pushing rebasing branches between repositories is practically non-existing (it only support fast-forward updates).

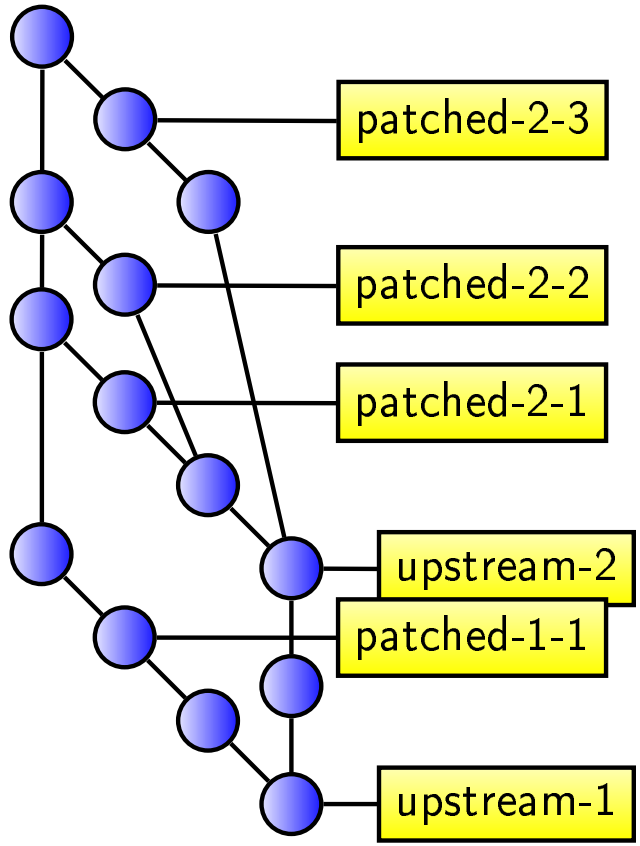


Git for Debian packaging

So git is great for managing patches by representing them with commits. But there two problems:

- Git should be used for storying history, too.
- Git's support for pushing rebasing branches between repositories is practically non-existing (it only support fast-forward updates).

Solution: store rebased branch in the history of a fast-forwarding branch.

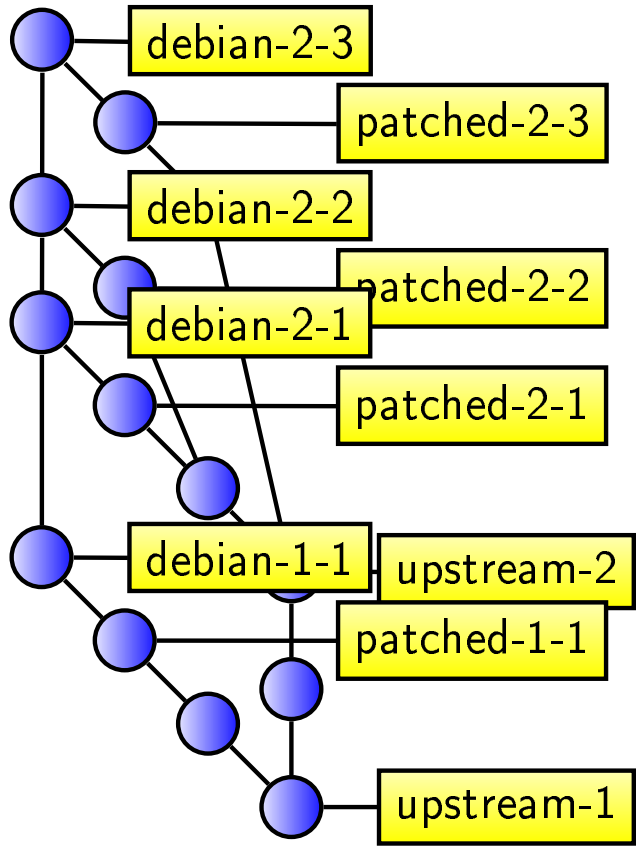


Git for Debian packaging

So git is great for managing patches by representing them with commits. But there two problems:

- Git should be used for storying history, too.
- Git's support for pushing rebasing branches between repositories is practically non-existing (it only support fast-forward updates).

Solution: store rebased branch in the history of a fast-forwarding branch.
Idea: also store modifications to debian/ there.



git-dpm

Idea of `git-dpm`:

- use git to manage patches

git-dpm

Idea of `git-dpm`:

- use git to manage patches
  - as described above with an embedded rebased patch

git-dpm

Idea of `git-dpm`:

- use git to manage patches
  - as described above with an embedded rebased patch
- use git to allow an otherwise unchanged packaging workflow

git-dpm

Idea of `git-dpm`:

- use git to manage patches
  - as described above with an embedded rebased patch
- use git to allow an otherwise unchanged packaging workflow
  - git is powerful enough, so one does not change the workflow too much
- do as much as possible with git as natively as possible

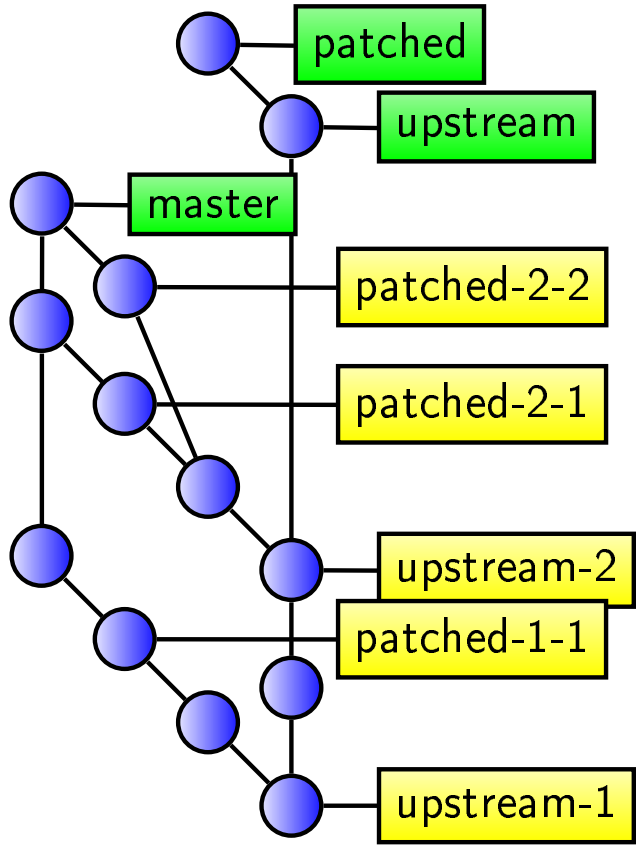
git-dpm

Idea of `git-dpm`:

- use git to manage patches
  - as described above with an embedded rebased patch
- use git to allow an otherwise unchanged packaging workflow
  - git is powerful enough, so one does not change the workflow too much
- do as much as possible with git as natively as possible
  - everything should be possible without git-dpm (and as easily as resonable possible)

git-dpm merge-patched

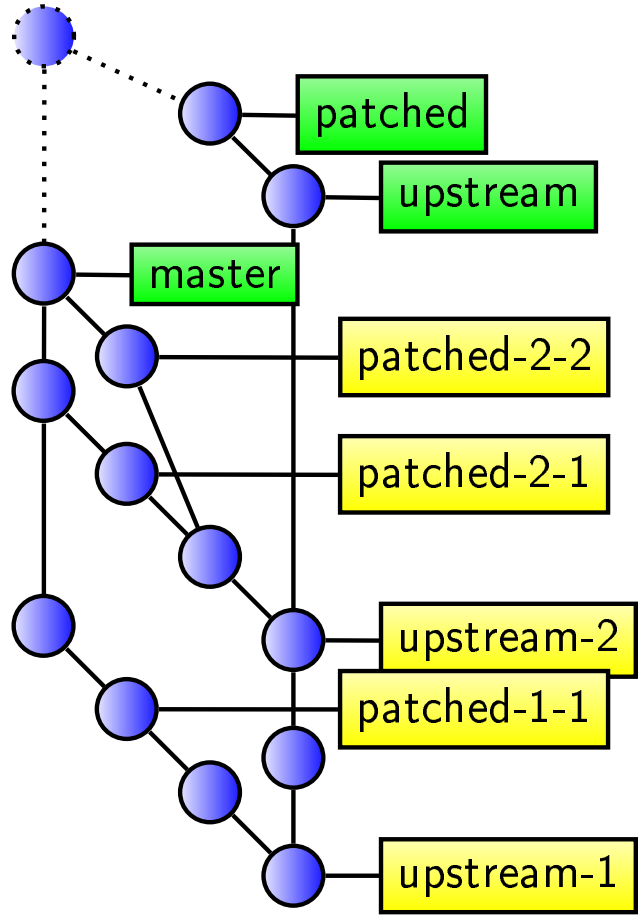
The above described method has one challange for git: A good merging method.
Consider this case: There was a new upstream. You have some patch against this (either a new one, or one of the previous ones survived because it was not included upstream). Now one needs a merge that does...



git-dpm merge-patched

Now one needs a merge that does...

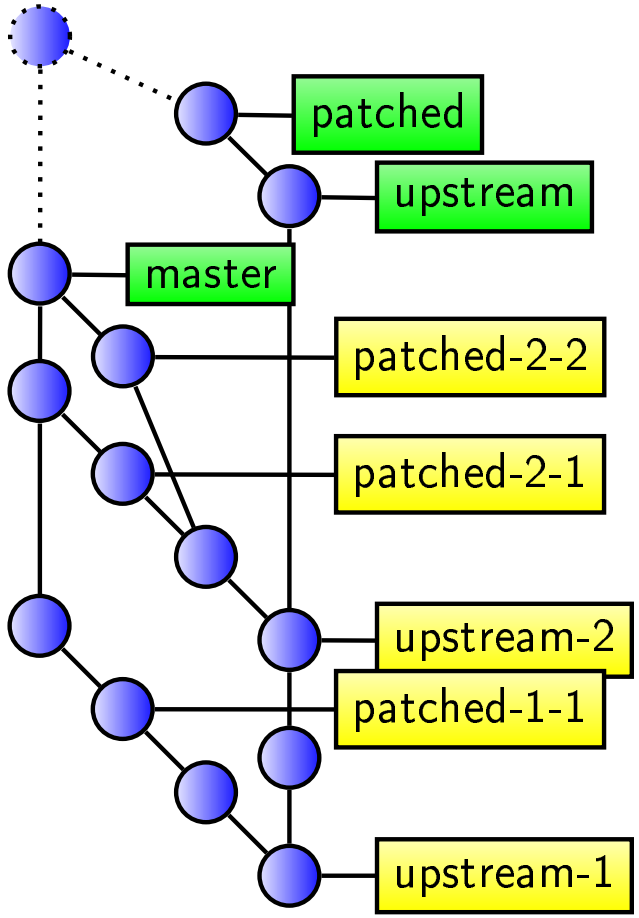
- Take `debian/` from your previous master branch (ignore upstream).



git-dpm merge-patched

Now one needs a merge that does...

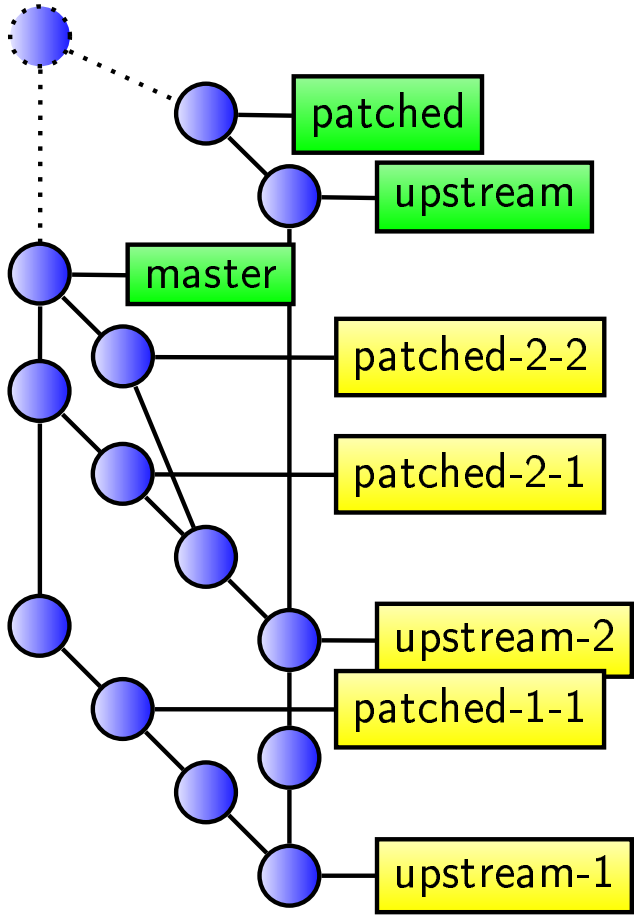
- Take `debian/` from your previous master branch (ignore upstream).
- Otherwise take everything from the new patched branch,



git-dpm merge-patched

Now one needs a merge that does...

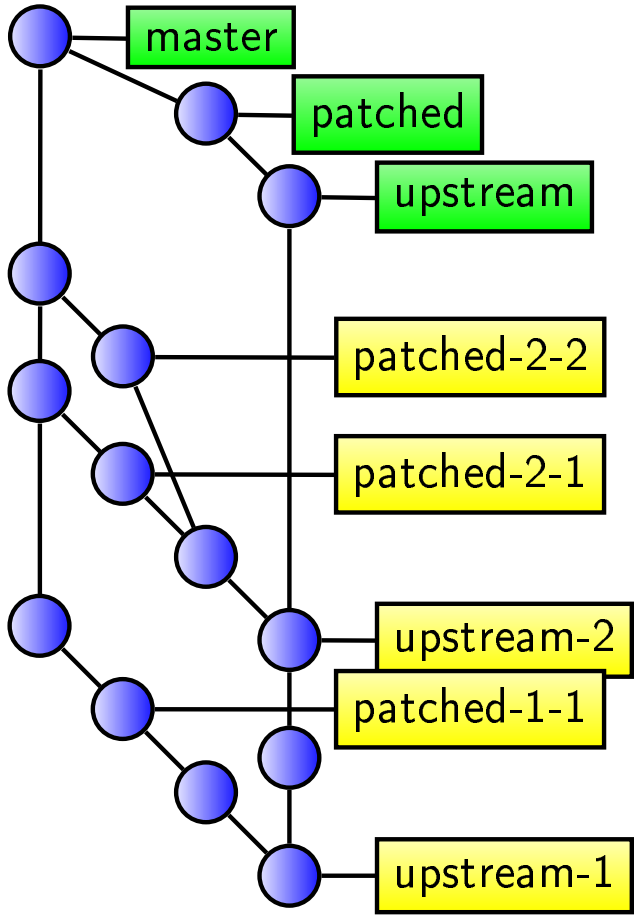
- Take `debian/` from your previous master branch (ignore upstream).
- Otherwise take everything from the new patched branch,
- except `.gitignore`, `.gitattributes`, ... files and



git-dpm merge-patched

Now one needs a merge that does...

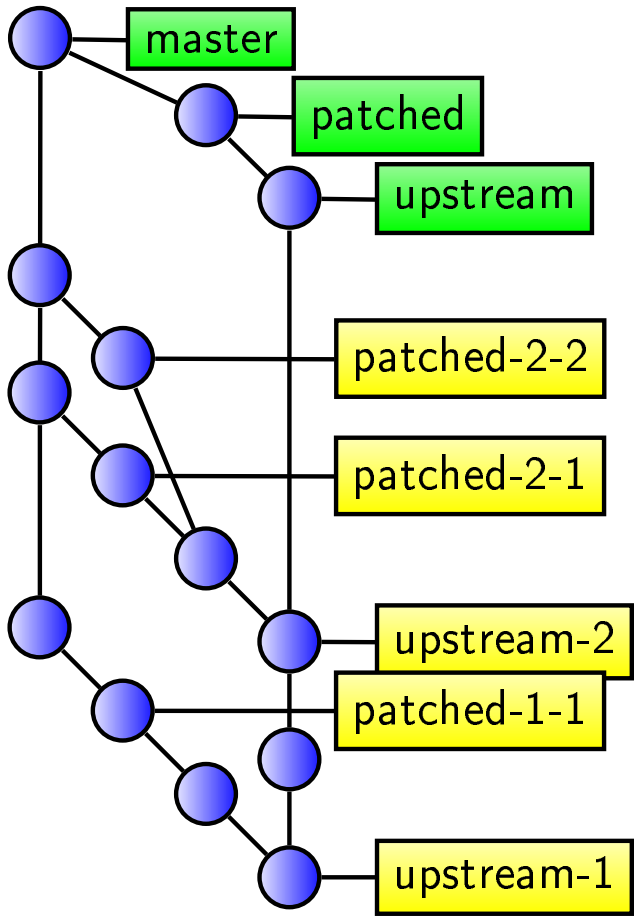
- Take `debian/` from your previous master branch (ignore upstream).
- Otherwise take everything from the new patched branch,
- except `.gitignore`, `.gitattributes`, ... files and
- except file deletions (so one can just clean files the upstream build system modifies, just as before using git).



branches used by git-dpm

As seen before, one always needs three points in the git history:

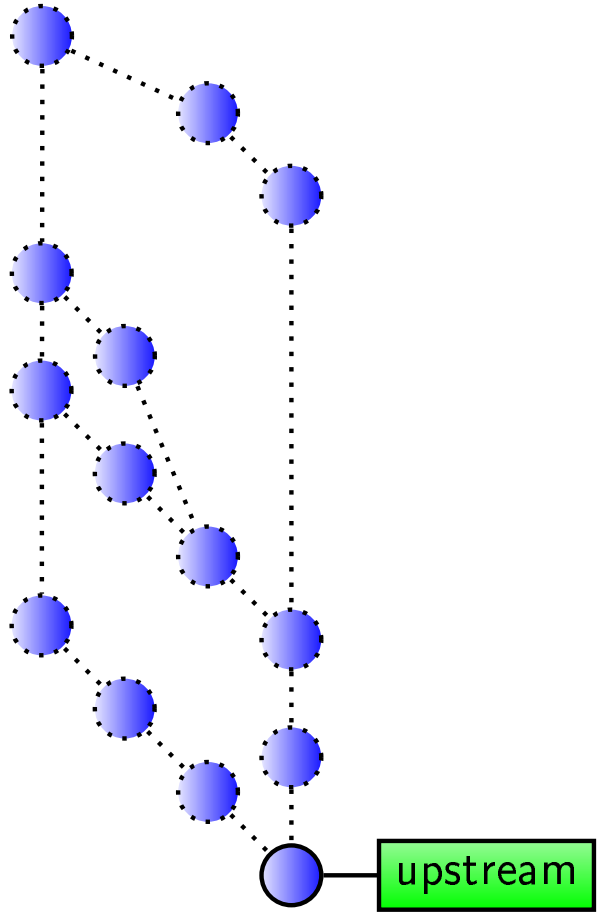
- an *upstream* branch for the upstream history (be it imported tarballs, upstream git commits, or imported tarballs on top of upstream git history)
- a *patched* branch based on the upstream branch, containing only changes to the upstream code.
- a *Debian* branch containing the Debian changes and having those other branches merged in (including previous states).



branches used by git-dpm

As seen before, one always needs three points in the git history:

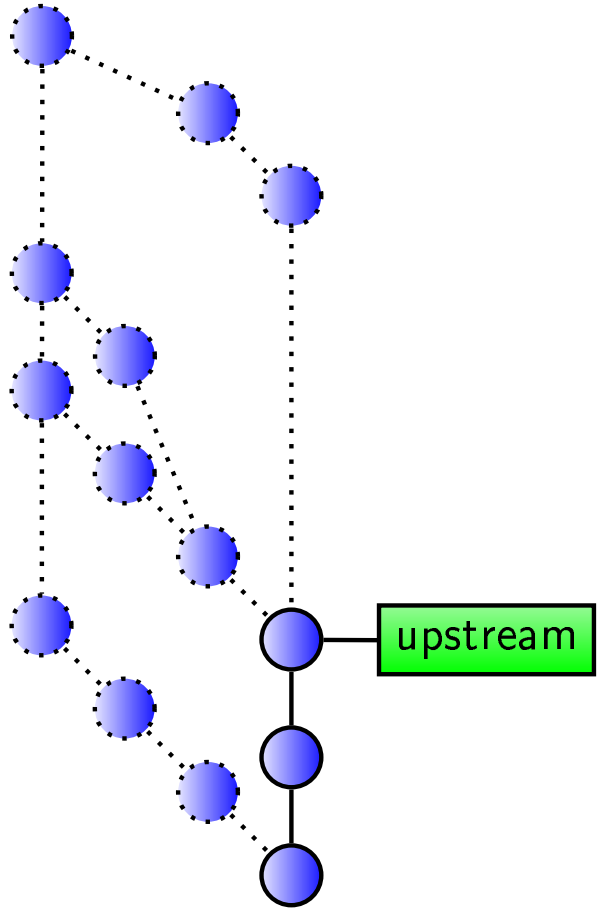
- an *upstream* branch for the upstream history (be it imported tarballs, upstream git commits, or imported tarballs on top of upstream git history)



branches used by git-dpm

As seen before, one always needs three points in the git history:

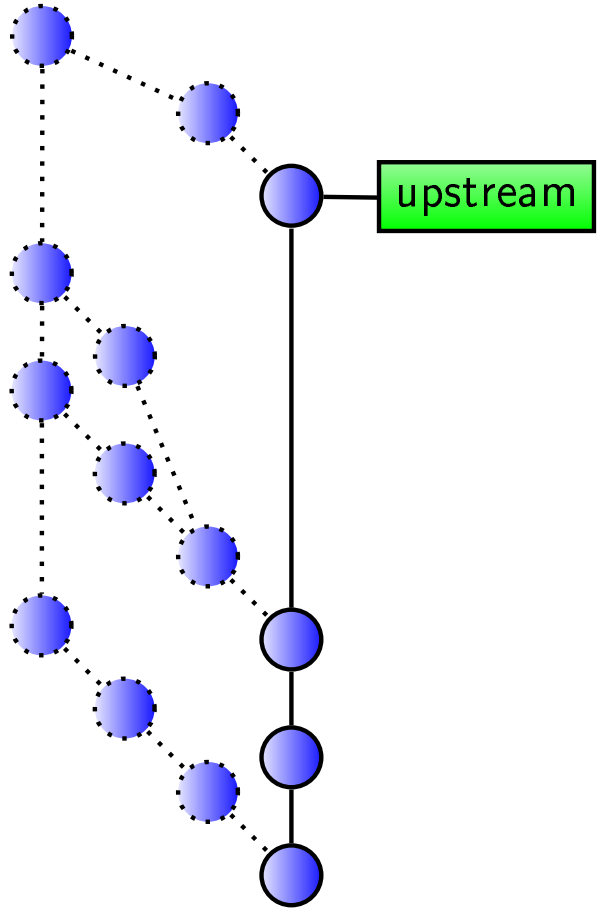
- an *upstream* branch for the upstream history (be it imported tarballs, upstream git commits, or imported tarballs on top of upstream git history)



branches used by git-dpm

As seen before, one always needs three points in the git history:

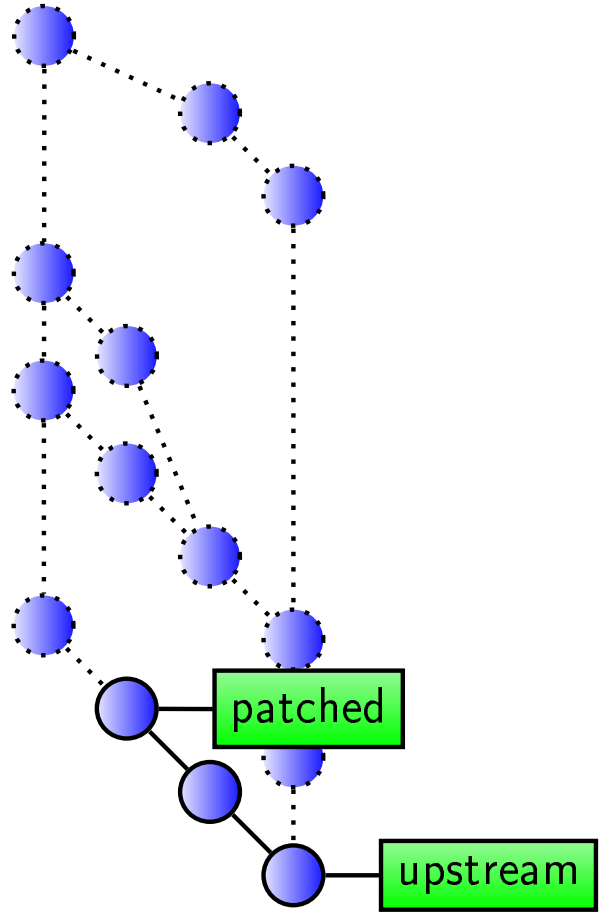
- an *upstream* branch for the upstream history (be it imported tarballs, upstream git commits, or imported tarballs on top of upstream git history)



branches used by git-dpm

As seen before, one always needs three points in the git history:

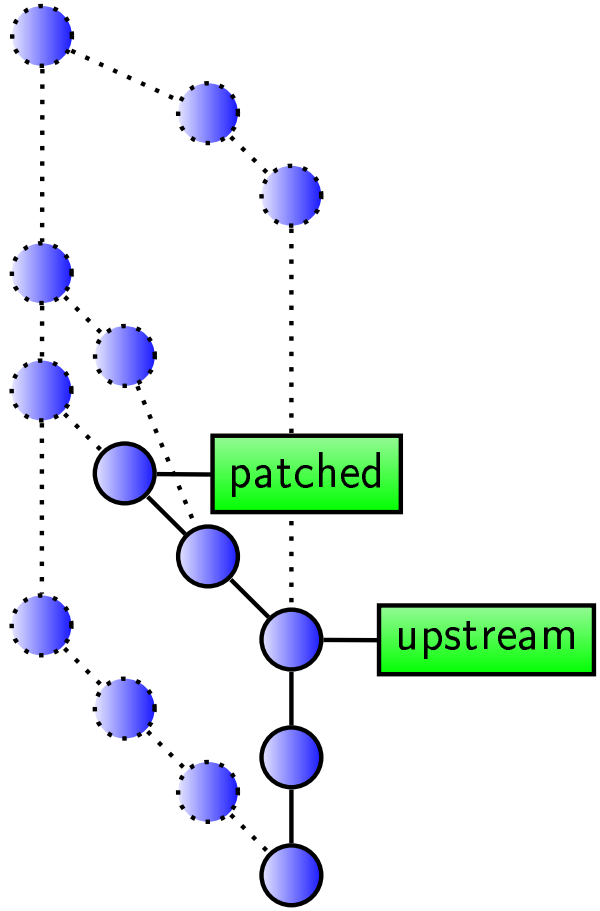
- an *upstream* branch for the upstream history (be it imported tarballs, upstream git commits, or imported tarballs on top of upstream git history)
- a *patched* branch based on the upstream branch, containing only changes to the upstream code.



branches used by git-dpm

As seen before, one always needs three points in the git history:

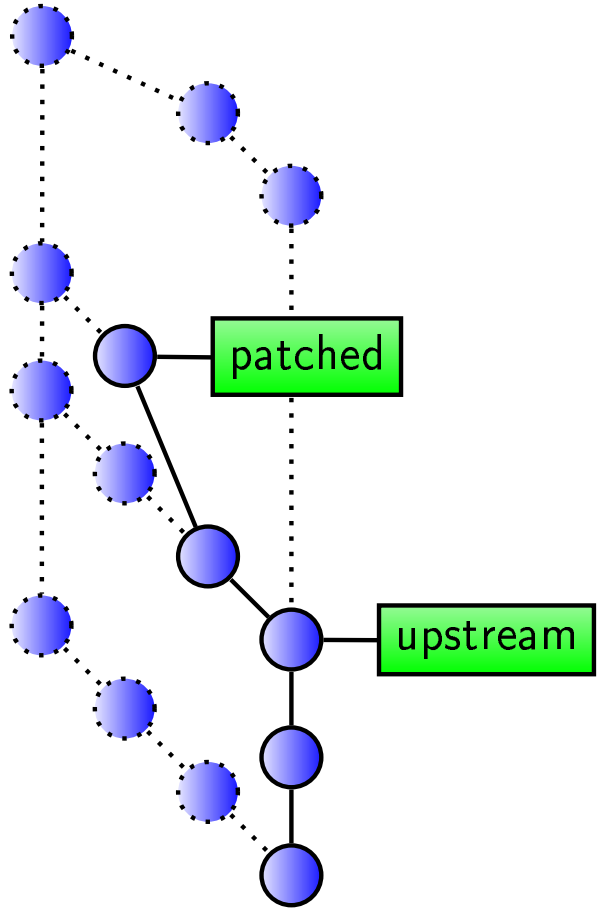
- an *upstream* branch for the upstream history (be it imported tarballs, upstream git commits, or imported tarballs on top of upstream git history)
- a *patched* branch based on the upstream branch, containing only changes to the upstream code.



branches used by git-dpm

As seen before, one always needs three points in the git history:

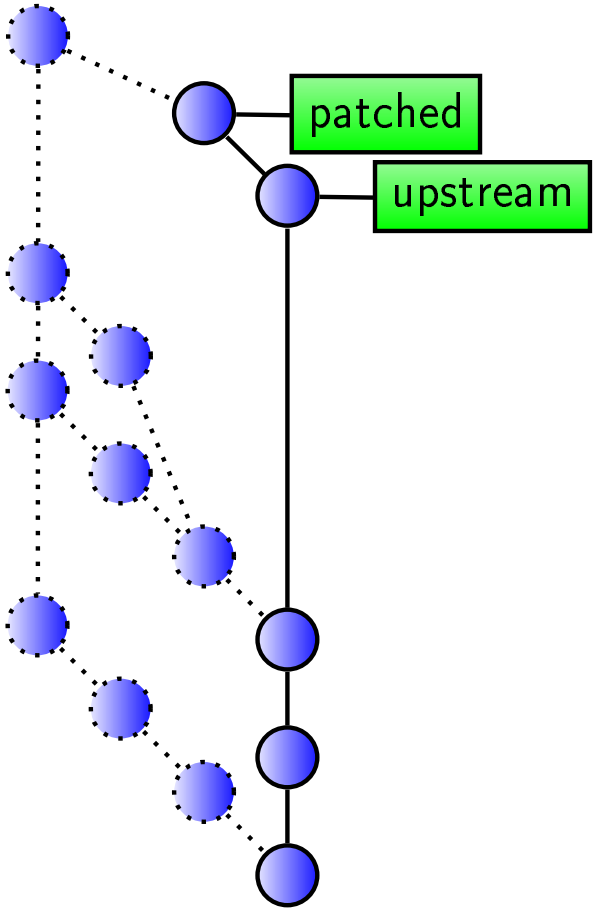
- an *upstream* branch for the upstream history (be it imported tarballs, upstream git commits, or imported tarballs on top of upstream git history)
- a *patched* branch based on the upstream branch, containing only changes to the upstream code.



branches used by git-dpm

As seen before, one always needs three points in the git history:

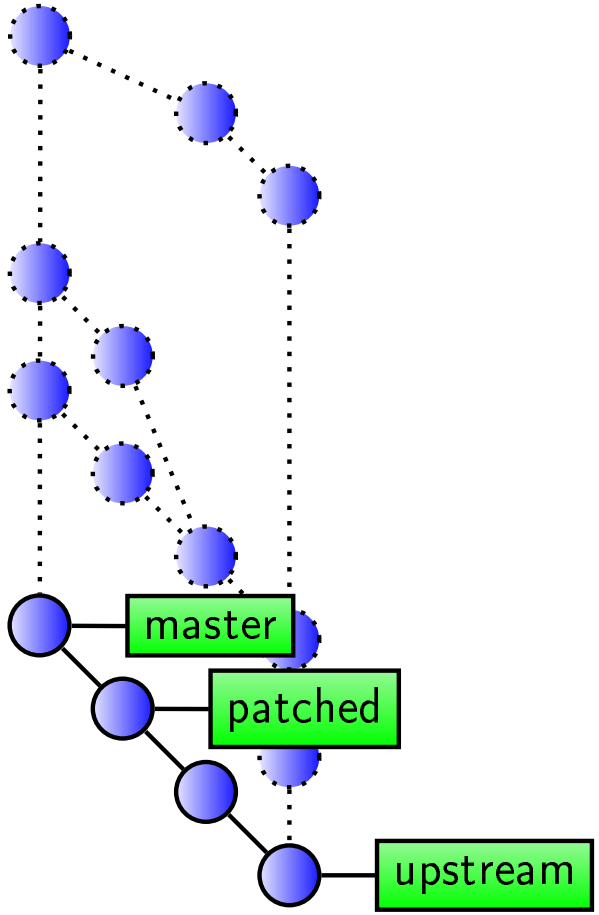
- an *upstream* branch for the upstream history (be it imported tarballs, upstream git commits, or imported tarballs on top of upstream git history)
- a *patched* branch based on the upstream branch, containing only changes to the upstream code.



branches used by git-dpm

As seen before, one always needs three points in the git history:

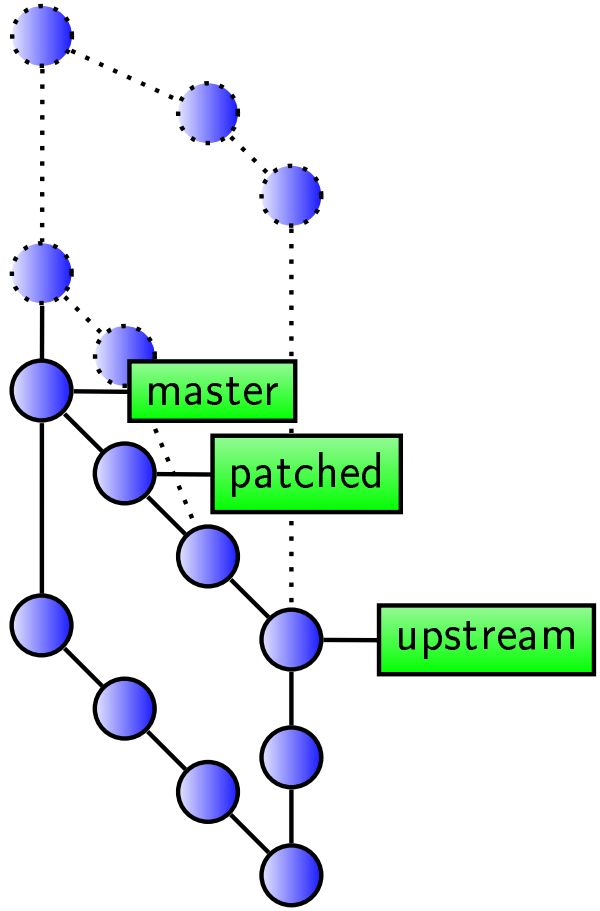
- an *upstream* branch for the upstream history (be it imported tarballs, upstream git commits, or imported tarballs on top of upstream git history)
- a *patched* branch based on the upstream branch, containing only changes to the upstream code.
- a *Debian* branch containing the Debian changes and having those other branches merged in (including previous states).



branches used by git-dpm

As seen before, one always needs three points in the git history:

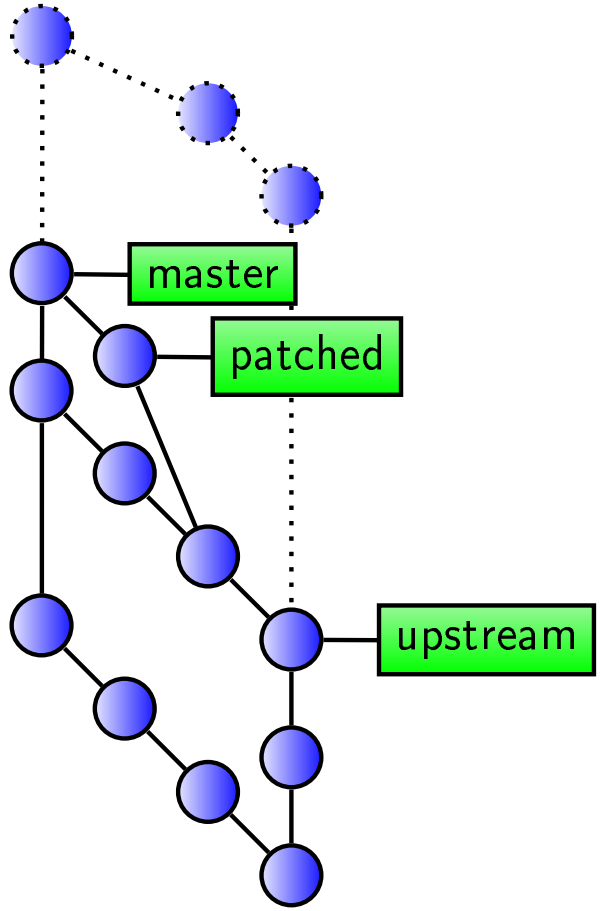
- an *upstream* branch for the upstream history (be it imported tarballs, upstream git commits, or imported tarballs on top of upstream git history)
- a *patched* branch based on the upstream branch, containing only changes to the upstream code.
- a *Debian* branch containing the Debian changes and having those other branches merged in (including previous states).



branches used by git-dpm

As seen before, one always needs three points in the git history:

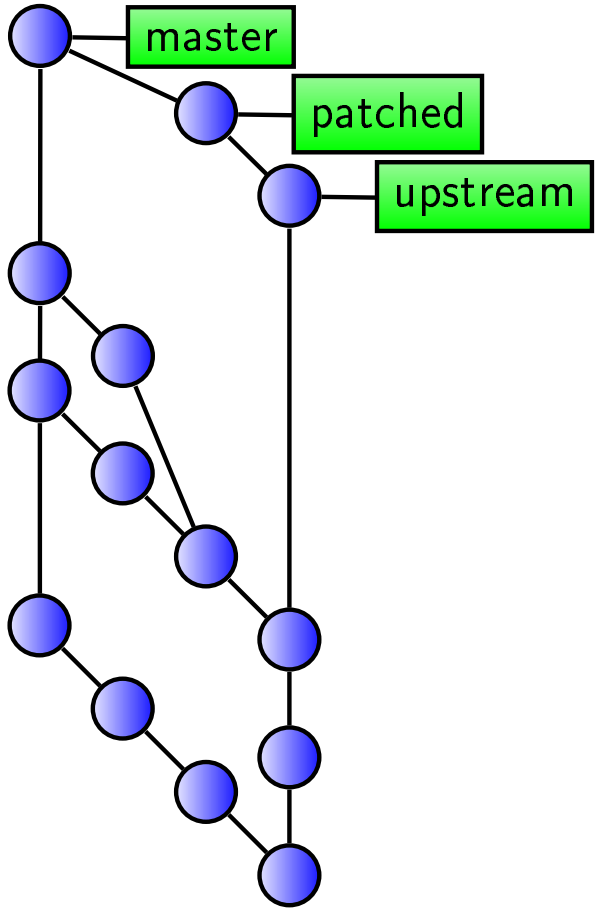
- an *upstream* branch for the upstream history (be it imported tarballs, upstream git commits, or imported tarballs on top of upstream git history)
- a *patched* branch based on the upstream branch, containing only changes to the upstream code.
- a *Debian* branch containing the Debian changes and having those other branches merged in (including previous states).



branches used by git-dpm

As seen before, one always needs three points in the git history:

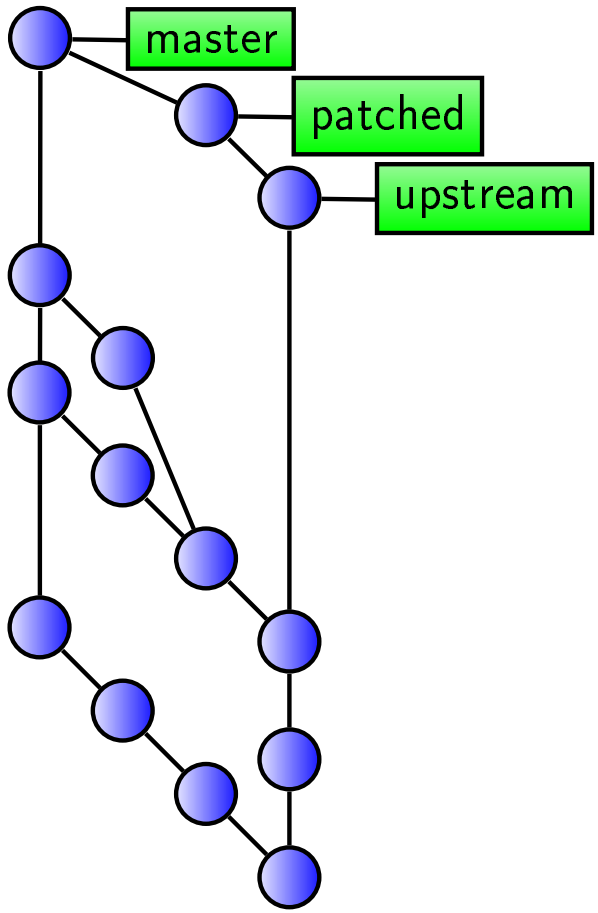
- an *upstream* branch for the upstream history (be it imported tarballs, upstream git commits, or imported tarballs on top of upstream git history)
- a *patched* branch based on the upstream branch, containing only changes to the upstream code.
- a *Debian* branch containing the Debian changes and having those other branches merged in (including previous states).



branches used by git-dpm

As seen before, one always needs three points in the git history:

- an *upstream* branch for the upstream history (be it imported tarballs, upstream git commits, or imported tarballs on top of upstream git history)
- a *patched* branch based on the upstream branch, containing only changes to the upstream code.
- a *Debian* branch containing the Debian changes and having those other branches merged in (including previous states).

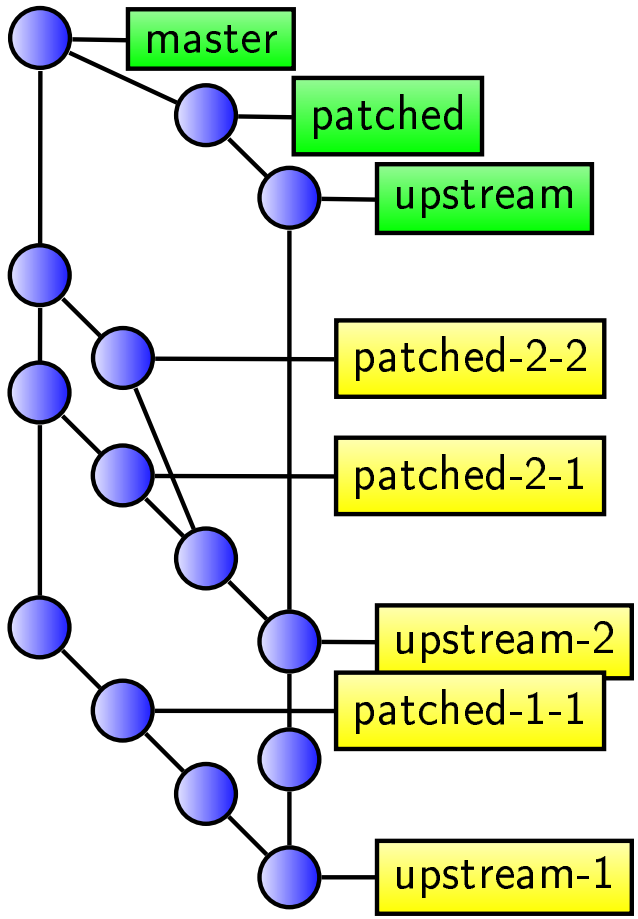


Only the Debian branch needs to be shared and published.

branches used by git-dpm

As seen before, one always needs three points in the git history:

- an *upstream* branch for the upstream history (be it imported tarballs, upstream git commits, or imported tarballs on top of upstream git history)
- a *patched* branch based on the upstream branch, containing only changes to the upstream code.
- a *Debian* branch containing the Debian changes and having those other branches merged in (including previous states).



Only the Debian branch needs to be shared and published.

branches used by git-dpm II

Unless configured differently, git-dpm assume that those branches are called

Debian branch	patched branch	upstream branch
`master`	`patched`	`upstream`
something	`patched-`*something*	`upstream-`*something*

So if it is called in a branch called `wheezy-patched` it assumes that to be belonging to a branch `wheezy` and an branch `wheezy-upstream`.

branches used by git-dpm III

Not all three branches are not suiteable to be globally visible git branches:

- As described above, git cannot share rebased branches itself very well.

branches used by git-dpm III

Not all three branches are not suiteable to be globally visible git branches:

- As described above, git cannot share rebased branches itself very well.
- Additionionally git is not very good in keeping related branches in sync at all.

branches used by git-dpm III

Not all three branches are not suiteable to be globally visible git branches:

- As described above, git cannot share rebased branches itself very well.
- Additionionally git is not very good in keeping related branches in sync at all.

Only the Debian branch needs to be published and the other branches are recorded by git-dpm stores the state of the related branches in `debian/.git-dpm`.

branches used by git-dpm III

Not all three branches are not suiteable to be globally visible git branches:

- As described above, git cannot share rebased branches itself very well.
- Additionionally git is not very good in keeping related branches in sync at all.

Only the Debian branch needs to be published and the other branches are recorded by git-dpm stores the state of the related branches in `debian/.git-dpm`.

- disavantage: the related branches are not directly visible via git means

branches used by git-dpm III

Not all three branches are not suiteable to be globally visible git branches:

- As described above, git cannot share rebased branches itself very well.
- Additionionally git is not very good in keeping related branches in sync at all.

Only the Debian branch needs to be published and the other branches are recorded by git-dpm stores the state of the related branches in `debian/.git-dpm`.

- disavantage: the related branches are not directly visible via git means (but those related to released versions are tagged, and they are easy to find with `git log` or `gitk`.)

branches used by git-dpm III

Not all three branches are not suiteable to be globally visible git branches:

- As described above, git cannot share rebased branches itself very well.
- Additionionally git is not very good in keeping related branches in sync at all.

Only the Debian branch needs to be published and the other branches are recorded by git-dpm stores the state of the related branches in `debian/.git-dpm`.

- disavantage: the related branches are not directly visible via git means (but those related to released versions are tagged, and they are easy to find with `git log` or `gitk`.)
- advantage: With only a single branch head to care about, you get all of git's advantages. You can just branch off a different development line with `git branch`, reset to the previous state with `git reset` and push it around and even merge easy cases.

git-dpm update-patches

Having modifications stored in git commit, one wants to export them to a `debian/patches/` series, so one can create a `3.0` source package.
One can do this manually with something like:

```
git format-patch -o debian/patches upstream..patched |
sed -e 's#^debian/patches/##' > debian/patches/series
```

or one can use `git-dpm update-patches`:

- does not need any arguments, but determines those automatically
- calls merge-patched first if necessary
- removes all old patches
- records that this state of the patched branch was exported
- commits the result

git-dpm update-patches

Having modifications stored in git commit, one wants to export them to a `debian/patches/` series, so one can create a `3.0` source package.
One can do this manually with something like:

```
git format-patch -o debian/patches upstream..patched |
sed -e 's#^debian/patches/##' > debian/patches/series
```

or one can use `git-dpm update-patches`:

- does not need any arguments, but determines those automatically
- calls merge-patched first if necessary
- removes all old patches
- records that this state of the patched branch was exported
- commits the result (including the new `debian/patches/` directory

building a source package

Committing `debian/patches/` into the Debian branch has some advantages I consider quite important:

- A checkout of a Debian branch looks essentially like a Debian source package extracted with `dpkg-source -x`.

building a source package

Committing `debian/patches/` into the Debian branch has some advantages I consider quite important:

- A checkout of a Debian branch looks essentially like a Debian source package extracted with `dpkg-source -x`.
- You can just call `dpkg-buildpackage` from a `git checkout`.

building a source package

Committing `debian/patches/` into the Debian branch has some advantages I consider quite important:

- A checkout of a Debian branch looks essentially like a Debian source package extracted with `dpkg-source -x`.
- You can just call `dpkg-buildpackage` from a `git checkout`.

This implies everyone can just clone a git repository managed by someone else with git-dpm, get the `.orig.tar` by calling `pristine-tar` or downloading that from somewhere, make some modifications, commit that into new patch using `dpkg-source -commit` and build and upload it and does not care for git-dpm at all

building a source package

Committing `debian/patches/` into the Debian branch has some advantages I consider quite important:

- A checkout of a Debian branch looks essentially like a Debian source package extracted with `dpkg-source -x`.
- You can just call `dpkg-buildpackage` from a `git checkout`.

This implies everyone can just clone a git repository managed by someone else with git-dpm, get the `.orig.tar` by calling `pristine-tar` or downloading that from somewhere, make some modifications, commit that into new patch using `dpkg-source -commit` and build and upload it and does not care for git-dpm at all (of course if using git-dpm again you need to apply that patch into the patched branch then).

upstream branch

The restrictions on the upstream branch `git-dpm` imposes are:

- It must not contain files not in the `.orig.tar` and needed in the build.
  - As the patches are generated relatively to this branch, no patch generated by git-dpm can introduce a file already in this branch.
  - Having additional files is otherwise OK but every file in the upstream branch but not the `.orig.tar` must be deleted in the Debian branch to not confuse `dpkg-source`.

upstream branch

The restrictions on the upstream branch `git-dpm` imposes are:

- It must not contain files not in the `.orig.tar` and needed in the build.
  - As the patches are generated relatively to this branch, no patch generated by git-dpm can introduce a file already in this branch.
  - Having additional files is otherwise OK but every file in the upstream branch but not the `.orig.tar` must be deleted in the Debian branch to not confuse `dpkg-source`.
- It must contain all the files from the `.orig.tar` file that are needed to build the package verbatimly.
  - Missing files found in the `.orig.tar` is OK, but only if you do not need them at package build time (for example autogenerated `configure` scripts you regenerate at build time anyway). For consistency ideally also delete those files in your `debian/rules clean` target to make a clone of your repository look more like a `dpkg-source -x` with `debian/rules clean` run.

upstream branch

The restrictions on the upstream branch `git-dpm` imposes are:

- It must not contain files not in the `.orig.tar` and needed in the build.
  - As the patches are generated relatively to this branch, no patch generated by git-dpm can introduce a file already in this branch.
  - Having additional files is otherwise OK but every file in the upstream branch but not the `.orig.tar` must be deleted in the Debian branch to not confuse `dpkg-source`.
- It must contain all the files from the `.orig.tar` file that are needed to build the package verbatimly.
  - Missing files found in the `.orig.tar` is OK, but only if you do not need them at package build time (for example autogenerated `configure` scripts you regenerate at build time anyway). For consistency ideally also delete those files in your `debian/rules clean` target to make a clone of your repository look more like a `dpkg-source -x` with `debian/rules clean` run.

The easiest way it just to use the full contents of the `.orig.tar` as HEAD commit of your upstream branch.

recording and importing upstream files

- `git-dpm prepare`
  will make sure the correct `.orig.tar` is available or try to recreate it with pristine-tar. It will also make sure or create the upstream branch as local git branch.

recording and importing upstream files

- `git-dpm prepare`
  will make sure the correct `.orig.tar` is available or try to recreate it with pristine-tar. It will also make sure or create the upstream branch as local git branch.
- `git-dpm (record-)new-upstream`
  will record the current state of your upstream branch and record the checksum of the named `.orig.tar` file to belong to this. i.e.: you create a upstream branch any way you want and call this to tell git-dpm to remember it in `debian/.git-dpm`.

recording and importing upstream files

- `git-dpm prepare`
  will make sure the correct `.orig.tar` is available or try to recreate it with pristine-tar. It will also make sure or create the upstream branch as local git branch.
- `git-dpm (record-)new-upstream`
  will record the current state of your upstream branch and record the checksum of the named `.orig.tar` file to belong to this. i.e.: you create a upstream branch any way you want and call this to tell git-dpm to remember it in `debian/.git-dpm`.
- `git-dpm rebase-patched`
  will call `git rebase` with the correct arguments to rebase your patched branch on the new upstrem branch (**read** and follow the instructions git gives you to resolve any possible conflicts).

recording and importing upstream files

- `git-dpm prepare`
  will make sure the correct `.orig.tar` is available or try to recreate it with pristine-tar. It will also make sure or create the upstream branch as local git branch.
- `git-dpm (record-)new-upstream`
  will record the current state of your upstream branch and record the checksum of the named `.orig.tar` file to belong to this. i.e.: you create a upstream branch any way you want and call this to tell git-dpm to remember it in `debian/.git-dpm`.
- `git-dpm rebase-patched`
  will call `git rebase` with the correct arguments to rebase your patched branch on the new upstrem branch.
- `git-dpm import-new-upstream`
  This creates a new commit in your upstream branch with the contents of the given `.orig.tar` file and records that as new upstream. Ypu can give additional parents this commit should have (useful to have upstream's git history included in there).

the patched branch

Collaboration gets easier if the patchd branch cannot get stale. Thus git-dpm by default removes it whenever it is not currently needed (i.e. after merging it in).
The suggested workflow is:

- `git-dpm checkout-patched`
  Creates the patched branch as git branch and checks it out.
- add new commits, amend commits, edit patches with interactive rebase, . . . , . . .
- record the new state and merge it back using `git-dpm merge-patched`, `git-dpm update-patched` or `git-dpm dch`.

When collaborating on a project and only one participant changed the patch branch, you can just merge the Debian branches (or let `git pull` merge them for you). If both did so, check out both patched branches, merge them, do an interactive rebase to linearize the result and chose which commits to use and then merge that back.

git-dpm dch

`git-dpm dch` is a wrapper around `dch` that

- calls dch (passing any arguments to it after a `--`)

git-dpm dch

`git-dpm dch` is a wrapper around `dch` that

- calls dch (passing any arguments to it after a `--`)
- calls `update-patches` (including `merge-patched`) if necessary,

git-dpm dch

`git-dpm dch` is a wrapper around `dch` that

- calls dch (passing any arguments to it after a `--`)
- calls `update-patches` (including `merge-patched`) if necessary,
- calls `git commit`
  - giving it a commit message based on the changelog commit message

git-dpm dch

`git-dpm dch` is a wrapper around `dch` that

- calls dch (passing any arguments to it after a `--`)
- calls `update-patches` (including `merge-patched`) if necessary,
- calls `git commit`
  - giving it a commit message based on the changelog commit message
  - passes arguments as `-a` (so one can also include other `debian/` changes in the same commit.

git-dpm import-dsc

`git-dpm import-dsc`

- is there to import a .dsc file
- is the easiest way to start a git-dpm project
- will try to import any patches found in the package to commits.

git-dpm import-dsc

`git-dpm import-dsc`

- is there to import a .dsc file
- is the easiest way to start a git-dpm project
- will try to import any patches found in the package to commits.

There are many different options. For example (see manpage for more):

- `--use-changelog`
  Treat `.dsc` files like a vcs you are importing and set the author, date and message of the commits from the changelog.

git-dpm import-dsc

`git-dpm import-dsc`

- is there to import a .dsc file
- is the easiest way to start a git-dpm project
- will try to import any patches found in the package to commits.

There are many different options. For example (see manpage for more):

- `--use-changelog`
  Treat `.dsc` files like a vcs you are importing and set the author, date and message of the commits from the changelog.
- `--patch-system`
  Explicitely name a patch system the package uses to import packages from. Mostly useful to name explicitely are `none` to not import anything and `history` that will split directly-applied 1.0 diffs into one patch per imported version.

git-dpm tag

`git-dpm tag` is a convienience wrapper to create tags for one release. For each for the three branches a tag is created.
The names are configurable, but the defaults are

		––named
Debian	debian*%e*-*%v*	*%p*-debian*%e*-*%v*
patched	patched*%e*-*%v*	*%p*-patched*%e*-*%v*
upstream	upstream*%e*-*%u*	*%p*-upstream*%e*-*%u*

With *%p* package name, *%e* epoch, *%v* version without epoch, *%u* version without epoch and revision.

git-dpm tag

`git-dpm tag` is a convienience wrapper to create tags for one release. For each for the three branches a tag is created.
The names are configurable, but the defaults are

		––named
Debian	debian*%e*-*%v*	*%p*-debian*%e*-*%v*
patched	patched*%e*-*%v*	*%p*-patched*%e*-*%v*
upstream	upstream*%e*-*%u*	*%p*-upstream*%e*-*%u*

With *%p* package name, *%e* epoch, *%v* version without epoch, *%u* version without epoch and revision.
I usually tag once I upload the resulting source package, but your milage may vary.

git-dpm record-dsc

With `git-dpm record-dsc` you can store a signed source package you created by using `pristine-tar` to store the `.debian.tar` and storing the signed `.dsc` in a branch called `dscs`.

- Any time left?
- Any questions?