How to Manage Stacked Diffs with Git

Kaung Htet

November 22, 2024

Contents

1 Stacked Diffs

- Today I am going to talk about stacked diffs.
- You might have also heard it as stacked diffs/stacked branches/stacked PRs workflows.
- This talk is more focused on `How to manage staked diffs/branches?` rather than `Why?`
  - Most resources online don't really talk about `how` and point to `Phabricator`, `Graphite`., etc
- Talk is also tailored more to our repositories.

2 My Background

- Started out with `svn` (very brief)
- Mercurial `(hg)`,
- Git `(git)`

3 What are Stacked Diffs?

- A workflow concept that invovles stacking `a series of small, atomic and dependent changes` on top of one another.
- Allows code reviews to happen on each small change, making them more efficient and managable.

- Every **"commit" (diff)** becomes a code review.
  - That's the Phabricator approach.
  - Trying to do that without tooling in our context is unintuitive and impractical.
  - The essense to have work on top of local `master` and the tool will create a new remote branch for each commit and a respective new code review ("PR" in our speak) for each commit against `master`.

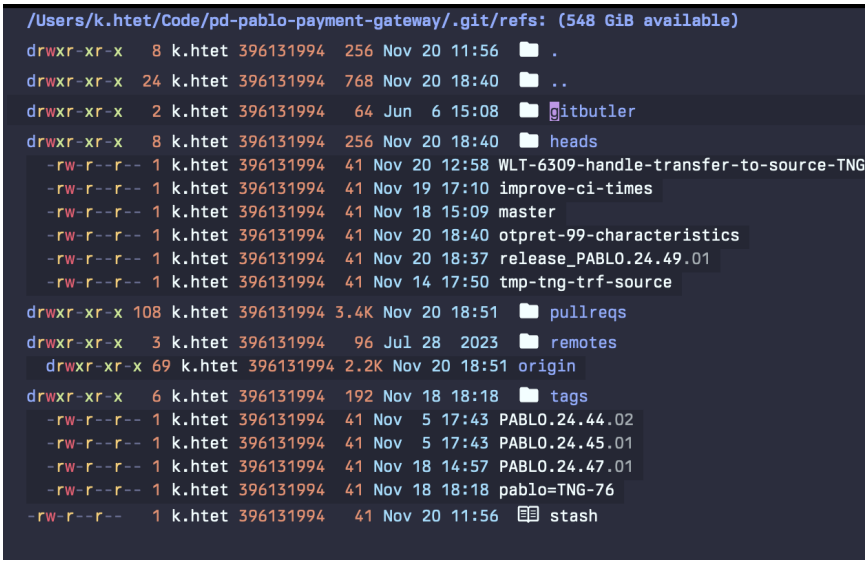
```
master (local):    A---B---C---D---E---F
                                \
                          (origin/master)

- D, E, F becomes individual branches
- D is PR against C
- E is PR against C
- F is PR against C

In your only using Github UI,
- in E PR view, you see everything from D+E.
- in F PR view, you see everything from D+E+F.

Landing PRs.
The merge happens for (C)
Tooling sync back to your local branch and rebase against origin-master
master (local):    A---B---C---D---E---F
                                    \
                                (origin/master)
```

4 Branches in `Git`

- There is no such thing as "branches"
- Branches are named references to commits.
  - Essentially 41-byte referecne pointing to a commit hash.
  - Exteremely cheap operation compared to other VCS.

4.0.1 Compared to other D-VCS?

- Git branches are still very flexible since it treats them **purely as movable pointers to commits**.
- e.g., **Mercurial** branches are permanent markers in commit history.



5 Stacking branches prone to be extremely painful

- In theory, stacked diffs seem like a pleasant workflow.
- Stacked branches are dependent on each other.
- Anytime there is an upstream change,
  - Every branch in the rest of the stack needs to be recursively rebased on top of one another to stay in sync again.
  - Child branches from D+2 onwards can be stuck in limbo with if D+1 is merged with `git merge --squash` to `trunk`.
- Potential of cascading merge conflicts.
- Can get real messy
  - If you are not crystal clear on what's happening. (With/without tooling).

- This talk will suggest workflows/tooling.
  - Without being clear on what's happening underneath, you are prone to a lot of nested mess.

6 True `merge` vs `squash merge` vs `rebase merge`

- A true `merge` brings in no changes but connects the two or more histories with a parent pointer to each.
  - **Preserves** the complete branch history and shows the relationship between branches.
- `Squash merge` **combines** all feature commits into a single commit on the base branch, loses branch relationships.
- `Rebase merge` replays commits for linear history, no merge commit, new hashes.

```
Starting state:
main        A---B---C
                \
feature         D---E---F


Regular Merge:   A---B---C---------M
                     \           /
                      D---E---F

Squash Merge:    A---B---C---S
                     \
                      D---E---F

Rebase Merge:    A---B---C---D'---E'---F'
```

7 Problem with `squash merge`

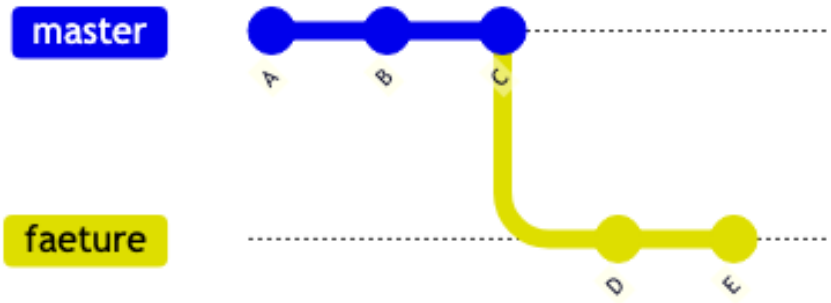
- `squash` merges are not true merges.
- It's simply one independent commit added to the target branch containing all the changes *copied* from the resulting diff of the source + target branches.

- Without any reference to the source branch, literally except for comments in the commit message.
- Really against squash merges to trunk.
- Personally a little iffy about this choice, but it makes sense for us given how we use PRs and branches.
  - At the end of the day the goal of `master` to have atomic changset in each commit that lands on it.

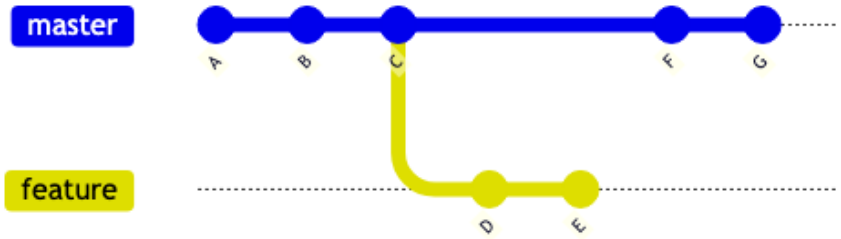
8 Rebase

- `Rebasing` is a core part of maintaing stacked branches.
- **3 Principles** to follow generally.
  - Keep branches small
  - `Rebase` on top of trunk to keep it up to date.
  - You want each branch to always be on top of each other `in a linear manner`.

8.0.1 Start a `feature` branch from `master`

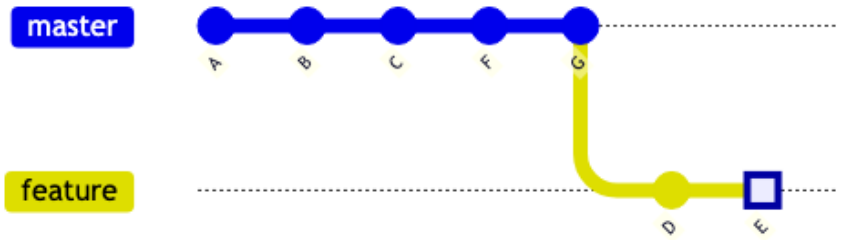


8.0.2 `master` has updates



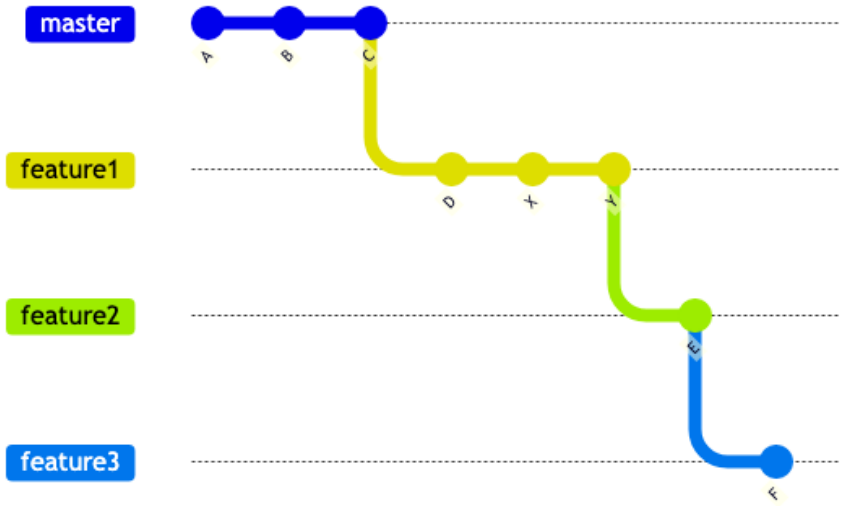
8.0.3 `rebase` feature onto `master`

```
git fetch origin master:master # pull down changes
git rebase master
```



9 Making stacked branches

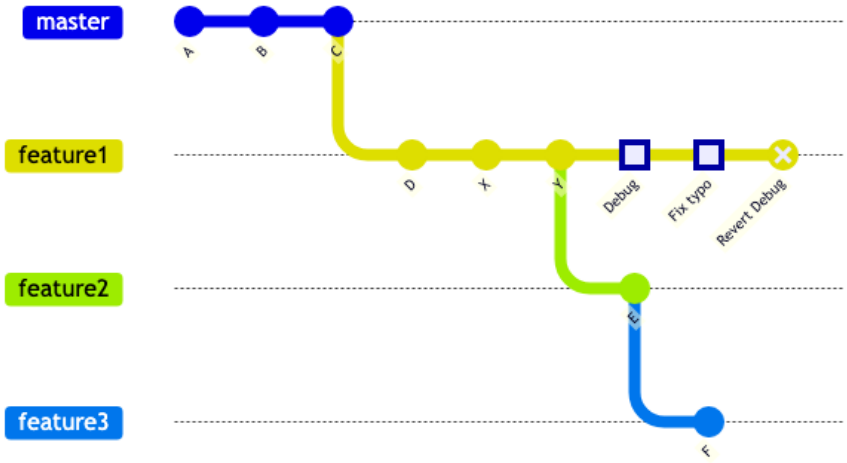
9.0.1 Create stacked branches



9.0.2 Keep your changset small

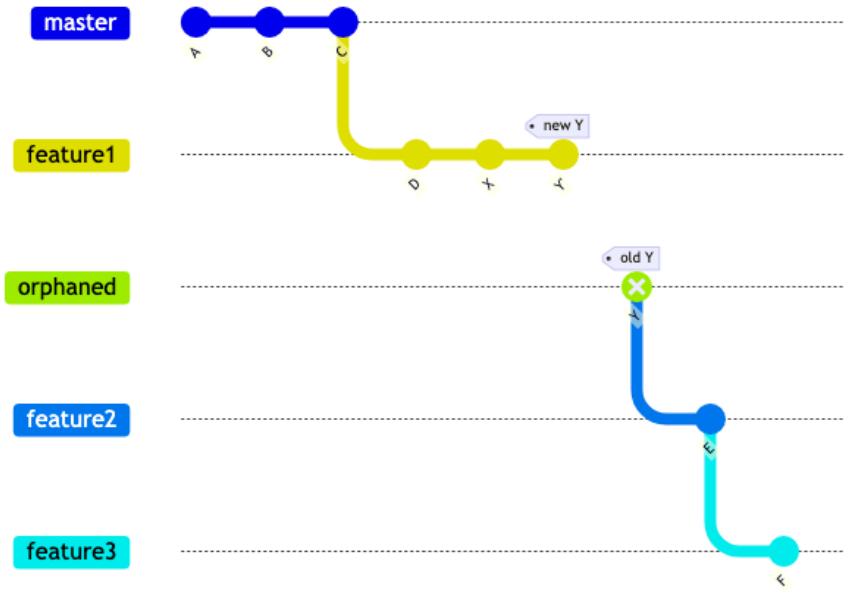
- Say you keep working on `feature1`

```
git checkout feature1
# ... more work here
git commit -am 'Debug'
# ... more work here
git commit -am 'Fix typo'
git revert HEAD^
```

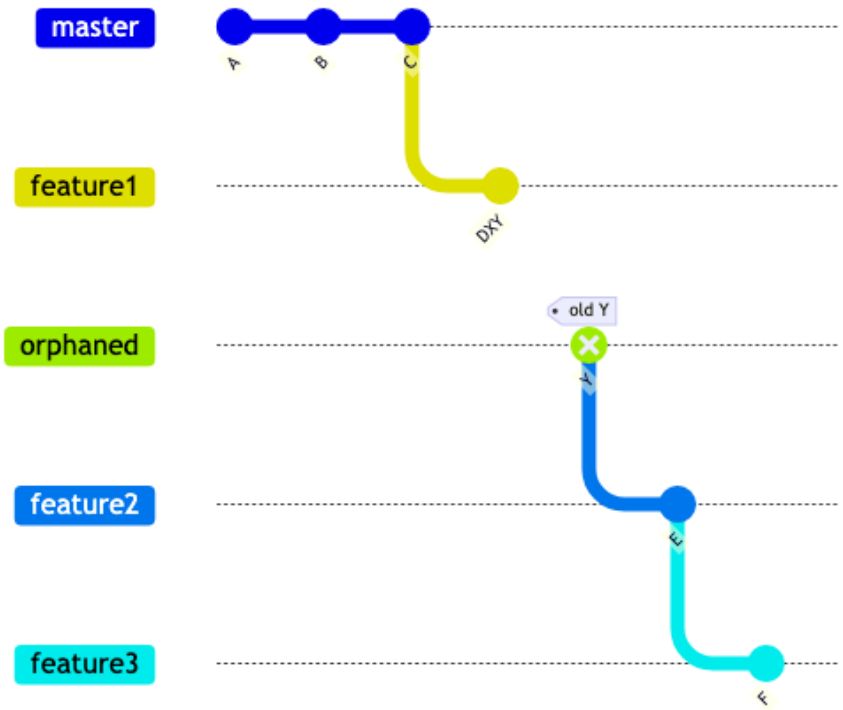


- `rebase` feature1

```
git rebase -i # DEMO?
```

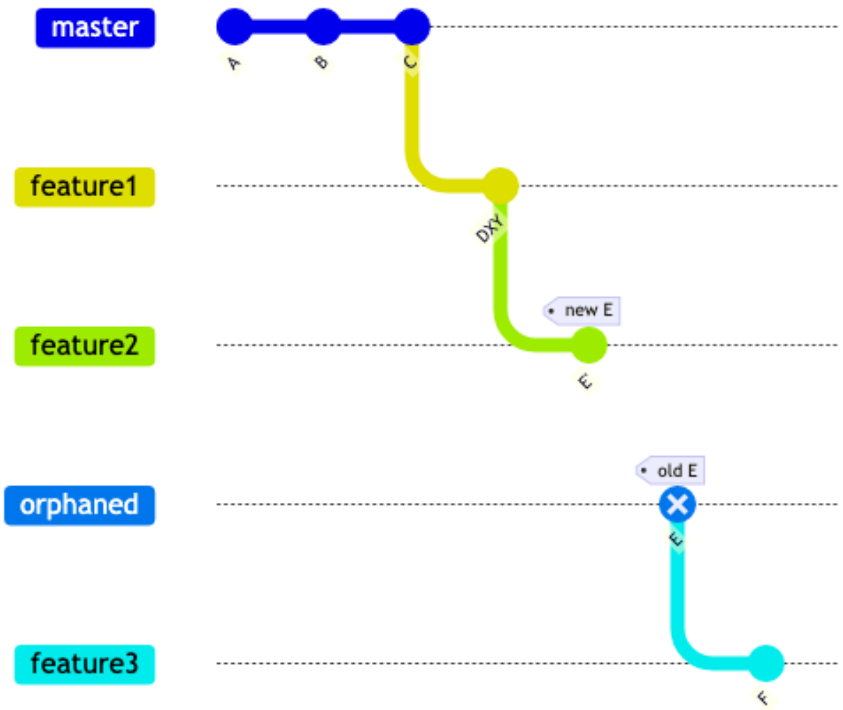


- In fact keep it as lean and atomic as possible
  - Not necessarily mean literally one commit

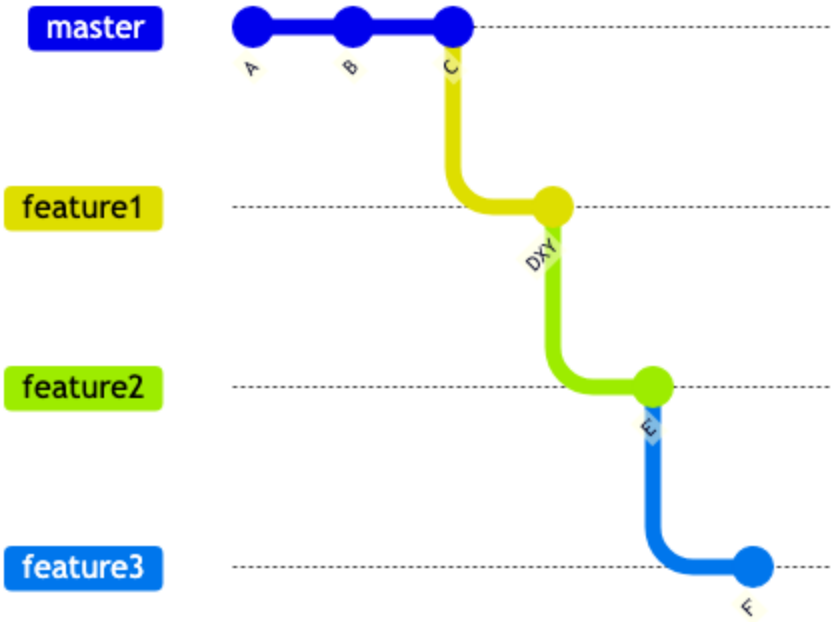


10 Rebase child branches one by one

```
git checkout feature2
git rebase feature1
```




```
git checkout feature3
git rebase feature2
```

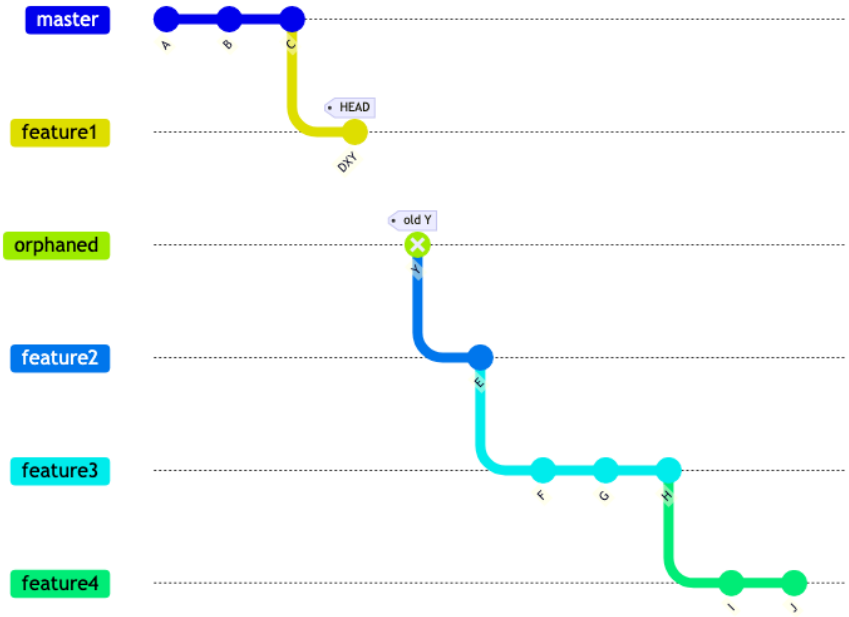


11 Can we do it in one go?

- Yes, enter `git rebase --onto` for surgically more precise rebasing.

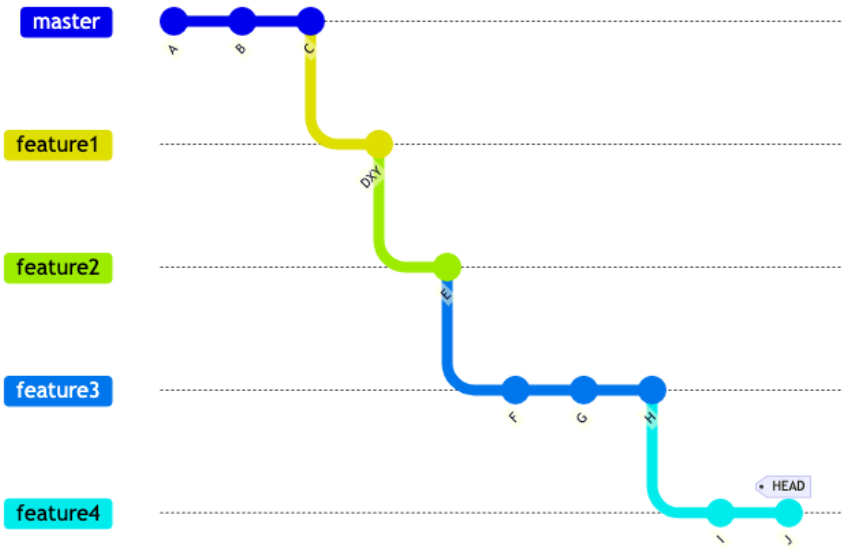
```
git rebase --onto <newparent> <old parent> HEAD
git rebase --onto <newparent> <old parent> <until>
```

11.0.1 Rewind the state after `rebasing` feature1



11.0.2 Go to the innermost child branch at the end of the stack.

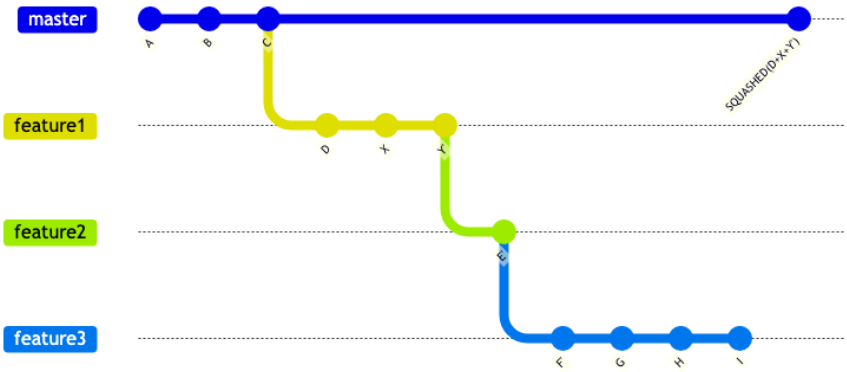
```
git checkout feature4
git rebase --onto feature1 feature2^ feature4 --update-refs
```



12 Merging `feature1` to `master`.

12.0.1 feature1 has been reviewed and finally merged to `master`.

```
git checkout master
git merge --squash feature1
# At github remote, it auto delete the remote branch (origin/feature1) -- Our settings
```

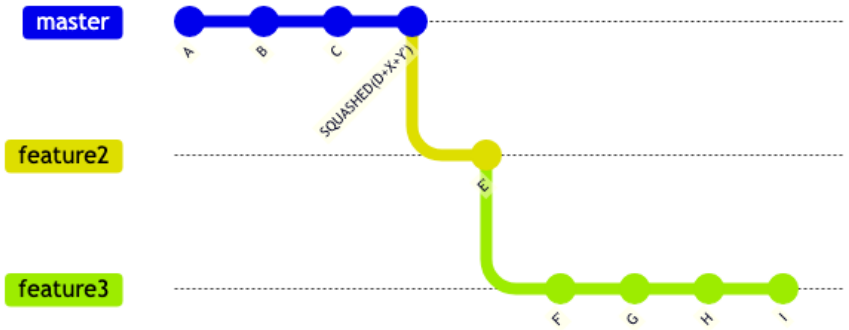


12.0.2 Now what?

- Your local feature1 branch still exits.

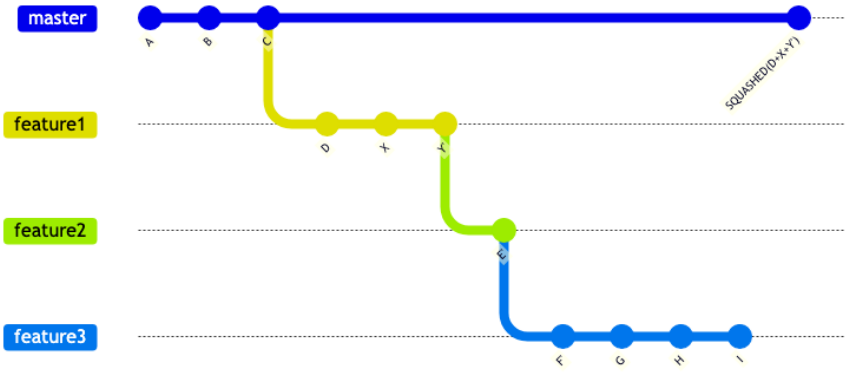
- After pulling, Git does not even know that it was merged to master
- When you delete, you have to use `-D` instead of `-d`.
- Anyway, let's continue the rebase.

```
git branch -D feature1
git checkout feature3
git rebase --onto origin/master feature2^ feature3 --update-refs

# still need to update the remote refs
git push --force-with-lease origin feature3:feature3
git push --force-with-lease origin feature2:feature2
```



13 What exactly is `--update-refs` ?

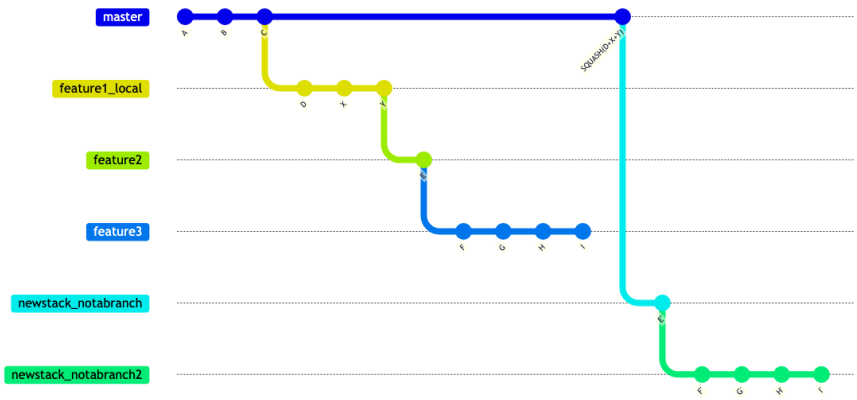
13.0.1 Go back to post squash merge state



13.0.2 Without `--update-refs`

- What you wanted still happens, old branch references still remain

```
git checkout feature3
git rebase --onto origin/master feature2^ feature3 --udpate-refs
git reset --hard origin/feature3
```



14 To sum up

- Keep changset small in a branch. Always clean it up. (`git rebase -i`)
- If a branch is merged, use `git rebase --onto` with `--update-refs` from the innermost branch

```
# after every merge, go to innermost branch
git rebase --onto <newparent> <oldparent> <until> --update-refs
```

```
# update the remote refs
for all branches from branch+1 to innermost branch
  git push -f origin <ref>
endfor
```

14.0.1 Can I automate this part.

- Yes, check out Git Town.

```
git town hack <my-feature>
# work on stuffs and create commits
git town append <branch2>
git town append <branch3>
git town append <branch4>
git town append <branch5>

# Ship branch2 for example
git town ship branch2

# And then from your youngest branch
git town sync
```