Git: Beyond Basics

Semmy Purewal

Who am I?

- Distant Past: Teacher/Professor
- Past: FANG engineer
- Present: Software Consultant (focus on DX, Process, Scaling, Hiring)
- Future: ??

Who am I?

- Distant Past: Teacher/Professor
- Past: FANG engineer
- Present: Software Consultant (focus on DX, Process, Scaling, Hiring)
- Future: ??

- Add me on LinkedIn or email me at [myfirstandlastnameseparatedbyadot]@gmail.com.

I acknowledge…

- Every team is different
- What works in one company will not work in all companies
- Shipping code should take precedence over new processes

I acknowledge…

- Every team is different
- What works in one company will not work in all companies
- Shipping code should take precedence over new processes

Also…

- Effective use of Version Control has made me a better engineer

What is Software Engineering?

Software Engineering

“Software Engineering is programming integrated over time.”

Titus Winters
Software Engineering at Google

Time implies History

Why is the history of your code important?

Code History

“History is who we are and *why* we are the way we are.”

David McCullough
American Historian

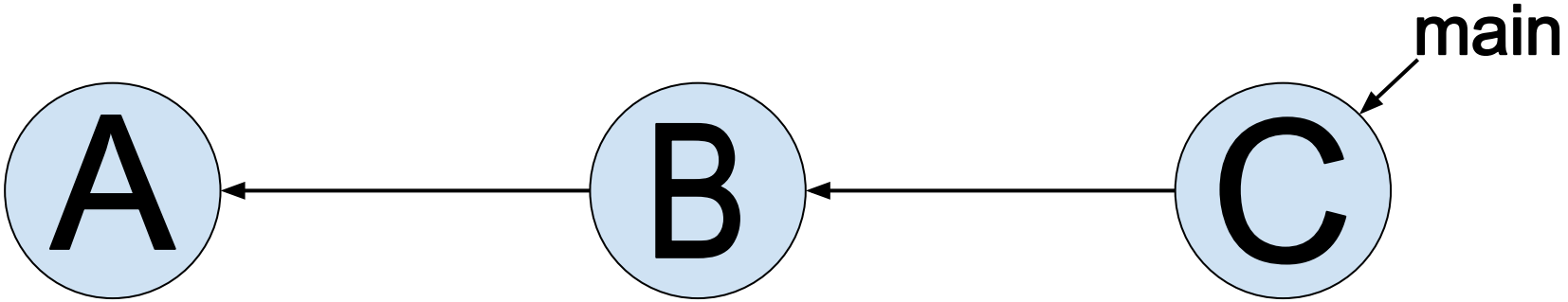
Version Control

If you are a professional, your version control skills are as important as your programming skills.

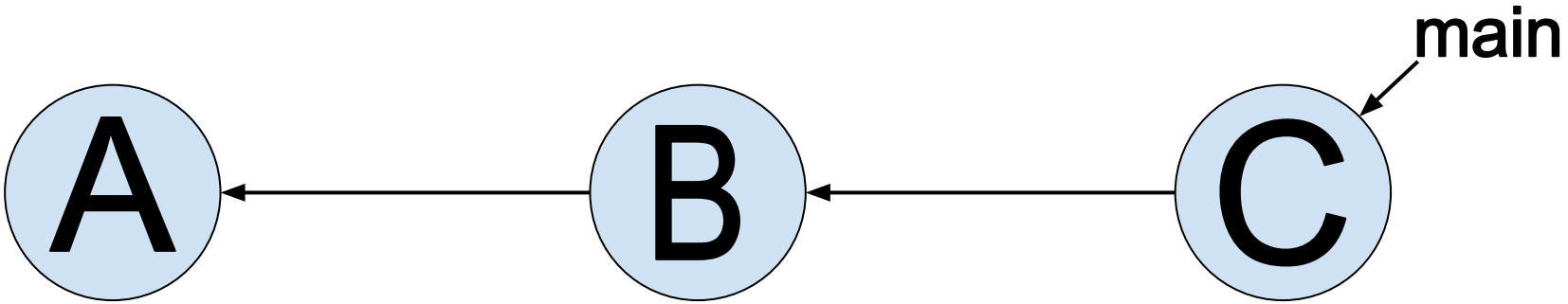
A Brief History of Version Control

Distributed vs. Centralized

Abstract Git History

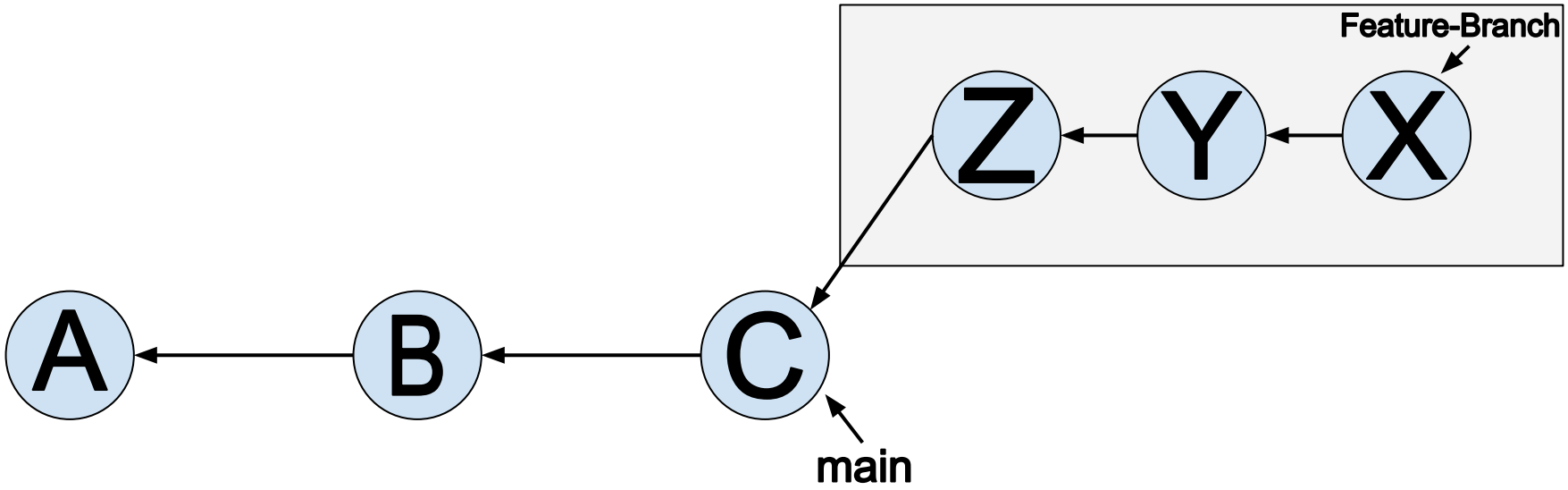


Abstract Git History

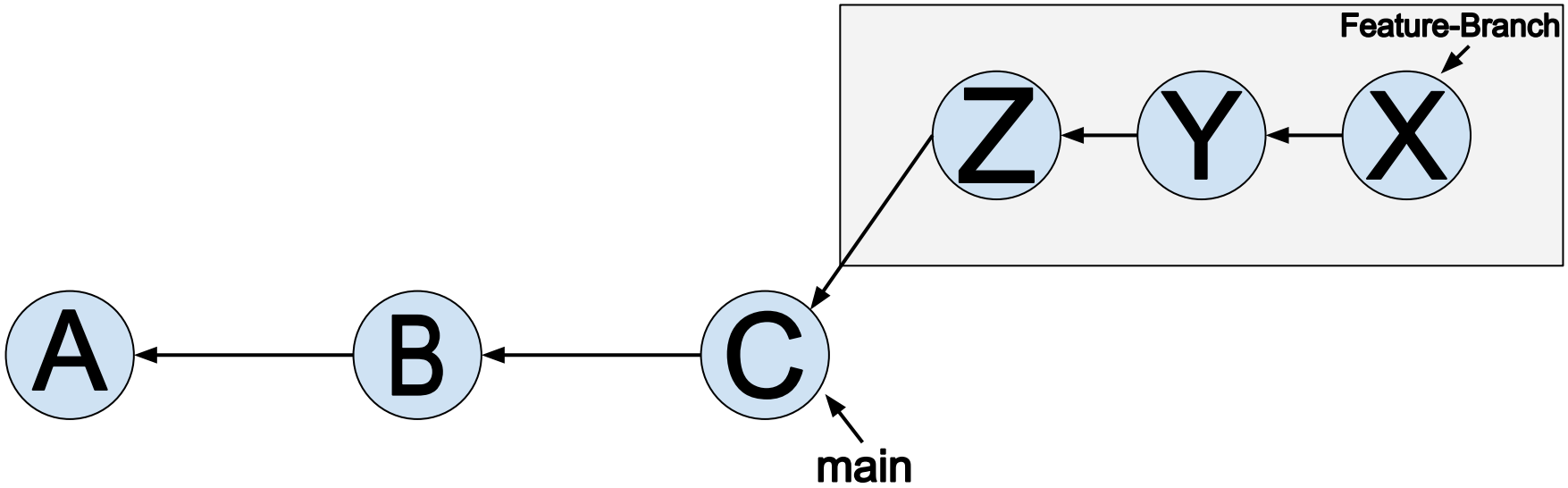


A git repository is a set of **immutable** commits. Each “normal” commit has an optional unique parent pointer. A **ref** (**main**) is just a reference to a specific commit.

Branches

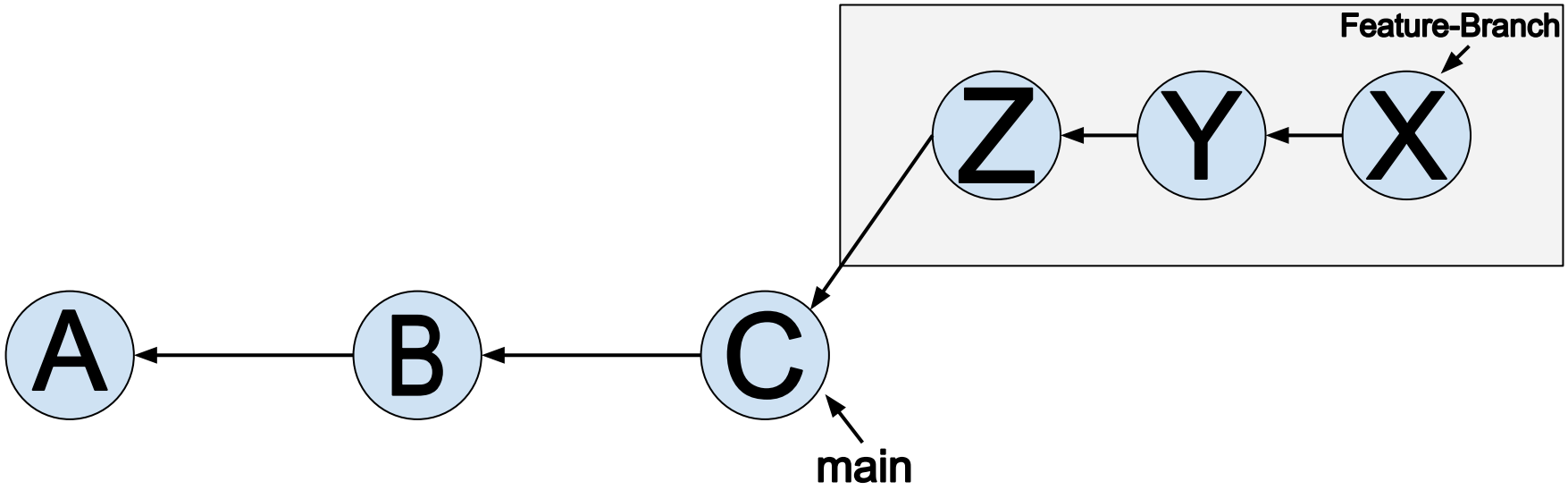


Branches



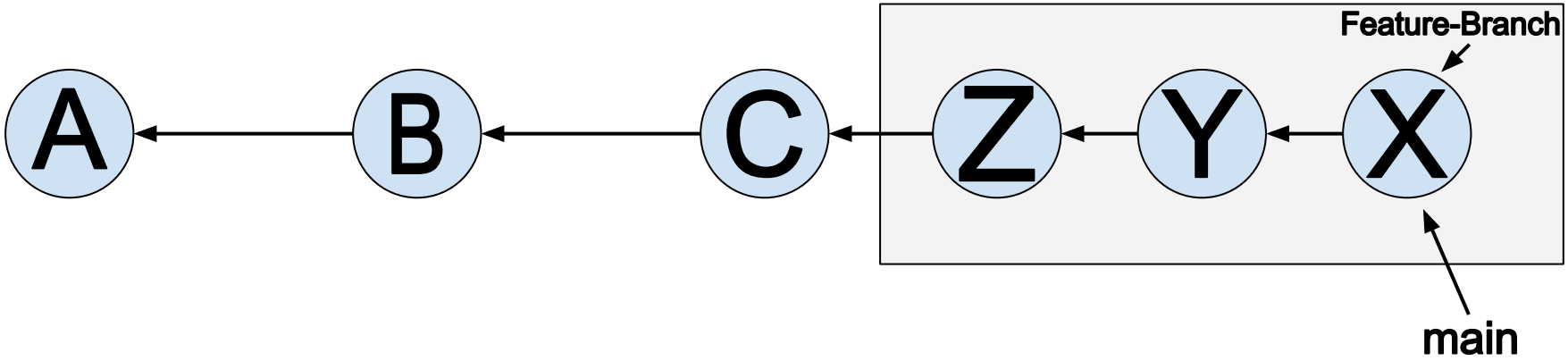
A branch is simply a new ref (**Feature-Branch**) to a specific commit.

Branches



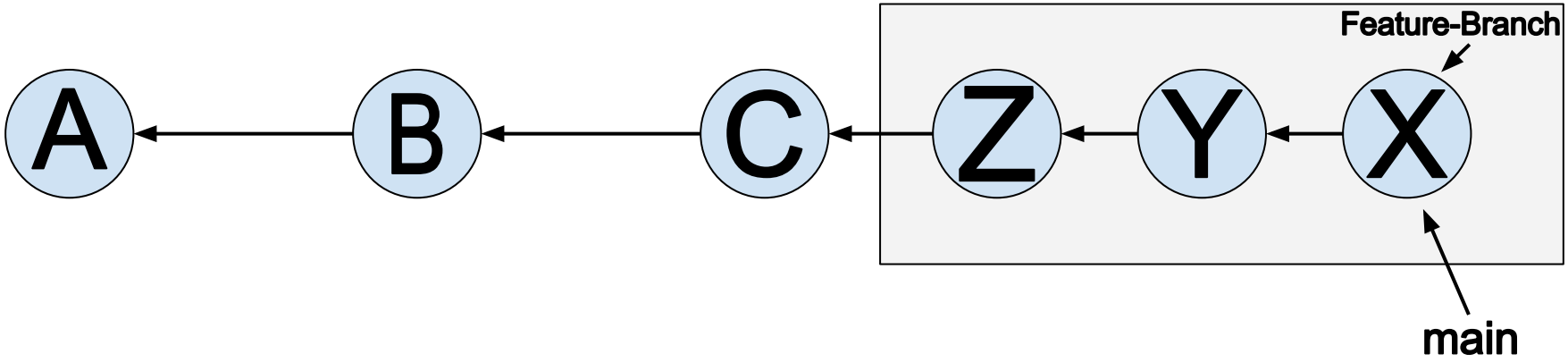
What if we want to “merge” **Feature-Branch** into **main**?

Fast-Forward



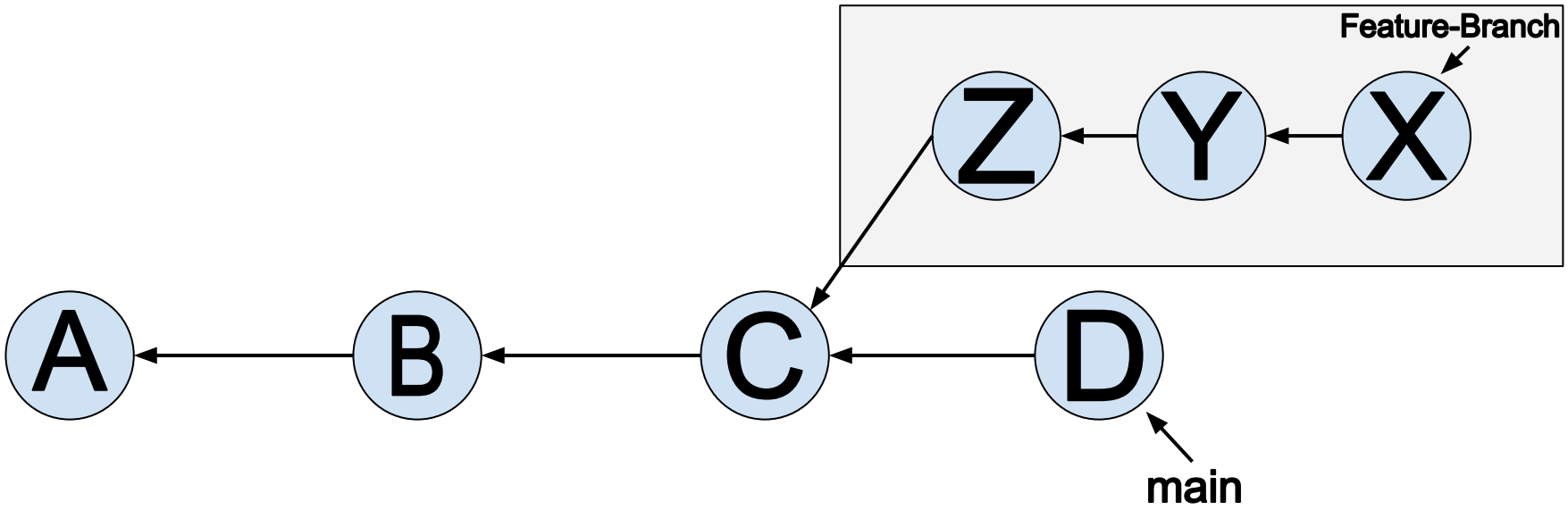
Fast-Forwarding simply updates the target branch's ref to point to the source branch's ref.

Fast-Forward



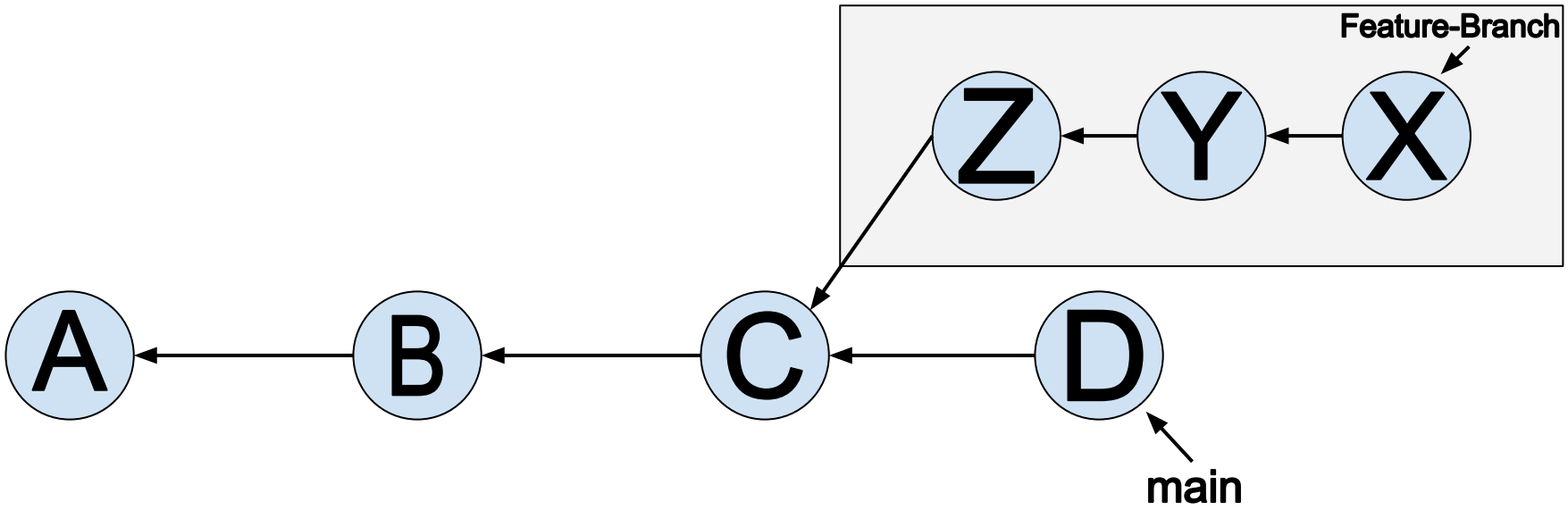
In a repo with lots of developers, fast-forwards are uncommon.

Branches



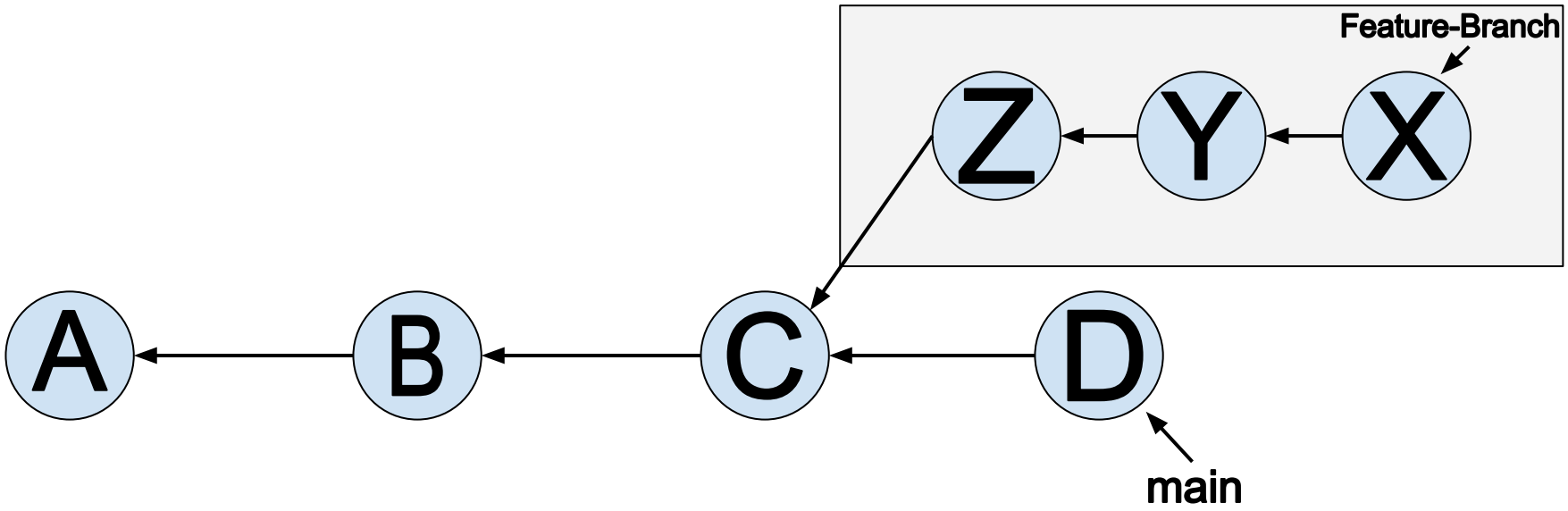
Refs can be updated independently of each other.

Non-Linear History



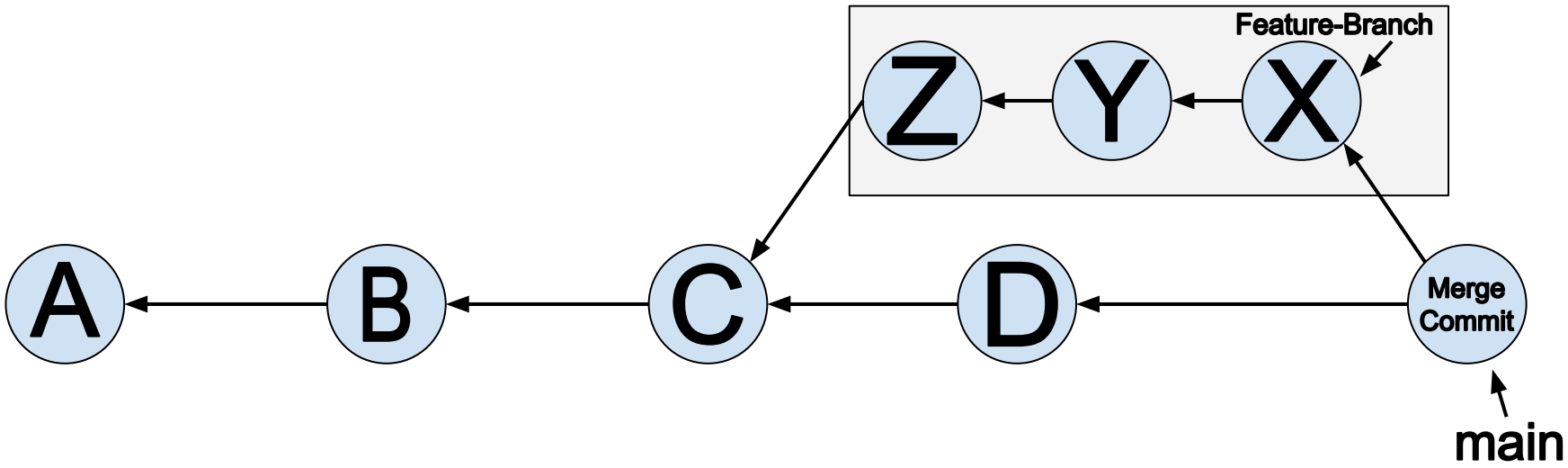
D is a commit that has been added to the **main** branch. **Feature-Branch** stays the same.

Branches



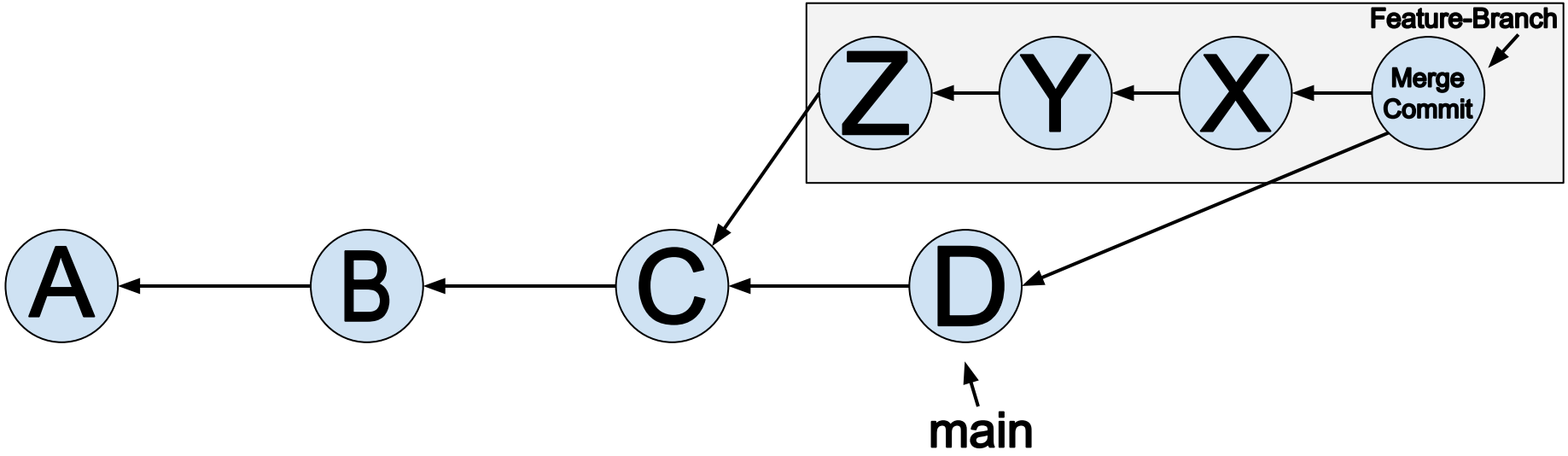
How do we reconcile these two refs so that all of the code is on **main**?

Merge Commit



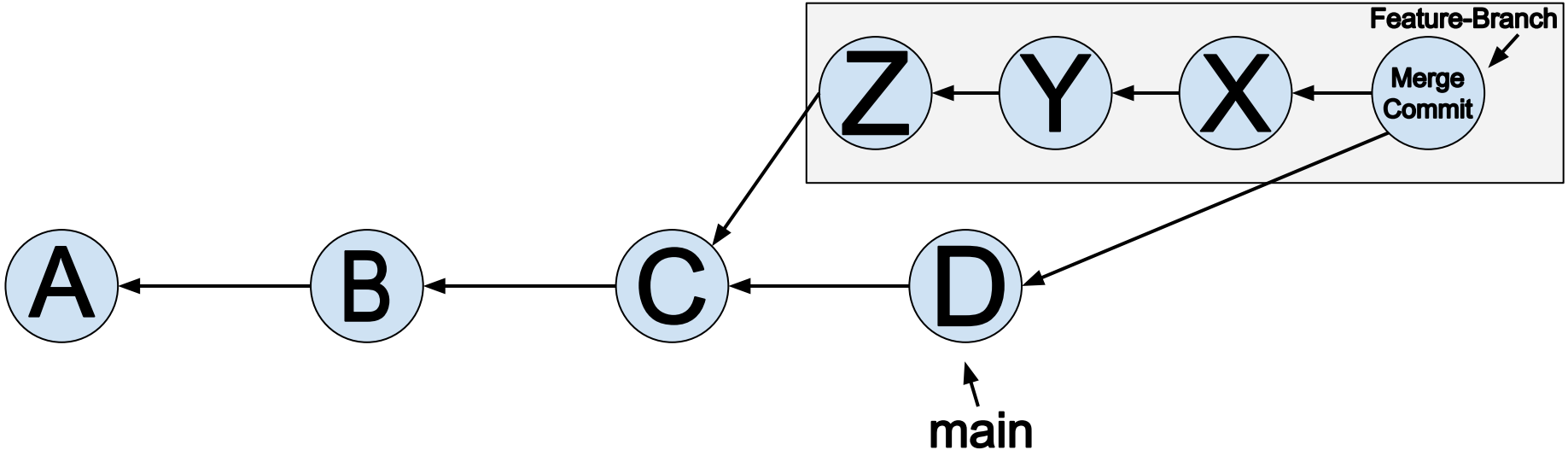
Merge commits are special in that they have two parent pointers. A merge commit can also contain changes, just like any normal commit.

Merge Commit

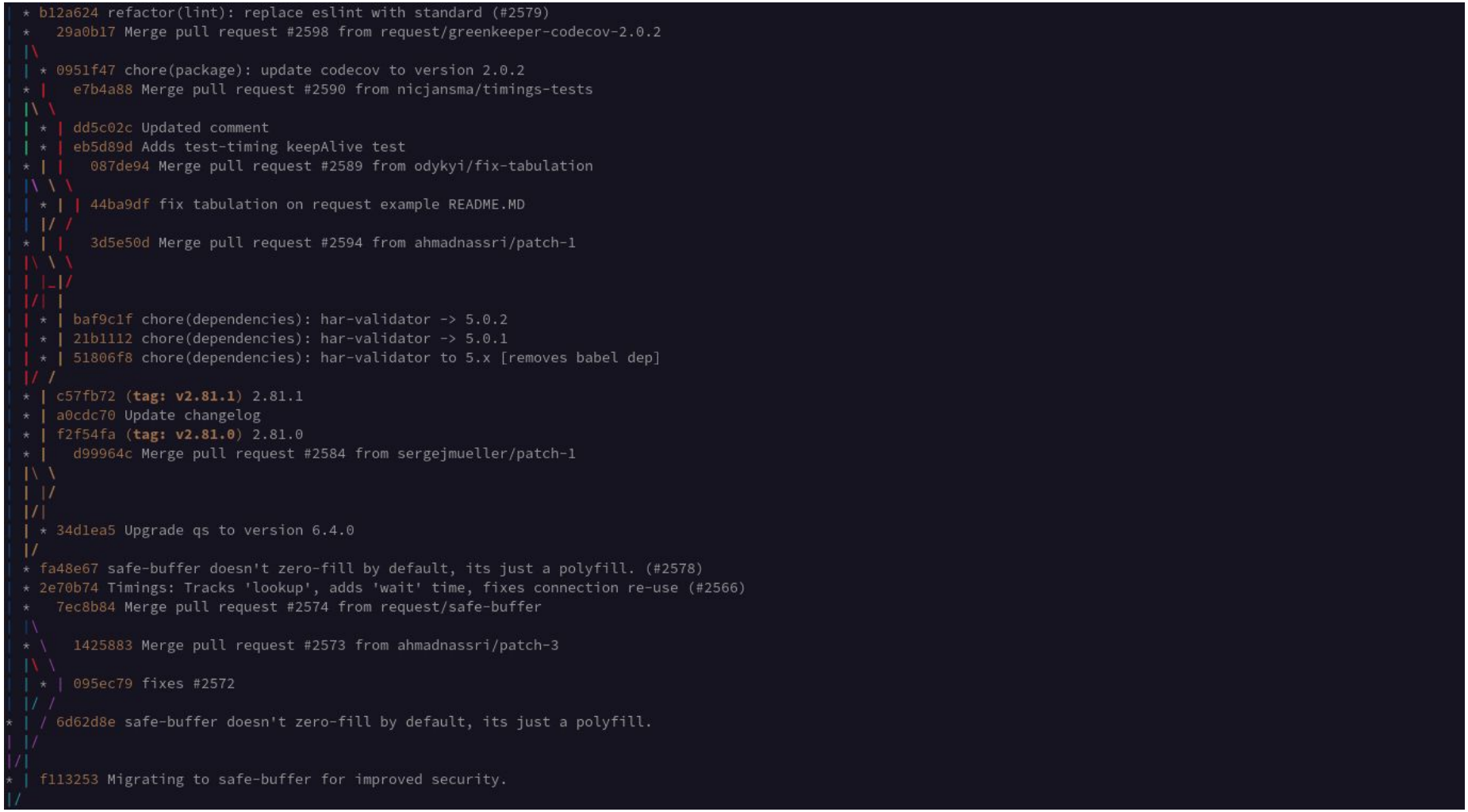


Similarly, if you want to update Feature-Branch to include D, you can merge main into Feature-Branch.

Merge Commit



But then you'll probably need another merge commit later when you merge back to main.

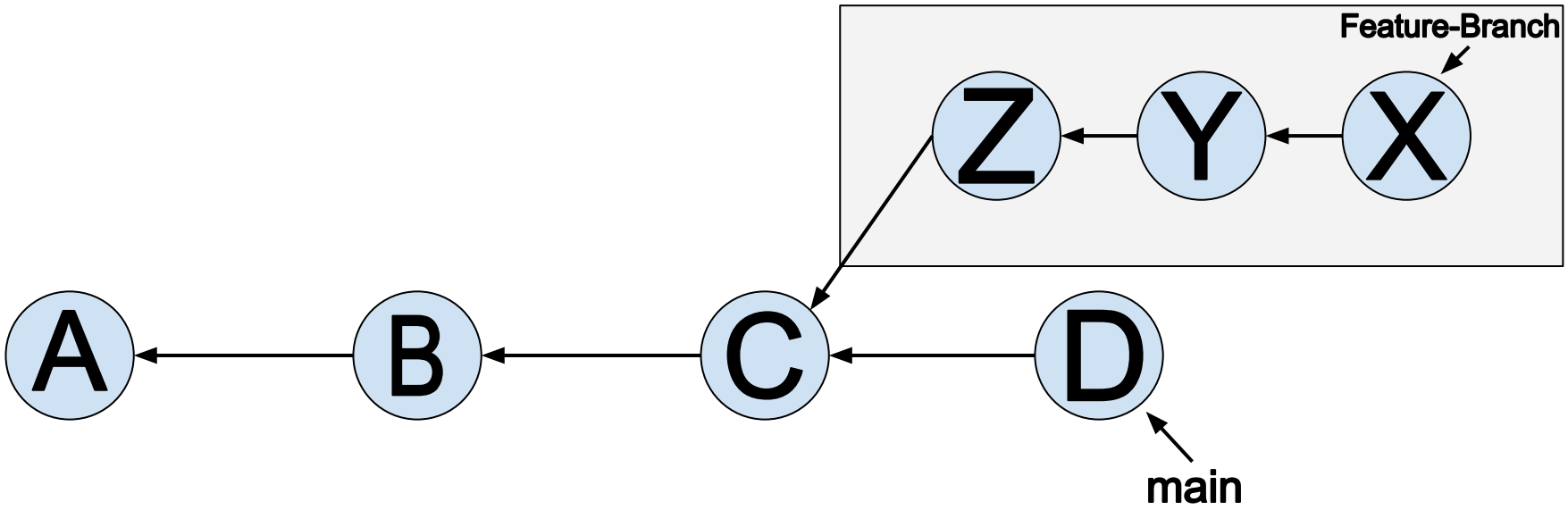

* b12a624 refactor(lint): replace eslint with standard (#2579)
*   29a0b17 Merge pull request #2598 from request/greenkeeper-codecov-2.0.2
|\
| * 0951f47 chore(package): update codecov to version 2.0.2
* | e7b4a88 Merge pull request #2590 from nicjansma/timings-tests
|\ \
| * | dd5c02c Updated comment
| * | eb5d89d Adds test-timing keepAlive test
* | | 087de94 Merge pull request #2589 from odykyi/fix-tabulation
|\ \ \
| * | | 44ba9df fix tabulation on request example README.MD
| |/ /
* | | 3d5e50d Merge pull request #2594 from ahmadnassri/patch-1
|\ \ \
| |_|/
|/| |
| * | baf9c1f chore(dependencies): har-validator -> 5.0.2
| * | 21b1112 chore(dependencies): har-validator -> 5.0.1
| * | 51806f8 chore(dependencies): har-validator to 5.x [removes babel dep]
|/ /
* | c57fb72 (tag: v2.81.1) 2.81.1
* | a0cdc70 Update changelog
* | f2f54fa (tag: v2.81.0) 2.81.0
* | d99964c Merge pull request #2584 from sergejmueller/patch-1
|\ \
| |/
|/|
| * 34d1ea5 Upgrade qs to version 6.4.0
|/
* fa48e67 safe-buffer doesn't zero-fill by default, its just a polyfill. (#2578)
* 2e70b74 Timings: Tracks 'lookup', adds 'wait' time, fixes connection re-use (#2566)
*   7ec8b84 Merge pull request #2574 from request/safe-buffer
|\
* \   1425883 Merge pull request #2573 from ahmadnassri/patch-3
|\ \
| * | 095ec79 fixes #2572
| |/
* | / 6d62d8e safe-buffer doesn't zero-fill by default, its just a polyfill.
| |/
|/|
* | f113253 Migrating to safe-buffer for improved security.
|/

Non-Linear History Trade-offs

- Pro: Tracks full history
- Pro: Maintains “True” history (arguable whether this is a pro?)
- Pro: Remove entire PR with single revert
- Pro: Generally easy as there are fewer merge conflicts

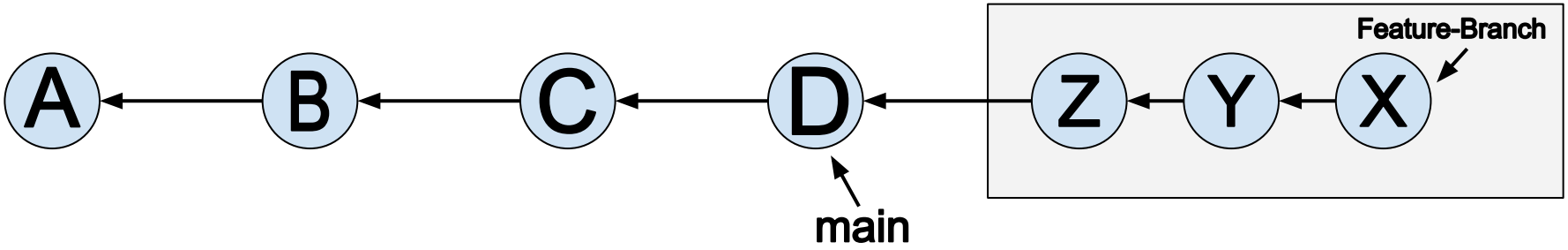
- Con: Graph Histories are Complicated
- Con: Merge commits will clutter your history
- Con: Encourages sloppy practices (arguable?)
- Con: Git Bisect becomes harder (historically)

Another Option?



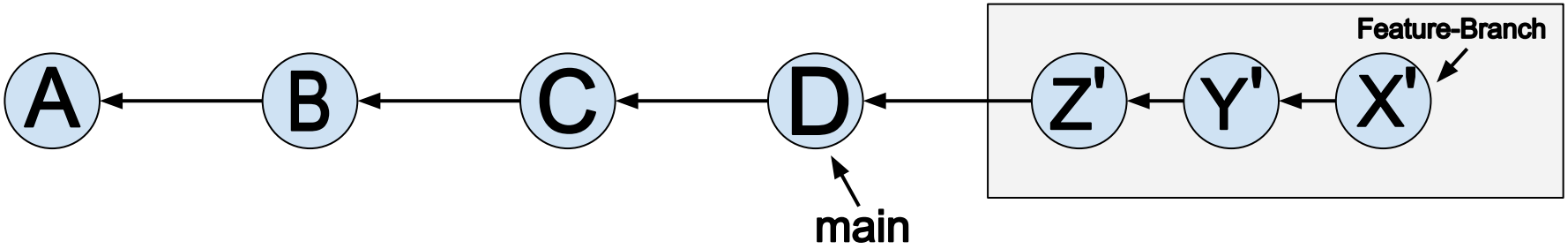
What if we could just make **Z**'s parent pointer point to **D**?

Rebasing (Conceptual Mental Model)



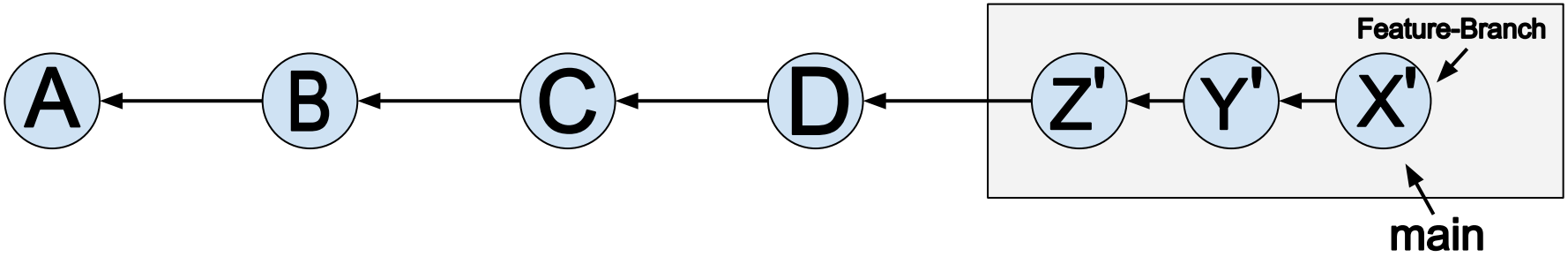
Simplest conceptual model: point the parent pointer of the first commit to the target ref.

Rebasing (What's Really Happening?)



In reality: create new commits by rewriting all commits of a branch onto the target branch (because commits in Git are immutable).

Fast-Forward



When your branch is on top of the branch you're merging into, you can fast-forward, which is just updating the ref and not creating any new commits.

Linear History Trade-offs

- Pro: Conceptually simpler
- Pro: Bisecting is easy
- Pro: Easier to remove a single problematic commit (maybe)

- Con: Requires more discipline to maintain
- Con: More potential for merge conflicts
- Con: Rewrites history (arguable?)

Isn’t Rebase Dangerous?

Don’t rebase shared branches (including main).

Opinion

A clean, linear history should be your goal.

My Workflow

- Branch, Hack & Experiment – add “WIP” commits
- Periodically rebase my branch against main
- Determine the “story” I want to tell for my reviewers
- Break up the large change into small, meaningful changes (the story)
- Make sure CI/Tests pass on each one
- Submit for review
- After review comments, rewrite history so it look like I did it right all along
- Rebase against main
- I never have “fix lint”, or “add comment”, or “change variable name” commits in my branches

Linear History Workflows

- Github: Rebase your branch locally to fix conflicts before merging
- Github: Force Push
- Github: Merge Queues

Clean History

- Every commit should pass CI tests/lints
- Prior to merging, all changes should be made in the appropriate commit
- Graphite: Stacked Diffs

Clean History Workflows

- Example: git amend
- Example: git rebase -i
- Example: rearranging commits
- Example: git add -i