Version Control Fundamentals

Michael Hirsch

CEDAR 2019 Workshop

Git vs. Mercurial

~2019 Market share (both started development in 2005):

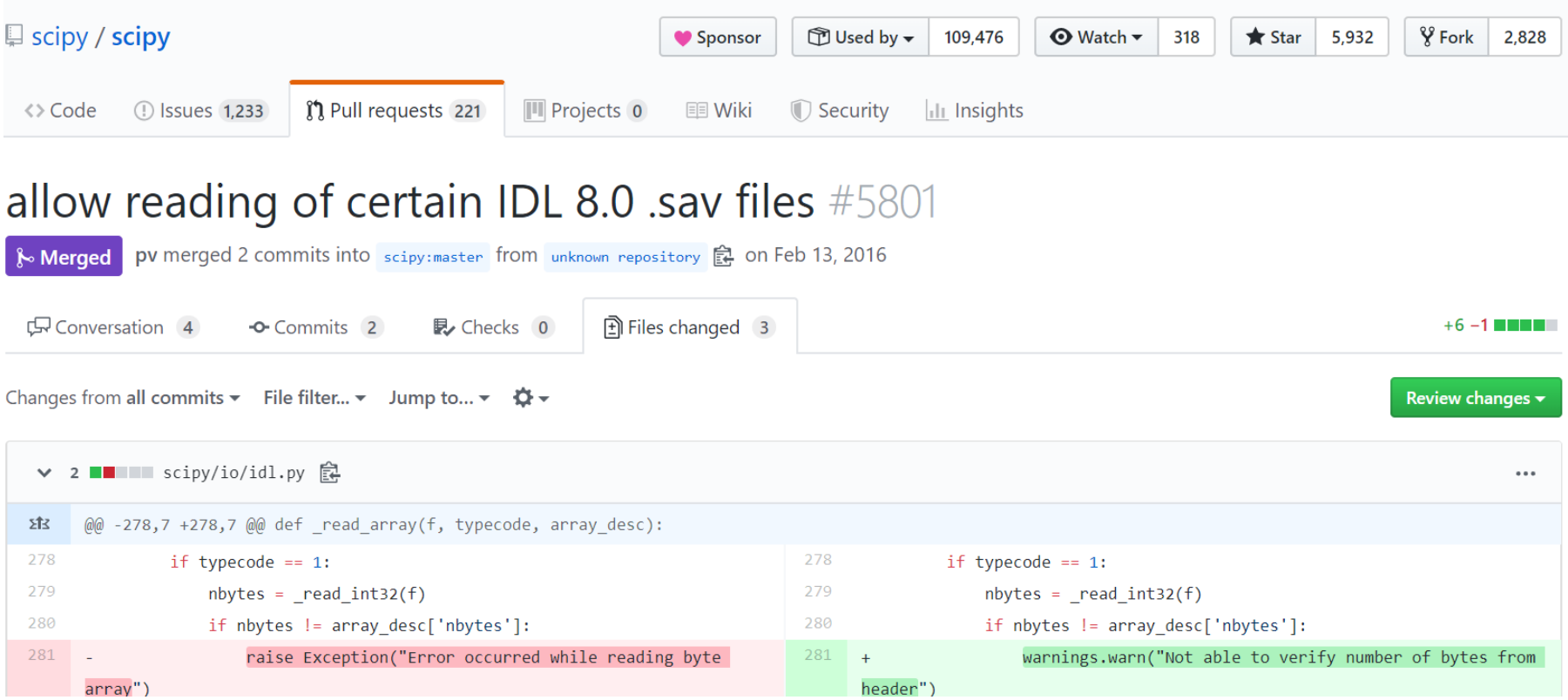
- Git: 80% (GitHub, GitLab, Bitbucket)
- Mercurial: 2% (Bitbucket)

Source: https://www.perforce.com/blog/vcs/git-vs-mercurial-how-are-they-different (Jan. 2019)

Version control (Git) in one slide

- `git clone` Copy a repo (includes history)
  - `git clone https://github.com/scivision/findssh`
- `git checkout` Choose a branch to edit
  - Keeps history uncluttered, allows trying risky things
  - `git checkout -b develop` Creates new “develop” branch to work in, without changing default “master” others see
- `git commit` Current code remembered in history
  - `git commit -am “adjusted scoring bonus”`
- `git push` Send revisions to repo(s) so others can use
- `git pull` Get other’s changes from repo
- `git merge` Joins branches, once new code is debugged
  - `git merge develop`
- `git tag` Convenient bookmark to a particular revision
  - git tag GRL2018
  - Use tagged revision by: `git checkout GRL2018`

Pull Requests



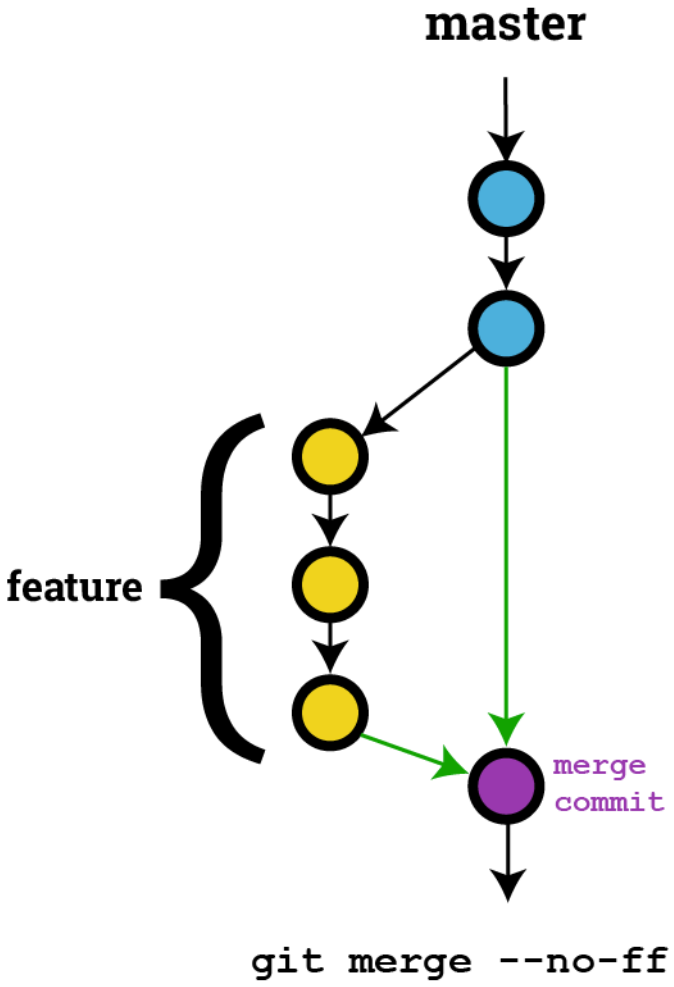
- anyone can make a Pull Request (PR) to your project code repo
- PR may take time to integrate upstream
- Workaround: fork project while changes are folded back into parent project
- Can simply make a new branch from original project “master” to test the PR

Code merging strategies

- No one right answer
- Safer methods are generally messier in project history
  - OK for small projects, but bigger projects try to avoid having 100K's of commits – slows Git down.

`git merge --no-ff` (safest, most verbose)

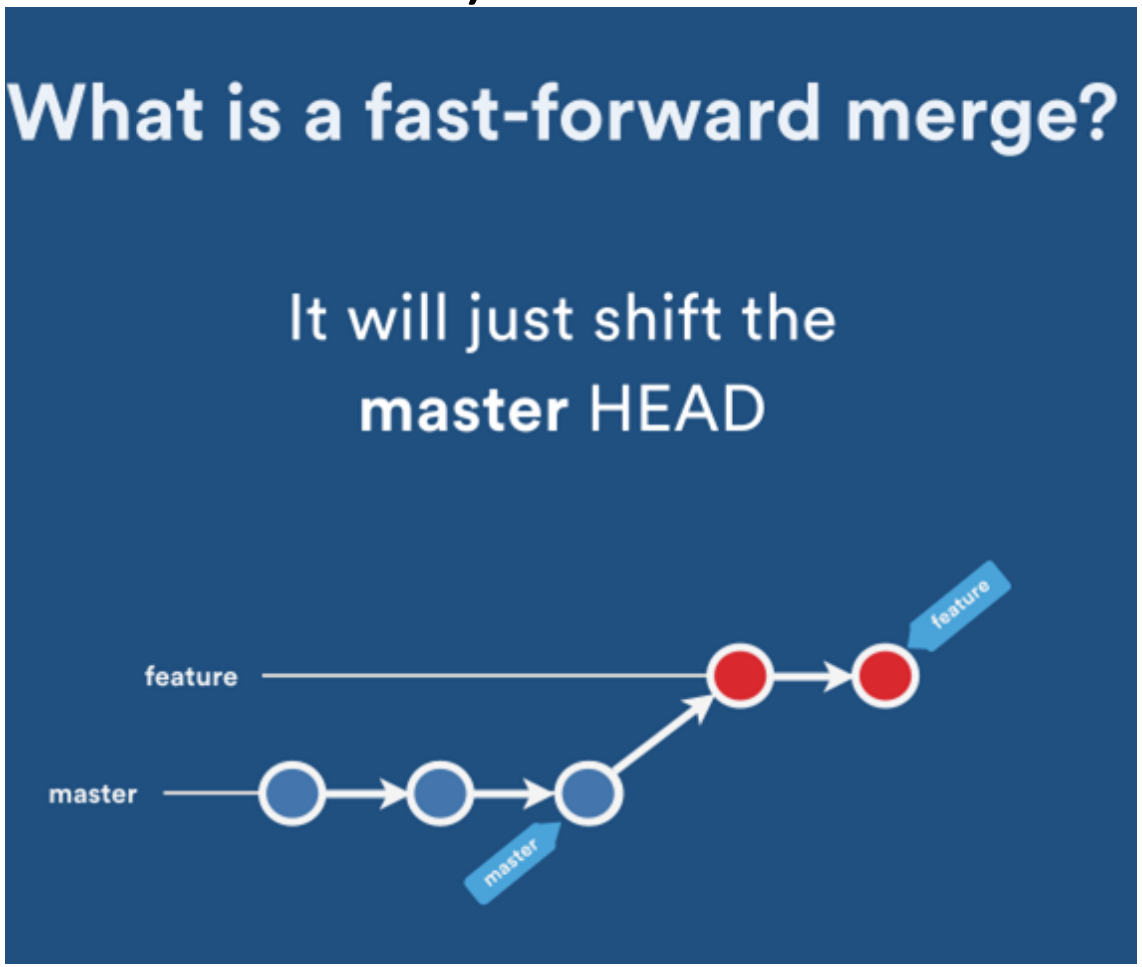
- Assuming you don’t delete feature branches, this method preserves the most history
- Adds an additional commit for the merge that points to the feature branch commits
- It works even if commits were added to master since you branched
    - Assuming the unit tests pass, that is, that the new code segments are still correct when used together



`git merge` (less verbose)

- I usually use “git merge” (which is a “fast-forward merge”)
  - “git merge --no-ff” is fine too
- “Risk” of “git merge”:

if someone committed to master since you branched, you may have to manually merge differences if the same lines of the same file were changed in those conflicting commits. Consider Meld for this case (or another merge strategy)



https://developer.atlassian.com/blog/2014/12/pull-request-merge-strategies-the-great-debate/

Git merge commit: example

Ready to do some new work:

```
git checkout master
git checkout -b newfeat1
```

Feature added, CI test completed:

```
git commit -am “feature1
complete”
git push -u origin newfeat1
```

Test passes, make merge commit

```
git checkout master
git merge [--no-ff] newfeat1
git push
```

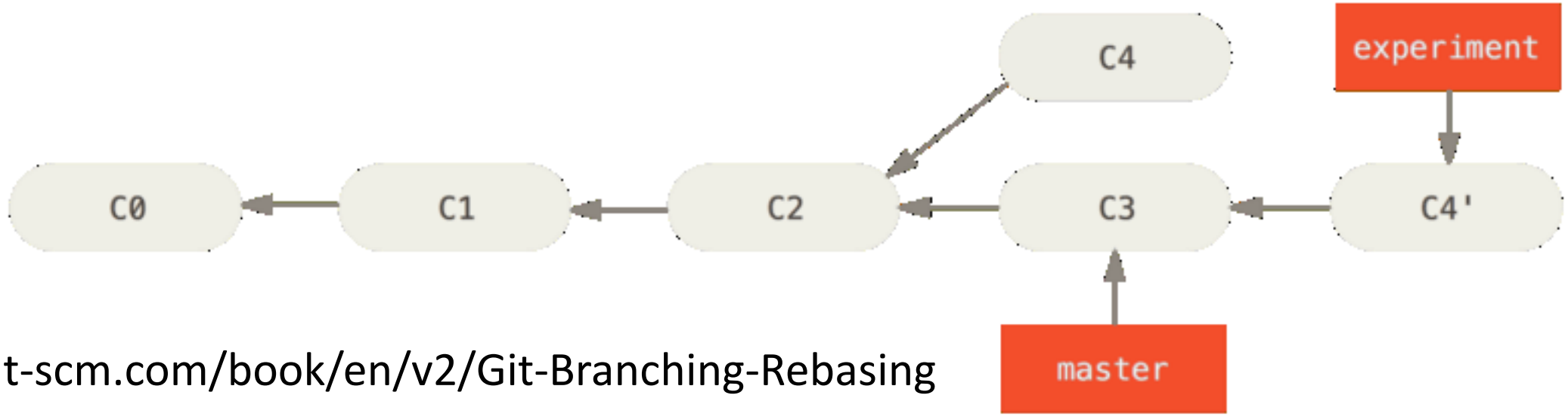
“git merge” conflict

If the same line(s) of the same file are edited on the branch merging into, a merge conflict results, that must be manually resolved

- Reduce occurrences of Git merge conflicts by:
  - Merging frequently (daily or when small unit of work is complete)
  - Break code up into a moderate amount of files vs. huge code files
- Another advantage of code modularity: mitigate merge messes

Advanced, but riskier: rebase merge

- Major projects: typically rebase branch to master before merge
- This is riskier: typos can wipe out a lot of work
- Git force push can irretrievably wipe out an unlimited amount of work
  - no one can recover, except from manual backups, if they were made
- Advanced Git users in experienced teams may use this paradigm
- Advantage: cleanest possible Git history--important for large projects, less so for class / small projects



https://git-scm.com/book/en/v2/Git-Branching-Rebasing

Risky Git commands

These commands have common, appropriate uses by experienced Git users.

But: they can irretrievably wipe out code & code history

“Force push”

```
git push -f
git push --force
```

“Reset”: erases local work, copies specified location

```
git reset origin/master --hard
```

“mirror push”: wipes out remote history

```
git push --mirror
```

Git backup strategies

offsite, automated mirroring: https://docs.gitlab.com/ee/workflow/repository_mirroring.html#pulling-from-a-remote-repository-starter

- DO NOT use Dropbox or similar services for raw .git folder, it will get corrupted!
- As a last resort, zipping up a Git project is “safe” to keep in Dropbox and the like