Git for teams

COSC202—Software Development

David Eyers & Steven Mills
Learning objectives

- Understand what Git branches are
- Describe what a Git merge commit is
- Explain how to use Git branches in team programming
- Contrast methodologies for using Git branches
- Outline common Git pain points
- Compare user-level access control against use of merge/pull requests for managing collaboration
Git workflows

• Git provides a very flexible set of tools
  • ... thus does not impose one way of working
  • For example, Linux code development used email lists to distribute and check commits
    • (Platforms such as GitHub hadn’t been developed yet...)

• Conventions have been established to ensure efficient management of actual project needs using Git

• First, its useful to introduce what Git branches are
Visualising Git repositories

- Filled circles are **commit nodes**
  - A commit saves all tracked files at time $t$
  - Commit messages are in black text
  - Letters on commits just for reference

- Arrows indicate **provenance edges**
  - Can read edges as "is derived from"
  - Merge commits have **multiple parents**
    - 'Y' derives from both 'T' and 'P'

- **Time** progresses up the page
Git branches label commits

• Rounded rectangles **show branches**
  • There are two: ‘main’ and ‘new feature’

• Git branches **point to a commit**
  • e.g., branch ‘main’ points to commit ‘Y’

• Branch pointers **can be updated**
  • In contrast, commits are read-only

• ‘HEAD’ is a special branch label
  • Points to where your next commit goes
  • Not shown; likely points to commit ‘Y’

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>Init the project</td>
</tr>
<tr>
<td>X</td>
<td>Add README</td>
</tr>
<tr>
<td>Q</td>
<td>Implement feature</td>
</tr>
<tr>
<td>K</td>
<td>Add tests</td>
</tr>
<tr>
<td>C</td>
<td>Implement an awesome feature</td>
</tr>
<tr>
<td>P</td>
<td>Fix tests</td>
</tr>
<tr>
<td>T</td>
<td>Hotfix a bug</td>
</tr>
<tr>
<td>Y</td>
<td>Release awesome feature</td>
</tr>
</tbody>
</table>
Committing to a branch

- Git depiction at earlier time
  - Your branch: “new feature”
  - You recently added commit ‘C’
- Commit to a branch involves:
  - Add code beyond commit ‘C’
  - Commit this code (commit ‘P’)
  - Branch label now at commit ‘P’
    - as shown on the following slide...
- Since ‘P’ derives from ‘C’:
  - Both commits considered to be on the ‘new feature’ branch
More on Git branches; Git tags

- Why use Git branches?
  - Facilitates independent working
- Note that there’s a default branch:
  - Recent repositories: it’s called ‘main’
  - Older repositories: it’s called ‘master’
- Also, note the Git tag ‘v1.0’
  - Unlike branches, commits don’t move tags
  - Tags can record metadata into Git history
Methodology: Working on the main branch

• Some projects choose to try not to create branches
  • Over time, long-lived branches can become much harder to recombine into the main branch
  • Encourages making smaller, self-contained changes
  • Keeps focus on main code objective, avoids side-experiments

• But some projects can’t be managed sequentially
  • Fixing complex bugs or features: best use branches

• One notable exception: pair programming
  • Many tools (including VSCode) now allow multiple developers to work on the same code at the same time
Methodology: Working off the main branch

• Branches can be shared with your teams
  • It still isolates its commits from the main branch, though
  • e.g., your team can first review your “new feature”

• Time to integrate the “new feature”?  
  • ‘Merge’ it into the main branch  
  • See commit ‘P’ in the previous figure

• Merging from a branch doesn’t stop further commits happening on that branch in future
  • e.g., fixing a bug in the “new feature”
Methodologies: Feature branching

- The ‘feature branching’ methodology’s principle:
  - any new feature is developed in a separate branch

- Merging to the main branch “adds” that feature

- Feature branches are retained:
  - Later augmentation of feature happens on feature branch
Methodologies: Gitflow

- Example for reference: it’s viewed as overcomplicated
  - A set of shell scripts helps it be used

- Gitflow specifies many branch functions:
  - **main** branch has commits for release versions
  - **develop** branch is where development occurs
  - **feature** branches branch off development branch
  - **release** branch polishes for release
  - **hotfix** branches off main branch thence into develop
Some common Git pain points

- Git understands how to **merge** text files:
  - It performs **line-by-line comparisons**
  - Line changes too close? Git will refuse to auto-merge
- **Binary files**, e.g., JPEG images are treated whole
  - No differencing; no content merging
- Some text files may not have **stable line structure**
  - e.g., XML data can be reordered without changing meaning
  - Git may automatically merge in a way that is destructive
  - Git may just get horribly confused and tell you to merge
Team Git protocols

• You can develop a team protocol for Git use
  • Ensure Git supports you as best it can!

• Examples of protocol points to consider:
  • Agree to commit often—avoids falling out of sync
  • Know what branches are being used and why
  • Consider pair programming / live sharing
  • Try not to touch lots of files without signalling why
  • Agree who’s going to edit files that might not auto-merge
  • Make good use of synchronous team communications
Web-based repository access control

- Common to push your code to a central server
  - e.g., our University of Otago GitLab server, for COSC202

- However, central servers support **access control**
  - Project owner chooses which users can push directly to project

- Beyond ‘Owner’, GitLab levels of access: (decreasing)
  - **Maintainer**—e.g., can’t remove data from project
  - **Developer**—e.g., can’t manage team
  - **Reporter**—e.g., can’t change codebase
  - **Guest**—very few privileges
Open source collaboration

• Say you’re working on a huge open source project
  • You **definitely** want contributions from anyone
  • … but **do not want** to manage user-level access control

• Solution: use GitLab **Merge Requests**
  • Unknown user works on their own copy (fork) of your repository
  • User raises a merge request through GitLab, which you review
  • GitLab then merges in their changes to your repository

• Note that GitHub instead calls these **Pull Requests**!

• Merge / Pull Requests are not built into Git itself