Lecture 7: Code forges & version control

Lecture goals

- Improve client interactions
- Use GitHub's tools for tracking, planning, and review
- Select an appropriate branching strategy for a product/organization
- Maintain a healthy commit history

Client interactions

Required course deliverables

- Feb 11: Project plan
- Mar 4: Report #2
- Mar 7-24: Presentation #1
- Mar 25: Report #3
- Apr 15: Report #4
- Apr 18-9: Presentation #2
- May 10: Final delivery

Intermediate report contents depend on methodology

- Report #2
    - Waterfall: Requirements document
    - Iterative: Draft requirements, first prototype
    - Agile: User stories, first sprint plan and delivery
    - (all): Architecture analysis

Client meetings feedback

- Future meetings may be smaller
    - Rotate team members
- Arrive on time
- Separate moderator and recorder
- Devs should drive meeting (following agenda)
- Reserve sufficient time

- Tips for remote meetings:
  - Cameras on if possible
  - If multiple devs in one location, use one connection (or else headphones)
  - Charge device
  - Eliminate distractions if sharing screen
  - Moderator role is extra important
    - Follow agenda
    - Call on speakers
    - Announce decisions

Recordings

- Not allowed in many contexts
  - Banned in secure facilities
  - May violate privacy law/regulations
  - May violate copyright, trade secret, export law
- Not welcome by many parties
  - Do not assume recordings are okay; plan to not rely on them – note-taking is a professional skill
- **Must** ask for permission before recording

- Recording of lectures is **not** allowed
  - Infringes copyright
  - Risk of FERPA violation
  - Violation of Cornell policy

Project plan feedback (preliminary)

- Imprecise methodologies
- Neglecting client deployment
  - Where will your client run the improved application?
  - What deployment will acceptance testing be performed on?
- Coordination plans without client input
- No slack in schedule, MVP not identified
- Unclear writing inhibits comprehension

    - Be precise

Code forges and version control

Project setup guides (internal projects)

- Teams work on forks of upstream repositories
    - All activity private to class
    - No expectation to contribute upstream
- Teams working on different features share a repository
- Project documentation should include a tutorial to facilitate orientation tasks
    - Get code
    - Build app
    - Test build
    - Run app
    - Generate docs
    - Steps to commit
- Streamlined walkthrough
    - Link to details elsewhere
- If joining a new project without a tutorial, write one!
    - Best written/updated by developers just joining a project

Code forges

- Centralized version control
    - Single source of truth
    - Policy enforcement
- Code browser
    - Permalinks for shared view
- Issue tracker
    - Track tasks
    - Track bugs
- Project planner
    - Kanban board
    - Assign and prioritize issues
- Code review

- Automation
  - Continuous integration
  - Continuous deployment
  - Issue triage
- Documentation browser
- Releases
- Metrics

Demo: GitHub browser and issues

Permalinks

- Click line number, then "...", then "Copy permalink"
  - Embeds current commit hash in link
  - If branch name is used instead, link could point to different code in the future
- Use permalinks whenever adding links to permanent documentation (issues, reviews, postmortem reports, etc.)

Projects/boards

- Projects -> Create a project
  - Template: Basic kanban
    - Avoid automation until experienced with manual process (requires discipline when creating issues, PRs, or else can lead to inaccurate boards)
- Each CS 5150 team should have their own "project" linked to their shared repository
  - Kanban boards may fit some methodologies better than others; "Milestones" are an alternative way of prioritizing and tracking work
- Projects can track "cards," "issues," and "pull requests". CS 5150 recommends tracking issues.

Issues

- Issues -> New issue
  - Click "Projects" to link with project
  - Click "Assignees" to assign to a developer
  - Add checkboxes to create subtasks

    - Use labels to organize into categories (e.g. “bugs” vs. “enhancements”) or dispositions (“duplicate,” “wontfix”)
    - (Later, we will automatically link pull requests with issues, not the other way around)

Version control

- Software engineering: "Programming integrated over time"
- Version control: Code tracked over time

- “Time machine;” helps answer questions:
    - What was the state of the code when this bug was encountered?
    - Which changes created the bug?
    - How many releases/results are affected by the bug?
    - Who can explain this surprising piece of code?
    - What else had to change when this feature was introduced?
    - Which version of the code should I use for my task?

Version control history

- Today: ubiquitous in software engineering
    - Even though metadata is not part of final product
- Prevents confusion over which version of code to use
    - Avoid confusing filenames
    - Avoid losing changes
- Granularity
    - File locking
    - Atomic commits
    - Line-level merging
        - Assumes composability – convenient, but not guaranteed

Version control taxonomy

Centralized

- Metadata stored in central sever
- Single source of truth enforced by architecture
- Requires network connection

- Binary objects handled well
- Can track user behavior
- Examples: SVN, Perforce

Distributed

- Metadata duplicated in every clone
- Source of truth determined by policy
    - Often centralized in practice
    - Hierarchies allowed
- Fully featured when offline
- Binary objects often expensive
- Examples: Git, Mercurial

Monorepo vs. many repos

- A single, monolithic repository for all of an organization's code brings certain benefits (read Google paper):
    - Large-scale refactoring
    - Simplified dependency management
- Often impractical with current distributed VCSs
    - Techniques: shallow clones, sparse checkouts, centralized large file storage, filesystem tricks
- Other techniques can be used for dependency management

Commit basics

- Atomic change affecting any number of files
- Metadata: author, timestamp, message
    - Git distinguishes “author” from “committer”
- Parent commit: represents state of repository prior to changes
    - Linear history: every commit has a single parent and child
    - Branch: multiple commits with the same parent
    - Merge: commit with two parents
        - Conflict: different changes in the lineages of both parents affect the same lines of code

Branch management

- Trunk/master/main

    - Canonical latest version of "ready" code
    - Should be kept in buildable state
- Development branch
    - Long-lived branch for iterating on work in progress
    - Merged with trunk when "finished"
    - Problem: merges are painful; postponing them makes them more painful
    - Shift-left: "if it's painful, do it more often"
- Trunk-based workflow
    - Keep changes small (may queue in issue branch)
    - Merge immediately to trunk
    - Requires continuous testing
- Release branch
    - Tracks version of software released "in the wild" (think hardware products)
    - Provide stability
    - Cherry-pick bugfixes

Demo: pull request

- Create issue branch
    - For open-source repo, create fork if not a maintainer
- Use commits to save work
- Clean up commit history using `rebase`
- Push to PR
- Respond to review by pushing additional commits or force-pushing revised history
- PR merge strategies (rebase, merge, squash)