0. git practice

(A carryover from last lecture)

Section overview

We covered git fundamentals last time, but didn't get to practical matters.
Sometimes a flexible tool is easier to use with guidelines.

- branches
- commits
- merges

Branches

- develop a feature on its own “feature branch”
- push feature branch commits to the same branch on origin (backup your commits)
- don’t use any long-lived branches besides master (at least not until real users are using the production app)
- use review apps for open pull requests, when possible

Commits

- on the feature branch, create small, fine-grained commits, about one thing (ideally)
- (this makes code reviews easier)
- use good commit messages
  - target audience: a developer who inherits your code and has to maintain it
  - first line: imperative tense (i.e. a command), <= 50 characters
  - empty line
  - paragraphs or lists explaining rationale: why were these changes necessary, and why did you do it this way instead of some other way?

Merges

When merging a feature branch into `master`:

- rebase the branch atop master
- *squash* your commits into a single commit with a great message
- fast-forward (i.e. linear) merge to master, push, and delete the feature branch locally and remotely

Merge recipe

- git fetch origin
- git log –oneline –graph –decorate HEAD origin/master
- git rebase master –onto origin/master
- (ensure tests still pass)
- git rebase -i origin/master (and set all but the first commit to "squash")
- git push –force-with-lease (to update the pull request)
- git checkout master
- git merge –ff-only <feature-branch>
- git push
- git branch -d <feature-branch>
- git push origin –delete <feature-branch>

1. Agile software development

Why is agility important?

Think, Pair, Share

TLA Corp. engages Acme Software to build an app. Acme reads the project description and quotes $250k over 3 months. They go build it, and deliver it just in time. TLA reviews it and says it's not what they asked for. TLA has people waiting on the software, and Acme has another project lined up that they need to start on. How can they resolve this disagreement?

What went wrong?

- Reliance on a written document over in-person, collaborative explorations
- Predicting relatively far out into the future, and committing to that
- No opportunity for iterative feedback from the client!
- A locked-in timeframe and budget doesn't mix with changing requirements

Why do it this way?

This approach might seem brain-dead, but there are good reasons for it.

- How can Acme commit to a budget unless the requirements are nailed down?
- How can TLA decide whether the app is worth building without a budget?
- How can requirements be nailed down without something being written down?

The problem

- I've never met a client who can imagine in full detail what they want.
- Even if they could, how can they (efficiently) get the details of that vision in my head?
- Even if they did, how can I have confidence about my estimates?
- Predicting the future with any accuracy is hard!
- (Arguably harder in our field, given how fast things change.)
- With everything nailed down, *change is the enemy*.
- In practice, this approach necessarily leads to inflated estimates.

Introducing agility

- in general, *agility* is useful in a changing environment
- key idea of agile software development: expect change, and handle it gracefully
- deemphasizes planning and predicting the future
- instead emphasizes *iteration*: taking things as they come (not before), and responding to new information smoothly

1. Agile software development

How can we achieve agility?

Extreme Programming (XP)

Extreme Programming (XP) was devised as a reaction to the planning-heavy approach that used to be prevalent.
Core values:

- communication
- simplicity
- feedback
- courage

XP Values #1: Communication

- focus on people, not documentation
- developer's goal: try to see the world through client's eyes and understand how they think
- client brings the *what*: idea, vision, goals
- developers bring the *how*
- collaboration is essential
- like using your phone to navigate through an unknown city
- as developers, your job is to be the executive of the client's will: make it so

XP Values #2: Simplicity

- simplicity of process
  - don't solve tomorrow's problems today
  - only plan (loosely) a week or two into the future
- simplicity of code
  - don't over-engineer
  - only built what you know that you need now (YAGNI)

XP Values #3: Feedback

- give client opportunities for feedback
- on a ~daily cadence, client can answer questions from the team to clarify their desires
- on a ~weekly cadence, client can see the current app-in-progress, provide feedback on the current state, and set near-term priorities
- example: a feature is 80% done. Client can decide whether the remaining 20% is higher priority than the next feature (or whether only 10% is).
- iteration saves time: less time spent planning and less time on wasted effort

XP Values #4: Courage

- a.k.a. fearlessness
- difficult not to worry about the overall project
- difficult to trust your collaborators

XP Practices

- client meetings to get client feedback and steer/establish new priorities
- pair programming
- test-driven development and well tested code
- refactoring
- small releases
- continuous integration

Setting client expectations

This sounds nice from a dev's perspective. But how can clients decide whether to pursue a project, if they don't know how much it will cost?

- Planning & exploration phase of work, to tackle biggest uncertainties
- Then rough estimates from an experienced dev, with padding based on complexity, uncertainty, and potential risks of getting it wrong, to establish a budget
- Client has a go/no-go decision
- Then maintain focus on highest-priority tasks, and communicate the state of the budget to the client frequently

Effective client meetings

What should happen at client meetings?

- show them what you've done recently (e.g. walk through tasks on Trello board and their corresponding features in the app)
- invite feedback: anything look off? any new requests?
- establish priorities: what's most important to client at this point?
- in general, the client is the expert for relative values of features, devs are the experts for the costs of features (in terms of time), and decisions about priority need both inputs

2. Programming activities

Think, pair, share

Think, pair, share

What proportion of your time programming is spent:

- actually writing code
- debugging code you wrote
- learning some new technology
- being distracted
- other

Programming activities

How essential are each of these activities?

- actually writing code
- debugging code you wrote
- learning some new technology
- being distracted

Minimizing distractions

- block off time to program
- close messaging and email apps
- put your phone away
- break bigger tasks up into tiny ones
- be aware of which tasks feel hard or overwhelming

3. Pair programming

What and why

What is pair programming?

Pair programming is two people working as a single unit to write code.

- One monitor, one keyboard, one mouse
- Two chairs, two people
- A “driver” and a “navigator”

Efficiency

But isn't that wasteful? Actually, no.

- you maintain higher focus, so get done sooner
- the code quality is higher, so spend less time chasing bugs
- studies tend to show that PP is at most 25% slower, but often on par with or even faster than solo programming
- each dev knows more about the total codebase, and fewer areas of the codebase have a "truck number" of 1
- help debugging, and diversity of experience to bring to a task

Difficulties

- verbalizing your thought processes
- exhausting your verbal circuits
- coordinating working times
- sounds intimidating! emotional resistance (even when you know better)

Other upsides

- the data shows that most people end up really enjoying this
- you learn and grow much faster as a developer (and a teacher)
- you get to learn ancillary things that are harder to learn otherwise, e.g. about the CLI, or tools, or git, or editor features, etc.