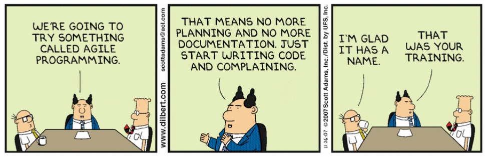
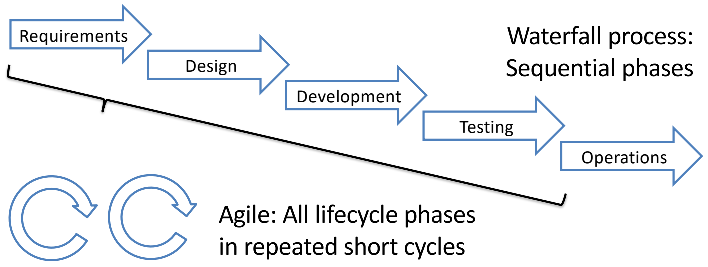
Recall: Plan & Document ⇒ Agile


WE'RE GOING TO TRY SOMETHING CALLED AGILE PROGRAMMING.
THAT MEANS NO MORE PLANNING AND NO MORE DOCUMENTATION. JUST START WRITING CODE AND COMPLAINING.
I'M GLAD IT HAS A NAME.
THAT WAS YOUR TRAINING.
www.dilbert.com

Dilbert 11/26/17



Waterfall process:
Sequential phases

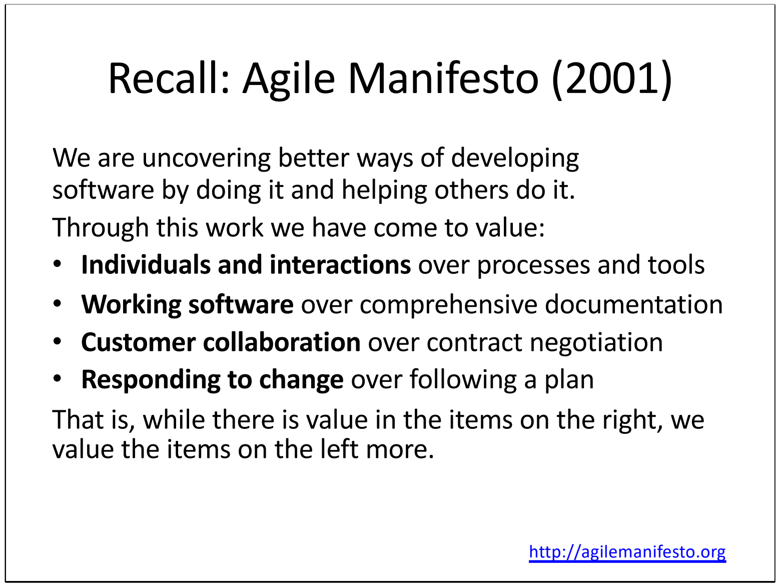
Agile: All lifecycle phases
in repeated short cycles

Waterfall: Sequential phases of project (like a cascading waterfall).

In contrast Agile, implements multiple iterations of those lifecycles in short repeated cycles. Embraces change as a fact of life: continuous improvement instead of a single planning planning phase. Team continuously improves working but incomplete prototype until customer satisfied (with customer feedback on at each 1-2 week iteration).

Note that when we talk about agile, we are talking as a project management “philosophy” (like P&D is a description of more than just Waterfall). Scrum, Extreme Programming (XP) are specific methodologies guided by the Agile philosophy.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



Recall: Agile Manifesto (2001)

We are uncovering better ways of developing software by doing it and helping others do it.

Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

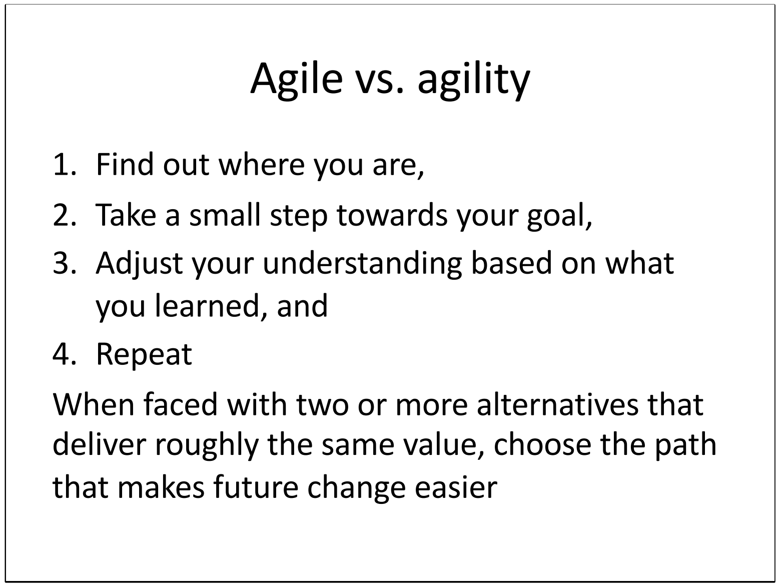
That is, while there is value in the items on the right, we value the items on the left more.

http://agilemanifesto.org

We saw the first last time with BDD (Behavior Driven Design). What are some other examples of these values in practice?

- PropTypes are working software instead of documentation
- The Given-When-Then tests document scenarios and serve as tests

http://agilemanifesto.org

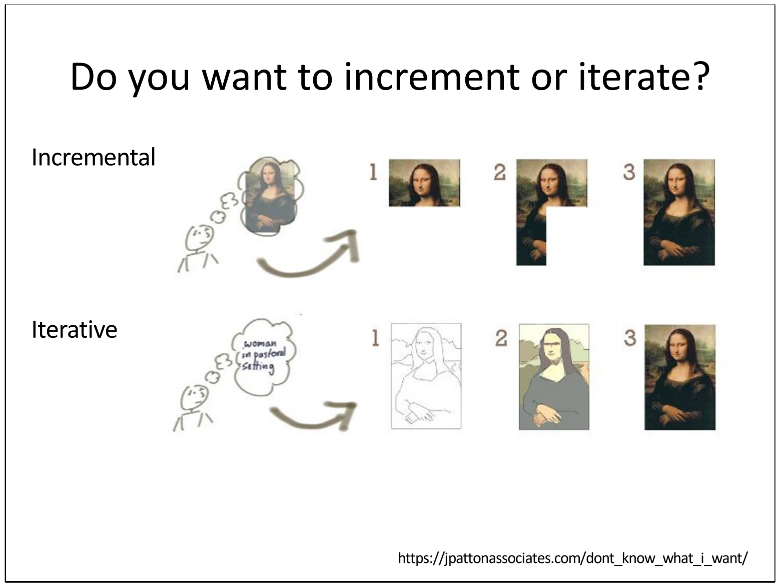


Agile vs. agility

1. Find out where you are,
2. Take a small step towards your goal,
3. Adjust your understanding based on what you learned, and
4. Repeat

When faced with two or more alternatives that deliver roughly the same value, choose the path that makes future change easier

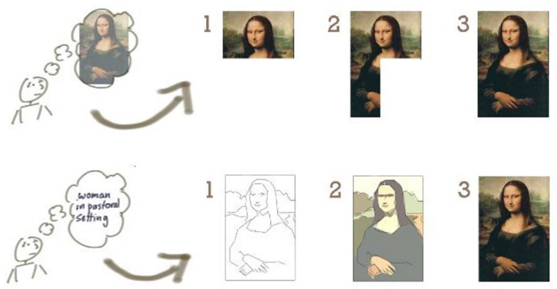
Adapted from Dave Thomas (https://www.youtube.com/watch?v=a-BOSpxYJ9M)



Do you want to increment or iterate?

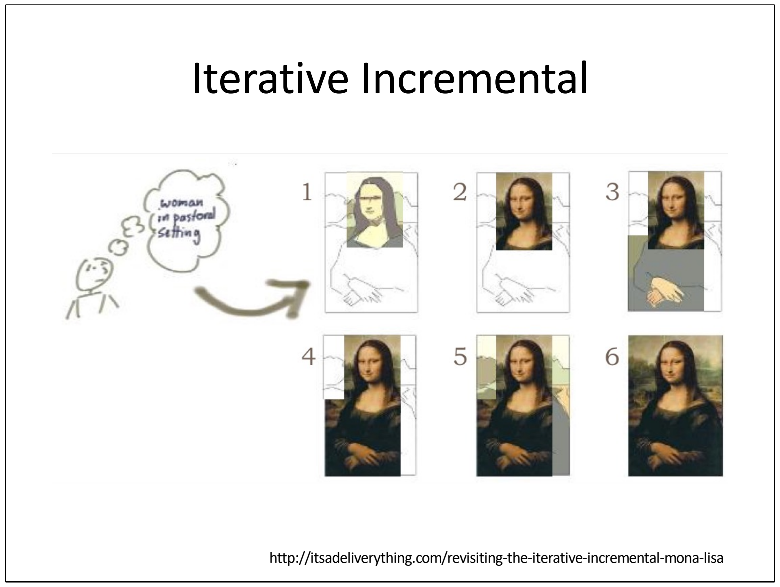
Incremental

Iterative

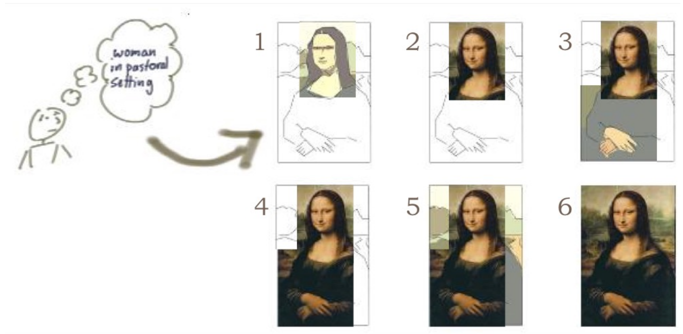


https://jpattonassociates.com/dont_know_what_i_want/

- Incrementing calls for a fully formed idea that is built a bit at a time, and thus requires having a fully formed idea.
- Iterating allows you to move from vague idea to realization, i.e., “iterating” builds a rough version, validates it, then slowly builds up quality/capabilities. The catch is we are addressing the entire problem at one time, i.e., we are working on the entire image. This is both challenging and has the risk that we end up with everything partially completed.



Iterative Incremental



http://itsadeliverything.com/revisiting-the-iterative-incremental-mona-lisa

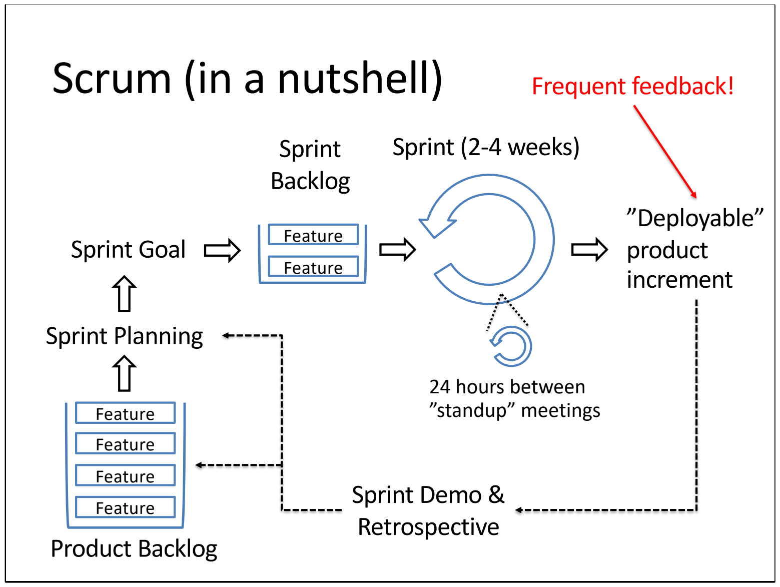
Here we work both ways, that is in increments, but some of those increments are iterative improvements to existing functionality.

“The sprint adds completely new features, based on user stories, hence expanding the scope of the functionality offered – that makes it Incremental. But each Increment is also likely to refine existing functionality – that makes it iterative.”

What do we get if we stop at each step … ?

1. Realized it isn’t a good idea. Stop there.
2. Have our top functionality!
3. Started to complete 2nd tier priorities…

What is the catch? There is big jump before step 1 (where we are setting up the outlines of the image). It takes experience to make good decisions at this phase that set you up for success later. That is to have that general sense of what you will need in the future and make decisions today in anticipation of those needs. For example, I don’t have a server yet, but I know how the data will likely be structured and so can design from my front-end accordingly. One goal of the project is to help use develop that experience.


Scrum (in a nutshell)

"Each Sprint has a definition of what is to be built, a design and flexible plan that will guide building it, the work, and the resultant product."

Sprint planning: A time-boxed planning meeting to determine
* Which features will be delivered in the upcoming sprint, and
* Decide how this work will get done (i.e., design the system, define specific work items, breaking up any larger tasks).

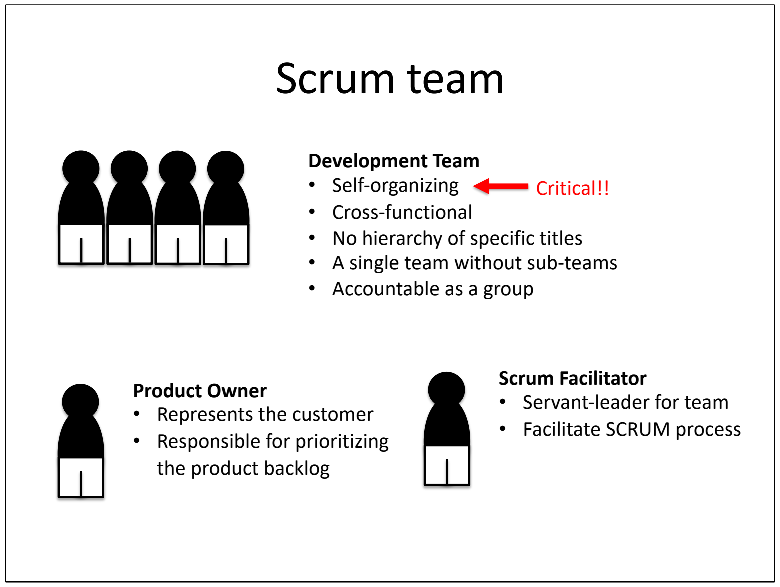
Daily Scrum: A daily standing meeting of no more than 15 minutes. Each person briefly describes
* What they did since yesterday to help the team meet the Sprint Goal
* What they plan to do today
* Any impediments that will prevent the team from meeting the Sprint Goal
The goal is efficient communication, quick decision making and quick resolution of any impediments.

Sprint Demo: A meeting at the end of the iteration to demo the new release (and there should be a new release). Team only demos "done" features.

Sprint Retrospective: A meeting to reflect about Sprint itself
* Identify things that worked and things that didn't
* Make a plan for how to make the next sprint work better

* Pay particular attention to the sprint velocity (rate at which work is accomplished). As we will discuss the goal is constant velocity.

Adapted from Mountain Goat Software
https://www.mountaingoatsoftware.com/uploads/presentations/Getting-Agile-With-Scrum-Norwegian-Developers-Conference-2014.pdf



Scrum team

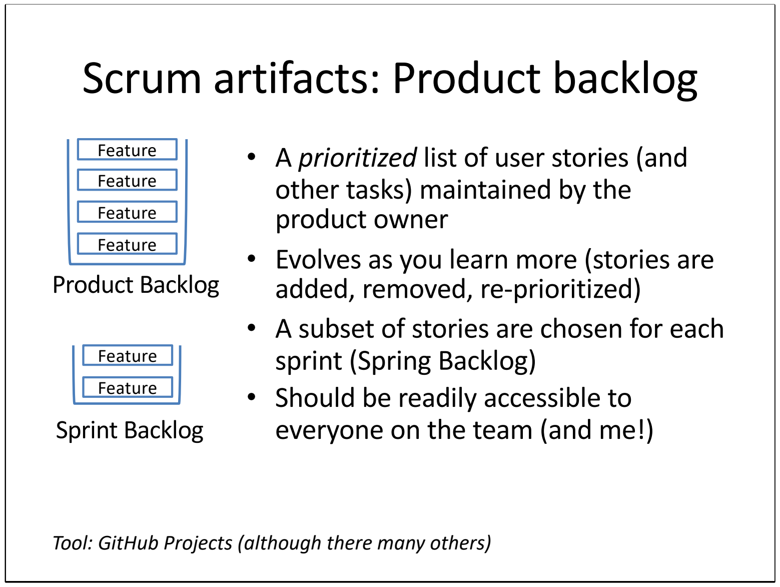
Development Team
- Self-organizing ← Critical!!
- Cross-functional
- No hierarchy of specific titles
- A single team without sub-teams
- Accountable as a group

Product Owner
- Represents the customer
- Responsible for prioritizing the product backlog

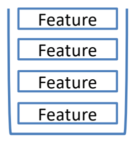
Scrum Facilitator
- Servant-leader for team
- Facilitate SCRUM process

If you have an external customer, I suggest selecting one person to the product owner and the point person for interacting with the customer. If you have an ”internal” customer, the proposer will likely play this role.

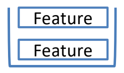
The scrum facilitator is more traditionally called the “Scrum Master”. The original intent was that person had mastered the Scrum process, not that they were in charge somehow. Since that is a loaded term and not actually a good descriptor of that role, we will call that person the “Scrum Facilitator”.



Scrum artifacts: Product backlog



Product Backlog

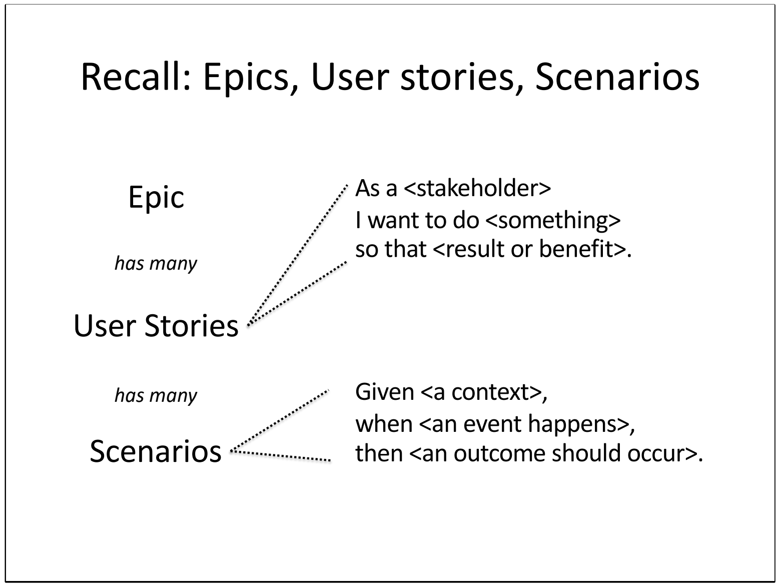


Sprint Backlog

- A *prioritized* list of user stories (and other tasks) maintained by the product owner
- Evolves as you learn more (stories are added, removed, re-prioritized)
- A subset of stories are chosen for each sprint (Spring Backlog)
- Should be readily accessible to everyone on the team (and me!)

Tool: GitHub Projects (although there many others)

Setting the Sprint Goal is a function of the Product Backlog (that is the set of features to build), the current state the application and the capacity and past performance of the Development Team. That is, you want to set a realistic goal based on past and predicted development velocity.



Recall: Epics, User stories, Scenarios

Epic

has many

User Stories

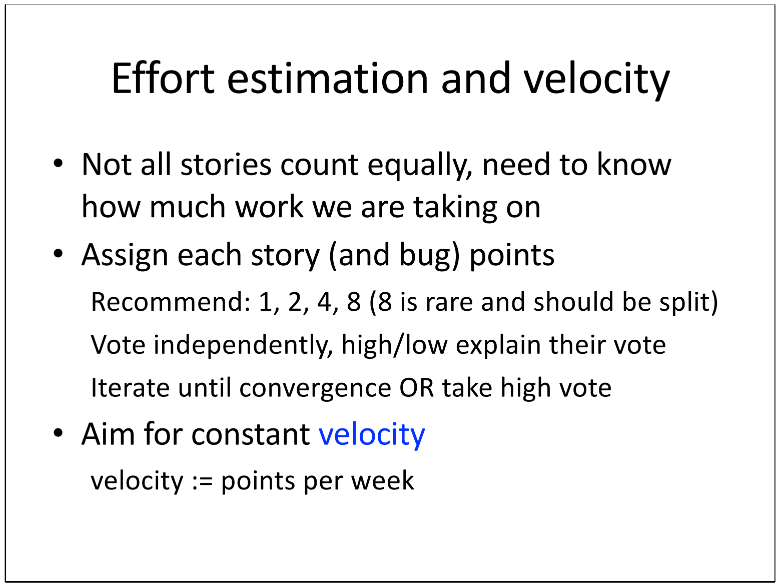
As a <stakeholder>
I want to do <something>
so that <result or benefit>.

has many

Scenarios

Given <a context>,
when <an event happens>,
then <an outcome should occur>.

Not all work items may be user stories. Some work-items will be bugs, Sometimes a task is necessary but far removed from the user, e.g., read an arbitrary byte range from a local or remote file.


Effort estimation and velocity

- Not all stories count equally, need to know how much work we are taking on
- Assign each story (and bug) points
  - Recommend: 1, 2, 4, 8 (8 is rare and should be split)
  - Vote independently, high/low explain their vote
  - Iterate until convergence OR take high vote
- Aim for constant velocity
  - velocity := points per week

You will often see Fibonacci schemes… Why? What is the difference between 5-6 (within error bounds)? "Studies have shown that we are best at estimating things that fall within one order of magnitude (Miranda 2001; Saaty 1996)"

What if I don't know how to approach a given user story? Should I just give it 4 points? User stories should not be so complex that you don't know how to approach implementing it. Recall an INVESTable story is Estimable. If your story is not estimable, you may need a spike, or to refactor/decompose it into a more approach form.

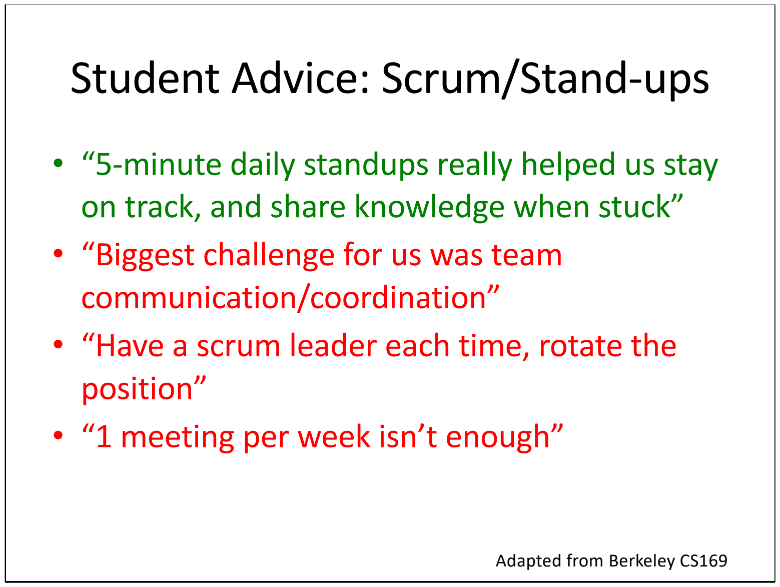
Why constant velocity? That means are working in a predictable and sustainable way!

For last 3 iterations, Team Blue's (#003F84) average velocity is 8, Team White's is 4. Which, if any, of the following comparisons between the Blue and White teams is valid?

A. Blue has more developers than White
B. Blue is twice as productive as White
C. Blue has completed more stories than White
D. None of the above

Answer: D

Since each team assigns points to user stories, you cannot use velocity to compare different teams. However, you could look over time for a given team to see if there were some iterations that were significantly less or more productive.

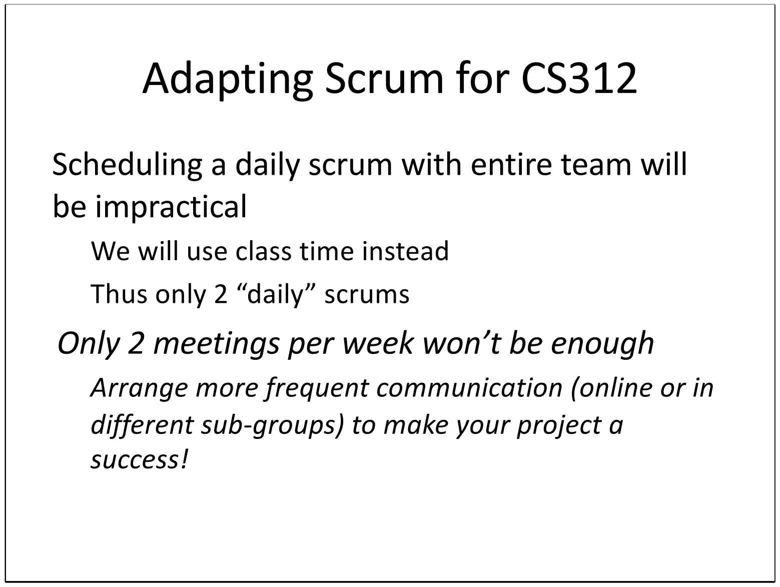


Student Advice: Scrum/Stand-ups

- "5-minute daily standups really helped us stay on track, and share knowledge when stuck"
- "Biggest challenge for us was team communication/coordination"
- "Have a scrum leader each time, rotate the position"
- "1 meeting per week isn't enough"

Adapted from Berkeley CS169

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



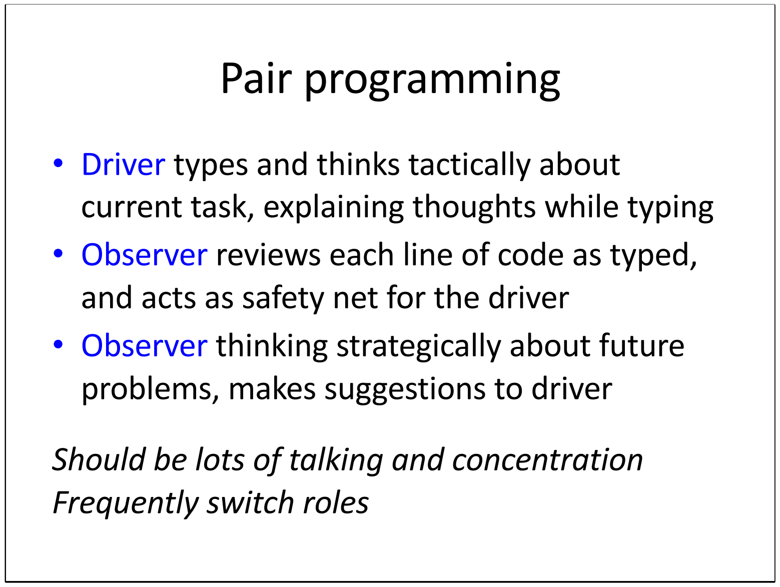
Adapting Scrum for CS312

Scheduling a daily scrum with entire team will be impractical

- We will use class time instead
- Thus only 2 “daily” scrums

Only 2 meetings per week won’t be enough

- *Arrange more frequent communication (online or in different sub-groups) to make your project a success!*


Pair programming

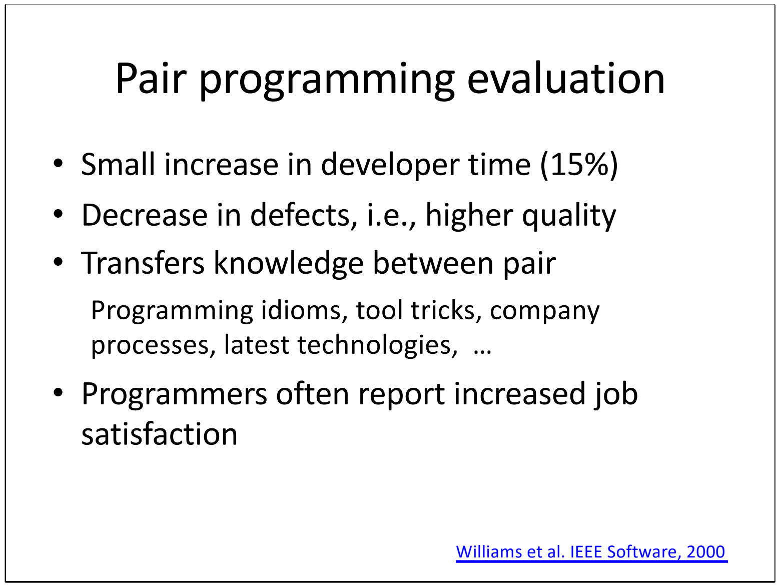
- Driver types and thinks tactically about current task, explaining thoughts while typing
- Observer reviews each line of code as typed, and acts as safety net for the driver
- Observer thinking strategically about future problems, makes suggestions to driver

Should be lots of talking and concentration
Frequently switch roles

Driver is thinking short term, Observer long term…

Why would a company do this?
What do you think the overhead of PP is (total developer time for PP vs. solo)? 100%? Or…

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



Pair programming evaluation

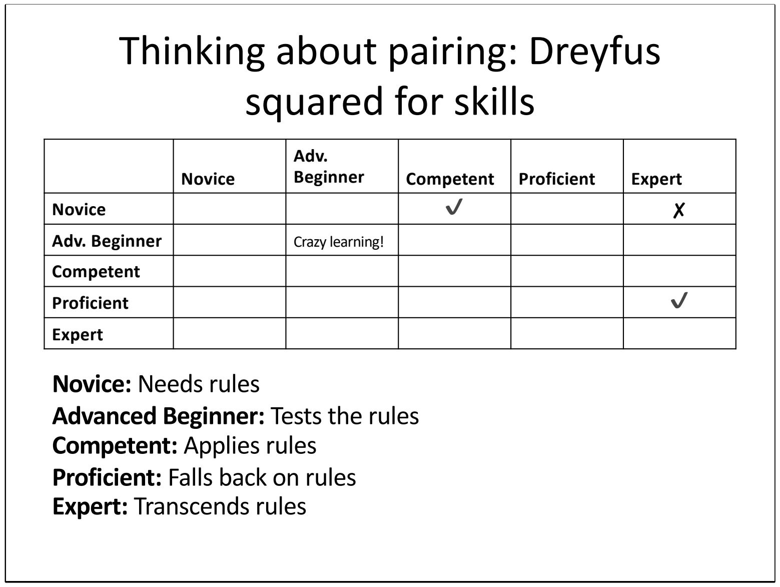
- Small increase in developer time (15%)
- Decrease in defects, i.e., higher quality
- Transfers knowledge between pair

  Programming idioms, tool tricks, company processes, latest technologies, …
- Programmers often report increased job satisfaction

Williams et al. IEEE Software, 2000

https://collaboration.csc.ncsu.edu/laurie/Papers/ieeeSoftware.PDF

I encourage "promiscuous pairing", regularly swapping partners so that eventually everyone is paired

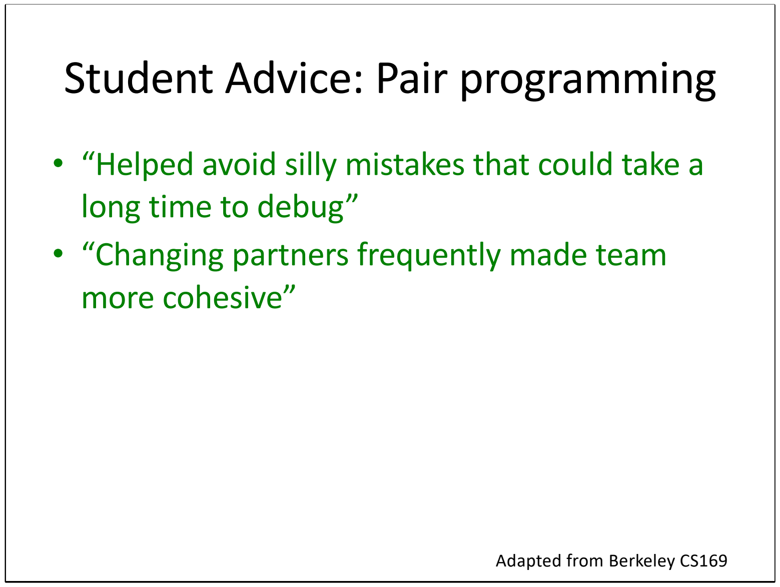


Thinking about pairing: Dreyfus squared for skills

	Novice	Adv. Beginner	Competent	Proficient	Expert
Novice			✓		✗
Adv. Beginner		Crazy learning!			
Competent					
Proficient					✓
Expert					

Novice: Needs rules
Advanced Beginner: Tests the rules
Competent: Applies rules
Proficient: Falls back on rules
Expert: Transcends rules

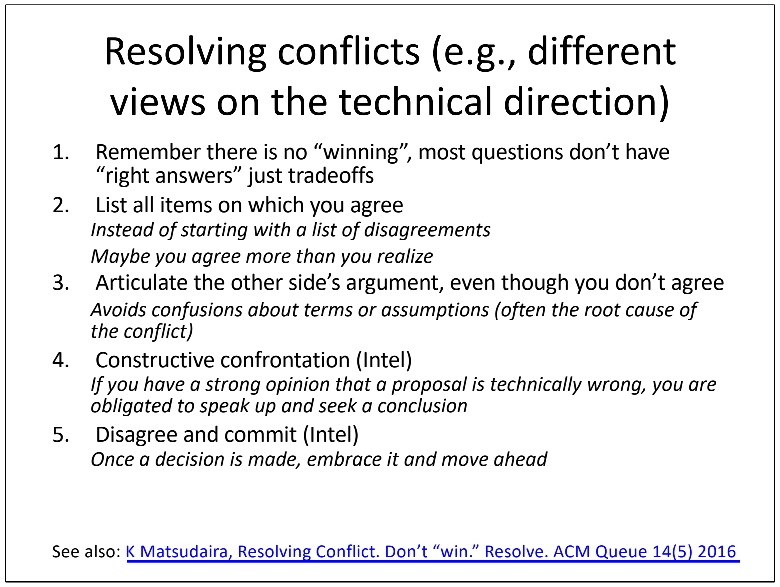
Adapted from Dan North
Dreyfus model of learning: https://www.youtube.com/watch?v=lvs7VEsQzKY&t=312s



Student Advice: Pair programming

- "Helped avoid silly mistakes that could take a long time to debug"
- "Changing partners frequently made team more cohesive"

Adapted from Berkeley CS169

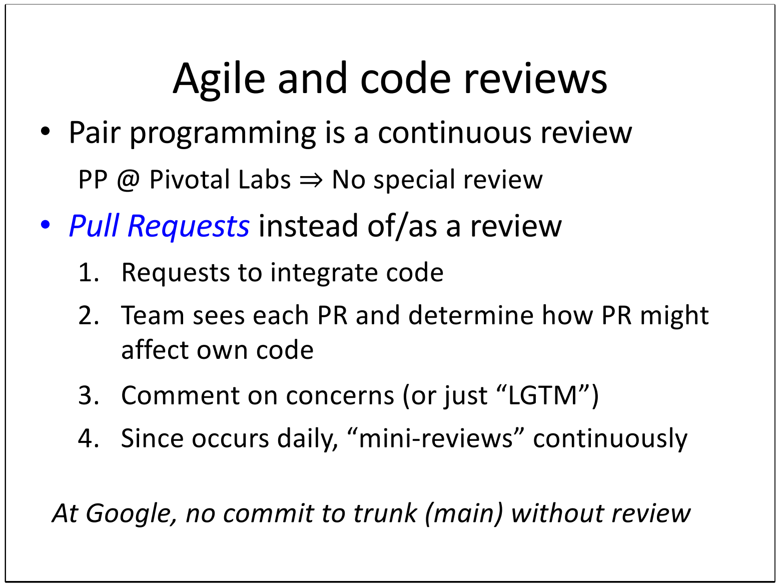


Resolving conflicts (e.g., different views on the technical direction)

1. Remember there is no “winning”, most questions don’t have “right answers” just tradeoffs
2. List all items on which you agree
   Instead of starting with a list of disagreements
   Maybe you agree more than you realize
3. Articulate the other side’s argument, even though you don’t agree
   Avoids confusions about terms or assumptions (often the root cause of the conflict)
4. Constructive confrontation (Intel)
   If you have a strong opinion that a proposal is technically wrong, you are obligated to speak up and seek a conclusion
5. Disagree and commit (Intel)
   Once a decision is made, embrace it and move ahead

See also: K Matsudaira, Resolving Conflict. Don’t “win.” Resolve. ACM Queue 14(5) 2016

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



Agile and code reviews

- Pair programming is a continuous review

  PP @ Pivotal Labs ⇒ No special review

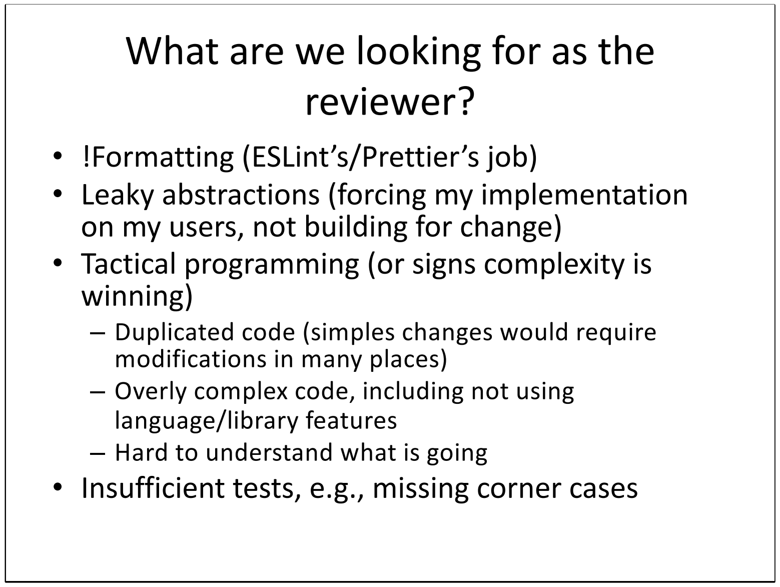
- *Pull Requests* instead of/as a review
  1. Requests to integrate code
  2. Team sees each PR and determine how PR might affect own code
  3. Comment on concerns (or just “LGTM”)
  4. Since occurs daily, “mini-reviews” continuously

At Google, no commit to trunk (main) without review

Opening PR means expecting other team members to review and comment on those changes, even if review is just to say, “Looks good to me” (LGTM). Depending on review outcome, PR may be merged, closed (withdrawn) or revised before merge.

Code review can be very effective. The detection rate for defects in code review is 55-60%! https://blog.codinghorror.com/code-reviews-just-do-it/ But for it to be meaningful the code needs be to be reviewable. A PR that changes thousands of lines across 10s of files? Really difficult to review. That is another motivation to keep user stories Small (the S in INVEST). Aim for more smaller merges than fewer large and far-reaching merges.

https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext



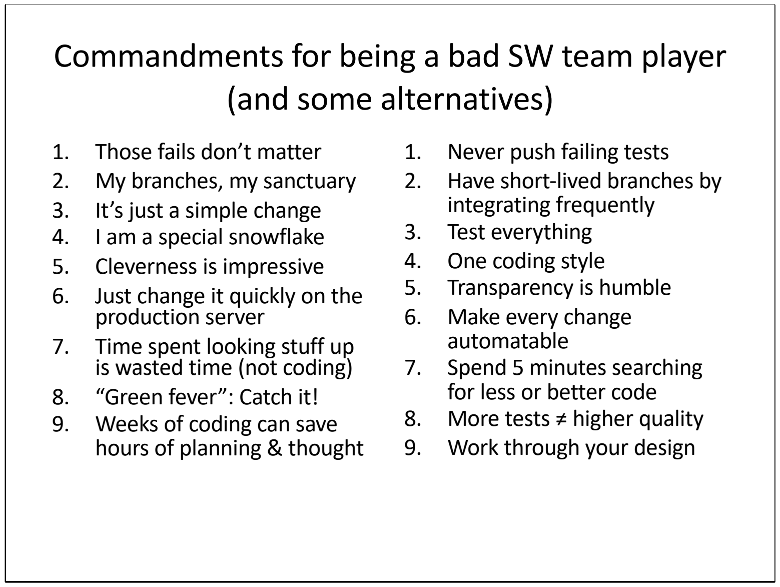
What are we looking for as the reviewer?

- !Formatting (ESLint’s/Prettier’s job)
- Leaky abstractions (forcing my implementation on my users, not building for change)
- Tactical programming (or signs complexity is winning)
  - Duplicated code (simples changes would require modifications in many places)
  - Overly complex code, including not using language/library features
  - Hard to understand what is going
- Insufficient tests, e.g., missing corner cases

TL;DR; Be humble, assume the best of intentions, ask don’t demand. Recall the goal is to make your team better, diminishing your teammates is counter to that goal.

A helpful guide: https://github.com/thoughtbot/guides/tree/master/code-review

1. Accept that many programming decisions are opinions. Discuss tradeoffs, which you prefer, and reach a resolution quickly.
2. Ask good questions; don't make demands. ("What do you think about naming this :user_id?")
3. Good questions avoid judgment and avoid assumptions about the author's perspective.
4. Ask for clarification. ("I didn't understand. Can you clarify?")
5. Avoid selective ownership of code. ("mine", "not mine", "yours")
6. Avoid using terms that could be seen as referring to personal traits. ("dumb", "stupid"). Assume everyone is intelligent and well-meaning.
7. Be explicit. Remember people don't always understand your intentions online.
8. Be humble. ("I'm not sure - let's look it up.")
9. Don't use hyperbole. ("always", "never", "endlessly", "nothing")
10. Don't use sarcasm.

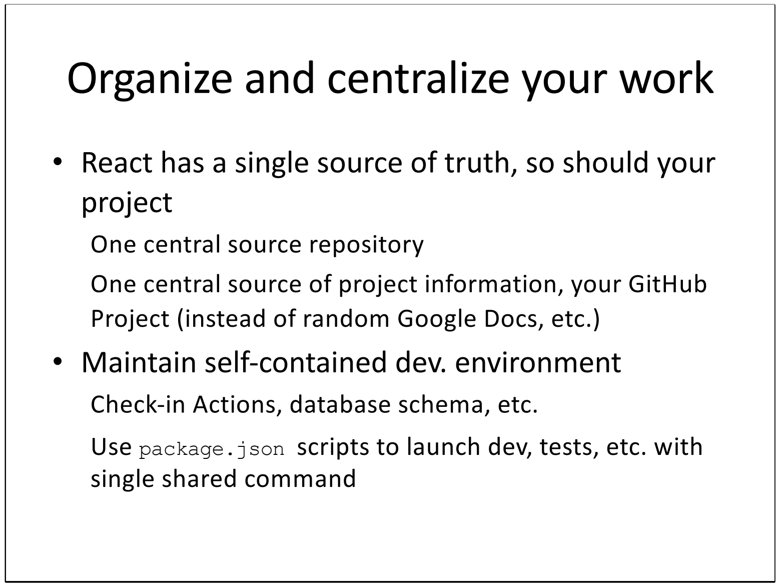


Commandments for being a bad SW team player (and some alternatives)

1.	Those fails don’t matter	1.	Never push failing tests
2.	My branches, my sanctuary	2.	Have short-lived branches by integrating frequently
3.	It’s just a simple change	3.	Test everything
4.	I am a special snowflake	4.	One coding style
5.	Cleverness is impressive	5.	Transparency is humble
6.	Just change it quickly on the production server	6.	Make every change automatable
7.	Time spent looking stuff up is wasted time (not coding)	7.	Spend 5 minutes searching for less or better code
8.	“Green fever”: Catch it!	8.	More tests ≠ higher quality
9.	Weeks of coding can save hours of planning & thought	9.	Work through your design

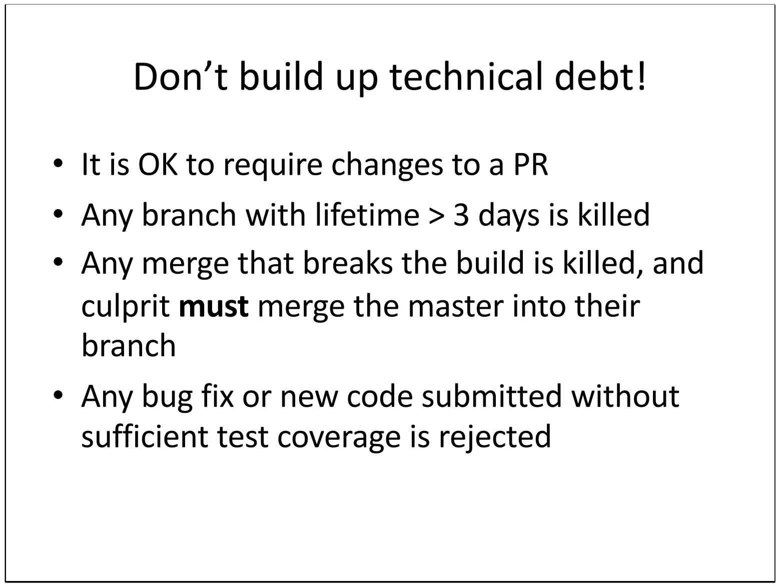
A specific anti-pattern for our projects is having one person own a specific layer/tier, e.g., the backend, the database, etc. And only they can/do make changes. Instead, aim to spread that knowledge around. If you create the first instance of something, e.g., setup the database, make that initial implementation an example for how others would extend that code with their own features. Recall our goal is to have teammates working on features, in their entirety, in parallel instead of working on the front and back-end in parallel.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



Organize and centralize your work

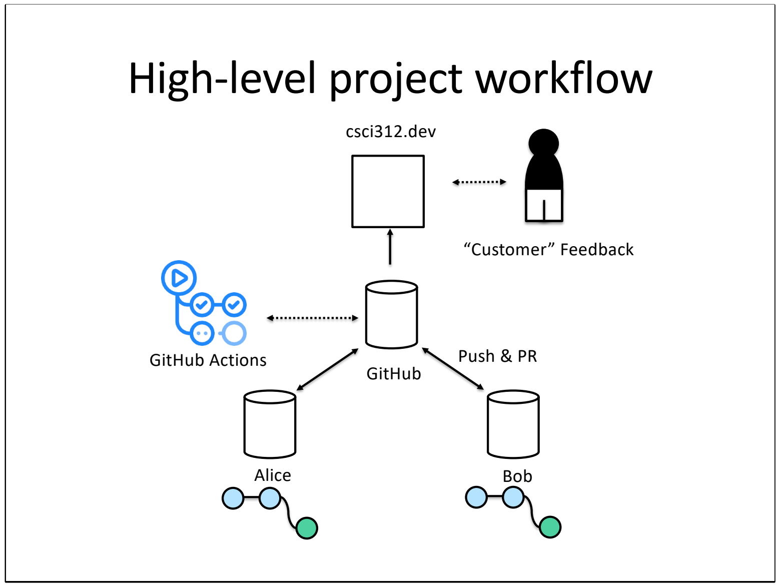
- React has a single source of truth, so should your project
  - One central source repository
  - One central source of project information, your GitHub Project (instead of random Google Docs, etc.)
- Maintain self-contained dev. environment
  - Check-in Actions, database schema, etc.
  - Use `package.json` scripts to launch dev, tests, etc. with single shared command



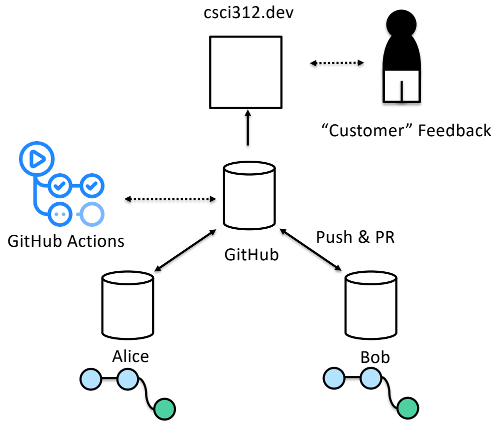
Don’t build up technical debt!

- It is OK to require changes to a PR
- Any branch with lifetime > 3 days is killed
- Any merge that breaks the build is killed, and culprit **must** merge the master into their branch
- Any bug fix or new code submitted without sufficient test coverage is rejected

Some of these may be a bit extreme in practice, but the larger message is to not abandon good processes in pursuit of short-term progress.

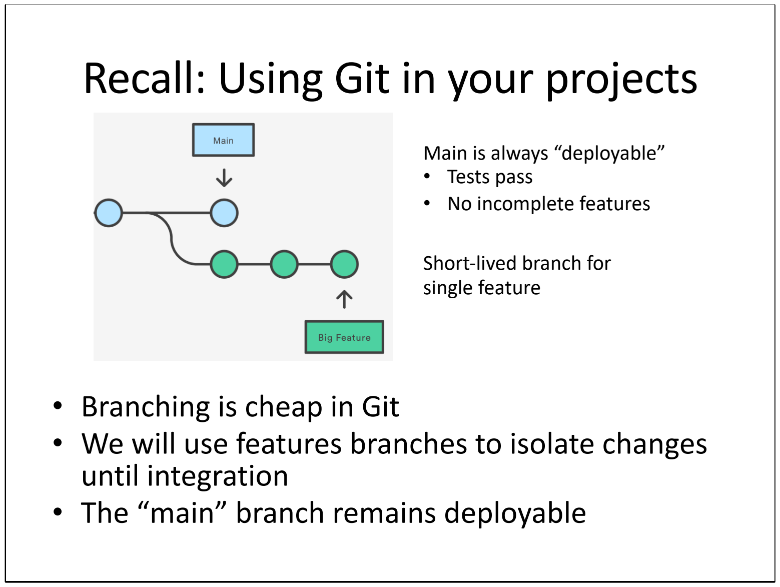


High-level project workflow

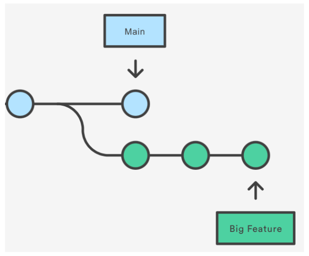


Although I want to note that we are aren't actually deploying directly from GitHub (although we could if wanted to…). Instead, someone will deploy, but they should only deploy the main branch as currently exists on GitHub.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



Recall: Using Git in your projects



Main is always “deployable”
- Tests pass
- No incomplete features

Short-lived branch for single feature

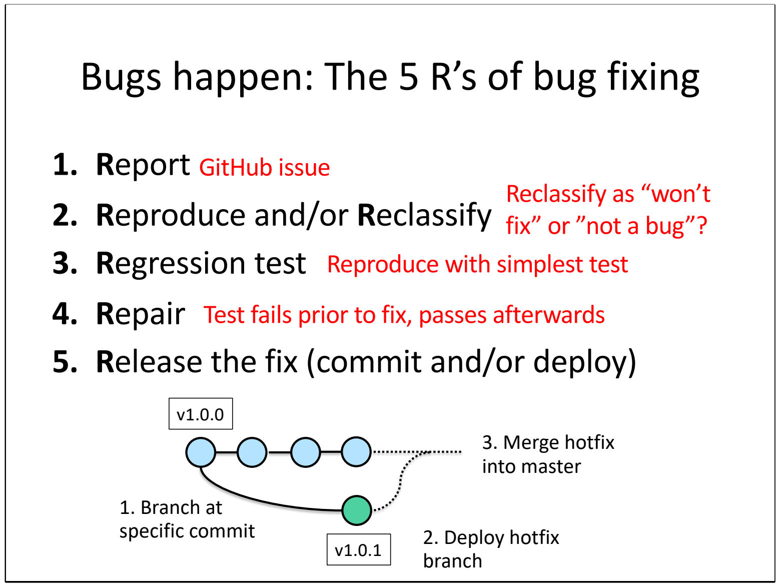
- Branching is cheap in Git
- We will use features branches to isolate changes until integration
- The “main” branch remains deployable

There are many Git workflows. We will adopt one particular workflow that works well for our kind of project and infrastructure. We will use of “feature” branches for all development.

A note about Git branch naming. The “main” branch used to be named “master” (and is still by default). That naming is no longer used as it is a charged term (https://www.acm.org/diversity-inclusion/words-matter) that does not reflect the relationship between branches. We will use “main” (although you will still see ”master” in some projects and documentation).

A key attribute is “short lived”. This workflow is most effective when branches only live for a brief period (as opposed to days or weeks) and most feature development can start main.

https://www.atlassian.com/git/tutorials/using-branches



Bugs happen: The 5 R's of bug fixing

1. **R**eport GitHub issue
2. **R**eproduce and/or **R**eclassify Reclassify as "won't fix" or "not a bug"?
3. **R**egression test Reproduce with simplest test
4. **R**epair Test fails prior to fix, passes afterwards
5. **R**elease the fix (commit and/or deploy)

v1.0.0
3. Merge hotfix into master
1. Branch at specific commit
v1.0.1
2. Deploy hotfix branch

Differing views on assigning points to bug fix…

Release approach:

- Rationale is that release branch provides a stable place to implement fix(es). The release branch provides a long-lived branch that manages these incremental fixes.
- For details: https://reallifeprogramming.com/git-process-that-works-say-no-to-gitflow-50bf2038ccf7

Other terms you will encounter is cherry-pick, which allows you to pull specific commits from one branch to another.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.