Plan & Document ⇒ Agile

Waterfall process:
Sequential phases

Agile: All lifecycle phases in repeated short cycles
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
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
Do you want to increment or iterate?

Incremental

Iterative

https://jpattonassociates.com/dont_know_what_i_want/
Iterative Incremental

http://itsadeliverything.com/revisiting-the-iterative-incremental-mona-lisa
Scrum (in a nutshell)

Frequent feedback!

Sprint Goal

Sprint Planning

Product Backlog

Sprint Backlog

Feature

Feature

Feature

Feature

Feature

Feature

Sprint Demo & Retrospective

"Deployable" product increment

24 hours between "standup" meetings
Scrum team

Development Team
- Self-organizing
- 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 Master
- Servant-leader for team
- Facilitate SCRUM process
Scrum artifacts: Product 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!)

*Relevant tools: Google Doc, GitHub issues, Trello, Pivotal Tracker, ...*
Recall: Epics, User stories, Scenarios

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

- User Stories
  - has many
  - Given <a context>,
    - when <an event happens>,
    - then <an outcome should occur>.
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
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
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”
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*
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
Thinking about pairing: Dreyfus squared for skills

<table>
<thead>
<tr>
<th></th>
<th>Novice</th>
<th>Adv. Beginner</th>
<th>Competent</th>
<th>Proficient</th>
<th>Expert</th>
</tr>
</thead>
<tbody>
<tr>
<td>Novice</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Adv. Beginner</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Crazy learning!</td>
</tr>
<tr>
<td>Competent</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proficient</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expert</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Novice:** Needs rules  
**Advanced Beginner:** Tests the rules  
**Competent:** Applies rules  
**Proficient:** Falls back on rules  
**Expert:** Transcends rules
Student Advice: Pair programming

• “Helped avoid silly mistakes that could take a long time to debug”
• “Changing partners frequently made team more cohesive”
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
High-level project workflow
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 without review*
What are we looking for as the reviewer?

- Formatting (ESLint’s job)
- Leaky abstractions (forcing my implementation on my users...)
- Hard to maintain code
  - Duplicated code (not DRY)
  - Overly complex code
  - Global variables and other “foot guns”
  - Poor variable names and needed comments
- Insufficient tests, e.g. missing corner cases
Using Git in your projects

- Branching is cheap in Git
- We will use features branches to segregate commits until integration
- Most branches will eventually die (many long-lived branches is not agile)

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

Short-lived branch for single feature
Recall: Git is distributed

- git checkout feature
- git rebase master
- git checkout master
- git pull origin master
- git push origin feature
- PR
The golden rule of rebase (and any re-writing of history)

• Never modify public history (commits)
  If anyone else could see this branch (e.g. you pushed to GH), don’t use rebase, --force, or any command that alters history

• When in doubt it is OK to just merge
You try to push to a remote branch and get a “(non-fast-forward) error: failed to push some refs [...]" message. What should you do?

A. Use "--force" argument to force Git to complete the push

B. You must still have merge conflicts. Manually fix those conflicts then push.

C. There have been intervening commits to remote branch. Pull then push again.
Conflicts happen: Merge commits

On branch feature
Unmerged paths: (use "git add/rm ..." as appropriate to mark resolution)
both modified: App.js

Git identifies the conflicts:

here is some content not affected by the conflict
<<<<<<< master
this is conflicted text from master
=======
this is conflicted text from feature branch
>>>>>>> feature branch;

Fix all conflicts then add updated files and commit to complete the merge
Bugs happen: The 5 R’s of bug fixing

1. **Report** GitHub issue
2. **Reproduce and/or Reclassify** Reclassify as “won’t fix” or “not a bug”?
3. **Regression test** Reproduce with simplest test
4. **Repair** Test fails prior to fix, passes afterwards
5. **Release the fix (commit and/or deploy)**

![Diagram]

- **v1.0.0**
- **v1.0.1**
- 1. Branch at specific commit
- 2. Deploy hotfix branch
- 3. Merge hotfix into master
Suppose you discover that your most recent release contains a bug whose regression test will require extensive mocking or stubbing because the buggy code is convoluted. Which action, if any, is NOT appropriate in this situation?

A. Do the refactoring using TDD on the release branch, and push the bug fix as new code with those tests
B. Do the refactoring using TDD on a different branch, push the bug fix as new code with tests, then cherry-pick the fix into release
C. Create a regression test with the necessary mocks and stubs, painful though it may be, and push the bugfix and tests to release branch
D. Depending on project priorities and project management, any of the above might be appropriate
Commandments for being a bad SW team player (and some alternatives)

1. Those fails don’t matter
2. My branches, my sanctuary
3. It’s just a simple change
4. I am a special snowflake
5. Cleverness is impressive
6. Just change it quickly on the production server
7. Time spent looking stuff up is wasted time (not coding)
8. “Green fever”: Catch it!
9. Weeks of coding can save hours of planning & thought

1. Never push failing tests
2. Have short-lived branches by integrating frequently
3. Test everything
4. One coding style
5. Transparency is humble
6. Make every change automatable
7. Spend 5 minutes searching for less or better code
8. More tests ≠ higher quality
9. Work through your design
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 (instead of random Google Docs, etc.)

• Maintain self-contained dev. environment
  Check-in DB, Travis, Heroku configurations
  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 rebase against master
• Any bug fix or new code submitted without high test coverage (e.g. 90%) is rejected
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 are required

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!