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 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.
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)
• 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. The catch is we have to address the entire scope at one time, i.e. we are working on the entire image (a risky approach).
“[In the] pure [iterative] example the team starts to address the entire scope at the same time – which adds considerable risk in software delivery.

“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.
"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
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.

Each iteration I want you to select a different Scrum Master. This person will be responsible for facilitating the Scrum process.
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.

---

**Scrum artifacts: Product backlog**

- **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, ...*
User stories should be:
• Specific
• Measurable
• Achievable
• Relevant
• Time-bound

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 to implementing it. If they are, you should go back to your stakeholders to refactor the user story into a set of simpler tasks that you do know how to approach.

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

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

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

“promiscuous pairing”, regularly swapping partners so that eventually everyone is paired

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
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>✗</td>
</tr>
<tr>
<td>Adv. Beginner</td>
<td></td>
<td>Crazy learning</td>
<td></td>
<td></td>
<td></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

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 Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
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.
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 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 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/](https://blog.codinghorror.com/code-reviews-just-do-it/)


Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
What are we looking for as the reviewer?

• !Formatting (ESLint’s/Prettier’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

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.
What is different about the state of this repository compared to what you have experienced as solo developers? Commits after branch (your teammates working)

https://www.atlassian.com/git/tutorials/using-branches
Another “friendly” behavior is to squash your feature commits to just one (makes it easier to review).

What happens when you need to make changes to your PR. Commit and push to remote feature branch, will be added to PR.
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.

Answer: C

(A) will rewrite shared history; (B) is not necessarily true; (C) this error is created when there are changes to the remote branch that haven't been fetched to the local repository.
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

You will need to add all files you change with git add (is if you were making a commit), before git commit to indicate that the merge is complete

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

Answer: A

We don't want to do extensive development, such as refactoring, on the release branch. Instead we either only want to make simple fixes (can have complex tests) or use other branches.

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

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 (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

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