# CS 312 Software Development

**Agile Development**

---

## Manifesto for agile software development (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.
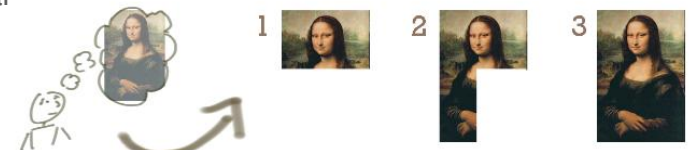
---

## 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
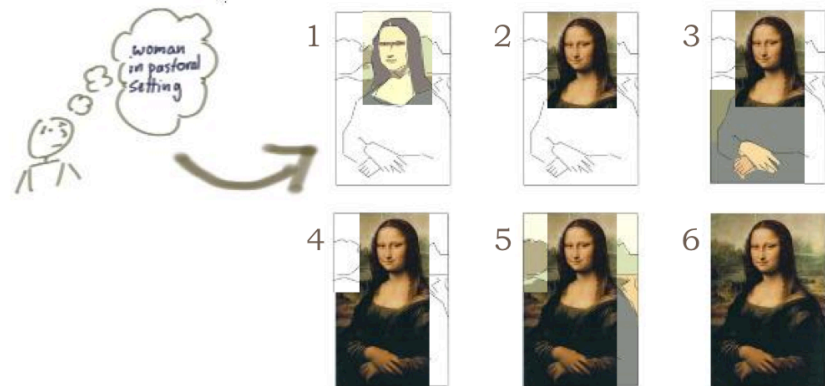
---

## Do you want to increment or iterate?

Incremental



Iterative

## Iterative Incremental
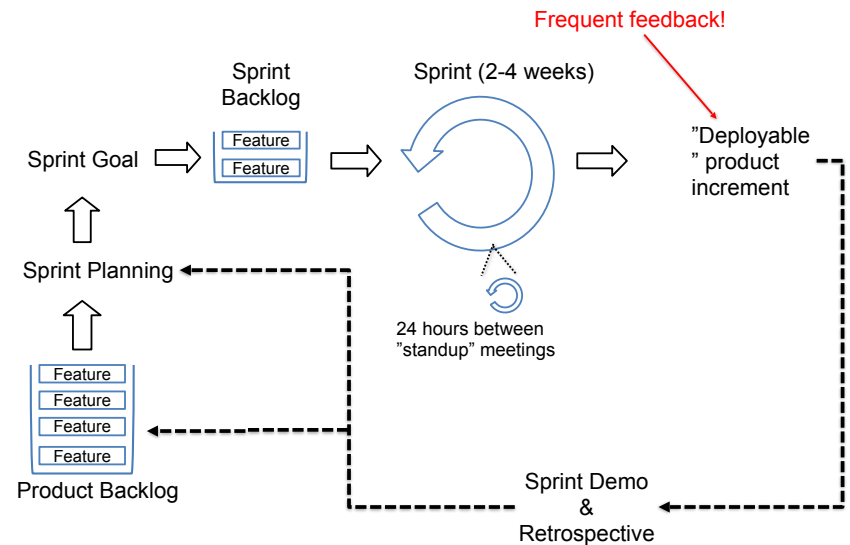


http://itsadeliverything.com/revisiting-the-iterative-incremental-mona-lisa

## Scrum (in a nutshell)



Frequent feedback!

Sprint Backlog

Sprint (2-4 weeks)

Sprint Goal

Feature
Feature

"Deployable" product increment

Sprint Planning

24 hours between "standup" meetings

Feature
Feature
Feature
Feature

Product Backlog

Sprint Demo & Retrospective

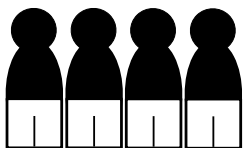## Scrum team



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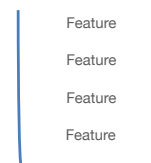- A single team without sub-teams
- Accountable as a group

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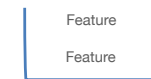- Responsible for prioritizing the product backlog

**Scrum Master**
- Facilitator for team
- Facilitates SCRUM process

## Scrum artifacts: Product backlog



Feature
Feature
Feature
Feature

Product Backlog

Feature
Feature

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!)

*Relevant tools: GitHub issues, Google Doc, Trello, Pivotal Tracker, …*

## 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

---

**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

---

# Agile and code reviews

- Pair programming is a continuous 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/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

# How to write reviews
**…and not alienate anyone on your team**

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

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

# Revisiting the Pull Request
**using rebase**
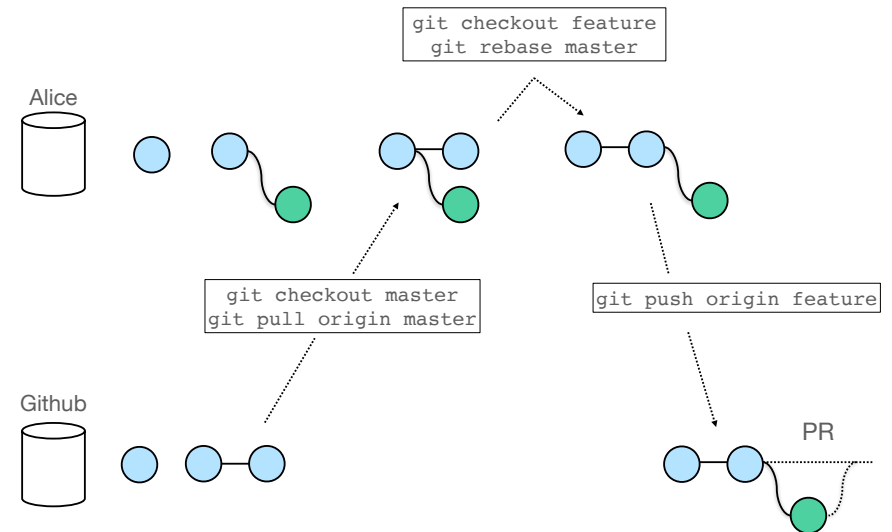


# The golden rule of rebase
(and any re-writing of history)

- 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 any command that alters history
- When in doubt it is OK to just merge

# Conflicts happen: Merge commits

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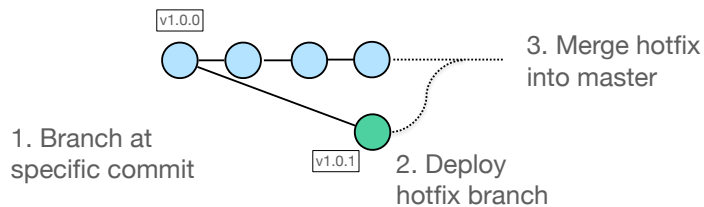
Git identifies the conflicts:

```
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. **R**eport
2. **R**eproduce and/or Reclassify
3. **R**egression test
4. **R**epair
5. **R**elease the fix (commit and/or deploy)

GitHub issue

Reclassify as "won't fix" or "not a bug"?

Reproduce with simplest test

Test fails prior to fix, passes afterwards

v1.0.0

3. Merge hotfix into master

1. Branch at specific commit

v1.0.1    2. Deploy hotfix branch

---

**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 |

---

# 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!*