Project Management

INTERSECT – Bootcamp 2023

Dave Rumph

What is Project Management?

- Definition:
  - Project management is the use of specific knowledge, skills, tools and techniques to deliver something of value to people. The development of software for an improved business (or scientific) process, the construction of a building (or scientific instrument), the relief effort after a natural disaster, the expansion of sales into a new geographic market—these are all examples of projects.
    - (from Project Management Institute website)

Who needs project management (PM)?

- If a group of people are working together, some amount of project management methodology will help
    - How large, long and complex the project is will determine how much project management discipline will be helpful
    - Generally, the more time and/or people involved, the more PM helps
- If you're writing scripts for your own research to test out some ideas, and the project will extend some time, you (and "future you") will probably benefit from some PM
- If you're experimenting with something for just one week, you probably don't need project management
    - But are you sure it's really just "one week"?

Phases of a Software Project

- [Most of] the phases of software development:
  - Determine software requirements
  - Software design
  - Implementation
  - Software testing
  - Software documentation
  - Release/deploy the software to the customer
  - Ongoing maintenance
- These phases can be mixed, or (sometimes) reordered
  - Line between requirements and design is often blurry
  - Implementation, testing and user docs often proceed concurrently
  - Test Driven Development (TDD): Tests are written first
- Definition of roles vary
  - E.g. the “customer” or “user” may also be the author of the software, or may be quite removed

Software Development Process?

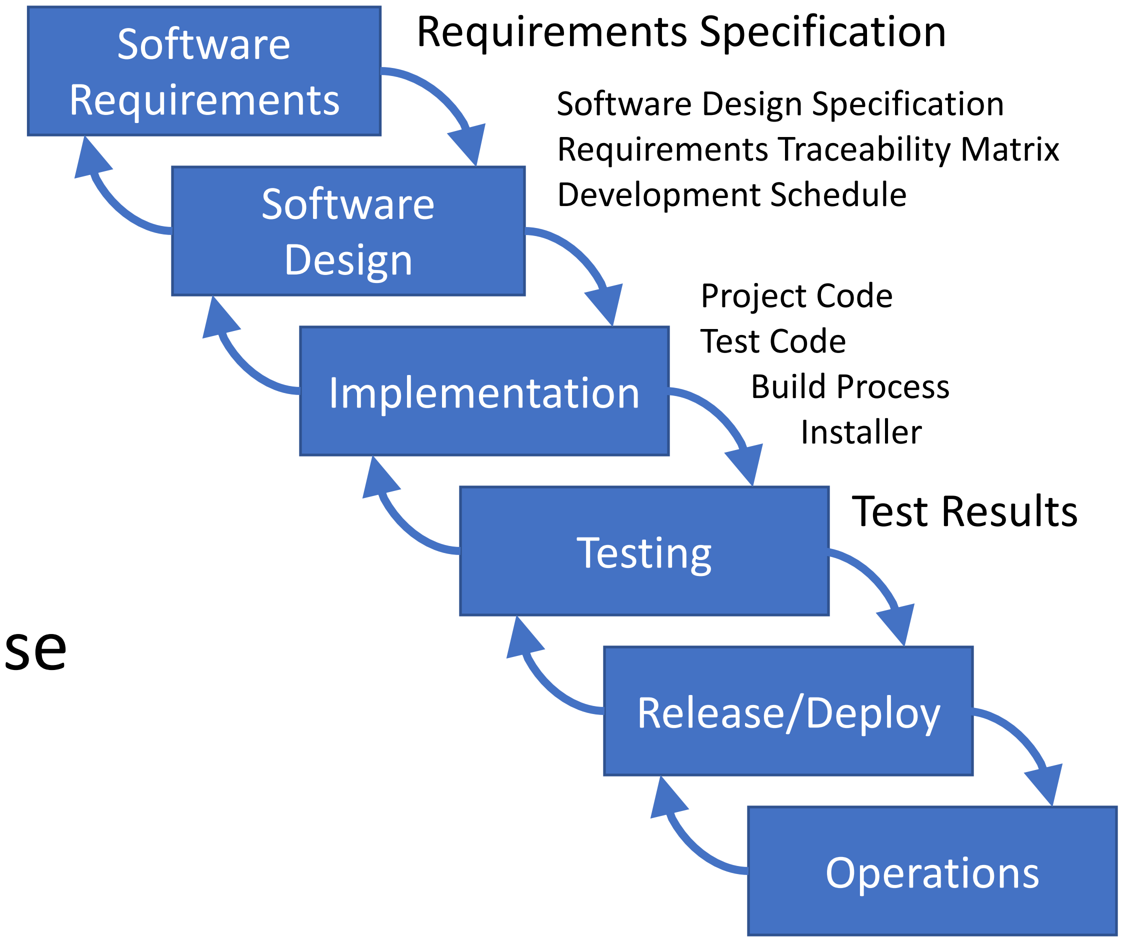
[Most of] the phases of software development:

- Determine software requirements
- Software design
- Implementation
- Software testing
- Software documentation
- Release/deploy the software to the customer
- Ongoing maintenance

- Idea: try to complete each phase before the next phase begins
    - Generally referred to as the **waterfall model**
    - Each phase produces specific deliverables, which feed into the next phase
    - “Progress flows downwards through the phases, like a waterfall.”

The Waterfall Model

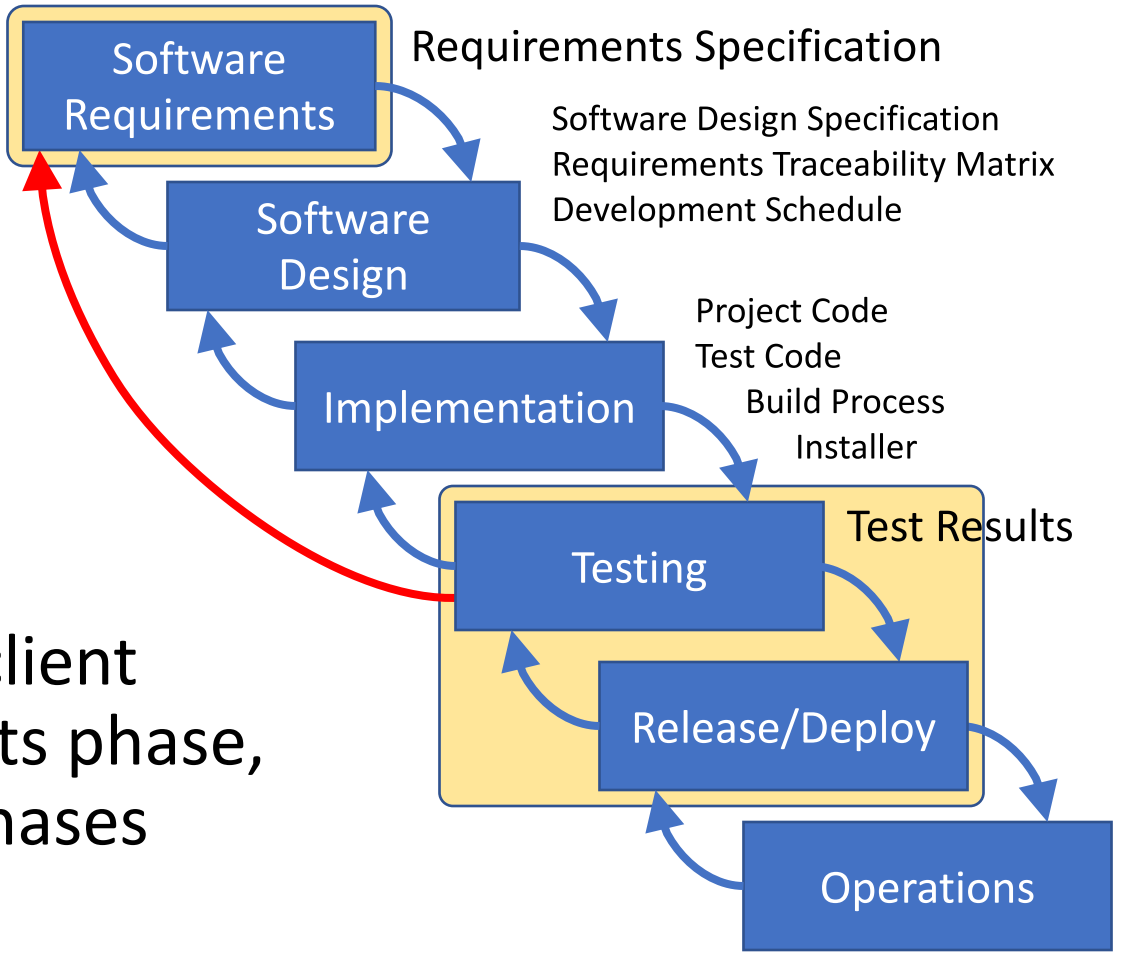
- Each phase produces artifacts that feed into the next phase
  - Assumption: each phase's artifacts are actually correct!
- Ideally, each phase will also identify any issues/flaws in the previous phase
  - Iterate between phases to fix issues



The Waterfall Model: In Real Life

- A much more common scenario: miscommunication of software requirements!
  - May not be identified until client demos, or formal acceptance testing

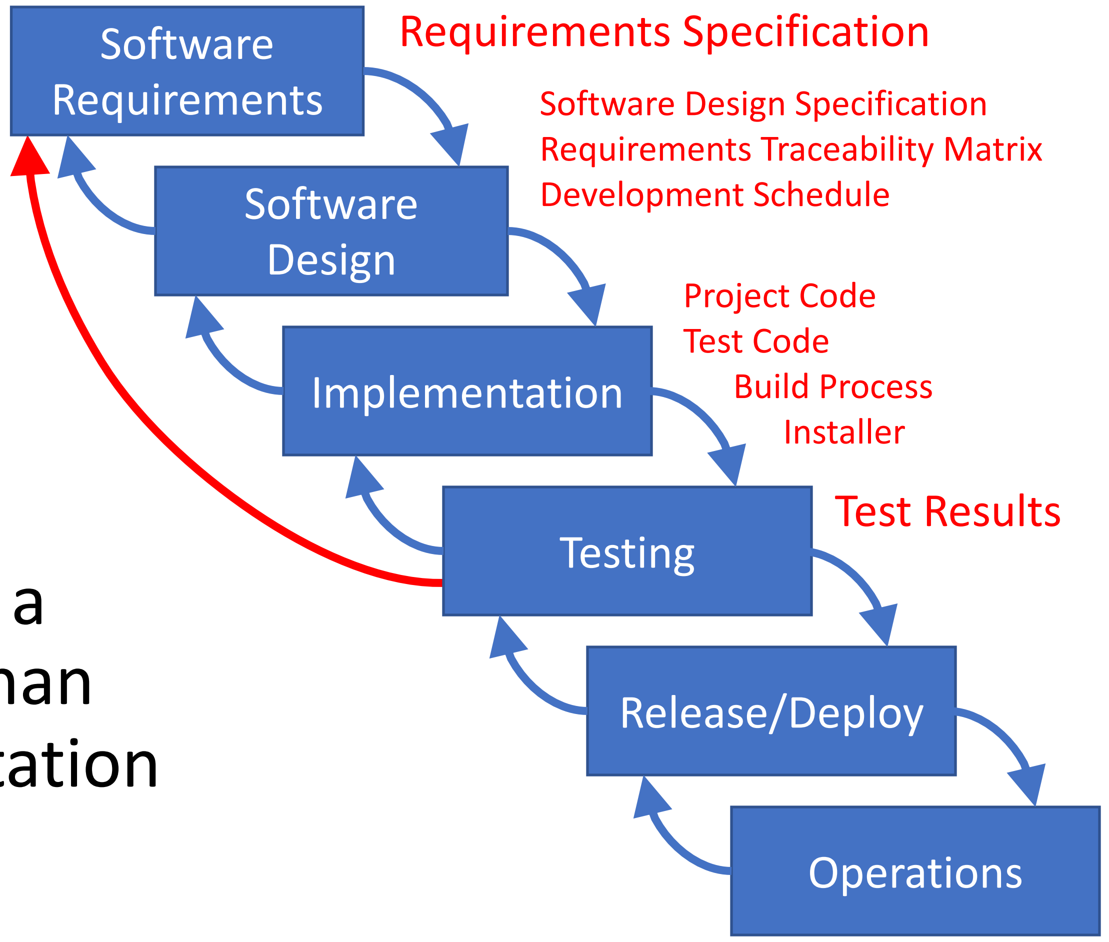
- In traditional waterfall models, the client is often only involved in requirements phase, and in acceptance-testing or later phases



The Waterfall Model: In Real Life

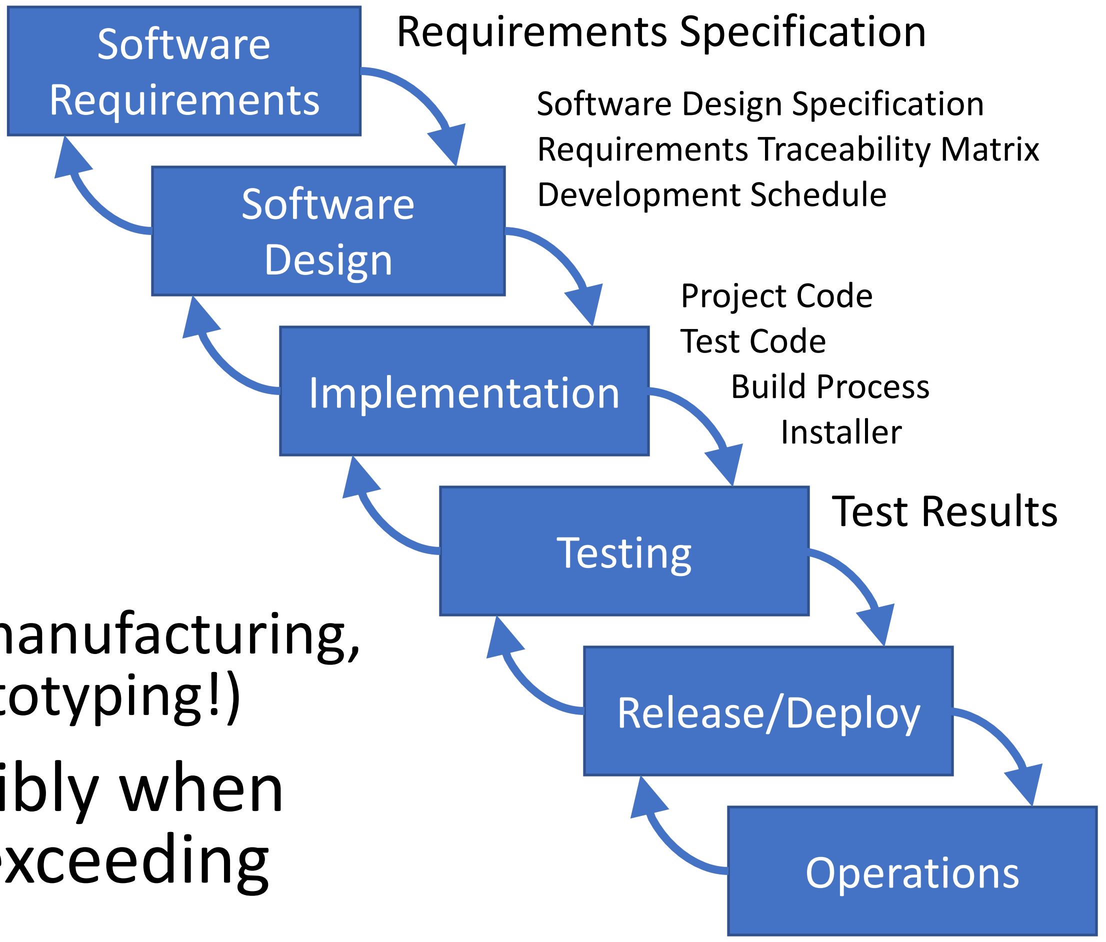
- In this case, *very significant* amounts of work may need to be reviewed, revised or discarded

- It’s 50x to 200x less expensive to get a requirement right in the first place than to fix a requirement after implementation
  - Boehm and Papaccio, 1988



The Waterfall Model

- When does the waterfall model work well?
- Waterfall model works well when:
  - Software requirements are very clear, well understood, and unchanging
  - Technical implementation details are also very well understood
  - (Generally considered more suited to manufacturing, but this is also changing with rapid prototyping!)
- Inversely, waterfall model works terribly when these things aren't present, greatly exceeding time and financial budgets



Software Development Methodologies

- Waterfall-based models don’t handle lack of knowledge, or change, very effectively
  - Poorly understood or changing software requirements
  - Poorly understood technology or architecture choices
  - (Lack of software-engineering discipline kills virtually all approaches)
- At the other end of the spectrum are **agile methodologies**
  - These methodologies are optimized to deal with poorly understood and/or changing requirements during development
  - Also tend to incorporate other practices to improve software quality
- Started becoming more popular in mid 1990s, as issues with waterfall approach became more and more evident

https://agilemanifesto.org/

The 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.

Core Value of Agile

- (Nearly) everything about Agile Development is based on one value:

Identify and eliminate sources of *WASTE*

Individuals and Interactions over Processes and Tools

- Good processes and tools are important, but it’s more important to make sure that people are happy, productive and communicating
- If a process or tool isn’t working for people, change it!
- Agile methods favor lightweight, widely accessible tools over fancy, expensive software packages
  - e.g. whiteboards in public areas
  - e.g. simple text formats like markdown and restructuredText for documents
- Agile methods promote a sustainable pace for software development
  - i.e. a pace that can be sustained indefinitely; no heroics

Individuals and Interactions over Processes and Tools

- Agile methods incorporate very regular communication of status, identification of issues, and retrospectives/postmortems on efforts
  - Preferably face-to-face interactions
  - A common approach:
    - Daily interactions between software developers
    - Planning and retrospectives every 1-4 weeks
- Agile methods also respect the time of individuals
  - Meetings are time-boxed so that software developers can focus on the problems they are solving
  - e.g. daily status meetings time-boxed to 15 minutes
  - e.g. planning meetings time-boxed to 1 hour per week of development effort

Working Software over Comprehensive Documentation

- Requirements specifications and design documents have one main purpose: to help get to useful, working software
  - Customers generally don’t care about them once they have their software!
- Agile methods generally maintain only documentation that allows the project to be developed, used and maintained, and no more
  - A common agile approach: “just barely enough” documentation
  - Focus on what developers can’t easily figure out looking at the code: the “big picture,” and “why?” rather than “what?”
  - Do work “at the latest *responsible* moment”
    - Guards against wasted work from changing requirements
- Agile methods generally use lightweight representations for software requirements, UI designs, software architecture/design, etc.
  - e.g. a whiteboard sketch of a UI workflow or domain model
  - e.g. a 3x5 card with a brief user story described on it
  - e.g. GitHub Issues to capture feature requests as well as bugs

Customer Collaboration over Contract Negotiation

- “Contract negotiation” means: “Give me a specification of what you want, then leave me alone so I can go get it done.”
  - Can occur between team members, as well as between a team and its client
  - Tends to make processes less flexible and adaptable to change
- Agile expects requirements, or the understanding of requirements, to change!
  - Encourages constant collaboration within teams, and between developers and clients, to ensure that everyone’s needs and goals are being met
  - Some agile approaches do limit opportunities to change requirements to reduce “requirements churn”

Responding to Change over Following a Plan

- When requirements change, the current plan may no longer be the right plan
- Agile methods adapt to change by reviewing and revising plans iteratively, over short time periods
- Agile methods tend to only have detailed plans for the short- to medium-term
  - It doesn’t make sense to devise detailed plans for the long-term, if circumstances are likely to change
  - If [when] changes come, any effort spent on detailed long-term plans would have been wasted

Scrum

- **Scrum** (and variants) is one of the most widely used agile methodologies
- Initial phase: **project envisioning**
  - How long this takes depends on the scope of the project
    - May be as much as 2-4 weeks
- At a high level, the client and the team define the following:
  - A project vision – what the project is intended to accomplish
    - No longer than a few pages at most, and often much shorter
  - A project backlog – a list of key features to be implemented
    - In envisioning, the features tend to be high-level and broad
    - Typically expressed as "User Stories"
    - Future efforts will refine the details
  - An initial roadmap – a series of releases that include features in the backlog
    - The first release should be the "Minimal Viable Product"

Sprints

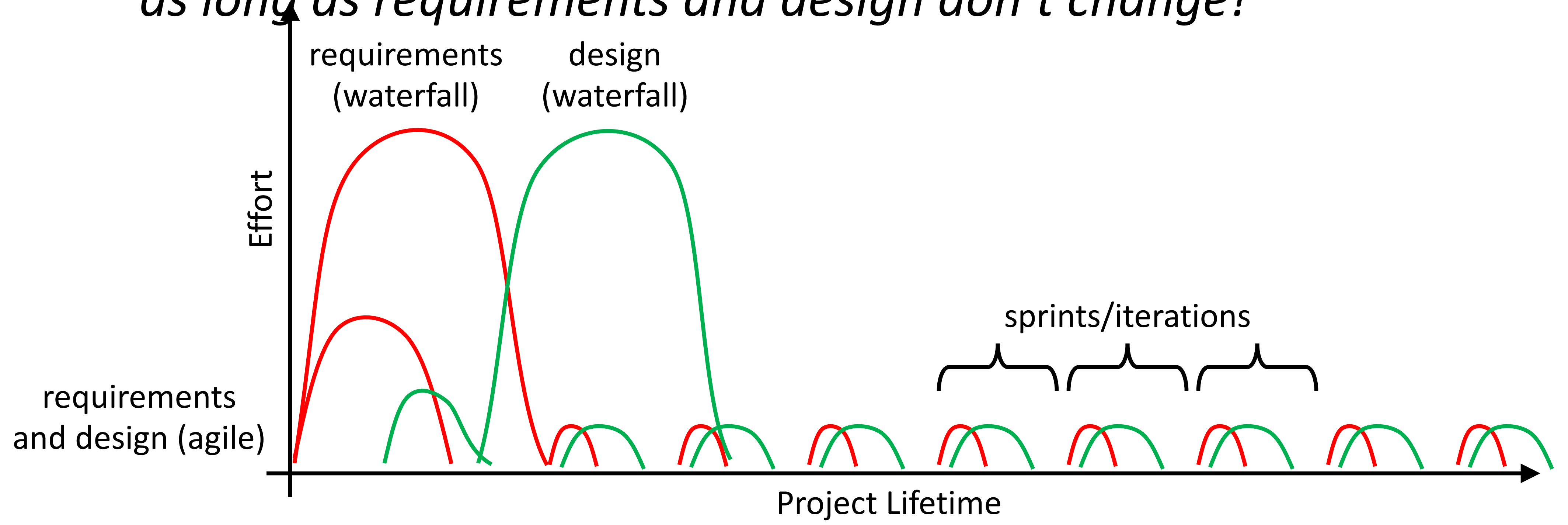
- After project envisioning is completed, development occurs in a series of time-boxed iterations (1 week to 4 weeks) called **sprints** or **iterations**
- Sprints start with a **planning meeting**, time-boxed to 1 hour per week
  - e.g. for a 4-week sprint, the planning meeting may be no longer than 4 hours
- The developers choose features from the project backlog to be implemented in the sprint
  - Larger features are decomposed into smaller features; details are discussed
  - Feature priority is determined by the client and the team, based on relevance to the product roadmap, risk/uncertainty, level of difficulty, etc.
    - ("MoSCoW" – stay tuned!)
  - Implementation cost is determined by the developers
- The developers commit to completing those tasks within the sprint

From Sprint to Sprint

- At the end of the sprint is a **sprint demo**, and a **retrospective**
  - Tends to be a short meeting, or is first part of next sprint's planning meeting
- Completed work is demonstrated to the client / stakeholders
  - Agile places high priority on *working software*, and on *customer collaboration*
- Everyone discusses lessons learned in the sprint
  - What unexpected issues had to be dealt with, and what was learned
  - What can be done better in future sprints
- A sprint may include releasing/deploying software, or it may not
  - Depends on where the team is with respect to the product roadmap

Agile vs. Waterfall

- Agile does requirements gathering and design incrementally, mostly just before implementation
- Overall, may be more costly than a well-executed waterfall approach, *as long as requirements and design don't change!*



Defining the Work

- In scrum, the **project backlog** is the specification of what work remains to be done on the project
- User stories – descriptions of features that can be implemented within a single sprint
- Epics – features that will require multiple sprints to complete
  - Epics contain user stories that correspond to the epic
- Bugs – software defects encountered in completed work
- Periodically, **backlog grooming** occurs
  - Stories and epics that are no longer relevant to the project, are removed (e.g. because a requirement was later found to be unnecessary)

Requirements as User Stories

- Describe software features or capabilities from a user’s point of view
  - Regular user
  - Administrator
- Written in “story” form:
  - “As a user, I can view my data”
  - “As a user, I can edit my data”
  - “As an administrator, I can change the access rights of a user”
- Non-functional requirements:
  - “The system is robust against changes in the database format”
  - “The system responds to a database commit in less than 50 ms”
  - (Non-functional requirements may not become separate tasks)

Kanban

- Japanese for “Board”
- A lightweight method for tracking work on a project
  - Originally done using a board marked into four columns and Post-It notes
  - Integrates well with GitHub
- Best for small to medium size projects
- Demo

MoSCoW

- A lightweight technique for prioritizing work
  - Typically done in the sprint planning meeting
- Focus is on the **next** software release/sprint
  - **M**ust have
    - Without this, we don’t have a useable release
  - **S**hould have
    - The release would be less valuable or compelling without this, but would still be worth releasing
  - **C**ould have
    - Nice to have, but not essential
  - **W**on’t have
    - Maybe not ever, but too much for this time
- Work on Must Haves
- Work on Should Haves if there is time remaining in the sprint

Hands-On: Setup

- MoSCoW
  - Use your GitHub repo from yesterday’s “Issues Tracking” session
  - Add Issue labels for “Must have”, “Should have”, “Could have” and “Won’t have”
- Kanban
  - Create a GitHub Kanban project board

Hands-On

- Sprint Planning
  - In a group of about four:
    - Create a Work Backlog based on the User Stories on the right, expressed as GitHub issues
    - Rate your issues for the “next” sprint using MoSCoW
  - Individually:
    - Populate the Work Backlog section with all of your issues
    - Move those rated “Must have” and “Should have” to “Ready to be Developed”

- User Stories:
  - As a user, I can display my image data using preexisting tools
  - As a user, I can specify an input and an output file name
  - As a user, if I don’t specify an output file name, a reasonable one is generated
  - As a user, I can display “before” and “after” images such that I can easily see differences

Debrief

- Did you encounter any problems during the activities?
- Do you think you could use these techniques in your own projects?
- Are there any areas where you would modify or customize the recommendations for your context?