Midterm Preparation
CSE 112 Spring 2022

Name: ______________________
PID: ________________________
Team # 15

Part 1: The questions here include material from lecture, slides, content from 112 provided already and some information beyond the course easily findable online. The Midterm will be composed exclusively of a time appropriate set of these questions reduced. Note we may modify logic or order of questions to combat direct memorization.
You may work together on your team to answer these questions together and prepare for the test. Your team must return your final team agreed upon answers back to your TA/tutor mentor contact by Wednesday May 4th at 6pm including your provided questions.

1) Definitions

A) In your own words try to precisely define Software Engineering. (Hint: Focus on the second word)

The process of designing, building, and maintaining software that helps its users in a way that considers good engineering principles like safety, reliability, quality, and longevity.

B) Define what you think Code Craftmanship involves?

Code Craftsmanship involves following rigorous development methodologies such as: agile development, automating testing, and design patterns which have an emphasis on creating "high quality" code. Further programmers should have pride in their work similar to craftsmen in their specific field.

C) Define what you think Software Manufacturing involves?

- the process of producing software in ways similar to the manufacturing of tangible goods (definition from Wiki)
- Designing and developing a piece of software with the intention of replicating it
- Modularization?
- Following a rigid formula / plan to produce code

D) Does Software Engineering contain these ideas or is it separate and different? Explain your opinion?

- code craftsmanship: included in software engineering (but not the core)
- software manufacturing: based on software engineering (to create the product for manufacturing)

E) The Atlantic article mentioned in the first homework suggested that our industry is a bit loose in its use of the word Engineer, why did the article's author feel that is the case? Article link

- Software products don't always last long, and crash/have errors that lead to devastating results more often than other engineering fields.
- A typical engineer would not implement "Duct tape" fixes aka quick and dirty fixes that do not address the root issue.

- What software engineers build is much different than what average engineers of other professions build. One can easily change/iterate code, but it is much more difficult and costly to change something once it is already built, say a building or bridge. This resulted in software to be built in a style that differs from other engineered goods.

2) If a super tool/language/framework existed that could make our developer productivity significantly better, what downsides might we encounter if we adopted that super thing?

Source: Slide 63 from 2-Making a Software Engineer.

In industry, we see that there is an over emphasis in searching for "one true tool". This can cause issues because a general super tool might not be able to do a single thing as well as a specific tool meant for the job (think of a swiss army knife vs a screwdriver). A good programmer does best with good tools, yet tools have trade-offs just like anything else, so it is best to have the correct tool for the job rather than a single tool for all jobs

From Lecture: Might be difficult to understand how to use the tool, entity that makes the tool (google, facebook) might have significant influence.

3) It is a commonly held belief that people are the key to productivity, even more than tools. If this assertion is true and then we should aim to have great people. It is also suggested that 10x programmers exists which are programmers are many times more productive than others.

a) Do you believe in a 10x programmer?

[✓ **Yes**] No

b) If you do believe in this, how does one become a 10x programmer?

- "Built different" - be born with the mental and physical capabilities and motivation to withstand long, productive hours of coding
- Accumulate lots of experience with multiple facets of programming
- Make the right decisions the vast majority of the time

c) Do you believe that a process can be built where the effects of the human programmer could be vastly minimized?

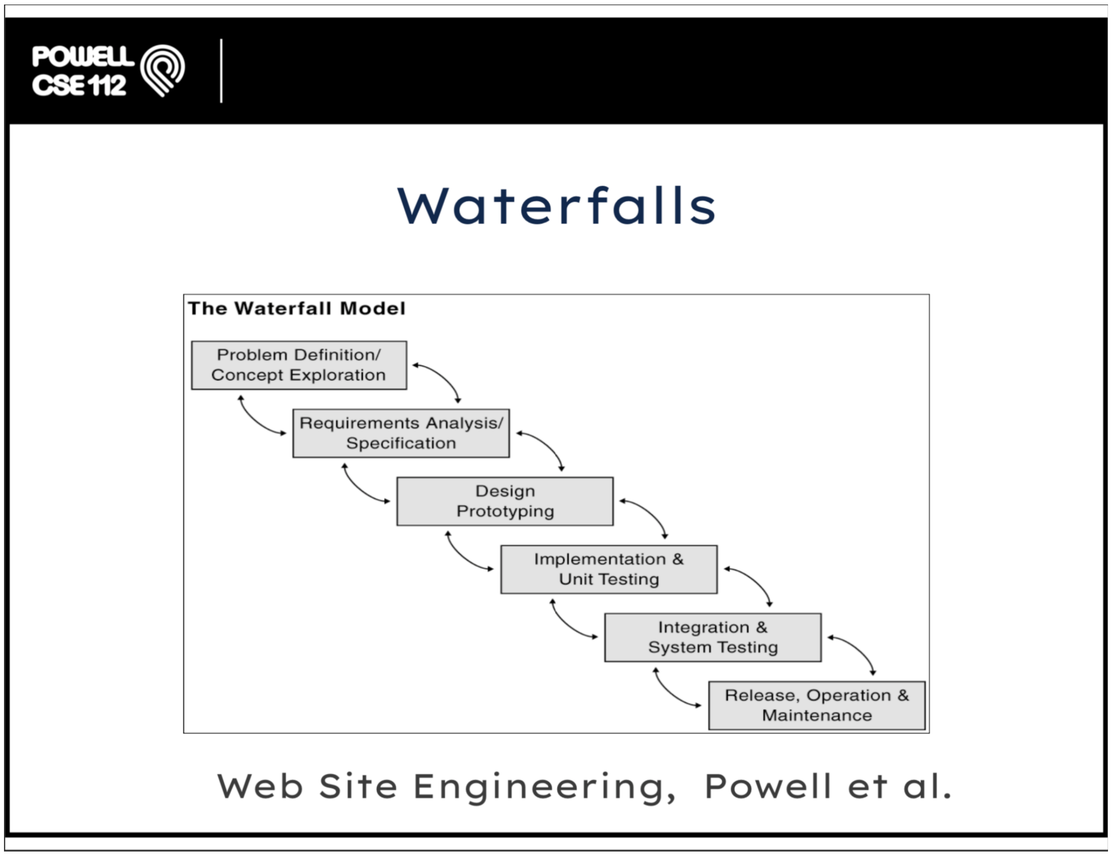
Yes [✓ **No**]

d) If you believe you can design a more human interchangeable process, what would we have to guarantee with our programmers?

If you don't believe we can build such a process provide your reason(s) why.

We don't believe we can build such a process because human problems require human solutions. Building an omniscient process such as this would likely require figuring out human-level artificial intelligence, which seems infeasible in our lifetime. Customer problems need to be addressed with empathy and understanding for the needs of people.

4) One popular philosophy for building software is often termed (BDUF) "Big Design Up Front". This approach is more traditionally known as the waterfall model. Draw a diagram below describing the rough steps of this model and give each a name. Describe next to the diagram the purpose of each step.


POWELL CSE 112

Waterfalls

The Waterfall Model
Problem Definition/ Concept Exploration
Requirements Analysis/ Specification
Design Prototyping
Implementation & Unit Testing
Integration & System Testing
Release, Operation & Maintenance

Web Site Engineering, Powell et al.

Description:

Problem Definition:
- Specify what is the problem / what's the main function of the application.

Specification:
- Specify the core parts of the program at a high level.

Prototyping:
- Start implementing basic cases of the specification. This also involves throwing out bad prototypes and settling on a good prototype

Implementation & UI testing:
- Work out all the details of the functions, and test them accordingly.

Integration & System Testing:

- Merge all functions together and ensure that the application still "works"

Release, Operation & Maintenance:

- Release the product and keep track of it to see if there are further enhancements or end-user problems that the team haven't caught in the development cycle.

5) The model presented in #4 has been criticized for a number of legitimate reasons. Write two criticisms below, suggest a solution for the criticism and a counterpoint to the criticism.

Criticism	Solution	Counterpoint
Even with the best planning, things can change, throwing a wrench in your plans and wasting a lot of effort	Be less specific and more big-picture with planning, implement iterative principles for smaller details	Requires much more communication, may not be viable for every situation
Customer is less involved after the beginning and gives little input throughout the process. Potential for late requests to derail the project or for customers to be feel unhappy being left out	Update the customer more during development, letting them know about the progress and such; consider customer's input during the process	Changing specs can be frustrating and make it difficult for the team to make progress

6.) Discussing incremental approaches in contrast to the model discussed in #4 & #5.

a) Name at least one instance "name" of an incremental Software Engineering approach (earn an extra point if you can name two others).

AGILE, SCRUM, RAPID, LEAN

b) What gains we get from these incremental approaches that counters some of the problems of the BDUF approaches?

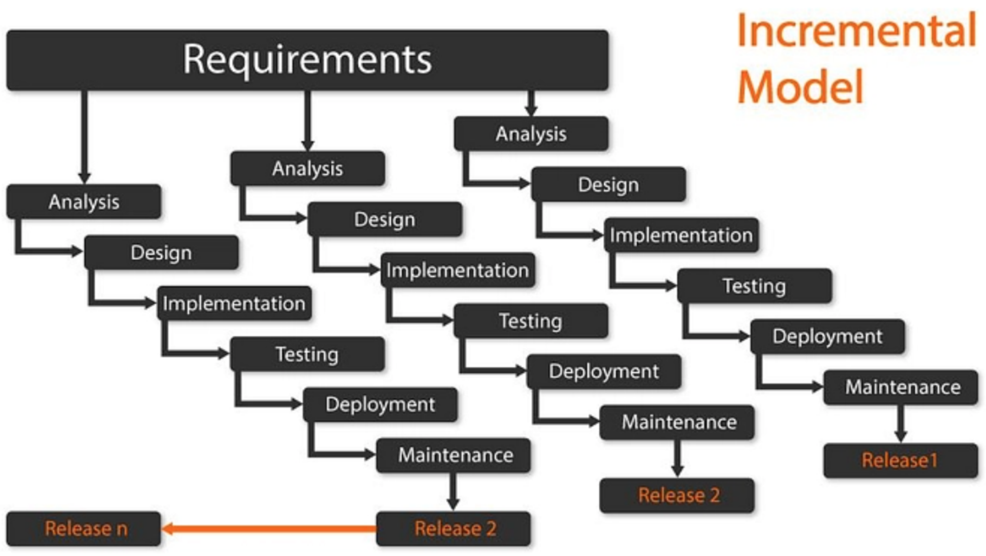
It is impossible to know everything up front, so being incremental allows programmers to face challenges as they come instead of guessing and being wrong later down the line. Another plus is that we can see app development sooner, which makes programmers more motivated and makes execs happy as they can see something being built rather than just planned.

c) What problems exist in the incremental approach that we should be aware of?

- Requires constant (and good) team communication

- Sometimes issues get pushed over to the next iteration, which are pushed to the next iteration, and so on… (not solving it at all until it's too late)
- Customers assume the software is ready to ship even when it's not

d) Draw a diagram to explain incremental approaches.



7)
a) The Google team study and the HRT methodology emphasis a number of dynamics and characteristics that lead to high performing teams. Describe these characteristics.

1. Psychological Safety
   a. Feeling safe to make mistakes and to be heard.
2. Dependability
   a. Team members trust others and can rely on them in crunch times, as well as get things done
3. Structure and Clarity
   a. Team members understand their roles and their responsibilities, as well as the overall structure of the project/team
4. Meaning of Work
   a. Team members find the work personally important
5. Impact of Work
   a. Team members believe what they are doing matters in relation to others, which will make them more invested in the success of the product

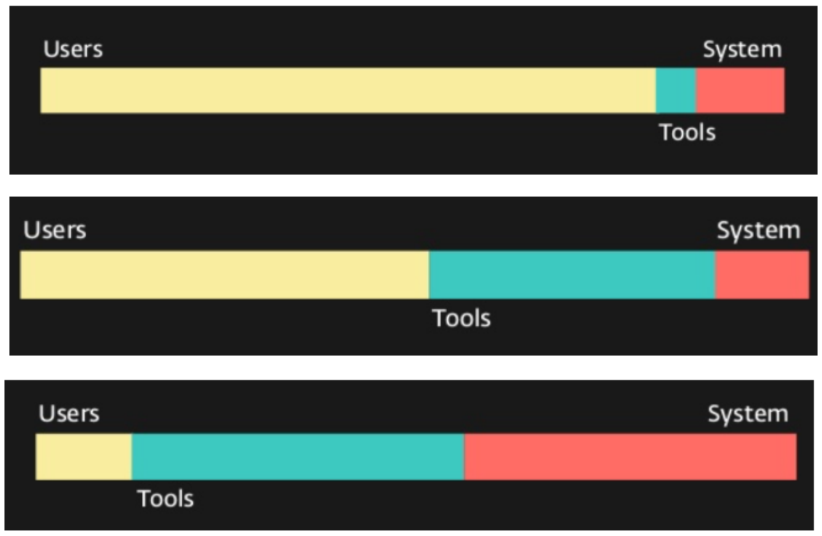
b) Google's research describes one characteristic as overwhelming all others, explain what it is and why you think that leads to good team outcomes.

Trust/Psychological safety - Feeling safe to make mistakes and to be heard. It allows team members to learn from mistakes, and be open about them so that they can be properly addressed. This will lead to a happier work environment, and allow problems to be seen earlier than if a programmer felt ashamed of a mistake and tried to hide it so they won't be blamed.

c) Describe the reverse of 7b. What aspects of individuals on teams lead to toxic outcomes. In other words what don't you want for your software engineering team?

Hostile environment - for example, programmers with high egos can make others feel worry or shame about their work which can lead to insecurity in the team setting. This can take a toll on the psychological safety of the group. Further, another toxic aspect of a teammate is talking over others, which makes them feel unheard.

8) The professor has expressed in numerous manners in class that there is an inherent tension between DX (Developer Experience) and UX (user experience). Provide an example of how an improved DX might lead to poor UX or the reverse if you like (improved UX and worse DX).
An improved DX means that more time is being spent to create developer tools and a better developer experience, and less time is being spent improving the user experience. Even though development may be faster in the long term, for short lived applications more time should be spent improving UX.



9) Explain how the statements "You can't be the tester" and "You must be the tester" are both simultaneously true for software engineers.

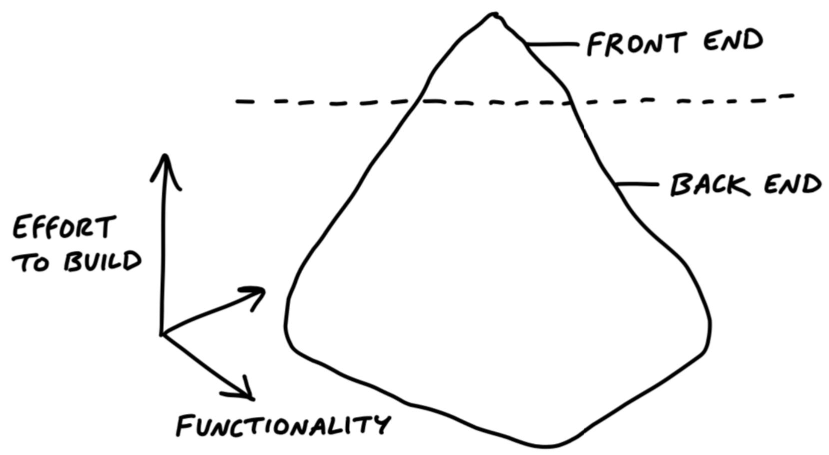
Can't be the tester, because you know the ins and outs, you know all the secrets and bugs.

You must be the tester because you need to check obvious fixes, you need to avoid obvious bugs that you are creating, ex. a writer has an editor, but the writer is still in charge of giving passable rough drafts.

10.) What does the "iceberg model" mean? Drawing a diagram is allowed. What does this idea suggest you should do in your software development process? Are there cases were the advice you just gave is not a good idea?

Source: Page 125 of Shape Up PDF (lecture slides channel in slack).

The iceberg model: In comparing working software with an iceberg, what we see (the frontend) is just the tip of the iceberg out of the water. The majority of the iceberg (the backend, docs, tests, ci/cd pipeline, etc.. in this scenario) is under water out of sight. Some suggestions for the software development process, it can be helpful to factor out the UI as a separate scope of work if possible, it can also be helpful to separate a back-end that is complex into individual concerns and tackle them that way. This advice wouldn't be a great idea if the UI was interdependent with the back-end complexity. Ultimately, the goal is to define features and things that can be finished and integrated in separate stages instead of waiting until the end and hoping it all works.



11.) Suppose that to beat the other teams your team has decided to add numerous exciting features to the application you're building. Explain, in software engineering terms, how feature additions could cause your project great risks or result in potentially unacceptable trade-offs.

- More code = higher complexity = more bugs
    - When features are blindly added without careful consideration, it can greatly increase the complexity of the software with little benefit. With increased complexity it becomes harder to debug the software.
- Becomes hard to maintain
    - Because of the complexity, it becomes difficult to maintain the software in the

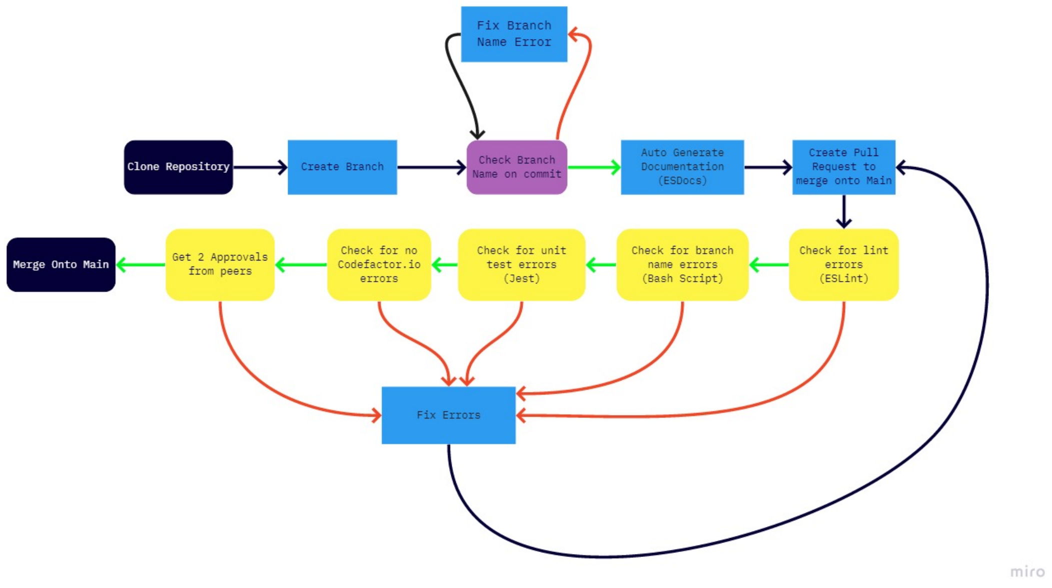
future

- Because we focus on new features, old bugs and issues are pushed off and ignored, which can come back later to haunt us.

Despite all the worries you presented in the previous questions many products still add as many features as possible. Provide some explanation(s) of why they might do that.

- Software engineering teams might add new features based on users' feature requests.
- Adding new features makes the product feel new to the users
- To please execs and managers. If they don't have a technical background they might not appreciate smaller performance improvements, but instead value new and possibly code-base breaking features.
- More money is brought in with new features

12.) The "conveyor belt" idea of a CI/CD system has been a big emphasis of this course. Draw your team's conveyor belt. Annotate each step to indicate the purpose or value of the individual step (in other words what benefit does the step provide?)


Fix Branch Name Error
Clone Repository
Create Branch
Check Branch Name on commit
Auto Generate Documentation (ESDocs)
Create Pull Request to merge onto Main
Merge Onto Main
Get 2 Approvals from peers
Check for no Codefactor.io errors
Check for unit test errors (Jest)
Check for branch name errors (Bash Script)
Check for lint errors (ESLint)
Fix Errors
miro

Our conveyor belt is the yellow steps in the image above. If at any of the following stages the pipeline fails, it stops the PR from being reviewed by others and is sent back to the pusher to fix.

1. The **linter** ensures that there is a standard code style and convention, to make it seem like the project was coded by a single person
2. The **branch name** bash script ensures that the branch naming is following the convention we set
3. The **Unit Tests** that we created are now tested at this next stage
4. **Codefactor.io** is used then for a sanity check and to ensure that there are no other glaring errors
5. Now the branch is **reviewed by two team members**, and if approved, the branch is

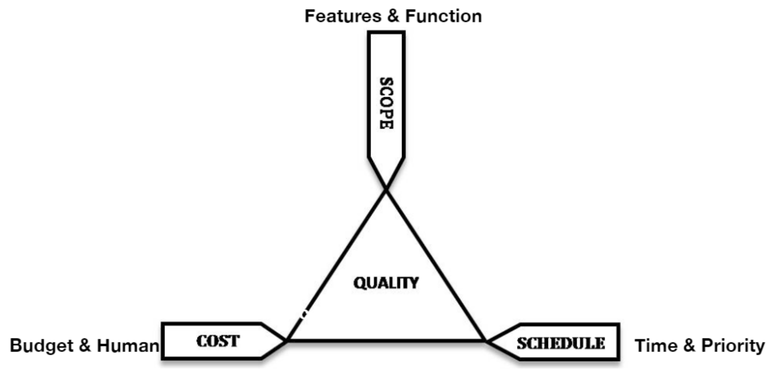
merged with main

13) What is having the process from #12 being ironed out being stressed so forcefully instead of letting us write code? What downsides can you come up with if we got down to building apps instead? What would be better about skipping the "belt" set-up effort?
This conveyor belt gives us an easy, replicable process that anyone new to the team can use to participate effectively. Without this ladder, it might be really challenging for new people to understand our CI/CD pipeline and make code base changes. However, for something like a quick, single person piece of software, like a script or small app for personal use, this could be major overkill. I think for any project at scale, especially when new people are being constantly onboarded like in a corporate environment, it's probably worth it.

14) Due to some recently uncovered "glitches," your boss Shannon Lumbergh has asked you to write code for a new internal payroll system at your company Initech. The intended budget for the project is $50,000. Discuss the impact of the fixed budget constraint on your project. Mention what will be necessary to ensure software success, given this constraint.

- With a fixed budget, we now have a two variable equation with scope and time. We have to come up with a budget and scope that allows us to stay within budget X + Y < $50k. Proper planning is required to ensure software success within these bounds.
- With a fixed budget, not a lot of manpower can be hired
- Only small amount of features can be implemented
- Little to no additional feature can be added during development (which will cost more money and time) or little maintenance provided post-launch
- Smaller and fixed budget will take a long time to develop (lower priority/ incentive)

15.) Draw the iron triangle of project management and briefly explain what it means.



Scope / functionality: How many features are in the program
Cost: The amount of money / manpower allocated for the project, also limits the quality and quantity of functions implemented, or even some sort of pipeline is simplified (like testing)
Schedule: The amount of time the team has to implement the functions, limited time means priorities have to be made.

- It is impossible to maximize all three factors as that means the program will not get

released at all
    - No deadline, not important >>> no incentive to develop
    - Too many functions, really long time / impossible to implement
    - Money and manpower isn't infinite
- Cost and schedule defines the limit of the scope of the project
- Similarly scope and cost allow developers to have a good estimate on the development time
- Scope and time also makes teams easier to calculate the financial cost.

16.) Explain how a coding standard could be a problem for a time constrained project.
Explain how non-adherence to a coding standard catches up with our code bases.
Give at least two reasons why you think people don't follow coding standards rigorously.

A coding standard can be a problem for a time constrained project because it takes time for SEs to acclimate to the standard, which takes time away from outward/visible project progress. However, non-adherence to a standard makes code readability much more difficult, which will eventually slow down the team and refactoring code to the standard is much more time consuming than learning and using the standard in the first place. This results in an accumulation of tech debt.

1. People are comfortable with the style of programming they use and do not want to change
2. It *appears that it takes time away from feature progress to adhere to a coding standard.

17) The professor believes the distribution method, medium, economic environment, and social environment have all influenced software development practices greatly. Provide at least 3 examples of how the industry or medium has affected our development practices.

1. SE Leaders such as Bezos and Musk have influenced what processes other SE companies use (top down, bottom up, a mix of both, etc)
2. The medium of SE allows programmers to move fast and break things with minimal repercussions (one main difference between programming and other engineering fields).
3. Adding new features is usually prioritized over fixing and optimizing previous features due to the consumer demand for new features (and the money that comes with it). Silicon valley environment

18) What is test driven development (TDD)?

TDD is when you make the tests first and then use the tests as pseudo-specifications to model your code and develop your application.

x Why is TDD a good idea?

TDD is a good idea when one has a clear vision of the application, and can result in less bugs and errors due to trying to pass test cases.

Why is TDD a bad idea?
One can write easy tests, which short changes the process of TDD. Also TDD is not optimal when one is trying to build a fast and short-lived application.

Why wouldn't people use TDD?

It's boring, and there is a lot of up front work that prevents programmers from getting into programming right away.

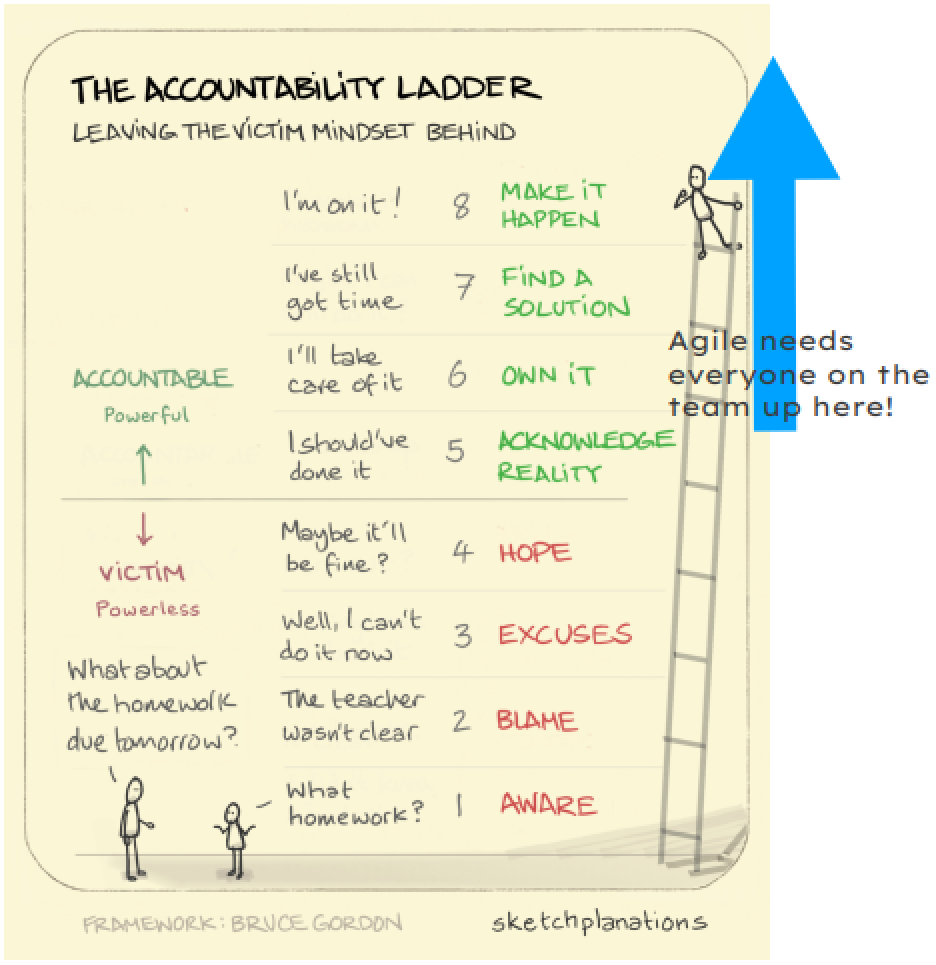
19) The professor seems to think that the hardest part of developing software is not at all the code, but the people involved, complete with all their irrationalities and foibles. Generally, we can expect to encounter three groups when building software: end-users, stakeholders (business owners / product owners), and fellow programmers. Write down what each group is likely to care about most and make a statement or two about how/why you should interact with them.

a) End users
- Want their app to work properly and responsively. The app does what they want. Doesn't really matter how it's done.
- Conversations on the app should be high level or surround things they interact with. Maybe describe how it went wrong when troubleshooting.

b) Stakeholders
- The project is made on time and on budget.
- Mostly discuss about the longevity of the app, trade off between quality and cost. Can talk about some technical obstacles about the app, but not at a super low level.

c) Fellow programmers
- Is it possible to implement some functions in time?
- How should one function of the app be implemented, where the programmers core -ilites are factored in.
- Discuss about individual functions, or overall program matches with the core -ilites with the team. Also about the possibility of adding new features and difficulties of adding/maintaining it.
- The discussion would most likely be at a lower level and also may apply agile/engineering principles.

20) An analogy about a ladder was made in the class and drawn on the board, what did the ladder describe (what's at the top of the ladder and what is at the bottom)? What was the point of the ladder, in other words what was it getting you to think about?



~~At the top of the ladder are concise and well thought out ideas, and at the bottom of the ladder are generalizations and unknowns. As developers we want to climb from the bottom of the ladder to the top before we start programming so that we can have a clear destination for the program. The ladder was to get us to think about narrowing our vision for things like features.~~

21) The Professor at various times warns about various different decisions made to embrace technologies like React, Angular, Ruby on Rails, Web3, and so on. In previous iterations the character Peter might encourage devs to embrace the new "hotness" as purposeful way to have you experience the concern, this time we cannot so give the Prof the benefit of the doubt that he isn't grouchy (even when he might be) try to figure out what the actual warning he is giving here.

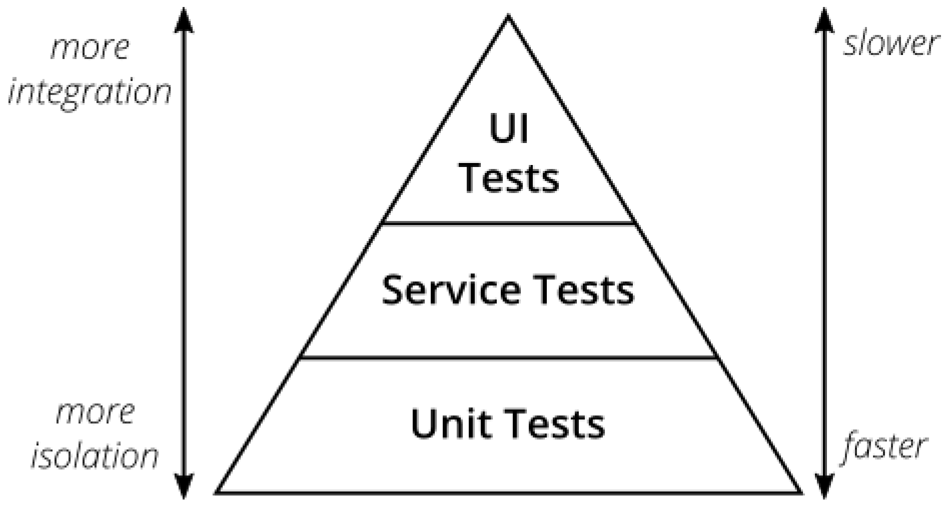
- Just because something is new and popular doesn't make it the right choice, be sure to consider the right tools for the job and justify your decisions.
- It's ok if you want to try a new technology for experience, but pivoting to it is not safe or

careful engineering.
- Further, new tech hasn't been tested as thoroughly as older tech, so there could be hidden mines ready to explode in the new tech.

22) Why should you put slower steps farther down your continuous integration pipeline (conveyor belt)?

- Having faster steps at the front allows the pipeline to break sooner if there are problems with those fast steps, then moves on to slower ones. This avoids bottlenecks and speeds up CI/CD

23) Draw the testing pyramid and explain each type of test. Make sure your explanation suggests what the type of test addresses and any pros/cons that may exist.



Unit tests - making sure that a certain unit (most likely a single function, or an object in OOP) in the codebase works as intended.
Pros:
- Tests internal structure of the software
- Makes sure smaller parts of the software are correct before joining them

Cons:
- Even if the unit tests pass, it doesn't entail a good user experience
- Individual features passing their tests may not ensure that emergent features work as well

Service tests - testing the general functionality of the software as a whole.
Pros:
- The test help the developer find bugs where the units are combined

- Increase test coverage

Cons:

- It would be harder to tell where the bug is base on the result of service tests
- Consume more time than unit testing (resource intensive)

UI Tests / E2E Testing - Testing the user interface of the application from the ground up, or sometimes referred to as E2E testing where we literally test the whole application from beginning to end to ensure the application flow behaves as expected.

Pros to E2E:

- Expand test coverage
- Ensure the correctness of the application
- Reduce time to market
- Cheaper cost

Cons to E2E"

- Slow execution, can take a very long time to test the whole thing.
- Inconsistent testing, meaning some tests may require maintenance and troubleshooting more often than others.
- A lot harder to debug and find where the point of failure is (can either be the function or the test itself)

24) Code Reviews can be quite difficult to perform. Explain why you should do them. Next explain what the biggest dangers badly run code reviews may lead to. Finally explain at least three tips on performing good code reviews.
More eyes and minds on code mean more chances for bugs to be spotted early and often.
Discussing code allows teams to develop a shared understanding of what "good" is and adhere to it.
Discussing and improving code allows novice devs learn patterns and approaches to coding from senior devs that educate during review.

Code review can be a way to reinforce coding conventions that the team agrees upon, if the code is difficult to understand for the rest of the team, then it could lead to poor developer experience.

Bad code reviewers are more like critics complete with lots of complaints, few solutions and often delivered bluntly. → destroy morale especially on a team with poor psych safety or toxic members. Also bad reviews can allow bad code to pass through.

Tips:

- Encouraging environments where people are not their code, and where people understand that it is not about "your code" but the shared code.
    - Suggesting improvements instead of lots of complaints
- Code should be reviewed
    - When it was created for freshness (esp. others)

    - Far after it was created for distance (esp. you)
- Write clear PR descriptions
-

25) The Professor has suggested that software is more than just the code itself but includes potentially many other artifacts (document, diagrams, etc.). Explain some of the artifacts that should exist. Next explain what happens when these artifacts are not available or kept up to date and what might happen to the software or team without them.

- pipeline documents and diagrams
    - showing how the pipeline works (branch, pull request, testing, merging)
    - (if not available/up to date) the team cannot understand what the pipeline does (ex. what tests the pipeline does)
- ADRs
    - including context, options, chosen option with reason, …
    - (if not available) the team cannot understand the reasoning behind the decision - We must capture the "why's" that drive the "hows"
    - (if not available) the decisions are not clear enough for the team if they are only recorded in meeting notes - members need to spend more time looking for decisions in meeting notes
    - (if not up-to-date) the team might make wrong implementations according to the not up-to-date documentation
- planning documents:
    - brainstorming documents, user stories, low-fi & hi-fi wireframes
    - (if not available) no guidance for the team to implement features/UI. For example, team members might implement different UIs if there is no wireframe document
- UML system diagrams:
    - UML (Unified Modeling Language) diagrams can be used to communicate a system design and user interactions
    - (if not available) no guidance for the team to connect UI and features together

26) There are 4 Agile Manifesto values write them here.

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

27) Enumerate the 12 Principles of the Agile Manifesto.

1. Customer Satisfaction by rapid delivery of software
2. Welcoming changing requirements even late in development
3. Working software is delivered frequently (weeks not months)
4. Working software is the principal measure of progress
5. Sustainable development, able to maintain a constant pace
6. Close, daily cooperation between business people and developers

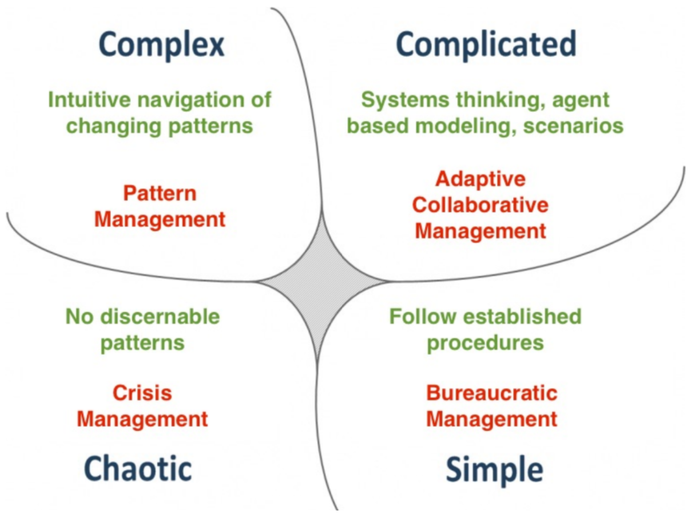
7. Face-to-face conversation is the best form of communication (co-location)
8. Projects are built around motivated individuals, who should be trusted
9. Continuous attention to technical excellence and good design
10. Simplicity - the art of maximizing the amount of work not done - is essential
11. Self-organizing teams
12. Regular adaptation to changing circumstances

28) What is Domain Driven Design and why is it important?
Domain Driven Design is a software development approach that focuses on the problem rather than the solution.
Development and design decisions are made with the domain in mind, oftentimes domain experts are consulted to improve and refine the application. This leads to software that better meets the needs of the domain.

29) Enumerate the four ideas of the Cynefin model beside the idea of something being Complex. The idea of some project being complex is unfortunately often self-inflicted. Explain why you think why introduced complexity might be taking place in modern SE.
1. Clear
2. Complicated
3. Chaotic
4. Confusion

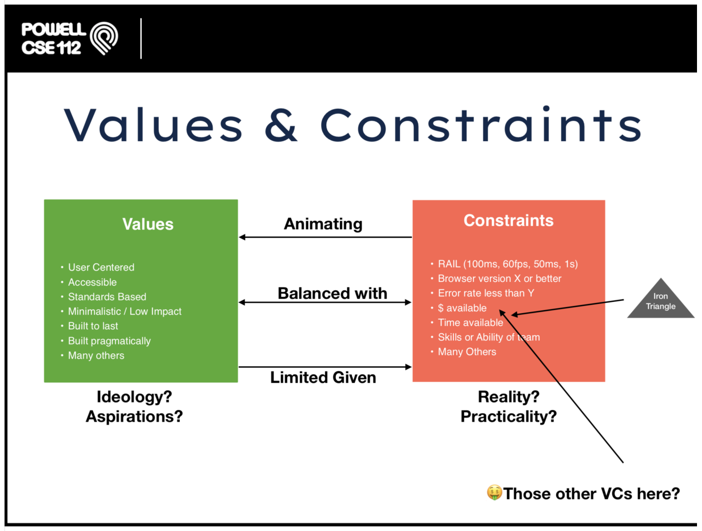
While essential complexity is necessary to give our application the capabilities it requires, accidental complexity can be detrimental to an application's success.

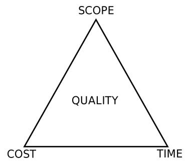
- Accidental complexity is often introduced as developers adopt new technologies, tools and deployment models, and add features.
- However, the developers are not always at fault, changes in management can create unnecessary complexity especially as the tension between developers and managers who want to adopt the latest agile spinoff escalates.

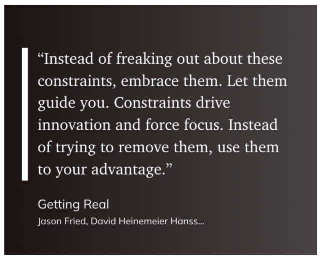

Complex
Intuitive navigation of changing patterns
Pattern Management
Complicated
Systems thinking, agent based modeling, scenarios
Adaptive Collaborative Management
No discernable patterns
Crisis Management
Chaotic
Follow established procedures
Bureaucratic Management
Simple

30) The Professor posited this idea of Software projects needing to be driven by the

idea of VC where VC is not Venture Capital. Explain this idea and discuss how it helps bring clarity to situations where choices might be in conflict.
VC: values and constraints

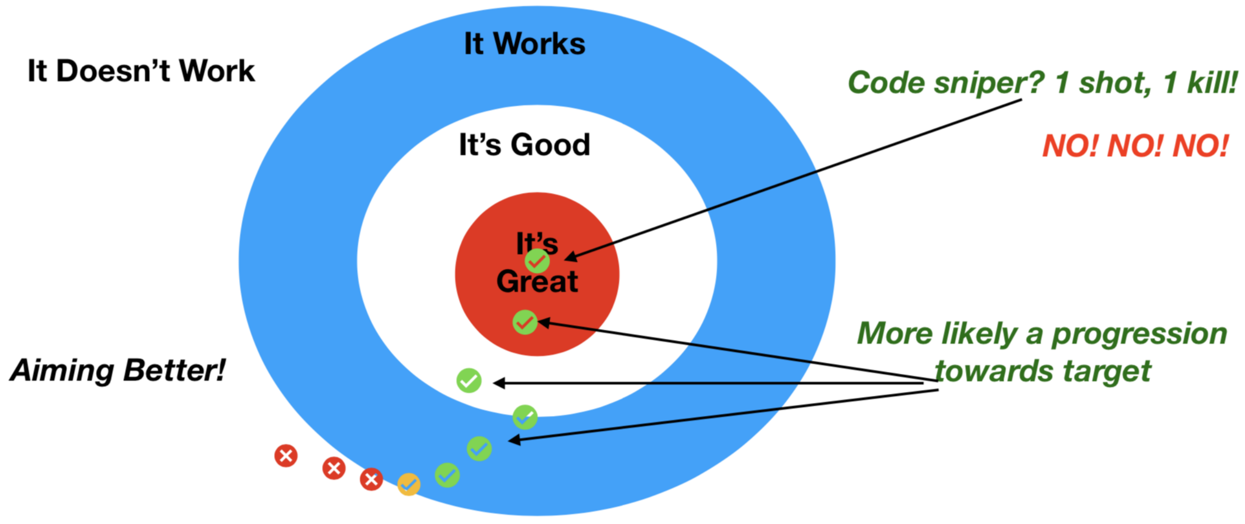





"Instead of freaking out about these constraints, embrace them. Let them guide you. Constraints drive innovation and force focus. Instead of trying to remove them, use them to your advantage."

Getting Real
Jason Fried, David Heinemeier Hanss...

Defining our project's values and using them in conjunction with constraints provides "guard rails" as we move iteratively towards our bullseye, helping us narrow down our choices in a reasonable manner. This is the premise of intentional engineering.


It Doesn't Work
It Works
It's Good
It's Great
Code sniper? 1 shot, 1 kill!
NO! NO! NO!
More likely a progression towards target
Aiming Better!

Part 2: The Wild Idea – Testing What Matters to You and Your Team
Your team has experienced many things in five weeks both in lecture, discussion and on your own. As a group come up with between 10 and 12 questions you want to make sure that each of the members of your team can answer about software engineering, your team, hard lessons, etc. that you think will serve you as software engineers. These questions will absolutely be on your team's midterm. However, to avoid you from just putting questions like "Is 3 an odd

number?" we must see the questions and approve them.
Write BOTH question and answer

Q. (Yizhou) What is the bus factor?

A. Bus factor is a numeric value that represents the number of people on the team/project that if hit by a bus would doom the project.

Q. (Yizhou What kind of bus factor would cause a risk on a team/project?

A. A low bus factor would cause a risk - unlike sports and symphonies, software engineering teams/projects do not have back-ups to replace the quitting member.

Q. (Yizhou) How to manage the risk?

A. Working with others can deal with this risk - learning from each other can increase the number of people understanding certain parts of the project, resulting in a higher bus factor.

Q. (Edgar) What are the four general parts of building and maintaining psychological safety in a group.

A. Modeling a team culture, allowing failure, avoiding blaming and empathizing with your teammates should be paramount when building psychological safety.

Q. What is tech debt? (Nick)

A. Tech debt is anything or any decision that leads to more work and stress later in a project or team's lifetime. Some examples are not following code standards, non-optimal tools, and focusing on new features rather than fixing problems.

Q. (Matt) Please describe Dietzler's Law.

A. 80% of what the user wants is fast and easy. The next 10% is possible but difficult. The last 10% is impossible. The user wants 100% of what they want.

Q. (Pablo) Is tech debt always a bad thing?

A. Not always. For very short term projects designed to be used and disposed, it doesn't really matter if maintenance is more difficult, as long as it completes its task. Also, if there's a hard deadline, it's possible that taking on that technical debt would be the only way to deliver on time. In that case, it's not ideal, but it's better then not shipping, or shipping something incomplete/non-functional.

Q. (Zhuoran) Describe top-down and bottom-up design and briefly explain why do we need both of them as programmers.

A. Top-down design focuses on breaking a core idea into smaller components and solving each one separately before piecing them together. Bottom-up design focuses on taking smaller components and piecing them into a larger system. We generally need both of them since we want both high-level planning and the ability to work things out individually as we go through the process of designing software.

Q. (Allen) According to the professor, should team/organization culture be defined or emergent?

A. Team/organization culture can be slightly initially defined, but ultimately it emerges and evolves. So whatever it becomes embrace your culture as long as it improves team function.

Q. Why are standups important? (Nick)

A. Helps get the team on the same page, people can more easily ask for help on blockers, and helps gel the team through a constant form of communication.