

Room: CS312S25

Peer Instruction Q & A

A
B
C
D
E

Grasp Gauge

GOOD UNSURE LOST

How are you feeling about the class material right now?

Ask a question

☒ Ask anonymously

Questions will appear once approved by your instructor.

👍 👎 😊 😐 😞 🙄 💡

Using the “question” feature of go/cs312-inclass, submit a comment on the following:

Think about an app or website that you have used in the past year (that is not Banner 😞):

- What is a feature that really impressed you, and why?
- *Or*, what was a challenge working with it, and why?

I suspect many of our complaints point to specific design/technical issues. But the root cause of those problems is likely not technical, but the outcome of problematic team structures and processes...

Software development involves programming, but it is not just programming. And our experiences bear that out. But how is it different?

Luiz André Barroso (Google VP of Engineering) on SW development: “It’s a team sport...”



In his 2020 Eckert-Mauchly Award Lecture in a response to a question about what educators should focus on, Luiz said the following:

“[...] if I look at the the folks that arrived at Google I think there are two things that I like to emphasize. One of them: being a good programmer is really important [...] The other thing that I think is important to recognize is a non-technical issue is that unlike you know when I entered the field the kinds of problems we solve today are at the degree of complexity and scale that a single superstar is much less likely to really move the needle [...]”.

“[...] we need to educate students to think of themselves as a team, and evaluate their performance not as much by their individual prowess, but in what ways their addition will make the team better. And I see that every day at Google, where the difference between a very good programmer and the “ninja” programmer in a very complex environment, where tens or hundreds of people need to work on the same project - the difference between those two is not their ability to program or their technical prowess, primarily, the difference between those two is how they work with their teams.[...]”

https://www.youtube.com/watch?v=Lv_eZX99IUU&t=2257s

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

How is SW engineering/development different from “programming”?

[SW engineering is] “The **multi-person** development of **multi-version** programs”
– Brian Randell

Parnas, D.L. (2011). Software Engineering: Multi-person Development of Multi-version Programs. In: Jones, C.B., Lloyd, J.L. (eds) Dependable and Historic Computing. Lecture Notes in Computer Science, vol 6875.

David Parnas (one of the early pioneers of this field) identified the following quote as the answer to this question... The “multi-person” aspect echoes the Barroso’s point. But what is “multi-version”?

Parnas further writes: “The focus of computer education has long been on algorithm design. When people teach programming, it is assumed that a program will be written by a single person. Moreover, they often assume that the ultimate user of the program will be the programmer. No time was spent on the possible need for several versions of the program and the consequent need to “design for change”. It was assumed that since software was not hardware, a program could not be hard to change.”

“multi-version” recognizes that programs are not static, they have multiple, perhaps many thousands of versions (some of which may coexist). And thus, we need to create programs with that change in mind.

He goes on: “Nothing is taught about the extra work needed if your program will be used by others or must be part of a multi-person project.” I suspect that sentence might capture your thoughts on about some (maybe all?) of your classes. It highlights one of the key aspect of this class that makes it different. Our goal is to specifically learn and practice techniques for working in a team to build “multi-version” software that can evolve with the needs of its users.

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

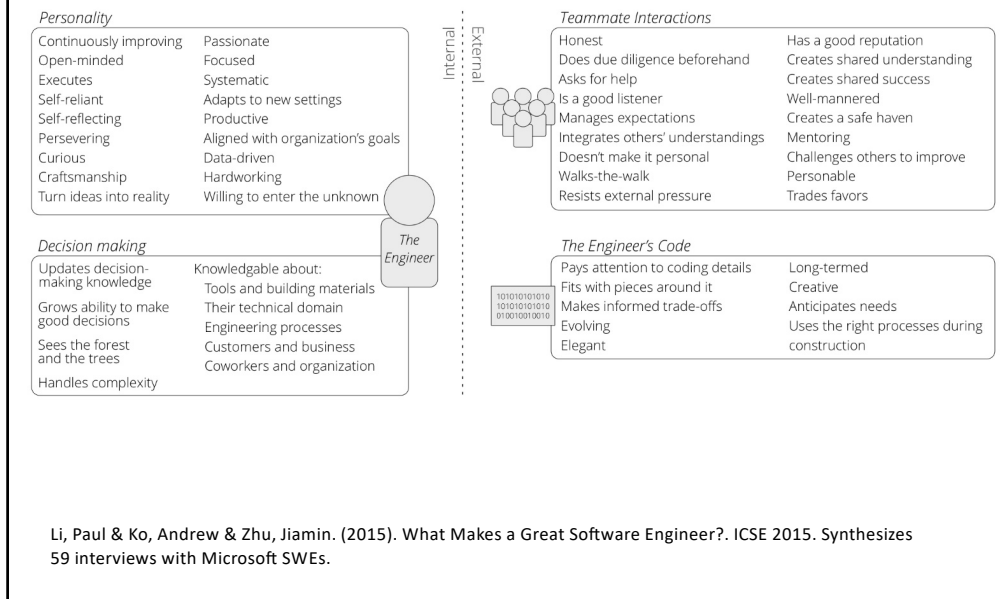
What are you hoping to get out this class?

CS312 learning goals:

1. Describe and employ modern methodologies for managing SW development
2. Use tools and services that support those processes, such as version control, GitHub, continuous integration, etc.
3. Describe and employ SW development principles, patterns and best-practices, such as test-driven development (TDD), DevOps, etc.
4. Describe, evaluate and employ technologies for full stack web development and single page web applications (SPAs)
5. Complete a large software development project as part of a team

Notice that most of these learning goals are about the process of SW development. Not specifically about web application development. Build a web application will be the specific domain in which we practice those skills. It is a means to an end, not the end itself.

What makes a great software engineer?



There is a lot on here... Our goal isn't to review these in detail. We will return this paper and this question many times. I want to highlight what is on here, things like how to be a great teammate, and what is not on here... specific technologies, or the ability to pound energy drinks and code all night. Many of those stereotypical attributes of "10X" programmers, "rockstars", "ninjas", or whatever you may call them are not actually that important. These are the attributes I want us to develop as part of this class.

Top 10 ways to have a bad time

10. Jump right into coding, that's where learning happens
9. Don't do the reading
8. Work alone, don't help your classmates
7. Resist learning new tools/techniques
6. Ignore advice (instructors, peers, CS169)
5. Focus on passing tests vs. learning
4. Ignore the honor code
3. Multitask in class, while coding, etc.
2. AI can handle that for me
1. "Process" just wastes valuable coding time

Learning new tools is lifetime project.

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

Context: Evolving ecosystem

Shrink wrapped	⇒	Software-as-a-Service
Monolithic	⇒	Services and serverless
On-premise	⇒	Cloud

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

Shrink wrapped software (SWS) ⇒ Software-as-a-Service (SaaS)

SWS

Client-specific binaries that must work in many HW/SW environments

- + Rich user experience
- Hard to maintain, with extensive compatibility testing required

SaaS

Online client-server model

- + One copy of SW, one HW environment (controlled by developers)
- + Easy to release updates
- + Easier to enable user collaboration
- Limited by online latency, capabilities of browser

What about mobile native applications?

SaaS tools are now so integral to how we work we don't really think about it (till we have to deal with Internet service in rural VT...), but it wasn't/isn't always this way.

What about mobile native applications? A counter example for this trend?

- + Use HW features unavailable in HTML5
- + *May* be faster...or not (many just HTML5 apps in native “container”)
- + Your brand is on user's home screen (though can get this in other ways)
- Harder to maintain with multiple platforms to support
- Upgrades now once again user's problem (although app stores mitigate some of that...)

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

Achieving modularity in SaaS?

Modularity encapsulates complexity into (relatively) independent units

One approach is Service-Oriented Architecture (SOA)

Most famously? expressed in 2002 Bezos mandate to Amazon*

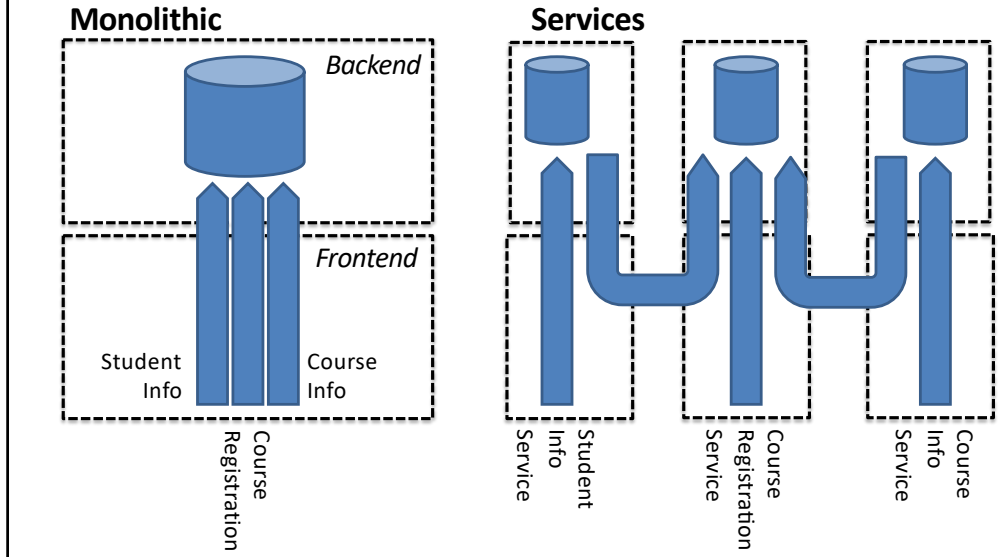
- All teams will henceforth expose their data and functionality through service interfaces.
- Teams must communicate with each other through these interfaces.
- There will be no other form of interprocess communication allowed...
- All service interfaces, without exception, must be designed from the ground up to be externalizable...
- Anyone who doesn't do this will be fired.

*Steve Yegge 2011 Blog Post

SOA: An architecture that focuses on discrete services instead of a “monolithic” design (where components are part of the same application and can often directly access all of the application’s data).

<https://courses.cs.washington.edu/courses/cse452/23wi/papers/yegge-platform-rant.html>

Monolithic \Rightarrow Multiple services?



What do we mean by services and service-oriented architecture (SOA) (also sometimes called microservices). As described in Bezos mandate, the system is decomposed into independent services. Each can be developed/deployed independently. In the ideal case the complexity of the system only "scales" with the most complex modules (not the system as a whole). In practice that ideal is not always achievable. And as you might image, this approach introduces its own challenges,. So why this mandate? In part it was to turn Amazon into a platform, not just an "application", and to do so they were going to "eat their own dogfood", i.e., use their internal tools like a customer.

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

Which of the following is a *disadvantage* of services-oriented-architecture (SOA) compared to a monolithic design? SOA:

- A. May be harder to debug & tune
- B. Results in lower developer productivity
- C. Complexity is a poor match for small teams
- D. Is more expensive to deploy than monolithic, because more hardware is needed to handle the same workload

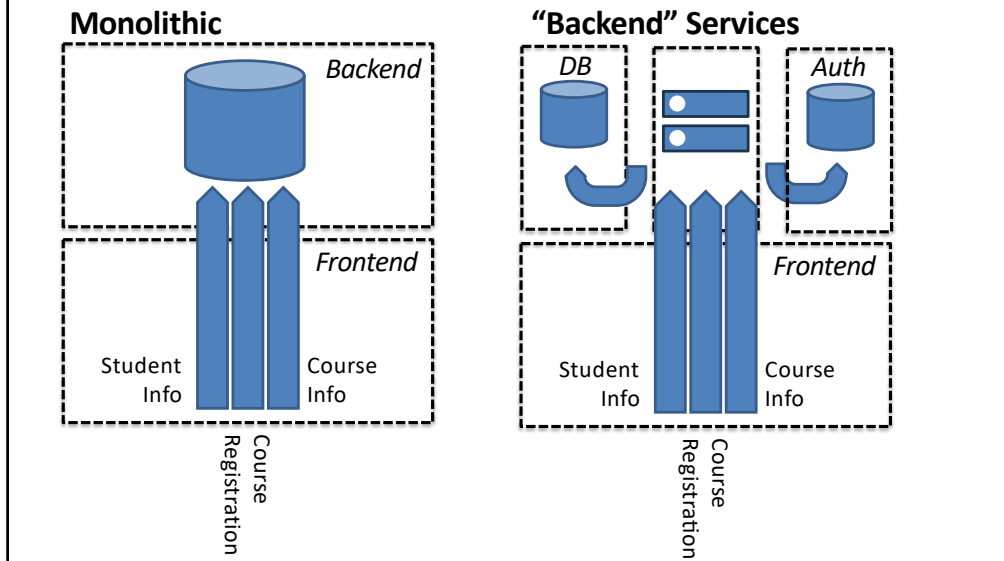
Answer: A.

Microservices can trade application complexity for system complexity. A microservice-based architecture can be more difficult to design and maintain at a system level, but each individual component of the system is greatly simplified. (<https://www.sitepen.com/blog/2017/02/20/microservices-and-spas/>)

Absolutes rarely work (moderation in all things). Services are not always appropriate and the trends towards SOA has in some cases swung back towards more monolithic applications. Why? Decomposing systems too much can introduce additional overhead and friction. And often that decomposition is not a function of separate functionality, but often separate parts of an organization (i.e., different teams). For example, part of Amazon Prime described how transforming their application back to a more monolithic approach substantially reduced costs: <https://www.primevideotech.com/video-streaming/scaling-up-the-prime-video-audio-video-monitoring-service-and-reducing-costs-by-90>

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

Monolithic \Rightarrow “Backend” services?



That trend back to monoliths, is sometimes only monolithic in a sense. A common approach today is not to decompose an application into multiple discrete services, but to build an application using multiple external services. That is use a separate “third party” service for authentication, for images, for real-time chat, etc. An extreme version of this are “serverless” architectures – a terrible name – in which the backend code you write is not stateful. Instead, it is interacting with multiple stateful services... but more on that in the future.

SaaS' 3 demands on infrastructure

1. **Communication:** Customers must be to interact with service
2. **Scalability:** Respond to fluctuations in demand or new services adding users rapidly
3. **Dependability:** Service & communication available 24x7

Cloud providers can offer all three on a pay-as-you-go basis (utility) at hard to match prices

Below a certain scale, it is hard to compete on price with experienced data center operators building warehouse-scale computers. Economies of scale and relentless optimization pushed down cost of largest datacenters many fold (estimates of 3-8X). Remember when thinking about cost, it is not just the cost to purchase the machine, but the cost to manage the space, power the machine, maintain it, etc. The actual hardware itself is only a part...

The barrier to entry is now very low (don't need to buy HW up front) and individual developers have access to same computing power as the big players. Infrastructure-as-a-service (IaaS), e.g., the "original" AWS, are increasingly becoming platform-as-a-services, PaaS. With a PaaS, the developer connects many "higher-level" services instead of provisioning the underlying servers, storage, etc.

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

Context: Accelerating change

- What makes software different than other engineering disciplines?
Change is easier (easier than for HW...)
- Easier frequently mistaken for easy
“Software Engineering” term arose from a 1968 NATO conference on the “Software Crisis”
- Current trends only increase expectations and the rate of change

The “Software Crisis” was a term to describe the “difficulties in producing high-quality software on time and within budget”. As computers became more powerful the complexity of problems that people could tackle also grew (but not “linearly”), exposing limitations in development processes. In his 1972 Turing Award lecture, Edsger Dijkstra’s (a participant in the 1968 conference) described the software crisis as:

The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.

https://en.wikipedia.org/wiki/Software_crisis

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

Plan & Document \Rightarrow Agile

“Plan-and-Document”:

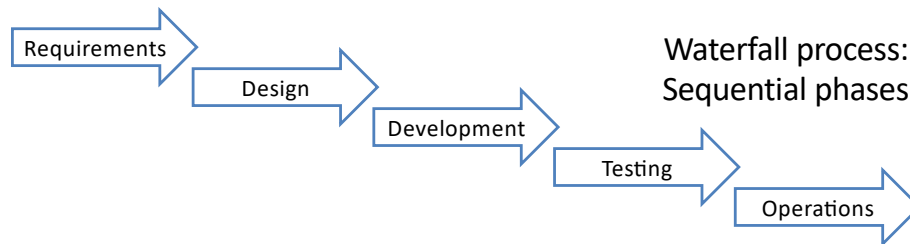
1. Before coding, the project manager makes plan
2. Write detailed documentation for all phases of the plan
3. Progress measured against the plan
4. Changes to project must be reflected first in changes to documentation and the plan

Implementations: Waterfall, Spiral, ...

How can be build software for this world...

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

Waterfall



Advantages:

- Errors are caught early (and more cheaply) before manifesting in next phase
- Extensive documentation is deliverable (can facilitate maintenance)

Waterfall: Sequential phases of project (like a cascading waterfall).

Spiral: "Spiralling" iterations of:

Determine objectives and constraints

Evaluate alternatives and identify and resolve risks

Develop and verify prototype

Plan next iterations

Spiral could be described as Waterfall with prototyping. Both involve a lot of planning and long phase changes (i.e., iterations can be long)

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

What is a major challenge faced by P&D processes such as Waterfall?

- A. Careful planning, then measuring progress against the plan
- B. Reacting to changes in a particular phase after that phase is done
- C. Using prototypes to get customer feedback

Answer: B

P&D assumes a phase is done once we move on. But we can imagine that during development, testing, deployment, etc. we observe we need to change the design. Revisiting the design process can be challenging at that point in these processes.

Change doesn't mean we made mistake. There are many reasons for change. We likely had incomplete information, or the needs of our users have evolved, or new technologies are available... Or probably all of those things.

In the software world change is not a problem to be fixed, but a fact to be coped with. That is, we want design processes with change in mind.

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

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.

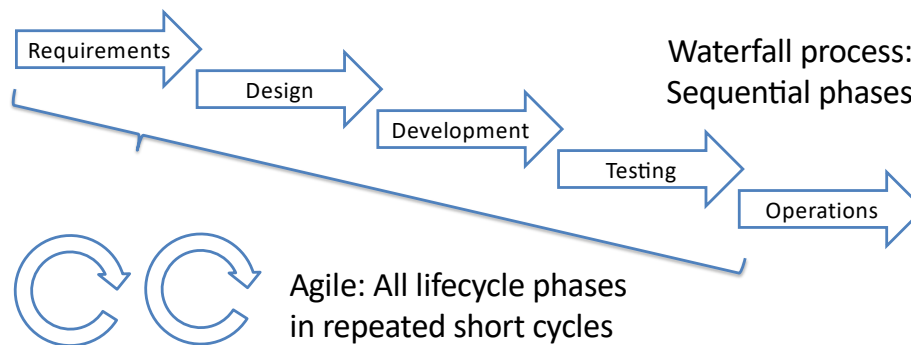
<http://agilemanifesto.org>

<http://agilemanifesto.org>

Plan & Document \Rightarrow Agile



Dilbert 11/26/17



Recall Waterfall features sequential phases (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 after each 1-2 week iteration).

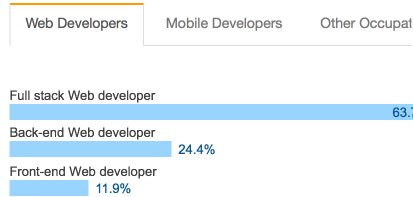
Note that when we talk about agile, we are talking as a project management “philosophy” (like P&D is a description of more than just Waterfall). Scrum, Extreme Programming (XP) are specific methodologies guided by the Agile philosophy.

In class we will implement one version of agile, in a way that is tailored to a class. Our approach is not the only one (or necessarily the best – a matter of opinion) or even appropriate for all applications/industries. But it will give us hands-on experience with these approaches to project management.

Agile is also a “brand” with consultants, etc. We are really aiming to be “lowercase a” agile, that is demonstrate agility, as opposed to the “capital A” Agile, the formal process that someone might want to sell you.

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

“Full-Stack”, “DevOps” and other buzzwords...



10,696 responses; select all that apply

[StackOverflow 2017 developer survey](#)

"A Full-Stack Web Developer is someone who is able to work on both the front-end and back-end portions of an application."[\[1\]](#)

DevOps? Cross-functional (no more silos) teams that:

- Apply “development” practices to operations, e.g., infrastructure as code
- Automate everything
- Integrate operations into developer role

The front-end is typically the portion of the application the user interacts with directly (and in our case runs in the browser), while the back-end is the portion of the application that provides resources (data) to the front-end, e.g., implements persistent datastores, business logic, and runs servers controlled by developer.

"Being a Full-Stack Developer doesn't mean that you have necessarily mastered everything required to work with the front-end or back-end, but it means that you are able to work on both sides and understand what is going on when building an application."

[1](<https://medium.com/coderbyte/a-guide-to-becoming-a-full-stack-developer-in-2017-5c3c08a1600c>)

“DevOps essentially extends the continuous development goals of the Agile movement to [continuous integration](#) and continuous delivery.”

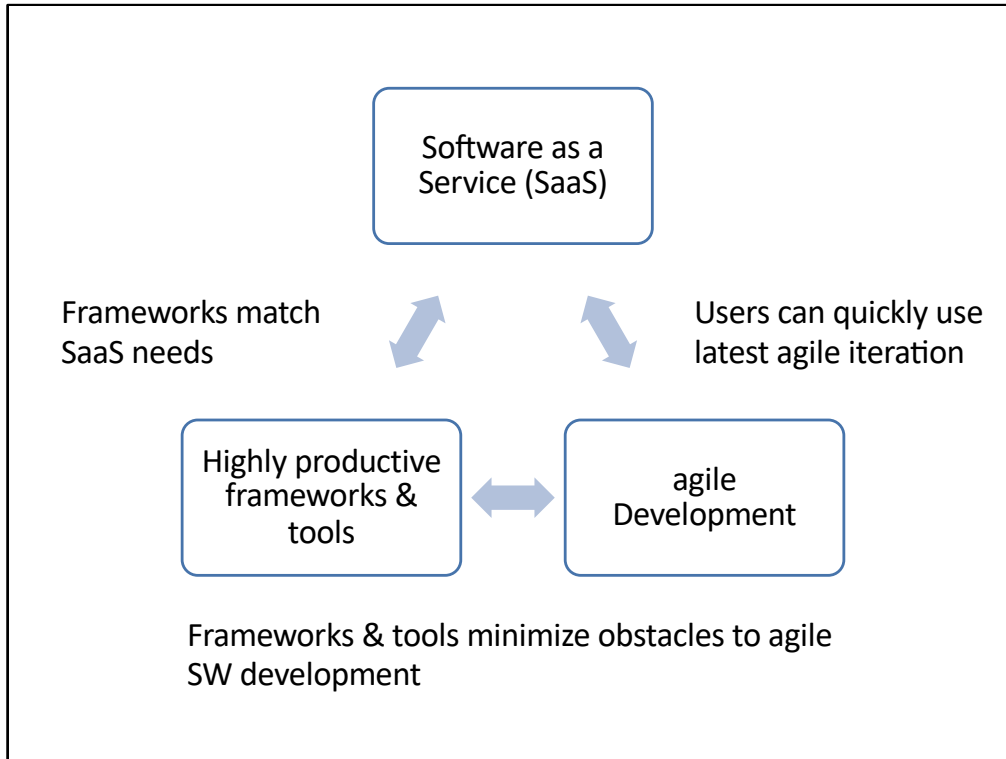
<https://www.ansible.com/blog/confessions-of-a-full-stack-devop>

<https://newrelic.com/devops/what-is-devops>

Summarizing our (the) landscape

- SW (can) evolve quickly to match user needs
- But doing so requires a development process that *embraces change*
- *agile* is a process that embraces change (as opposed to plan & document, etc.)
- SaaS is an ideal domain for agile processes
- Cloud gives everyone access to scalable HW and services for implementing SaaS
- Web applications are natural consumers of these services

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



Synergistic methods (deliver software as a service), tools (frameworks, etc.) and processes

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

Which aspect of the software lifecycle consumes the most resources?

- A. Design
- B. Development
- C. Testing/debugging
- D. Maintenance

Answer: D.

We will do a lot of "greenfield" development in class but that is not necessarily true of your future tasks. A lot of code has already been written, and SW maintenance, which isn't just fixing bugs, but also adding new features to existing SW, is ~60% of SW costs. Working with "legacy code" matters. Part of what we will learn is how to write code that is maintainable, and by virtue of working with large teams, how to work with code others have written. Whenever you are about to disregard legacy SW, remember that legacy code is successful code, otherwise it wouldn't still be around.

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

Beautiful code



earthcam.com

Beautiful code:

- Meets customer needs
- Easy to evolve

The “cruft” that makes enhancements expensive is the *technical debt* created by doing the easy thing, not the “Right Thing”

The only people that can see the Statue of Liberty’s hair are those that climb up the torch. Yet the artist(s) included hair anyway!

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

Tactical vs. strategic programming

- Tactical: The focus is getting something, anything, working
“You tell yourself that it’s OK to add a bit of complexity or introduce a small kludge or two, if that allows the current task to be completed more quickly. This is how systems become complicated.”
- Strategic: Working code is not enough
“It’s not acceptable to introduce unnecessary complexities in order to finish your current task faster, [...] Most of the code in any system is written by extending the existing code base, so your most important job as a developer is to facilitate those future extensions.”

Ousterhout, John K. . A Philosophy of Software Design

“All programming requires is a creative mind and the ability to organize your thoughts. If you can visualize a system, you can probably implement it in a computer program. This means that the greatest limitation in writing software is our ability to understand the systems we are creating.”

Complexity is our enemy. The tools and processes we will learn about are designed to help minimize and/or overcome complexity. But tools alone are not enough. We must design software to be simple or at least simpler. How can we do so? 1) Actively try at each moment to make our code simpler and more obvious, 2) encapsulate the complexity we can’t eliminate. We will see and do both this semester.

[click]

We have all done this (I am doubtful of anyone who says otherwise...) And the structure of most courses actually encourages “tactical programming”. Most programming assignments are “green field” (you start from scratch), there are often hard deadlines, and you rarely need to use the code you write in the future, so there is minimal cost to introducing complexity.

“Almost every software development organization has at least one developer who takes tactical programming to the extreme: a tactical tornado. The tactical tornado is a prolific programmer who pumps out code far faster than others but works in a

totally tactical fashion. When it comes to implementing a quick feature, nobody gets it done faster than the tactical tornado. In some organizations, management treats tactical tornadoes as heroes. However, tactical tornadoes leave behind a wake of destruction. They are rarely considered heroes by the engineers who must work with their code in the future. Typically, other engineers must clean up the messes left behind by the tactical tornado, which makes it appear that those engineers (who are the real heroes) are making slower progress than the tactical tornado.”

There are no absolutes – moderation in all things. We don’t to spend so much time coming up the “best” design that we don’t accomplish anything. We should think about this from an investment perspective. We are continually making small investment (say 10-20% of our time) make the system better. This may be proactive – spending a little more time up front to improve our design, or reactive – fixing a design problem instead of working around it.

Ousterhout, John K. . A Philosophy of Software Design, 2nd Edition

Signs complexity is winning

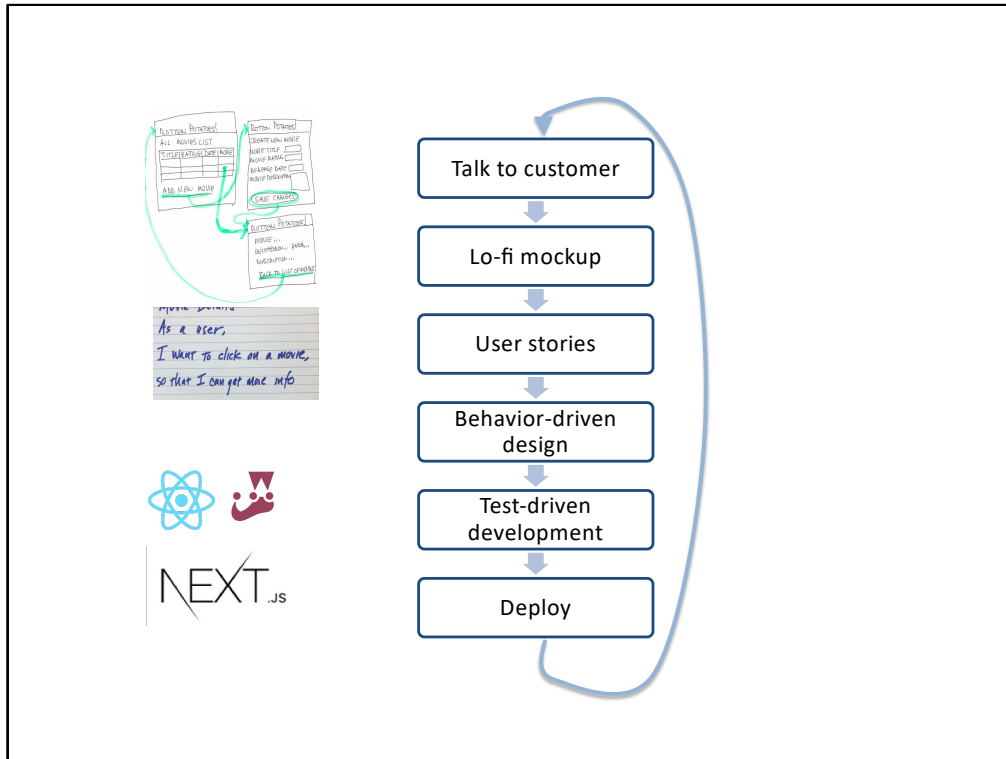
1. Change is hard: Seemingly simple changes require modifying code in many places
2. High cognitive load: Lots of work to figure out how to complete your task correctly
3. Unknown unknowns: Not clear what you need to change to complete your task

Ousterhout, John K. . A Philosophy of Software Design,

“Of the three manifestations of complexity, unknown unknowns are the worst. An unknown unknown means that there is something you need to know, but there is no way for you to find out what it is, or even whether there is an issue. You won’t find out about it until bugs appear after you make a change. Change amplification (change is hard) is annoying, but as long as it is clear which code needs to be modified, the system will work once the change has been completed. Similarly, a high cognitive load will increase the cost of a change, but if it is clear which information to read, the change is still likely to be correct. With unknown unknowns, it is unclear what to do or whether a proposed solution will even work.”









More generally we will see this warning signs in slightly different forms – different people use different descriptions, but there are crosscutting ideas about the difficulty of making changes/improvements.

Ousterhout, John K. . A Philosophy of Software Design, 2nd



The workflow we will implement has some key “technical” steps, but also some “non-technical” steps that are equally important for ensuring that we develop the right software, i.e., software that solves the user’s actual problem, not just software that works correctly. Unlike a class assignment, where there is (often) a very clear task and problem to solve, in the project (and any real-world situation) you will need to figure out the problem too! That is figure out what you should be building!

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

Technology	Tools	Process
HTML JavaScript CSS Frameworks Databases    	VCS Testing frameworks Linter Continuous Integration (CI) PaaS & IaaS    	Agile (Scrum) Test-driven development (TDD) DevOps Design patterns

Here are some of the tools and technologies we will use. These specific tools and technologies are a means to an end. I care more that you learn the “why” for these tools than the “how” of any particular technology or framework.

This class has a lot of moving parts. And I have made many choices for tools, techniques, methodologies. In some cases, it is a choice among many similar alternatives where there is no "right" answer. I don't want you to get bogged down in questions of whether any particular tool or technology is the best in a technical or other sense (we often are optimizing for the best class experience, not the best tool). You may be more familiar with the alternatives. As we will talk about it in a moment, I ask you to "do the class" with the class tools. Doing so will make your teams more efficient and be opportunity for you to learn new tools/techniques. At the same time, if our approach seems to conflict with best practices, don't hesitate to ask me about it - we will both get a lot out of the ensuing discussion.

What I ask of you

“Do the class”

- Commit to the CS312 tools and processes
Perfect practice makes perfect
- Be a ~~good~~ great teammate
Be responsible for your learning, don't get left behind
Use your knowledge to make your team better

One goal of this course is to practice effective SW development methodologies. That is the process itself is important. Our goal is to practice strategic programming, i.e., working code is not our only goal. Recall that *perfect* practice makes perfect. I want you to “slow down” and make those investments and the structure of the course is intended to enable you to do so!

We have a wide variety of backgrounds:

- Some of you have learned these technologies in/for another class
- Some of you have used these technologies in an internship or summer research, and
- Some of you have never touched these technologies before...

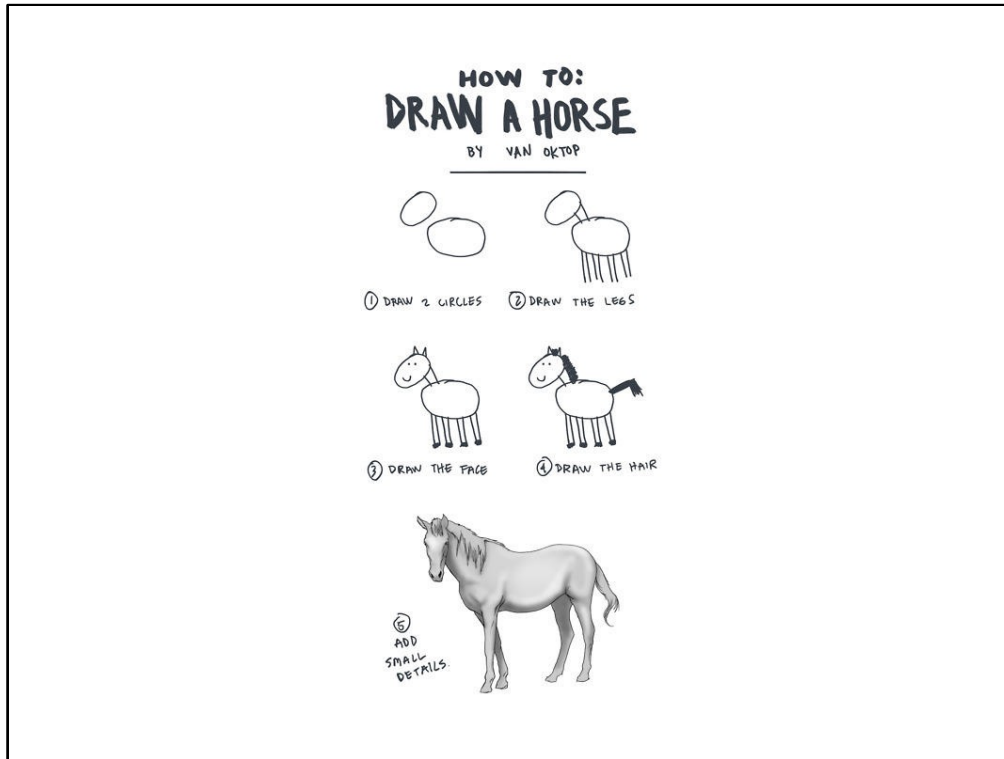
In some of these aspects, CS312 will be similar to your future working environments but more challenging (a company doesn't usually create a team of composed only of new developers). To overcome these challenges, I expect you to:

- * Be responsible for independently picking up the details of unfamiliar tools or technologies.
- * Put in the effort to make sure you don't get left behind. Use the resources on the course page (and others) and make sure to ask me and others for help when you need it. Don't be the teammate that can't contribute because you don't know what is going on!

If you have prior experience, I expect you to:

- * Use your knowledge to actively make your team better, not to sit back in judgement or frustration. Recall the often the best way to learn is to teach (e.g., "see one, do one, teach one"); you will get more out of the class if you actively engage with all of your classmates, including those with less experience.

- * Understand you can't do it alone. It may seem like you can do the project better or faster by yourself, but the end product will actually be worse if the whole team can't (or doesn't) contribute.



It will feel like the course content only gets you to to step 4, but the project asks for step 5... and that is true to some extent (but also often self-imposed). Our goal is to be simultaneously realistic and optimistic about what comes next. Are we going to recreate an application that was developed by 100s of engineers over years in just 6 weeks? No. Are we going to build a more impressive application than we expect? Yes! We do so by not trying to become experts in everything in a day or a week or semester (i.e., add all the “small details” at once) but by focusing on learning one thing at time (i.e., learning to draw the mane, then learning to draw the tail...) and doing that repeatedly. Amazing stuff will happen!

That learning will often be independent. In part, because your particular project will need specific techniques/technologies that we couldn’t possibly cover. That is true for your future endeavors too (regardless of your path). The chance the you took a class in the exact thing you end up working on is very slim (it may not exist yet!). You will have to learn it. And that is in part what we are trying to practice here.

Being a great teammate

TABLE 4. GREAT SOFTWARE ENGINEERS' ENGAGEMENT WITH TEAMMATES. ATTRIBUTES DISCUSSED IN DETAIL ARE IN BOLD.

Attribute and description	Excerpt that capture interviewees' sentiment
Creates shared context —molding another person's understanding of the situation while tailoring the message to be relevant and comprehensible to the other person.	"Most compellingly relate the value of that abstraction as it goes to non-abstract to very abstract to each person... empathize with your audience... get them to get it." -SDE2, Windows
Creates shared success —enabling success for everyone involved, possibly involving personal compromises.	"Find the common good in a solution... express here's the value for you... It's a win-win situation." -Senior Dev Lead, Windows
Creates a safe haven —creating a safe setting where engineers can learn and improve from mistakes and situations without negative consequences.	"If you learn something from a failure, that's a wonderful sort of thing... [but not] If you're afraid of getting smacked upside the head... encourage the people to experiment, possibly succeed, possibly fail". -Senior SDE, Office
Honest —truthful (i.e. no sugar coating or spinning the situation for their own benefit).	"When you do make mistakes, you've got admit you made a mistake. If you try to cover up or kind of downplayed mistake, everybody will see it, it's super obvious. It affects your effectiveness." - Partner Dev Manager, Corp Dev

- **Creates shared context:** Molding another person's understanding of the situation while tailoring the message to be relevant and comprehensible to the other person.
- **Creates shared success:** Enabling success for everyone involved, possibly involving personal compromises.
- **Creates a safe haven:** Creating a safe setting where engineers can learn and improve from mistakes and situations without negative consequences.
- **Honest:** Truthful (i.e., no sugar coating or spinning the situation for their own benefit).

Li, Paul & Ko, Andrew & Zhu, Jiamin. (2015). What Makes a Great Software Engineer?. ICSE 2015. Synthesizes 59 interviews with experienced Microsoft SWEs.

As we noted before, being a great teammate is a skill in its own right that can be developed and honed (like any other skill). I wanted to highlight the observations about teamwork from the Li et al. paper.







Course logistics

- Content to review before class, with pre-class questions on PrairieLearn, to create time for in-class work (the “practicals”)
- Specification/satisfaction grading
- 4 programming assignments in weeks 1-5 prepare for the project
 - Meaningful attempt by initial deadline “unlocks” later final deadline
 - Combination of automated testing and manual feedback
- Ongoing “practical” exercises with automated testing
- Exam partway through the semester tied to class Learning Targets with optional retest during finals week
- Large team project starting in week 6!
- Ed discussion board for Q&A, go/cshelp for peer assistant hours

Specification/Satisfaction grading. No “points” or partial credit. Everything is graded on a form of satisfactory/not yet satisfactory. All elements have an opportunity to receive feedback and resubmit (for programming assignments, project) or retake a similar problem (exams). Final grade is determined by the bundles described in the syllabus. Why? A grade should reflect your demonstrated understanding of the material at the end of the course. Assessing your work is a necessary but imperfect proxy for assessing understanding. My goal and responsibility is to create the best structures possible for you to demonstrate your true understanding. And your corresponding responsibility is to do everything you can to make your work accurately reflect your true understanding. Note that is an ongoing experiment. It will almost certainly require tweaking. I welcome your feedback and am ready to change any aspects that are not working.

There is an extensive “Getting Started” page with software to install. Please “get started” so you are ready to go for next class and we can resolve any setup problems.

Course policies (in an AI world!)

-  Working together
-  Working jointly on the same solution (same or different computer) to an individual assignment
-  Searching online for docs, suggestions, StackOverflow, etc.
-  Searching for or using previous solution to problem, even if freely available online
-  Generative AI (ChatGPT, Copilot) with citation
-  Not citing LLM use, copying the assignment as the LLM prompt

Our goal in this course is to maximize our productivity by using professional tools and practices (of which generative AI, is definitely one!). I have GitHub turned on. I use it and LLMs to help assemble materials for this course. You are welcome and even encouraged to experiment with those tools in this course. Here, as in any professional situation, though, you are responsible for the code that you submit/commit. That AI generated the code it is not an excuse for code that doesn't work.

That said, simply dumping the assignment into an LLM as the prompt is not permitted. Doing so short circuits the learning process. In many cases you are learning to use Javascript et al. for the first time. Finding/generating code that seems to work but you don't know why doesn't help build the understanding you need to tackle these problems in the future. In our programming assignments there are lots of tests and other feedback to know if the code works. But in the projects and elsewhere, that isn't the case. You are responsible for both writing the code and ensuring it works. Magic code only makes that harder...