What are you hoping to get out this class?

CS312 learning goals:
1. Describe and employ modern methodologies for managing SW development, specifically Agile and Scrum
2. Maintaining and extending “legacy” code
3. Working with non-technical customers
4. Testing
5. Describe, evaluate and employ technologies for full stack web development and single page web applications (SPAs)
6. Complete a large software development project as part of a team

Alongside the more formal learning goals I hope you will gain some important development skills that are not necessarily are part of your other courses (and not necessarily specific to SW development).
Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license
These specific tools and technologies are a means to an end. We care more that you learn the “why” for these tools than the “how” of particular technologies, frameworks, etc.

This class has a lot of moving parts. And I have made a lot of choices for tools, techniques, methodologies. In some cases it is a choice among many similar alternatives where there to is no "right" answer. That is 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 discuss later, I ask you to "do the class" with the class tools. Doing so will make your teams more efficient. At the same time, if our approach seems to conflict with best practices, don't hesitate to ask me about it - we will both learn from the ensuing discussion.
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.
Context: Evolving ecosystem

- Shrink wrapped ⇒ Software-as-a-Service
- Monolithic ⇒ Services
- 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)**

<table>
<thead>
<tr>
<th>SWS</th>
<th>SaaS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client-specific binaries that must work in many HW/SW environments</td>
<td>Online client-server model</td>
</tr>
<tr>
<td>+ Rich user experience</td>
<td>+ One copy of SW, one HW environment (controlled by developers)</td>
</tr>
<tr>
<td>- Hard to maintain, with extensive compatibility testing required</td>
<td>+ Easy to release updates</td>
</tr>
<tr>
<td></td>
<td>+ Easier to enable user collaboration</td>
</tr>
<tr>
<td></td>
<td>- Limited by online latency, capabilities of browser</td>
</tr>
</tbody>
</table>

*What about mobile native applications?*

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 with bookmarks too)
- Harder to maintain
- Upgrades now once again user’s problem

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
This approach achieves code reuse, but at the level of whole services. Think of using Google Maps like a library.

A recent-ish variant of this approach is Microservices in which applications are composed from independently deployable services. https://martinfowler.com/articles/microservices.html

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
Traditional "thin client" model
1. User enters an address
2. Browser requests a resource from the address
3. Server at that address returns the resource
4. Browser renders the resource
5. User clicks link or fills in form
6. Browser makes new request to server
7. Server returns _new_ resource
8. Browser loads new resource and displays it

This is a "thin client" model in which all the real work happens and all state is maintained on the server. JavaScript may be used, but only for simple tasks like: form validation, layout tricks, and interactive - but stateless - features like an accordion.

Single page application or "thick" client

Google (with the help of some Microsoft technology) upturned this model with the introduction of Google maps. Prior maps applications, e.g. Mapquest, would request a new page from the server when you wanted to navigate. In contrast Google used an MS technology called AJAX which allows JavaScript to make requests to the server without loading a new page in the browser. In this "thick client" approach, much of the application functionality is implemented in the browser with communication...
happening behind the scenes. The result was (is) an experience much closer to a
desktop application. We would choose such an approach when we need to
implement an application with extensive user interaction that primarily interacts with
an API (that may already exist, or is needed to support other clients). This last aspect
makes SPAs natural consumers of (micro)services.
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

Correct: 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 a greatly simplified. (https://www.sitepen.com/blog/2017/02/20/microservices-and-spas/)

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license
Bezos’ 2002 services mandate

1. All teams will henceforth expose their data and functionality through service interfaces.
2. Teams must communicate with each other through these interfaces.
3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.
4. It doesn’t matter what technology they use. HTTP, Corba, Pubsub, custom protocols -- doesn’t matter. Bezos doesn’t care.
5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.
6. Anyone who doesn’t do this will be fired.

[Steve Yegge blog post (2011)]

https://plus.google.com/+ RipRowan/posts/ eVeouesvaVX

Radical at the time, but with hindsight we can see how this approach enabled Amazon AWS.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license
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 manifold (estimates of 3-8×).

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, is increasing becoming platform-as-a-service, PaaS, e.g. Heroku, in 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
Plan & Document ⇒ 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 in changes to documentation and the plan

Implementations: Waterfall, Spiral, ...

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

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

Correct: B.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license
Waterfall: Sequential phases of project (like a cascading waterfall).

In contrast Agile, implements multiple iterations of those lifecycles in short repeated cycles. Embraces change as a fact of life: continuous improvement instead of a single planning planning phase. Team continuously improves working but incomplete prototype until customer satisfied (with customer feedback on at each 1-2 week iteration).

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

In class we will implement one version of Agile, Scrum, and do so in a way a 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.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
“Full-Stack”, “DevOps” and other buzzwords...

"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 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, etc.).

"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
- SPAs are natural consumers of these (micro)services

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

Correct: 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 (adding new features to legacy 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

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”

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.
One goal of this course is to practice formal SW development methodologies. That is the process itself is important. And recall that perfect practice makes perfect.

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 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, we expect you to:

---

**What I ask of you**

---

**“Do the class”**

• Commit to the CS312 tools and processes
  
  *Perfect practice makes perfect*

• Be a good teammate
  
  *Be responsible for your learning, don’t get left behind*
  
  *Use your knowledge to make your team better*
* 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.
Being a great teammate

Table 4: Great software engineers’ engagement with teammates. Attributes discussed in detail are in bold.

<table>
<thead>
<tr>
<th>Attribute and description</th>
<th>Excerpt that captures interviewee’s sentiment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creates shared context — molding another person’s understanding of the situation while tailoring the message to be relevant and comprehensible to the other person.</td>
<td>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.” — SHRE, Windows</td>
</tr>
<tr>
<td>Creates shared success — enabling success for everyone involved, possibly involving personal compromises.</td>
<td>“Find the common good in a solution... express here’s the value for you... it’s a win-win situation.” — Senior Dev Lead, Windows</td>
</tr>
<tr>
<td>Creates a safe haven — creating a safe setting where engineers can learn and improve from mistakes and situations without negative consequences.</td>
<td>“If you learn something from a failure, that’s a wonderful sort of thing... but wait! If you’re afraid of getting smacked upside the head... encourage the people to experiment, possibly succeed, possibly fail.” — Senior SHRE, Offer</td>
</tr>
<tr>
<td>Honest — truthful (i.e., no sugar coating or spinning the situation for their own benefit).</td>
<td>“When you do make mistakes, you’ve got admit you made a mistake. If you try to cover up or kind of downplay mistakes, everybody will see it, it’s super obvious. It affects your effectiveness.” — Senior Dev Manager, Corp Dev</td>
</tr>
</tbody>
</table>

- **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).