Andrew Onozuka
Vivin Vinil, Alexander Tahan, Yuantian Zhou, Abhimanyu Srivastava, Merrick Qiu, Leo Lee, Joseph Del Val, Julia Poon, Jacob Felts

Week 1 | SE Intro - Demystification, Definition, and Defiance
- definition of SE, SE >> coding, more people oriented than people think, vs traditional engineering principles
- breadth of topics, type of projects, risk & impact, practitioner background, and more leads to variability in definitions

Week 2 | Individual Devs
- SE productivity is people driven, 10x devs, Postel's law also known as the robustness principle drives much of how the internet works. Roughly it states be permissive in what you accept, and strict with what you emit. This seems like a good plan in dealing with what we do (be strict) and what we encounter outside (be permissive)
- don't confuse confidence with competence, imposter syndrome, absolutism, you != your code, selfish vs selfless devs
- T shaped dev: A developer with too much breadth but no depth isn't great except in overview situations. Such folks often do better as project coordinators or in small orgs where we can afford to do less quality with less people. A developer with too much depth but lacking breadth often misses important external effects or opportunities. These developers also are more valuable in large groups as specialists, but may run the risk of being eliminated if the speciality falls out of favor. The best devs tend to be a "balanced T" and have breadth and depth within reason.
- you will fail before you succeed.

Week 3 | Groups
- Why Teams are Required, Team Formation Steps, Team Size and Communication, Team Composition - Same vs Varied, Team Composition - Star Players Benefits and Challenges, Team Topologies - Relating Org Ideas with Teams, Team Leaders, Team Challenges - Generally, Team Challenges - SE Specific

Week 4 | UCD
- a strong emphasis on the user during the constructive process - user centered design or UCD I often tend to think of this these days like moving from a earth centric to a heliocentric viewpoint as moving from a tech centric to a user and user centric view point.
- Law: You are != your users | Law: Users cannot be your designer | Tip: Avoid asking users what they want, instead infer it and verify it | Tip: Avoid "spooking the animals" if you decide to test or interview | Tip: You can't be everything to everyone, there will be no perfect usability, a11y, etc. Engineering pragmatism requiring you to balance trade-offs and to define what is "good enough"

<u>Sampling of UCD techniques:</u>
- UCD focused thinking → Persona generation, User Stories
- Observation → Direct or In-Direct - Analytics
- Interviews → Avoid guided questions, Listen, Aim for free form responses

<u>Common UCD Artifacts:</u>
- Personas → Beware of devolving personas too much into stereotypes, Consider your "mom" or other real people to avoid this challenge
- User Stories → These are considered Agile concepts, As a <blank> I want to <do blank> in order to <blank>
- Customer Journey Maps and Other Scenario Diagrams → Try to understand that your software lives in your user's world and is not their whole world, Their steps may occur over time and offline as well - most software needs to meet the world not be the world (better tell Zuck!)

<u>From UCD Research to Requirements</u>

Figuring out what they want is the first step which is then followed by making it work for them which has a range of -ilities. Both of these processes should happen before code is done and be documented in the form of design documents often ADRs (Architectural Decision Records) as well as the artifacts mentioned above (ex. Personas, etc.)

<u>Broad Requirements - Defining -ilities:</u> -ilities are often what may call Level 0 decisions which set some broad requirements a system must aim to adhere to. You should be careful working on projects that have not done the work to define these as once their effects are felt by actual end users or result in negative business outcomes, then people tend to come to look for those who "caused" it and they may view we SEs are to blame - even if in some sense this might be above our pay grade. TL;DR - Demand or define yourself system -ilities or put yourself or a project at a big risk

<u>User Focused -ilities</u> Some of -ilities like security, reliability, etc. are not user focused, but many are including:

- The system provides the required functions - **Utility**; Ability to access the systems and its function - **Availability**; Ability to access the systems within acceptable time - **Performance** - ility; Ability to be able to use the functions - **Accessibility**; Ability to be able to use the functions successfully - **Usability**; Ability to enjoy the functions - **Satisfaction** (Satisfiability)

These -ilities are architectural decisions that have to balance costs and some of them are at tension meaning that we might not be able to do everything in some perfect way (aka we must except trade-offs here)

<u>User Thinking Must Tie to System Thinking</u> | Danger: You must tie your user thinking to your system thinking, if you don't you will likely bake in a critical flaw that may hobble user acceptance.

<u>The Importance of Accessibility</u> | Accessibility is quite an important one because it can be discriminatory along physical ability as well as social position (think lack of hardware or connectivity) Accessibility (a11y) is not limited to just extreme cases like blindness and may be temporary or situational; Choices we make with technology can effect most all of these issues and sometimes inadvertently.; Example: non-semantic HTML is inherently inaccessible as it provides no hints to assistive devices. While it can be rectified your <div>-itis can catch up to you and may even result in legal liability

Week 5 | Process Models Overview

<u>Software Activities</u>: Requirements, Design, Development, Validation (aka Testing), Deployment, Operations

<u>Problem Solving</u>: Top-Down Design - start with high level and break down towards small pieces. The common reductionist style; Bottom-Up Design - using interactions, data, etc. to grow upwards; Middle-Out - a surprising concept that has us trying things sometimes completely out of order and still finding value via exploration; Generally I think you need to move between all the techniques as appropriate; An easy way to recall that realism of all approaches being useful might be to think of writing a book. Start with concept and outline - top-down. Write chapters and realize outline needs revision - bottom up. Get stuck and try sections or examples out of order - middle out

<u>Iron Triangle</u>: cost, scope, schedule, quality in the middle | will be trade offs no matter what

Agile: Flexible, iterative, customer involvement, quicker delivery, adaptable, collaborative teams.

Waterfall: Sequential, rigid, limited customer involvement, longer delivery times, risk mitigation through planning, extensive documentation.

Week 6 | Architecture

- Availability, Maintainability, Usability, Security, Performance, Reliability

Agile tends to focus on less global roles so architectural duties are not explicitly defined. In some ways, it would be thought to be emergent from the group with individual members making decisions. This lead to my point of Where's Waldo the Architect? for Agile. The architectural guru, astronaut, "all powerful", etc. is at the opposite end of the spectrum. The challenge here is that such a person is hard to find, likely becomes a single point of failure and frankly becomes too distant to make good decisions. My general thought is we need an architect and they should be experienced, but trust their hands on members if not be in the code with them!

- Architectural Decision Record (ADR)