[This document was copy-pasted from some slides… it may be stale]

**Whitewater Rafting**

**What is the difference between the picture on the left side and the picture on the right side?** Professional rafters vs amateur rafters

**How are the trips different?** There are a lot more rules / regulations for the professional trip → mostly for safety, but also for impact on the environment, some rules are to extend the life of the equipment. I can go cliff jumping on my own but not as a commercial guide (liability).

**What is the difference in skill between the guides?** The best of the professionals are probably a bit better than the amateurs. The worst of the amateurs are (hopefully) worse than the professionals. But for 75% of the boating community, there's not too much difference.

**Who determines whether the trip is a success?** For the amateurs, I decide whether it was successful. Did I enjoy it? If my friends didn't, then I won't invite them next time. For the professionals, it's the customers whose opinion matters (and to some extent the owner of the company); I might have a horrible time but if the customers had a great experience then it was successful.

**How much can the trip cost?** There are strong supply & demand economic forces on the professional trip → the company needs to make money but can't charge too much or nobody will book a trip. They have to amortize the equipment costs, pay for insurance, pay the employees, maintain the property, and in the end the owner needs to make a profit. For an amateur, the trip can be as fancy or as basic as I feel like.

**What is the timeline for each trip?** I can stay on the river as long as I like (within reason, of course). For a professional, there needs to be a consistent plan so that the guests can arrange their schedule accordingly.

**Professional Software Development**

**What determines if a project is success?** I do / Customer and business team (within budget, on time, sufficient quality - usability and reliability/accuracy)

**Who chooses the purpose / scope of the project?** I do / Customer, with input from computer scientist

**Is there a difference in skill between an amateur and professional computer scientist?** Not necessarily, although there are certainly novice coders who don't have the skill to be hired into a professional role

**What is the time frame involved?** However long I am interested / A schedule agreed upon by all parties that probably includes long term maintenance and customer support

**What are the budget limitations?** However much I want to spend / My company needs to pay me and make a profit, but if I charge too much then we won't have any business

**Product Failure Curves**

These two graphs show product failures over time. High values on this graph are bad because they mean that products are failing. Low values are good because it means that most things are working as intended.

The top picture shows the failure rate of a typical product such as a car. There are some lemons that fail right away. But once you remove those, there are relatively few failures for a long time. But eventually you get to the point where the equipment starts to break. Parts wear out and the product needs to be replaced.

You'd expect for the software failure rate to stay relatively low for the entire life of the product. After all, there's nothing to break because software is virtual, right?

But the bottom picture shows the failure rate of many software products. Notice all of the failure spikes! What happens is that programmers try to add new features to sell more copies of their product, but they introduce bugs at the same time. We can minimize this phenomenon with strong software engineering practices.

**Software Development Lifecycle Stages**

The SDLC is a process that is meant to minimize software failures. It is a series of stages that software "lives" through, much like the lifecycle of a butterfly. Let's see if you can guess some of the stages (hint: there's not *necessarily* 7 stages… might be more or less). Here are some potential answers:

1. Plan / Analyze
2. Design
3. Develop / Implement
4. Test & Integrate
5. Deploy and Maintain
6. Evaluate

**SDLC Diagrams**

Here are a variety of software lifecycle diagrams that I found on the internet. Notice that nobody can quite agree how many stages should be part of the SDLC. However, if you were to examine the diagrams closely, you would see that there are a lot of common themes and some groups just combine multiple stages into one. For instance: Analysis and Planning can be two stages in some diagrams and a single stage in others.

Notice that there are two different diagram shapes, the sequential and the cyclical. The older software development life cycles favored a do-it-once-and-do-it-right approach. They're on the top row: Waterfall Methodology and V Methodology. In this paradigm, you completely finish one stage before moving on to the next. It was a great first try and is still used today on smaller projects. But more recent versions of the software development methodology focus on the cycle part of SDLC. They are iterative, and develop the software incrementally in shorter periods of time called "sprints". This type of methodology is called "Agile" programming.

**Agile Programming**

This diagram includes the 5 stages that are common to all processes, organized into the Agile methodology:

1. Brainstorm: Plan and analyze user requirements (should I have said "Ideation" instead?)
2. Design: Create the overall software architecture with all its diagrams and documentation
3. Development: Write the main code for the program
4. Quality Assurance: test, evaluate, and fix any bugs in the software
5. Deployment: deliver the product and help the customer use it  
   (repeat with next version)

For the sake of establishing some consistency for the course, we're going to use these five stages as our official SDLC for the course.

Agile development has the advantage of adaptability. Software is designed in stages and the focus is on continuous improvement. Early sprints only contain a fraction of the functionality. Later sprints contain most/all of the functionality but might fix bugs or add new features. These cycles are important in today's (more) complex software market because the customer may not understand the full product requirements until the product is halfway finished.

Be careful not to confuse continuous improvement with a lack of planning. As new programmers, many of us just started writing code without any real plan for how it was going to turn out. This would be called "cowboy" style programming and it's very different from the Agile Development Methodology. Significant planning is required to make each sprint useful and to make sure that the early designs fully support the expected iterations of the product.

**Zeroing in on Software Requirements**

An entire course could be devoted to the Software Development Lifecycle, but I only have 75 minutes to cover it with the engineering students. So we're going to focus on one small part of the Agile SDLC: Software Requirements from the "Brainstorming" stage. We are going to focus on requirements because they are one of most important stages for the customer and business leaders to be involved in. And engineering students, you will likely be considered the *computer scientist's customer* for many of your projects.

Software Requirements are the key place in the SDLC where customers get to choose and customize the product they are purchasing. It's like when you are buying a new car and you go through all of the specifications. You choose the options that you want and then the company tells you what the price will be. These specifications (3.0 L V8 engine, AWD, manual transmission, roof rack, 19" rims, British racing green) are like the software requirements. Except that rather than choose them from a list, you have to make them up from scratch. Software Requirements are an agreement between you, the customer, and the company that is creating your product. Projects are bid and contracts are signed based on the Software Requirements. Then it's time for the computer scientists to head off on their own to create the product.

[There are a few slides that don't really need any explanation.]

**High Quality Software Requirements**

Pull out the Software Requirement Handout and have students read it on their own. Then discuss in groups what they have learned about software requirements.

The Ivy Hooks statement is from a talk at an International Council on Systems Engineering (INCOSE) symposium. She is a mathematician and engineer who worked for NASA on the Apollo Missions.

**Does Chat GPT Know How to Write Software Requirements?**

For fun, ask the students to use ChatGPT to come up with Dos and Don'ts of software requirements. Each group must add a unique item to each side of the list. Then ask the gr

**Sample Prompt:** How do you write high quality software requirements?

**Sample Answers:** Writing high-quality software requirements involves several key practices:

* …
* Consistency and Completeness: Ensure consistency across all requirements and completeness in covering all aspects of the software functionalities needed.
* Traceability: Establish clear traceability between requirements and other project artifacts (such as design documents, test cases, and user stories) to maintain alignment and understanding throughout the development process.
* Involvement of Stakeholders: Involve stakeholders, including end-users, clients, and development teams, in the requirement gathering and validation process. This ensures that requirements truly reflect user needs.
* Avoiding Assumptions: Do not rely on assumptions. Ensure that requirements are based on concrete information and not presumed needs or functionalities.
* Validation and Review: Regularly validate and review requirements with stakeholders to ensure they accurately represent the evolving needs of the project.
* Use of Templates or Standard Formats: Employ standardized templates or formats for documenting requirements. This helps maintain consistency and clarity across different requirements.
* Revision and Iteration: Embrace a process of iterative refinement. Requirements may evolve as the project progresses, so be open to revising and updating them based on changing circumstances or new insights.

**Improve the Requirement**

5 minute group exercise then come together with answers. Here are some potential answers:

1. "The system must have good usability"  
     
   Of course, a system should have good usability! Every tester and developer knows that. But to whom does it have to ‘feel good’ to? Describe the user group(s) and the knowledge expected from them. A common theme in this list of cringe-inducing requirements is their vagueness and lack of objective criteria.  
     
   Examples of measurable criteria are the time to complete a specified action. A better way to express this requirement is: “A customer service rep should be able to enter 3 issues in less than 15 minutes”. This is highly measurable.  
     
   Another way to get usability measurable is to set standards. If there is a documented company standard, you can state that the system should be built according to the standard.
2. "Each page of the system will load in an acceptable time-frame"  
     
   The requirement is ambiguous and not measurable. A better requirement would be something like this: "'Register Student' and 'Enroll Courses' pages of the system will load within 5 seconds."
3. "The system must provide the ability to create appointments and appointment invitations must be sent to clients”  
     
   We assumed at the start of the project that Dynamics 365 would be able to meet this requirement by leveraging the standard “Server-Side synchronisation” to send outlook meeting request to participants. Ex: When a user creates an Appointment in Dynamics 365, this appointment gets synched to the organizer Outlook Calendar and get sent to each attendee of the appointment.  
     
   However, during a clarification workshop, we discovered that the requirement is actually to send the meeting invites by text message. And that we should never send an outlook meeting request to clients.

**Summary**

Ask the students to help fill in the blanks. Help when they get stuck. Ask them to expand on some of the points.

**Answers:**

* sprints
* Brainstorm → Design → Develop → QA → Deploy
* brainstorm
* functional, non-functional, and acceptance criteria
* necessary, verifiable, attainable, and clearly defined