Introduction

What are Software Requirements?

Take a moment, and think back to the first bit of code you wrote for a course at Rose. Maybe it was in Python in CSSE120, or Maple in Calc, or Java in 220 or 230. What did you know before starting to write that code? Not specifically referring to the CSSE concepts or language syntax, but what did you know about what that code should do?

You probably knew the following about the program/method/etc.:

- Inputs:
  - The expected inputs
  - The types of the inputs
- Outputs:
  - The expected outputs
  - The types of the outputs
  - Any expected side effects
- A general description of what it should do to get there

Let's say you were given this problem statement:
"Write a Python function, printStars(n), that takes in a number greater than 0 and prints out a line with that number of * characters. Then, it prints out n-1 *s, then n-2, until only 1 star is printed. Then it prints an empty line (shown below as an _).

Example:

Function call: printStars(5)
Output:

```
*****
****
***
**
*
_
```

" [1]

What did we learn from that problem statement?

- Inputs:
  - n: the number of stars on the first line
  - Type of n: "number greater than 0"
- Outputs:
  - n lines of stars (The count of stars in each line decreases by 1)
  - 1 empty line

[1] Problem idea taking from CSSE120 problem set, rewritten from memory

    - Type of output: strings
- General Description:
    - A Python function for printing decreasing lines of stars.

With this information, you can dive right into coding the function. You know you'll need a loop and some print statements, and even if you're not too familiar with Python, loops, or prints, you can start researching and playing with these until you arrive at a successful solution.

Now think about this: Where did all that information come from? Most likely, it was provided by your instructor, a textbook, or some other previously prepared material.

Without that information, what would you have coded?

The information you used to write this function could be considered as software requirements, though not as formal as those we'll see later in the course. They are the ultimate goal of Software Requirements Engineering (SRE), but you can think of Software Requirements as "something that you can write code to."

Where do Software Requirements come from?

It's easy enough to locate the requirements source in a school course. Your instructors gave you prompts that told you what you needed to accomplish. And, you probably know why they provided those requirements. If we follow the example above, it may have been to help you learn how to use loops. In 230 it may have been to help you learn tree balancing, etc., but you generally know why you're writing a piece of code.

Now think about a larger program you've worked on, for example, your CSSE333 project. What did you know before going into that project?

That's probably a shorter list: You knew it needed to use a database and have some kind of front-end. If the project was from your lightning talk, you probably knew the domain well. Beyond that, the project was open-ended.

So how did you know what code to write? The initial problem statement made you write out the problem the system would solve and what functions the system would have. This was not busy work, or to make you write a paper because we love to grade them (we don't). This was your first exercise in SRE, and a necessary exercise before starting a project the size and scope of the CSSE333 project.

In the previous looping example, it was easy to sit down and start writing code because you knew exactly what the system should do. Was it that easy when you started your CSSE333 project? Did you know if you should use Eclipse/IntelliJ and start coding in Java, or if you needed a different IDE to support, say, an Angular app with NodeJS and Express?

In the CSSE333 project, you did not have software requirements, but **Needs** (client problems) and **Features** instead. Needs and Features are a high-level definition of a software problem and solution, and you can start to break out Software Requirements from them. These were likely never done formally for your CSSE333 project. Instead of starting to write code once you had software requirements, like in the looping example above, you probably started writing code when you had only Needs and Features. We will discuss Needs and Features more in the next reading.

Did this work well for your CSSE333 project? How many times did you message your team members and say, “Should X happen when the user does Y?” or “Did you already handle Z?” Your project team was small enough that this kind of management can work, though it gets tedious and often confusing.

Imagine someone came to you and said “I run a Humane Society, and managing all these paper records by hand is a mess. Can you write software for me?”

Would you be able to sit down and start coding?

In the looping example, there were concrete software requirements to tell you what to do. This leaves no confusion or fuzziness on what you should be implementing. In the CSSE333 example, there were some defined needs and features, though no concrete software requirements. This leaves some fuzziness on exactly what code should be written, but with a small enough team, good enough communication, and an internal client, the software can be completed.

In the Humane Society example, there is one need, no features, no defined software requirements, and an external client. Everything is fuzzy (even the residents of the Humane Society).

Resolving that requirement fuzziness is what Software Requirements Engineering (SRE) is all about.

In the CSSE333 example, you resolved the fuzziness not through SRE but by deciding as you went because you were the decision maker, because there was no external client. But in the Humane Society example, you are not the decision maker. You have a client, and that person is deciding which problems are most important. Why do they have that power? Because they’re generally the ones paying you to write the software, and they will be the actual end users of the system. This is where we must learn someone else’s domain to write software for them.

Why do we care?

You may be thinking, “My CSSE333 project turned out okay, and I know what a Humane Society does, I can code this!” And many software developers have tried this tactic before. Unfortunately, it frequently results in significant errors in the final software, meaning that the product either doesn’t do what the client wanted or does it in a way that’s worse than the status quo.

Leffingwell and Widrig’s Managing Software Requirements Second Edition covers this topic in more detail in Chapter 1. There are 10 copies in the library, and there is a free trial for the book online at the link below. Once you’ve read Chapter 1, return here to continue this reading.
https://www.safaribooksonline.com/library/view/managing-software-requirements/032112247X/

Does Leffingwell and Widrig’s summary still apply?

Computer Science and Software Engineering change by the month, it’s the nature of a field where little if anything is physical and work can be shared online in seconds. And you may have noticed that

Leffingwell and Widrig's book is from 2003, and the report they were citing in Chapter 1 was from 1994. So, has anything changed or gotten better in the last 15-20+ years?

According to the Standish Group's 2018[2] report on 2017 data, "In certain areas, there has been much improvement, while in other areas, things have gotten worse. The result is that the statistical numbers look almost the same as they did 20 years ago!" The report notes that the number of projects "on time, on budget, and on target" (how the Standish Report measures a successful project) was 31% in 1994, and in 2017 it was 36%.

The most progress has been seen in projects using agile methodologies, over the more traditional Waterfall[3] method. That said, medium sized agile projects still only have a 31% success rate. Overall, agile projects of any size have a 42% success rate, while waterfall projects have a 26% success rate. Would you hire a company knowing that, on average, the success rate was 42%?

Thankfully, the Standish Report contains much more detail to breakdown the factors that play into success or failure. In 2016, Standish introduced what they call the "winning hand" of project attributes, or those attributes that increase the likelihood of success for a software project. In the 2018 report, this winning hard consists of:

Attribute	**Standish Definition**
Small Project Size	6 team members (maximum) with a six month timeframe
A Good Sponsor	The person in charge of the project, a.k.a the decision maker. A Good Sponsor has low decision latency (i.e. they make decisions quickly) and helps support project direction and needs.
Agile Process	Standish notes that the agile process must be implemented correctly and that team members must be proficient in these processes and tools for this to be effective
Team Skills (aka Technical Skill)	Whether or not the team's technical skills match the project's needs. We'll call it Technical Skill for clarity in the reading below, to avoid confusion with the ability to work in a team.
Emotional Maturity	Teamwork, communication with team/client, problem solving, etc.

Are you surprised by any of these? We often think, "As long as my team has the technical skills, we'll meet our goals!" The Standish Group, compared what happens when a project team is highly skilled in Technical Skills or Emotional Maturity, but not both. Before reading on: Which do you expect to increase project success rates more: Technical Skills or Emotional Maturity?

[2] Johnson, Jim. Decision Latency Theory: It's all about the interval Boston, The Standish Group International, Inc, 2018.
[3] The Waterfall method of software development is a process in which each phase, such as requirements, design, implementation, etc. are fully complete before the next phase may begin.

The Standish Group found that a project with only high Team Skills had a 33% project success rate, while a team with only high Emotional Maturity had a 42% success rate. Teams with both of these characteristics raised their success rates to 50%. We often put a lot of weight behind technical skills, they are some of the easiest to teach and evaluate for, but it's worth taking note of the +9% change in success rate when your team favors Emotional Maturity over Technical Skills, and higher still when they favor both.

There is one thing not explicitly listed in this "winning hand" that we will talk about over and over the rest of the quarter. To find it in the top five attributes, you have to go back to the 2015[4] report. In that report, their top 5 "Factors for Success" were:

- Executive Sponsorship
- Emotional Maturity
- User Involvement
- Optimization
- Skilled Resources

Note that many of these are similar themes, though renamed.

- Executive Sponsorship → The Good Sponsor
- Skilled Resources → Team Skills
- Optimization fits into Agile Process

The key difference of note is that **User Involvement** appears to have been removed from the "winning hand" (or top 5 "Factors for Success" in 2015 terminology). This may seem an odd choice, but it is explained as follows:

> "'User Involvement' was the number-one factor of success for many years. This year, we listed it as number seven. It would still be close to number one, if not for agile process and emotional maturity. Agile processes incorporate users naturally and also provide for fast feedback. Emotionally mature teams also have a good relationship with their users and tend to bring them into the process. Therefore there is a reduced need to be proactive in bringing users into the project as a separate activity.
>
> By the same reasoning, non-agile projects and emotionally immature teams will require investment in user involvement skills. A large percentage of failed projects are projects that were completed, but then rejected by users and consequently not adopted. In addition, many projects were classified as 'challenged' not because they were rejected by users but because users were dissatisfied with the results."

Instead of discounting the need for User Involvement, Standish is instead relying on the fact that User Involvement is now such a consistent part of a properly implemented Agile Process, and the Emotional Maturity of teams, that it does not need to be called out as a separate task. They are in no way stating that User Involvement in the process is less important than it was, or that another attribute can substitute for it.

[4] https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf

Human Interaction is Key

In each of the project attribute lists above (either the 2018 or 2015 list), only 1 out of 5 of the attributes focuses on technical abilities, and none of them focus on the tools used to manage a software project. Improvement in project success rates is not about the technologies used by software engineers, or the advancements in these technologies over time, or even the technical capabilities of their teams. Project success rates are instead heavily influenced by human factors.

Software project success correlates strongly with improvements in Software Requirements Engineering practices, specifically those practices that focus on the communication between the humans on a project. **It's not about the technologies or the advancements we make in writing code, it's about the time and effort we put into understanding the stakeholders we're writing software for.**

What can we do about it?

Now that we know how detrimental requirements errors can be, and that those errors are usually due to the human side of the equation, what can we do about it?

Unfortunately, there is no silver bullet that can ensure you will never have an error in your requirements. However, being aware of where the pitfalls are, and the best practices to avoid them, can help prevent you from making these mistakes in future work. Throughout this course, we'll provide you with techniques to elicit and document requirements from a client, as well as methods to identify and correct requirement issues to ensure you're building software that meets client needs.