Learning Journal – Week 1: Introduction to Software Engineering & SDLC

This week, I went through the introduction to Software engineering and Software Development Life Cycle (SDLC). Through readings, self-quiz and discussion post, I learned a lot about why software engineering is important in our fast-moving digital landscape and the difference between any of the traditional engineering disciplines compared to software engineering at its core.

The first thing that hit me, and I discovered the hard way, is that software projects are uniquely complex. Unlike the construction of almost any physical structure, be it an aircraft or a bridge, which conforms to predictable laws of physics in mature processes, software development takes place in a volatile, ever-shifting landscape. Unlike other categories of engineering solutions (e.g. civil, mechanical), programming languages, development frameworks and methodologies evolve rapidly making the area a relatively immature and unsettled ground (Sommerville, 2016). To complicate matters, this only adds to the fragmentation of tools and practices making it challenging to really impose time-tested, standardized approaches to any software project.

As practiced, the SDLC framework provided a structured view of the phases of software development — it consisted of stages for requirements gathering, design, implementation, testing, deployment, and maintenance. It was quite fascinating to compare older paradigms, like Waterfall, to newer paradigms, such as Agile and Incremental models. Waterfall, although this linear approach is effective when requirements are not changing.. However, I learned that in most real-world projects, especially in software, requirements often change mid-way due to evolving client needs or market conditions. Agile, in contrast, accommodates change through iterative cycles, feedback loops, and close collaboration with stakeholders (Pressman & Maxim, 2020). This adaptability is what makes Agile and other iterative models so valuable in today's software development environment.

Reflecting on my discussion post, I realized that many software projects struggle not because of poor talent or effort but because of the invisible and intangible nature of software. It's much harder to detect bugs or bottlenecks early in the process when everything is abstract. Engineers working on physical structures can see and feel their materials, whereas software developers must visualize logic and interactions mentally or through simulation. This makes progress tracking and quality assurance more complicated.

I also found the concept of evolving requirements particularly relevant. I used Microsoft Word's extended development as an example of how shifting user demands and competitive pressures often lead to feature creep, which in turn delays timelines and increases complexity. In contrast, physical engineering projects like bridge construction benefit from more rigid and finalized plans that rarely shift once approved.

Overall, this week helped me appreciate the structured methodologies that support software engineering and how important they are to successful project outcomes. I now better understand the reasons for delays and failures in software projects and how selecting the appropriate SDLC model can mitigate many of these challenges. Going forward, I aim to apply this knowledge by critically evaluating which development models best fit different project contexts.

References:

Pressman, R. S., & Maxim, B. R. (2020). *Software engineering: A practitioner's approach* (9th ed.). McGraw-Hill Education. https://www.mheducation.com/highered/product/Software-Engineering-A-Practitioners-Approach-Pressman.html

Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson Education. https://www.pearson.com/en-us/subject-catalog/p/software-engineering/P200000003258/9780137503148

Wordcount: 484