Lecture 27: Software Engineering I

10/30/2020

Motivation

- In some ways, we have misled you about what programing entails:
  - 61A: Fill in the function
  - 61B: Implement the class according to our spec
- Always working at a "small scale" introduces habits that will cause you great pain later
- The goal of these lectures is to gain a sense of how to deal with the "large scale"
  - Project 3 will give you a chance to encounter "large scale" issues yourself

Complexity Defined

The Power of Software

- Unlike other engineering disciplines, software is effectively unconstrained by the laws of physics
  - Programming is an act of almost pure creativity!
- The greatest limitation we face in building systems is being able to understand what we're building! Very unlike other disciplines, e.g.
  - Chemical engineers have to worry about temperature
  - Material scientists have to worry about how brittle a material is

Complexity, the Enemy

- Our greatest limitation is simply understanding the system we're trying to build!
- As real programs are worked on, they gain more features and complexity
  - Over time, it becomes more difficult for programmers to understand all the relevant pieces as they make future modifications
- Tools like IntelliJ, JUnit tests, the IntelliJ debugger, the visualizer, all make it easier to deal with complexity
  - But our most important goal is to keep our software **simple**

Dealing with Complexity

- There are two approaches to managing complexity
  - Making code simpler and more obvious
    - Eliminating special cases, e.g. sentinel nodes
  - Encapsulation into modules
    - In a modular design, creators of one "module" can use other modules without knowing how they work

The Nature of Complxity

- What is complexity exactly?
  - Ousterhout: "Complexity is anything related to the structure of a software system that makes it hard to understand and modify the system

- Takes many forms:
  - Understanding how the code works
  - The amount of time it takes to make small improvements
  - Finding what needs to be modified to make an improvement
  - Difficult to fix one bug without introducing another
- "If a software system is hard to understand and modify, then it is complicated. If it is asy to understand and modify, then it is simple"
- Cost view of complexity:
  - In a complex system, takes a lot of effort to make small improvements
  - In a simple system, bigger improvements require less effort

Complexity

- Note: Our usage of the term "complex" is not synonymous with "large and sophisticated"
  - It is possible for even short programs to be complex

Complexity and Importance

- Complexity also depends on how often a piece of a system is modified
  - A system may have a few pieces that are highly complex, but if nobody ever looks at that code, then the overall impact is minimal
- Ousterhout gives a crude mathematical formulation:
  - C = sum(c_p * t_p) for each part p
    - c_p is the complexity of part p
    - t_p is time spent working on part p

Symptoms and Causes of Complexity

Symptoms of Complexity

- Ousterhout describes three symptoms of complexity:
  - **Change amplification**: A simple change requires modification in many places
  - **Cognitive load**: How much you need to know in order to make a change
    - Note: This is not the same as number of lines of code. Often MORE lines of code actually makes code simpler, because it is more narrative
  - **Unknown unknowns**: Worst type of complexity. This occurs when it's not even clear what you need to know in order to make modifications
    - Common in large code bases

Obvious Systems

- In a good design, a system is ideally **obvious**
- In an obvious system, to make a change a developer can:
  - Quickly understand how existing code works
  - Come up with a proposed change without doing too much thinking
  - Have a high confidence that the change should actually work, despite not reading much code

Complexity Comes Slowly

- Every software system starts out beautiful, pure, and clean
- As they are built upon, they slowly twist into uglier and uglier shapes. This is almost inevitable in real systems
    - Each complexity introduced is no big deal, but: "complexity comes about because hundreds of thousands of small dependencies and obscurities build up over time
    - "Eventually, there are so many of these small issues that every possible change is affected by several of them"
    - This incremental process is part of what makes controlling complexity so hard
    - Ousterhout recommends a zero tolerance philosophy

Strategic vs. Tactical Programming

Tactical Programming

- Much (or all) of the programming that you've done, Ousterhout would describe as "tactical"
    - "Your main focus is to get something working, such as a new feature or bug fix"
- The problem with tactical programming:
    - You don't spend problem thinking about overall design
    - As a result, you introduce tons of little complexities
    - Each individual complexity seems reasonable, but eventually you start to feel the weight
        - Refactoring would fix the problem, but it would also take time, so you end up introducing even more complexity to deal with the original ones
- The end result is misery

Strategic Programming

- "The first step towards becoming a good software engineer is to realize that **working code isn't enough**"
    - "The most important thing is the long term structure of the system"
    - Adding complexities to achieve short term time games is unacceptable
- Strategic programming requires lots of time investment
    - Try to plan ahead and realize your system is very likely going to be horrible looking when you're done

Suggestions for Strategic Programming

- For each new class/task
    - Rather than implementing the first idea, try coming up with (and possibly even partially implementing) a few different ideas
    - When you feel like you have found something that feels clean, then fully implement that idea
    - In real systems: Try to imagine how things might need to be changed in the future, and make sure your design can handle such changes

Strategic Programming is Very Hard

- No matter how careful you try to be, there will be mistakes in your design
    - Avoid the temptation to patch around these mistakes. Instead, fix the design.

        - E.g. Don't add a bunch of special test cases. Instead, make sure the system gracefully handles the cases you didn't think about