CS61B, 2021

Lecture 27: Software Engineering I

Backstory

Long ago, 61A had technical topics on Monday/Wednesdays and Friday was reserved for bigger picture topics.

- We're going to try out an experiment this semester, where for the next 3 weeks, one lecture per week will be dedicated to these sorts of topics.
- Very different flavor of lecture, much less "technical."

Let me know what you think!

Motivation for Today

In some ways, we have mostly misled you about what programming entails.

- 61A: Fill in the function.
- 61B: Implement the class according to our spec. Exceptions: Proj2 and proj3.

Always working at a “small scale” introduces habits that will cause you great pain later.

In these lectures, I’ll try to give you a sense of how to deal with the “large scale”, though it is a hard task because finding examples small enough to fit in lecture, but large enough to showcase the issues is very difficult.

- Project 2 and 3 give you a chance to encounter the issues yourself.
- I hope these lectures help with proj3 and reflect on your proj2 experience.

Credits

These lectures are heavily inspired by “A Philosophy of Software Design” by John Ousterhout.

- It’s cheap and very good!
- https://books.google.com/books?id=pD6-swEACAAJ&dq=philosophy+software+design&hl=en&sa=X&ved=0ahUKEwj9sZDmisvhAhXN6Z4KHcY6AYoQ6AEIKjAA
- https://www.amazon.com/Philosophy-Software-Design-John-Ousterhout/dp/1732102201

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.
- Civil engineers have to worry about the strength of concrete.

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 Complexity

What is complexity exactly? Ousterhout defines it thus:

- “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 easy to understand and modify, then it is simple”.

The Nature of Complexity

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.

Example: tilt in 2048

An Overly Complex Tilt Implementation

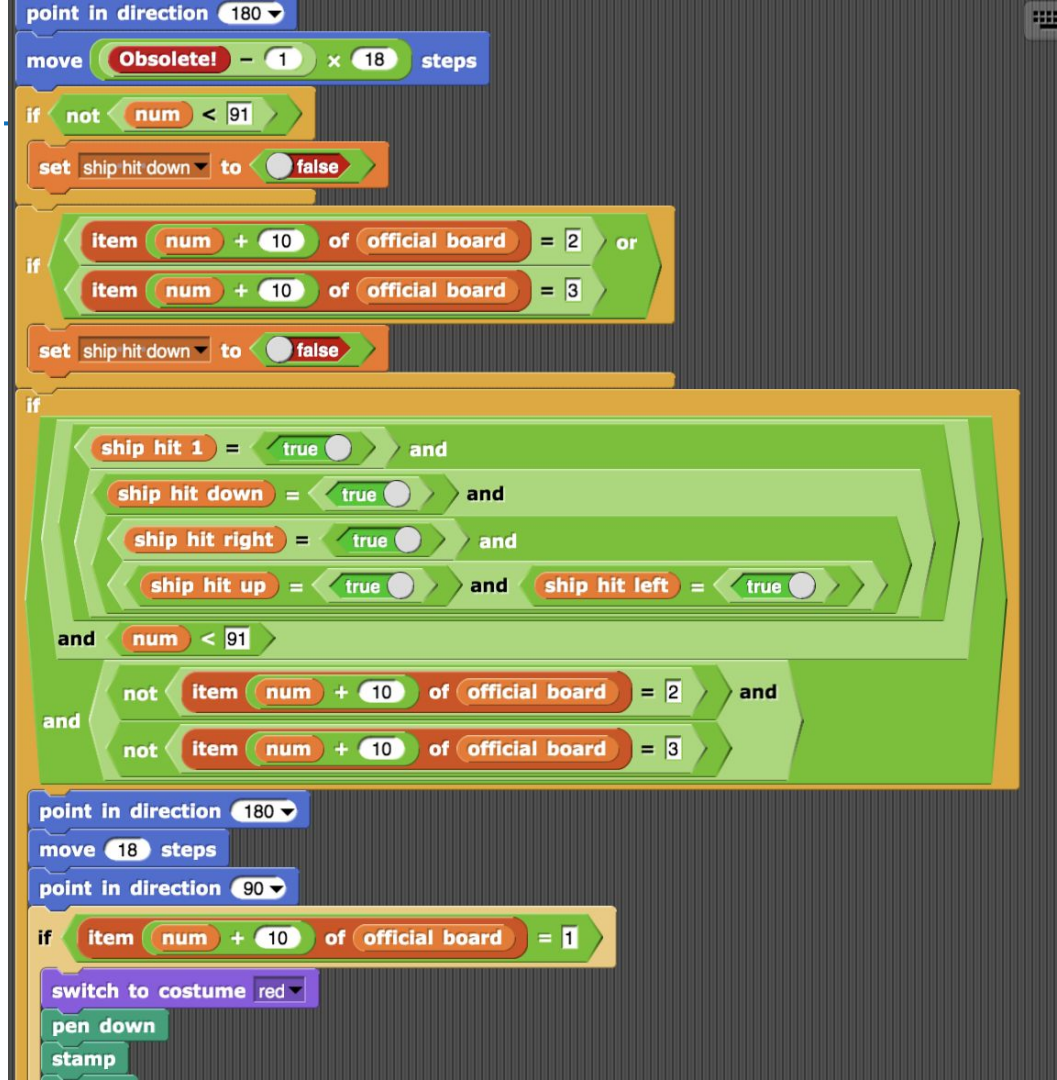
Let’s see an example of a tilt implementation that is way too complicated.

Complexity

Note:Our usage of the term “complex” in these lecture is not synonymous with ‘large and sophisticated’.

- It is possible for even short programs to be complex. Example from a former CS10 student (who was later a CS186 TA).

What makes this 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's book 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 the 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. Our overly complex 2048 was a good example of this.
- **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.
    - Incidentally, this is why I don't like ++ or --!

Symptoms of Complexity

Ousterhout describes three symptoms of complexity:

- **Change amplification**: A simple change requires modification in many places. Our overly complex 2048 was a good example of this.
- **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:** The 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.
  - You've felt this a lot in Gitlet!

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 or thousands of small dependences 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."

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."

This may seem like a silly criticism. Clearly, working code is good.

Tactical Programming

The problem with tactical programming:

- You don’t spend problem thinking about overall design.
- As a result, you introduce tons of little complexities, e.g. making two copies of a method that do something similar.
- 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.
  - Project 3 will give you a chance to feel this!

The end result is misery.

Strategic Programming

"The first step towards becoming a good software designer 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.

- And as novice programmers, it'll seem quite mysterious and hard. On project 3, try to plan ahead, but 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.
  - Example: In project 3, you'll need to support both input from the keyboard AND input from a String.

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.
  - Example: Don't add a bunch of special cases! Instead, make sure the system gracefully handles the cases you didn't think about.
  - Specific example: Adding sentinel nodes to SLLists.
- Indeed, it is impossible to design large software systems entirely in advance.

Tactical vs. Strategic Programming Case Study (Ousterhout)

As a startup, Facebook embraced tactical programming.

- “Move fast and break things.”
- Common for new engineers to push changes to the live site within their first week.
  - Very rapid development process in the short term.
  - Felt empowering!
- Facebook was very successful, but its codebase was a mess: “incomprehensible, unstable, few comments or tests, and painful to work with.”
- Eventually, motto became “Move fast with stable infra.”

Note: Arguably Facebook’s general attitude has done great harm. Will discuss in a future lecture.

Tactical vs. Strategic Programming Case Study (Ousterhout)

By contrast Google and VMware are known as highly strategic organizations.

- “Both companies placed a heavy emphasis on high quality code and good design.”
- “Both companies built sophisticated products that solved complex problems with reliable software systems.”
- “The companies’ strong technical cultures became well known in Silicon Valley. Few other companies could compete with them to hire the top technical talent.”

Real world projects and companies succeed with either approach!

- … but Ousterhout says it’s probably more fun to work somewhere with a nice code base.

Causes of Complexity

There are two primary sources of complexity:

- **Dependencies**: When a piece of code cannot be read, understood, and modified independently.
  - Example:
- **Obscurity**: When important information is not obvious.
  - Example: Naming our children “goodChild” and “badChild” make the code much easier to understand and reason about.

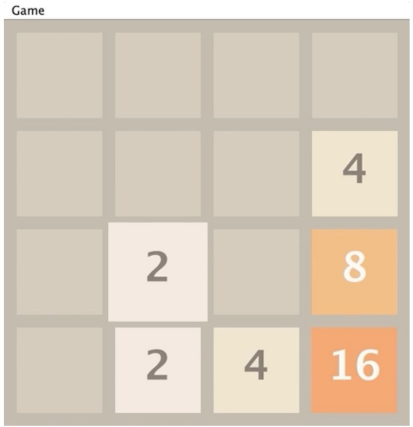
Seeking Obvious Code through Decomposition

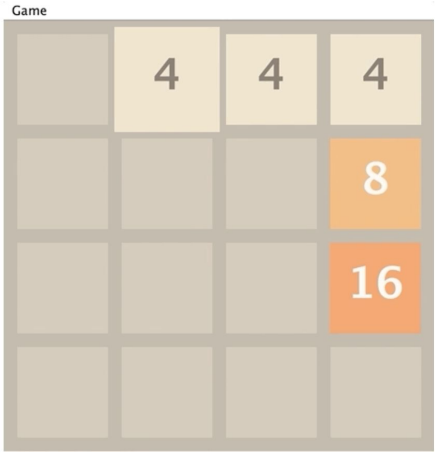
Decomposition Exercise

One of the most important principles for making code obvious is decomposition into parts, often implemented as helper methods.

Let's reflect on 2048.

- What are some useful ways to decompose the 2048 tilt operation?
- What are some useful helper methods (that you may have implemented)?

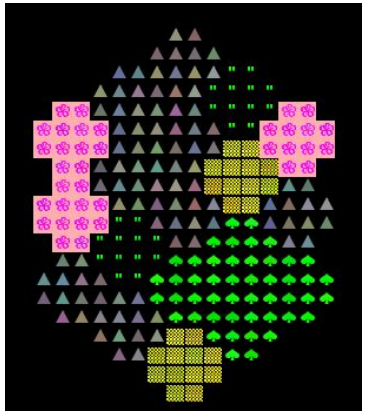


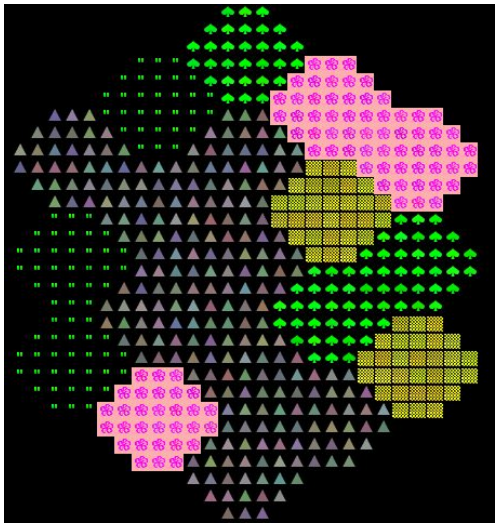


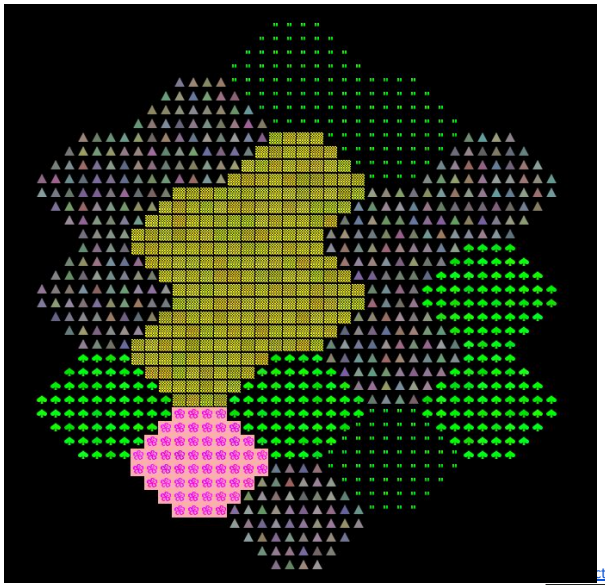
Decomposition Exercise

One of the most important principles for making code obvious is decomposition into parts, often implemented as helper methods.

Now let's try an exercise inspired by lab 12 (coming next week).







Goal: Generate Worlds

Suppose you were actually trying to write code to generate worlds like the ones below.

- What are some useful ways to decompose the world generation process?
- What are some potentially useful helper methods or classes?