Lecture 01: Welcome to OOD

May 9th, 2022

Agenda for Class:

- Go over Canvas / Syllabus
- Understand why we have OOD in the first place
    - Why does it exist as a class?

Canvas + Syllabus

- See Canvas

Why OOD?

- Easy answer: Writing software is hard
    - Stuff breaks, weird stuff happens, it's hard
- The issue - writing *good* software is **very** hard

The Value of Code

Consider the following Racket code:

```
(define sq(x) (* x x))
```

The code looks like it works, but anything and anyone can do it - it's not worth much

What code is valuable?

- Operating systems
    - Windows NT 3.1 - 4 million lines of code
    - Windows Vista - 50 million lines of code
    - Every project you've ever written combined - doesn't even scratch a million

> Takeaway: Big programs can easily create bugs

Adaptable and Extendable Code

- New iPhone Features
    - Spotify playlist shortcut
    - Under the screen touch id
    - Sharing battery
    - USB-C support
    - Photos into animation
    - No notch

Sometimes, the customer as a whole doesn't know what they want

> Takeaway: Software that lives for a long time **changes** and should adapt to change

How Do Software Engineers Solve the Problem?

Software Development Life Cycle (Waterfall Model)

1. You're given the problem
2. Analyze the *heck* out of the problem
    - Consider features
    - Limitations
    - What does the customer want?
3. Design a solution to the problem
4. Implement the design in `code`
5. Test the implementation RIGOROUSLY
6. Deploy the software
7. Evaluate the software
    - For us - self eval
    - In real world - customer feedback
8. Go back to step one

The *REAL* Software Development Life Cycle

1. Get the problem

2. Cursory analysis
3. Write the wrong implementation
4. Do a better analysis
5. Works, but incorrect design
6. Some more implementation + testing
7. Reanalyze + redesign
8. Repeat step 6
9. Iterate!
10. Deploy
11. Bug reports
12. Head scratching, confusion
    - What went wrong?
13. Do anything else
14. Tempted to reimplement from scratch
    - Costs WAY too much
    - Not feasible - our code should be flexible and extendable

The Main Questions

1. How do we write systems + deal w/ complexity of the problem?
2. How do we design w/ flexibility in mind?

Our solution for this class: **Object-Oriented Design (OOD)**

What is OOD?

Three main components

1. Information Hiding

- Keeping data private so other objects can't access it
- To drive a car you don't need to know how the engine works, you just know it *does* work

> **Objects should not know about the data representations of other objects**

2. Interfaces

- Expose operations to use from an object for the client

> **Interface: The line between client and provider**
>
> Client: Object that uses
>
> Provider: Object that implements interface

3. Polymorphism

- Same code works on different classes of objects
- Make our design loosely coupled

> **Loose coupling: Different components depend very little on details of other components**

SOLID Principles

Single Responsibility

- Each object has one purpose
- Car: Wheels turn, steering wheel turns the wheels, engine powers the car, etc.

Open or Closed

- Our design should be **open** to extension
- **Closed** to modification
- Should be able to add new features without gutting the code

Liskov Substitution

- If a Square is a Rectangle, a Square can be used anywhere a Rectangle can be used
- All subclasses of A should be able to be used where A is used.

Interface Segregation

- No client should be forced to depend on methods it doesn't use
- Introducing the idea of methods you CAN use and methods you CANNOT use
    - `public` vs. `private` distinction

Dependency Inversion

- Details depend on abstractions, but not the other way around
- Example from Fundies II: Double Dispatch

What Makes Software Good?

- Correctness
- Usable + Accessible
    - No need for extensive documentation
    - Usable by others
- Efficient
    - Usable within a reasonable amount of time + memory
- Works consistently among environments
- Extensibility + Backwards compatability
    - Modularity
- Security (reduced exploits + attack vectors)
- Scalability
- Maintainability

In this class we focus on two aspects:

- Correctness
- Efficiency

Design is about making CHOICES

- Between features
- Between different metrics (correctness vs. efficiency)

There is no silver bullet for design!

- EVERYTHING is tradeoffs

Client Perspectives vs. Implementor Perspective

- As the writers of systems, you are both the implementor and the client
    - Implementor: You write the code, etc.
    - Client:
    - User of the code but doesn't have access to implementation
    - Person who writes the tests