07 - Object Oriented Software Engineering

February 1, 2023

COMP2404

Darryl Hill

Overview

Software Engineering

What is software engineering?

It is a set of

- best practices
- strategies
- design patterns

used to deliver high quality software

Software Engineering

What is high quality software?

It is code that is

- reliable
- easily modifiable, extendable
  - all software is updated
  - what parts are updated will often influence design
- reusable
  - if we solve a problem it is enough to solve it once
- scalable
  - your code should work for all sizes of data and all numbers of clients
  - this is harder than you'd think

Software Engineering

Writing code that works and is correct is THE most important thing.

- You can't ship broken code.
- Correctness should be verified using **tests**
  - A couple of run-throughs is not sufficient

However we should also consider other aspects of design

- Code that makes updates and extensions easy is important over the lifetime of your product
- All actively used programs are updated and changed
- Once you stop updating, your product becomes obsolete

Design

Low level design involves making good parts

- Choosing what objects to use
- Protecting objects so the compiler catches errors
- Keeping interfaces abstract

High level design involves assembling the parts

- Developing an effective workflow
- Refactoring when necessary to more appropriate designs
- Separating objects into logical categories

Low Level Design

When considering your design, ask yourself:

- What objects do you need?
  - What data do they contain?
  - What behaviour do they need?
- Can you reuse classes from elsewhere?
- What do your classes have in common with each other?
- What information should be hidden in each class?

Low Level Design Principles

Classes should have a **single** responsibility

- Just as functions, classes should have narrow scope

Encapsulation

- Hiding unnecessary details
- Limit the ways someone can use your class to only those that are permitted
- This helps the compiler catch errors

Abstraction

- The act of generalizing your code
- Uses *encapsulation*

Principle of Least Privilege

- Design heuristic
- Helps us write good / well-encapsulated code
- Only allow access to what is absolutely necessary

Encapsulation

Encapsulation

- Hiding class implementation
  - Prevents hacking - using your class in ways it was not intended
  - This is not just about separating files
- Limiting access so that your class cannot be used incorrectly
  - Forcing the compiler to find and report errors for us
  - Compiler errors are preferable to runtime errors

We present a public interface that tells the user

- What our class does
- How to use it

Thus our class is used only as intended.

Principle of Least Privilege

It's a design heuristic that helps achieve proper encapsulation

Requires that you:

- grant permission to runtime objects only as needed
- never grant more permission than needed

Applies to

- variables, parameters, objects
- class members

Objects should be introverted.

Abstraction and Encapsulation

The biggest threat to timely software development is ***change***

- clients change their mind about what they want
  - more features, or something different
- designers misunderstand the requirements
- developers misunderstand the solution

Change is costly - good design mitigates this cost

- Updating spaghetti code is a terrible experience with terrible results
- Updating well designed code is easier and limits the places errors can occur

Abstraction and Encapsulation

Developers should

- design classes whose implementation can be changed without impacting other classes
- if the user class depends on implementation details, we cannot change our class without changing the user's class as well
  - our changes then break the user's code - very bad
  - this is why we hide implementation

Different pieces of code should communicate through an abstraction layer

- through interfaces
- abstraction layers is the main idea behind design patterns
- the more abstract the interface, the easier to modify the implementation

Abstraction and Encapsulation

Approach for good data abstraction:

- design objects that model the real world

A good class interface should be

- simple and intuitive
  - interface for files: `open('file')`, `save('file')`, `close('file')`
- not require knowledge of the implementation details
- be sufficient for future needs!

Abstraction and Encapsulation

For good encapsulation:

- Group together common data and functionality
- Grant least amount of access to other classes
- Use private or protected data members
- Maximize code reuse

High Level Design

Good approaches to software engineering are not always agreed upon.

- We will explore some ideas, but depending on the project you are working on, mileage may vary
- Try stuff, keep what works, reject the rest
- You often must conform to where you work, but your understanding of software engineering should keep evolving
- Keep asking questions
- Better to understand ***why*** certain decisions are made.

If you keep asking questions your software engineering skills will continue to improve.

High Level Design

What sort of architecture should you use?

- Decide a high-level approach, but stay flexible
- Often depend on what framework you are working in
  - Android? iPhone? XBox?
  - Each platform has made its own design decisions you must conform to

Don't fall into the trap of over-designing.

- Finding the exact design is often an iterative process.
- Some problems are revealed when you encounter them
- Use refactoring to introduce necessary design patterns.

Software Engineering - Workflow

What constitutes a good workflow - opinions vary - this is one option

- Requirements analysis
  - Determine what the client wants and translate that into an application
  - ***Use cases*** are particularly useful
- Initial design
  - Plan on using small pieces of code with clear purpose and interfaces
  - Apply design patterns or architecture styles to maximize flexibility
- Implementation
  - Follow (and update) the design
  - Stay flexible - unforeseen problems may be encountered
- Testing
  - Make sure what we build works.
  - Extremely important - you will write many, many tests in your career
- Iteration - repeat all these steps
  - A good design will eventually come into focus

Refactoring

Writing well designed code takes practice - people don't know to just "build a house"

- Iterate and update
- Often we see problems after we start coding
- Handling new problems or simply increased complexity often involves ***refactoring*** the design

Refactoring:

- Updating the software with a more appropriate design
- The function of the application should **NOT** change during refactoring
- We will see an example of **refactoring** after we learn **polymorphism**

Object Categories

An often useful breakdown of object categories:

- `control` objects (manage object interaction)
- `boundary` (UI/view) objects (interact with user / API)
- `entity` objects (store data about items)
- `collection` objects (storing many items)

With very general interfaces to communicate between categories.

Once we decide a category for an object, it becomes easier to determine the object's responsibilities.

Testing

There are many types and levels of testing.

We will discuss three main levels of testing:

- Unit testing
- Integration testing
- System testing

There are even more levels (alpha testing, beta testing, regression testing,etc), but we will stick with these.

Unit Testing

Unit testing is the testing of smallest components

- Usually individual classes
- Very straightforward if we have small, single purpose classes and functions

  `double Product::getPriceWithTax();`
- If we make a Product with a given price and tax rate, then the output is straightforward
- Most mistakes are in the edge cases
  - Should test with `price = 0.0; taxRate = 0.0;`, etc.

Integration Testing

Once we are certain individual classes work correctly we can see if they work correctly together

Test classes that

- use other classes
- contain other classes

`bool University::addStudent(Student*);`

- Can we print out the `Students` after?
- Do we have the correct number of `Students` after adding one?

Again, edge cases are where most errors occur

- Can I remove every `Student` then add them back?
- Can I print if there are zero `Students`?

System Testing

We integrate all of our system at once

- Also test system from User's perspective
  - Making menu selections for example

Again, test edge cases

- Can I print an empty `University`?
- Can I modify text areas that are meant for output?
- Can I use the app in unexpected ways?

This can include User **input** testing...

"What if I enter a character instead of an int?"

...but we won't focus on that in this class

Sanitizing user input can be a very large task