Overview

1. OOP Principles
   - Open Closed Principle
   - Favour Composition Over Inheritance
   - Single Responsibility Principle
   - Liskov Substitutability Principle
   - Law of Demeter

2. Refactoring
Object Oriented Programming Principles

The principles covered in class were:

- Open Closed Principle
- Favour Composition over Inheritance
- Single Responsibility Principle
- Liskov Substitutability Principle
- Law of Demeter
What’s open closed principle?
Open Closed Principle

A module should be open for extension but closed to modification.

You provide an abstract base class which gives the interface for the client to interact with. You then still have the freedom to extend the functionality through concrete classes without breaking client code.

Also known as ”Program to an Interface, not an Implementation”.
Problem:
1. We only have one class of shape (Square) at first, we want to calculate the sum of a vector of squares.
2. We add a new class Circle derived from Shape. How can we modify AreaCalculator::Area?
3. How to modify the code to follow open-closed principle?
Favour Composition Over Inheritance

Why?
We choose composition over inheritance because it is possible to modify the component at run-time while we are able to use it (whereas you can’t change your parent class at runtime.)
Favour Composition Over Inheritance

Problem:
A set of furniture: chair, bed and table in different colors: red, green and yellow.
1. If using inheritance,...
2. If using composition,...
3. If we add a new color now, ...
4. If we add a new type now, ...
Favour Composition Over Inheritance

When do you still need inheritance?

- We need a type hierarchy
- When using Polymorphism
What’s Single Responsibility Principle?
Each changeable design decision should be encapsulated in a separate module.
Single Responsibility Principle

Why:
- Code changes
- Low coupling
- Maintainability
- Testability and Debugging
Liskov Substitutability Principle

What’s Liskov Substitutability Principle?
Liskov Substitutability Principle

A derived class must be substitutable for its base class.
Liskov Substitutability Principle

- Objects accept the base class’s messages.
- Methods require no more than base class methods.
- Methods promise no less than base class methods.
Q: Is Square substitutable for Rectangle?

Example: lsp1.cc lsp2.cc lsp3.cc

source: Robert Martin, Agile Principles, Patterns, and Practices in C#
Law of Demeter

Also known as the principle of least knowledge.

Method `A::m1` can only call methods of

- `A` itself
- `A`’s data members
- `m1`’s parameters
- any object constructed by `A`’s methods
Law of Demeter Violations


class Foo {
public:
    void example(Bar b) {
        C & c = b.getC();

        c.doIt();

        b.getC().doIt();

        D d{};

        d.doSomethingElse();
    }
};
Sometimes, you will want to re-write your code, for one reason or another - perhaps to get rid of duplicate code, or to re-organize code to make it more concise, more modular, etc. Sometimes, you may *not* want to refactor - because there is a deadline coming up, or because the people you are working with are used to the code being a certain way (even if it is “wrong”).
Duplicate Code

What do you do if you have duplicate code...

- in the same class?
- in two related classes?
- in two unrelated classes?
Duplicate Code

In the same class: factor out common code into a private helper method.
In two related classes: put a helper method into a parent class.
In two unrelated classes: A little more interesting...

- Maybe they should be related? Could they be children of a single abstract parent class?
- Can the code be moved to a single class?
- Can the duplicate code be encapsulated in its own class?
What do you do if you have one large class, or one long method?
Break it up! Each module should handle a specific piece of functionality, and each function should have a specific task. A long function should be broken up into multiple functions and helpers, each of which does something small and concise. A function with a long parameter list is also a good sign that the function is trying to do too many things at once.
End