C# Best Practices

Clean Code Principles and Recommendations

Xavier Morera

Helping .NET developers create amazing applications

@xmorera / www.xaviermorera.com / www.bigdatainc.org

Let’s Talk About Clean Code

Any fool can write code that a computer can understand

Good programmers write code that humans can understand

by Martin Fowler

Clean Code

Clean Code

Code that is easy-to-read, maintain, extend, and change by any developer

Developers that look at the code should be able to

- Understand what the code does
- Why

Common Reasons to Prefer Clean Code

Software is intangible

Not constrained by the laws of physics

Can create an application in an infinite number of ways

The "better" your code is at the beginning

The easier it will be to modify and scale your application

I’ll Fix This Later…

```
public static bool CompareBooleans(bool a, bool b)
{
    if (a == b)
    {
        return true;
    }
    else
    {
        return false;
    }
}
```

Objectives of This Module

KISS

DRY

YAGNI

SOLID

Favor composition over inheritance

Separation of concerns

* And always write self-documenting code!

Writing Self-documenting Code

What does this mean?

Self-documenting code

Self-documenting Code

In a nutshell

Your code must express exactly what it does

- For example, using human readable names

Create variables with self-explanatory names

Write methods/functions that do one thing

Function That Does One Thing

```
Product CreateProduct()
{
    // ...
}
```

Function creates a product

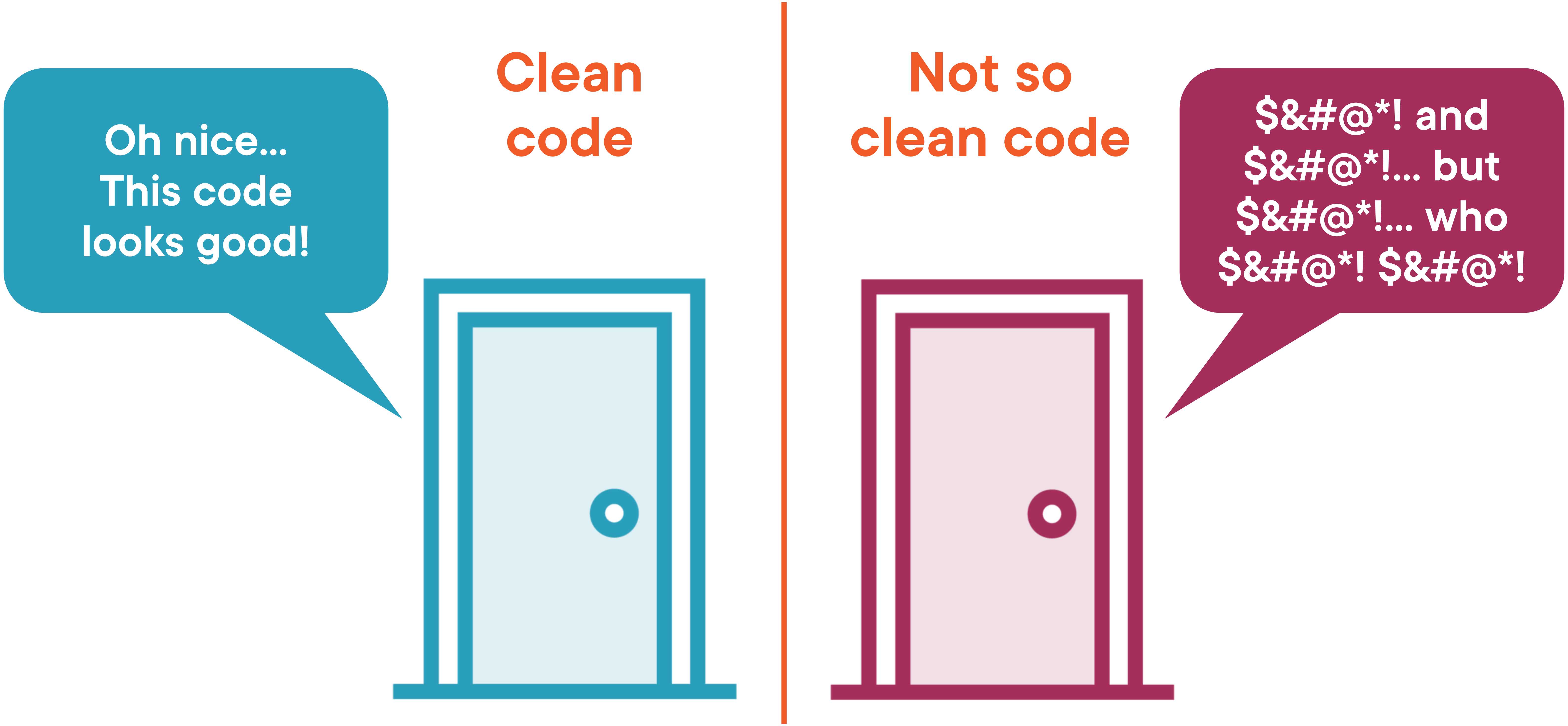
It is a great name!

“Code must express exactly what it does"

```
// Option A:
System.Drawing.Color.FromArgb(((int)(((byte)(0)))), ((int)(((byte)(64)))),
((int)(((byte)(64)))));

// Option B:
public int mul(int a, int b)
{
    int product = 0;
    for(int i = 0; i < b; i++)
        product += a;
    return product;
}
```

Is This Clean Code?


Oh nice... This code looks good!
$&#@*! and $&#@*!... but $&#@*!... who $&#@*! $&#@*!

Clean code

Not so clean code

KISS

Keep It Simple, Stupid

Simplicity is the ultimate sophistication

by Leonardo da Vinci

KISS

Make your code simple

- Avoid unnecessary complexity

If your code is simple

- Easier to read
- Thus, easier to maintain

Disclaimer

No hard rule on what is “simple code” and what is “unnecessarily complex code”

Many guidelines and suggestions

Simplicity is in the eye of the beholder

- There may be a fine line between both

Never try to show off using cryptic code

Demo

KISS

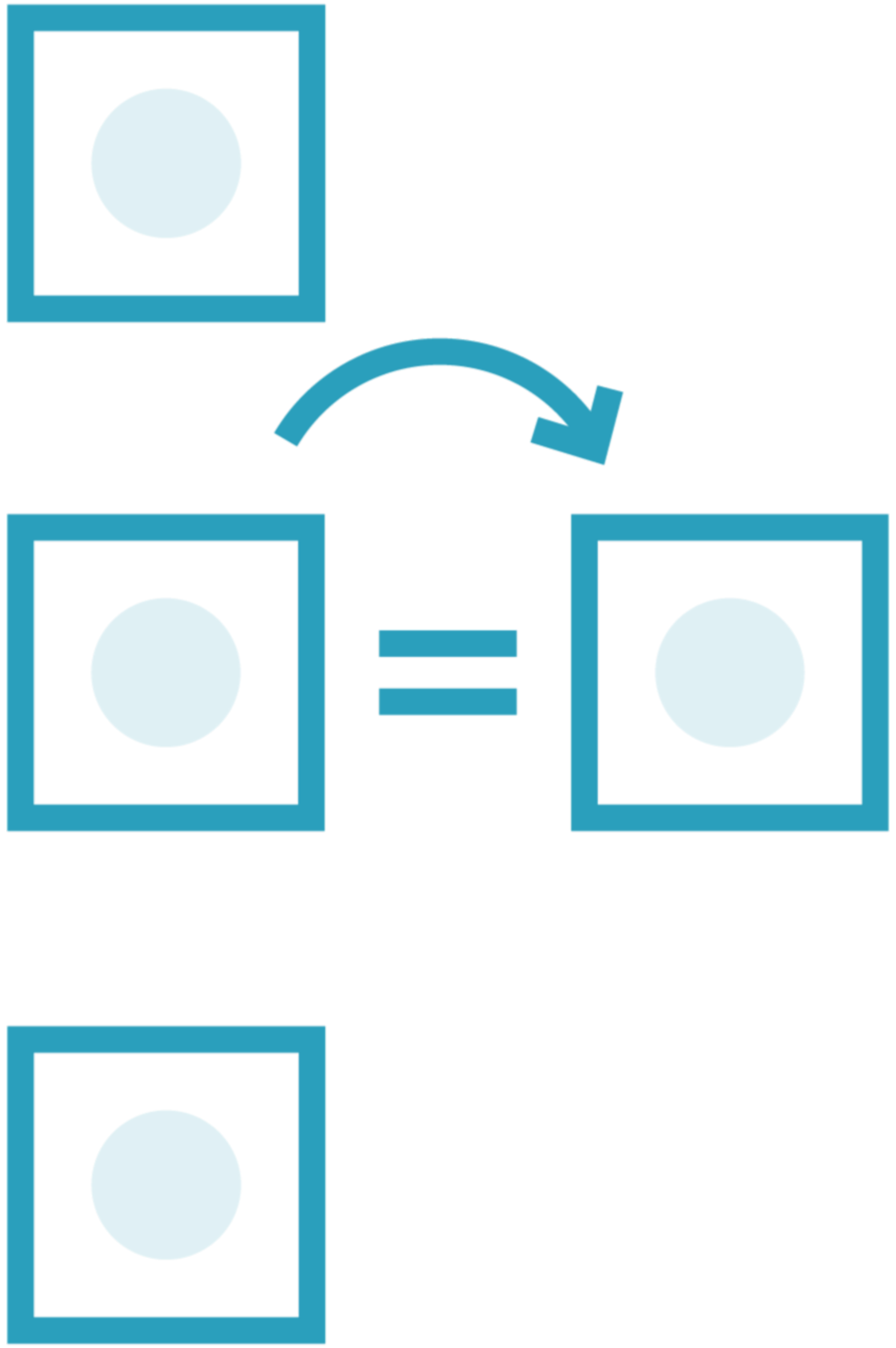
DRY

Don’t Repeat Yourself

Reduce the repetition of code

Goal

DRY



Just a few lines of code to perform an action

When a functionality is used multiple times

- Create a function with the statements
- Call every time the functionality is required

Every piece of knowledge must have a single, unambiguous, authoritative representation within a system

DRY principle (Pragmatic Programmer)

Demo

DRY

Write Everything Twice

WET

YAGNI

You Aren’t Gonna Need It

YAGNI

- Comes from the development methodology of Extreme Programming

- Avoid creating unnecessary functionalities

- Do not try to solve problems or implement solutions that do not exist, just because you think you'll need it

Demo

YAGNI

SOLID and the Single Responsibility Principle

SOLID

Acronym for five principles of object-oriented programming that help make code understandable, flexible, and maintainable

Five Principles

S	Single-Responsibility Principle
O	Open-Closed Principle
L	Liskov Substitution Principle
I	Interface Segregation Principle
D	Dependency Inversion Principle

Promoted by Uncle Bob

These principles were promoted by Robert Martin, known as Uncle Bob, originally in his 2000 paper Design Principles and Design Patterns

Single-responsibility Principle

S

Every class or module

- Has one responsibility
- Or specific functionality

If your class has many responsibilities

- Maybe it is time to split them in smaller ones

Applying the Principle

AND

When you are coding ask yourself

- What is the responsibility of this class?

If there is an "AND"

- Then it is required to break it up
- For example, `ProductAndWarranty`

The single responsibility principle applies to components and microservices

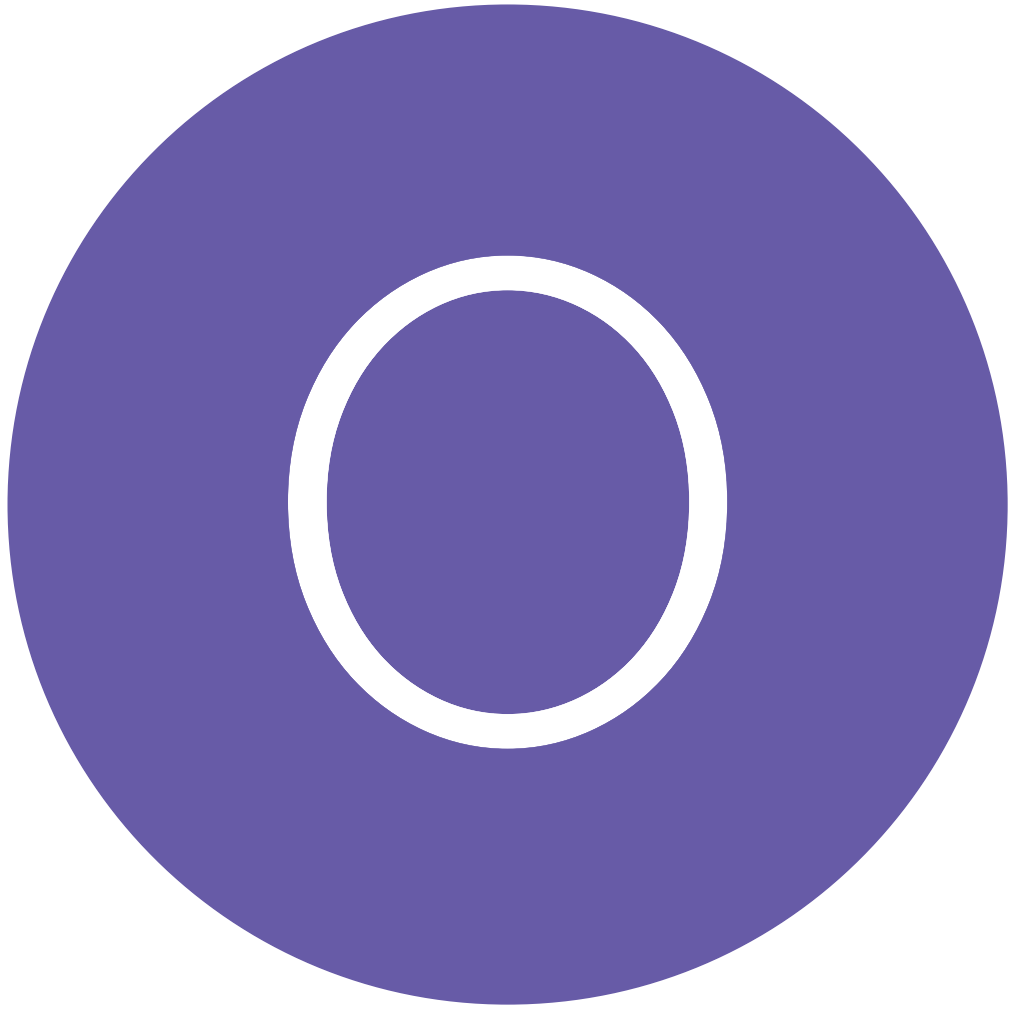
Demo

Single-Responsibility Principle

Open-closed Principle

The Open to extension but
Closed to modification

Open-closed Principle



Once a functionality has been implemented

- If requirements change over time
- Changes are implemented by adding new code

Most important principle of object-oriented design

Imagine changing your API response format frequently

- Would be a challenging scenario for users

Open-closed Principle

The Open-Closed Principle states that you should never rewrite the code

Demo

Open-closed Principle

Liskov Substitution Principle

Let Φ (x) be a property provable about objects x of type T.

Then Φ(y) should be true for objects y of type S where S is a subtype of T.

Liskov substitution principle

Any subclass object should be substitutable for the superclass object from which it is derived

Liskov substitution principle in a nutshell

Vehicle

Move method receives key to start vehicle

```
public class Vehicle
{
    // ...

    public virtual void Move(Key Key)
    {
        // Turn on
        // Move vehicle
    }
}
```

Inherits from Vehicle and starts with key

Truck

```
public class Truck : Vehicle
{
    // ...

    public override void Move(Key Key)
    {
        // Turn on
        // Move truck
    }
}
```

The Rules for Liskov

To achieve this, your subclasses need to follow these rules:

- Don’t implement any stricter validation rules on input parameters
  - Than those rules implemented by the parent class
- Apply at the least the same rules to all output parameters
  - As those applied by the parent class

Demo

Liskov Substitution Principle

Interface Segregation Principle

Interface Segregation Principle

I

A client should not be exposed to methods it does not need

Declaring methods that are not required

- Pollute the interface
- Leads to a bulky interface

Demo

Interface Segregation Principle

Dependency Inversion Principle

Dependency Inversion Principle

D

High-level modules should not depend on low-level modules

- Both should depend on abstractions

Abstractions should not depend on details

- Details should depend on abstractions

Demo

Dependency Inversion Principle

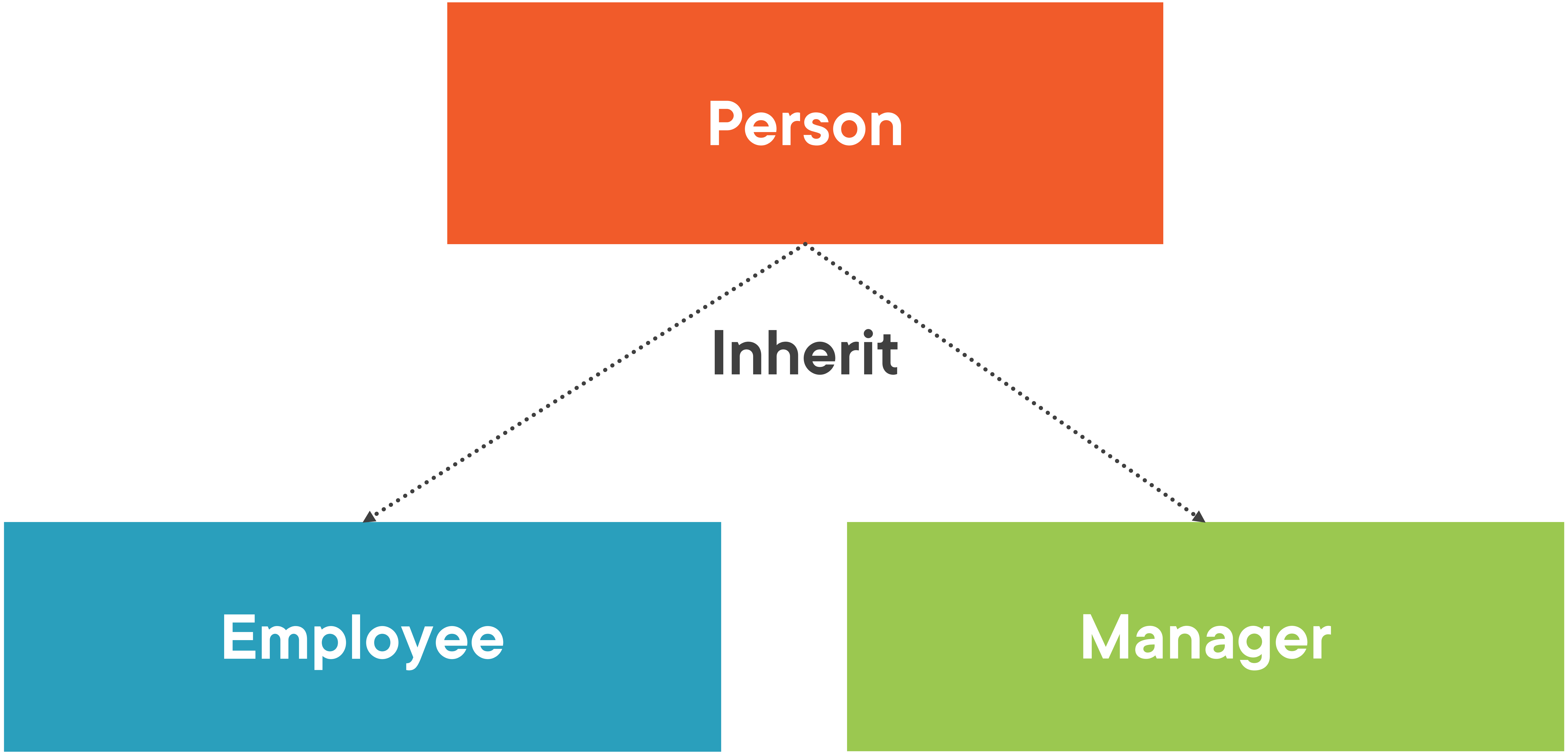
Favor Composition Over Inheritance

Favor Composition Over Inheritance

Design your types according to their functionality rather than nature

Inheritance makes your codes inflexible to later modifications

Inheritance

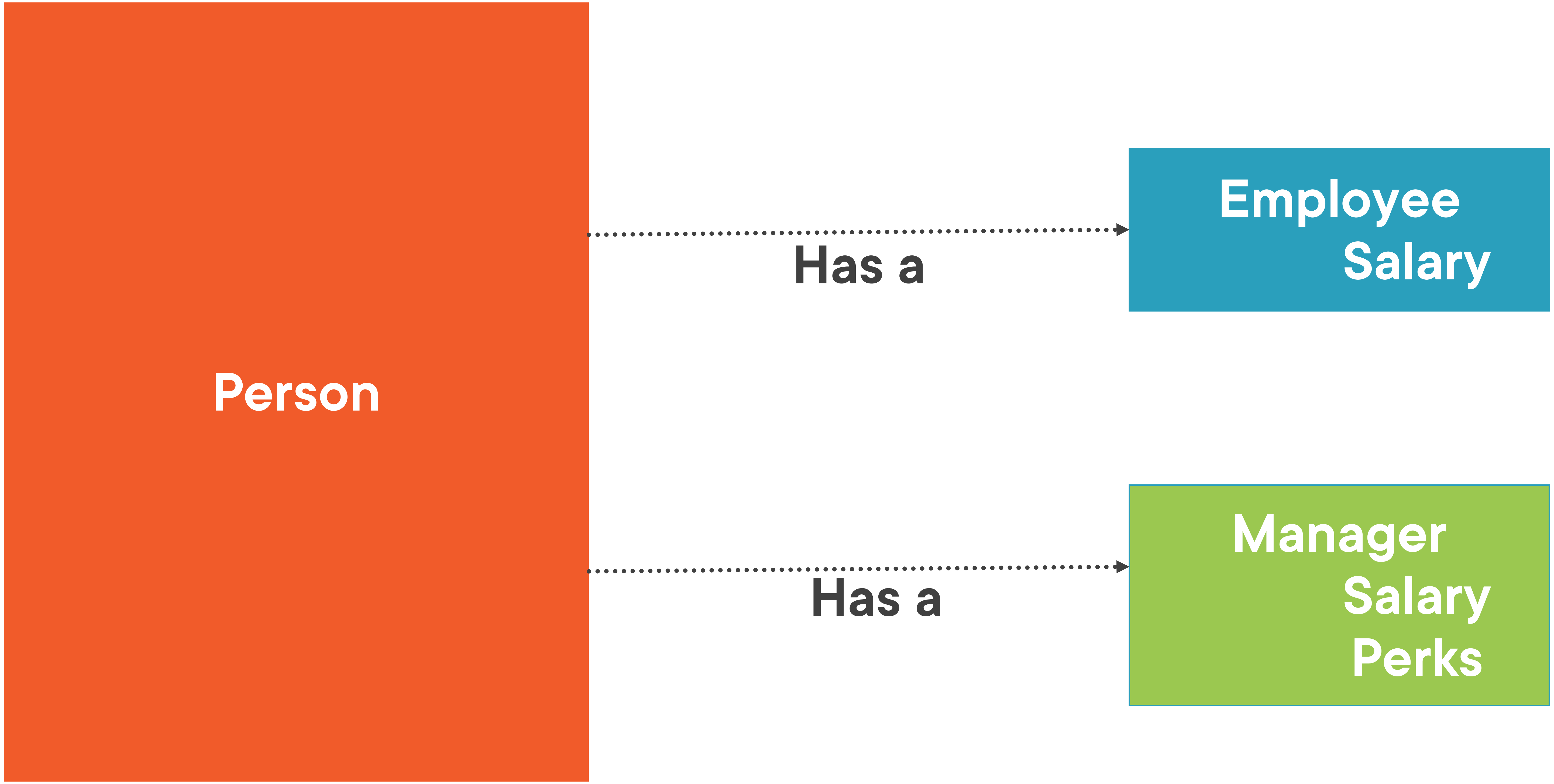


Composition

Class references one or more objects of other classes

Allows you to model a has-a association between objects

Composition



Person

Has a

Employee
Salary

Has a

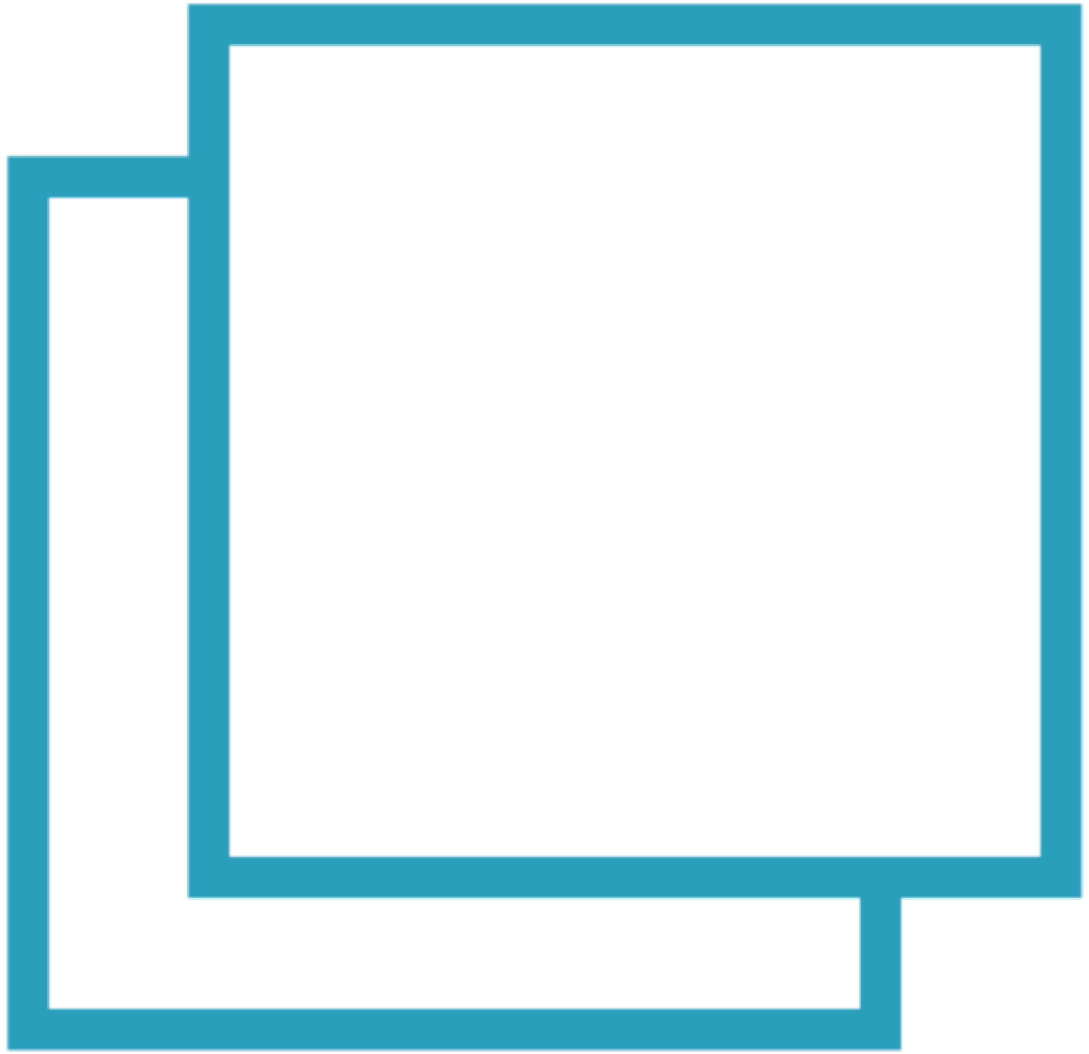
Manager
Salary
Perks

Demo

Favor Composition Over Inheritance

Separation of Concerns

Separation of Concerns



It is a principle used in programming

- Separates an application into units
  - With as little as possible overlapping
  - Between the functions of the individual units

Achieved by using modularization, encapsulation, and arrangement in layers

- Multi-layer architecture

Multilayer Architecture

CarvedRock.App contains the following components:

UI Layer

CarvedRock.Maui, CarvedRock.Web, and CarvedRock.Console

Business Logic Layer

CarvedRock.BL

Data Access Layer

CarvedRock.DA

Common Code

CarvedRock.Common

Takeaway

Write self-documenting code

KISS

DRY

YAGNI

SOLID

Favor composition over inheritance

Separation of concerns

SOLID

Single-responsibility Principle

Open-closed Principle

Liskov Substitution Principle

Interface Segregation Principle and

Dependency Inversion Principle