Good Software

D.R.Y. – Cohesion - Single-Responsibility – Coupling – Dependency Injection

Writing Software

How to write *GOOD* software.

S.O.L.I.D. Design Patterns

Cohesion

how much the parts of your code belong together

Cohesion

Cohesion

how much the parts of your code ***belong*** together

- **High Cohesion** – the parts of the class are tightly related
- **Low Cohesion** – the parts are lightly or not related

- **Which is better? Why?**
- **High cohesion is better because it means the parts make sense.**

Single-Responsibility

Do 1 thing well

Single-Responsibility

- Every module or class should have **only one reason** to change
- It should have responsibility over a single feature or functionality of the software

- Based on the concept of Cohesion.
- High Cohesion usually means your class likely has a single reason to change.
- If your class has functions or data not related to the class itself, then it has Low Cohesion and likely would have multiple reasons to change.

OOP: Single-Responsibility

- Imagine a Tank class that has a method for turning the turret and a method for sending a text message to your online friends.
- The Tank class might have to change if the speed of turning the turret changes AND it might have to change if the ways to text your friends changes.
- It really has **2 responsibilities** (tank behavior and texting behavior).

```
class Tank
{
    0 references
    void TurnTurret()
    {
        //logic for updating turret
    }

    0 references
    void TextFriends(string msg, List<int> friends)
    {
        //logic for sending a message to your friends
    }
}
```

Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

Coupling

how dependent pieces of code are on each other.

Coupling

Coupling

how dependent pieces of code are on each other.

- **Tight Coupling** means two pieces of code need each other directly.
- **Loose Coupling** means two pieces of code need each other but don't know specifically about the each other.

- **Which is better? Why?**
- **Loose coupling is better because it makes your code more flexible**

OOP: Coupling

Tight Coupling:

```
void DoSomething() //tightly coupled to the Circle class
{
        IShape _shape = new Circle(15);
        //or worse…
        Circle cir = new Circle(15);

}
```

OOP: Coupling

Loose Coupling:

```
void DoSomething() //loosely coupled now
{
    //the factory could return an Ellipse instead
    IShape _shape = ShapeFactory.CreateShape(Shape_Circle);
}
```

Dependency Inversion

Use indirections

Dependency Inversion

- High level and low level classes should not directly depend on each other.

- Abstractions should not depend on details but rather concrete classes should depend on abstractions.
  - C class should depend on the X interface but NOT vice-versa.

DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Is-a versus Has-a

OOP: IS-A vs HAS-A

- MetroBus and SchoolBus have an **IS-A** relationship with Bus.
  - MetroBus **IS-A** Bus.
- Bus **HAS-A** _passengers field. This is containment or composition.

Composition over Inheritance

- In general, **favor composition over inheritance**. **Why?**
  - Keeps the hierarchy chain simpler and more flexible.

- When possible, **program to an interface**. **Why?**
  - Looser coupling! Remember Dependency Inversion?

Summary

Good Software

- Don’t Repeat Yourself
- High Cohesion
- Loose Coupling
- Do 1 thing well
- Think before you code

For More Info

- S.O.L.I.D. Principles (https://en.wikipedia.org/wiki/SOLID)
- High Cohesion (https://en.wikipedia.org/wiki/Cohesion_(computer_science))
- Loose Coupling (https://en.wikipedia.org/wiki/Loose_coupling)
- Favor Composition over inheritance https://en.wikipedia.org/wiki/Composition_over_inheritance