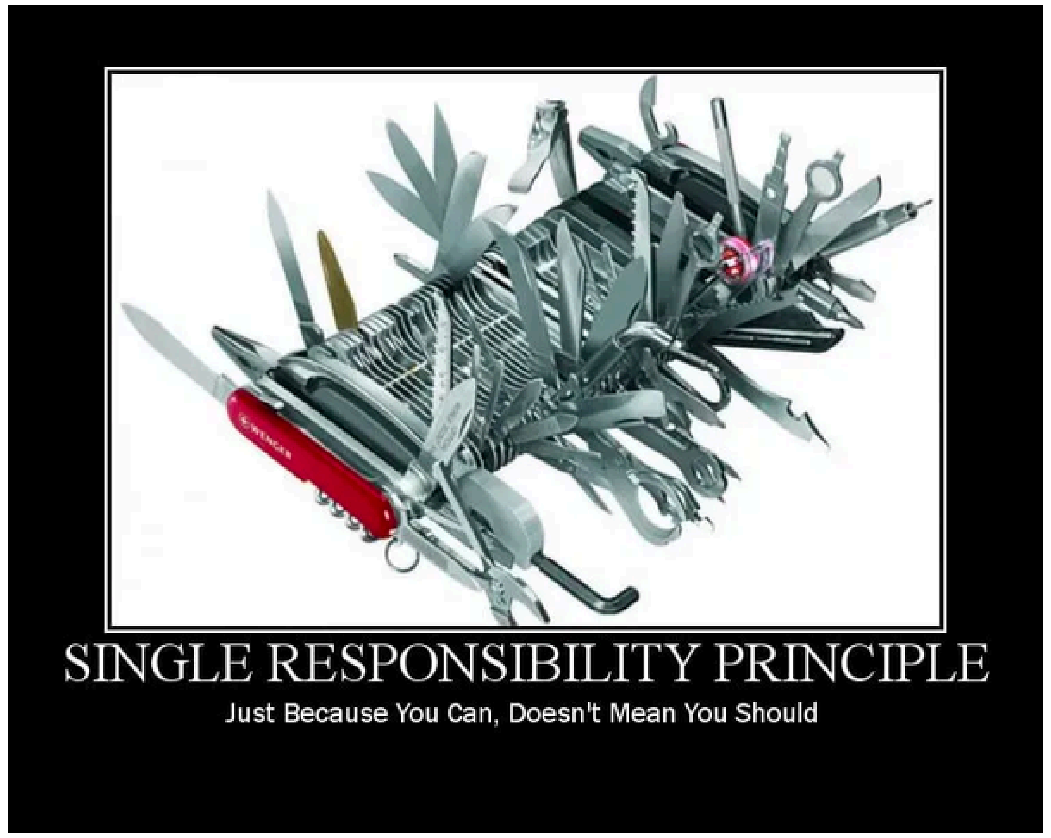
COMP 3550

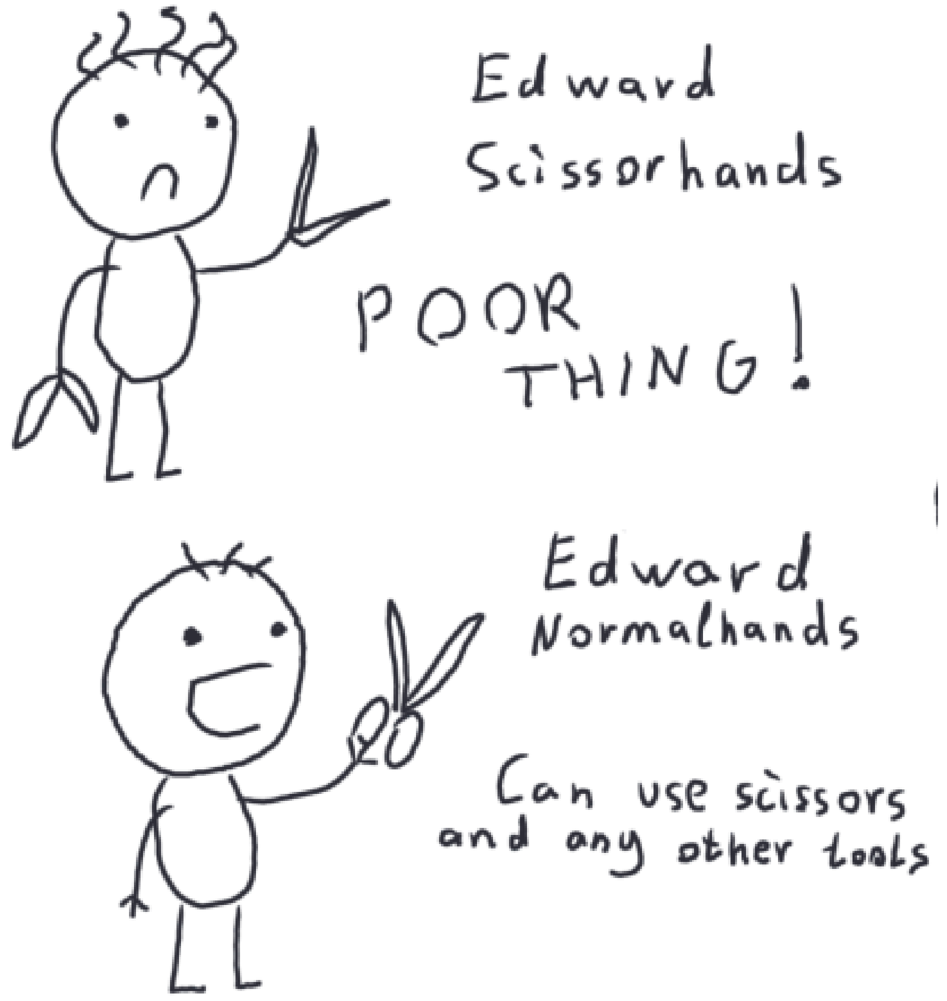
5.4 — SOLID (PART 2: ISP, DIP)

Week 5: Design Principles & Refactoring

REMINDER:


SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

SRP



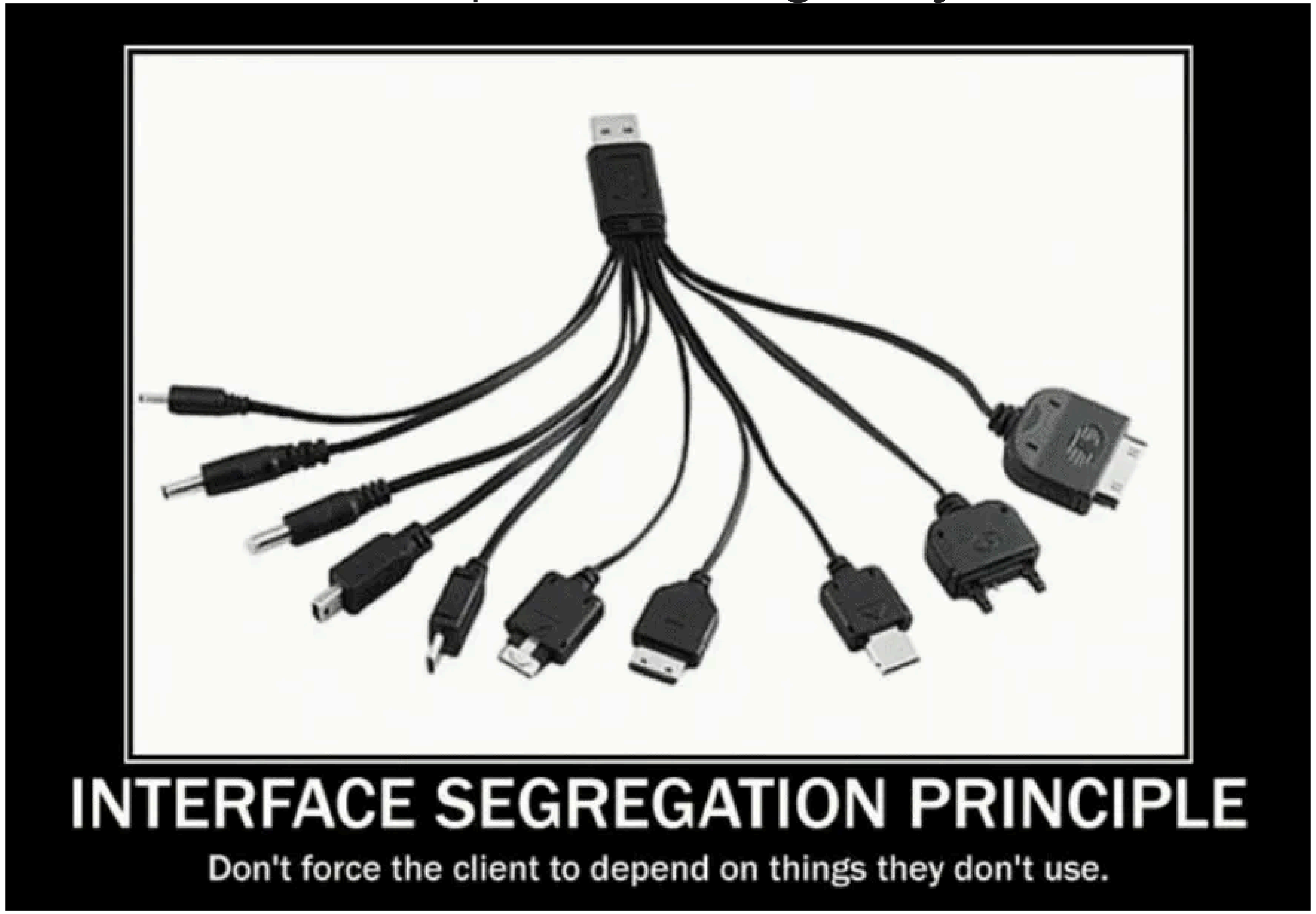
OCP

LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

LSP

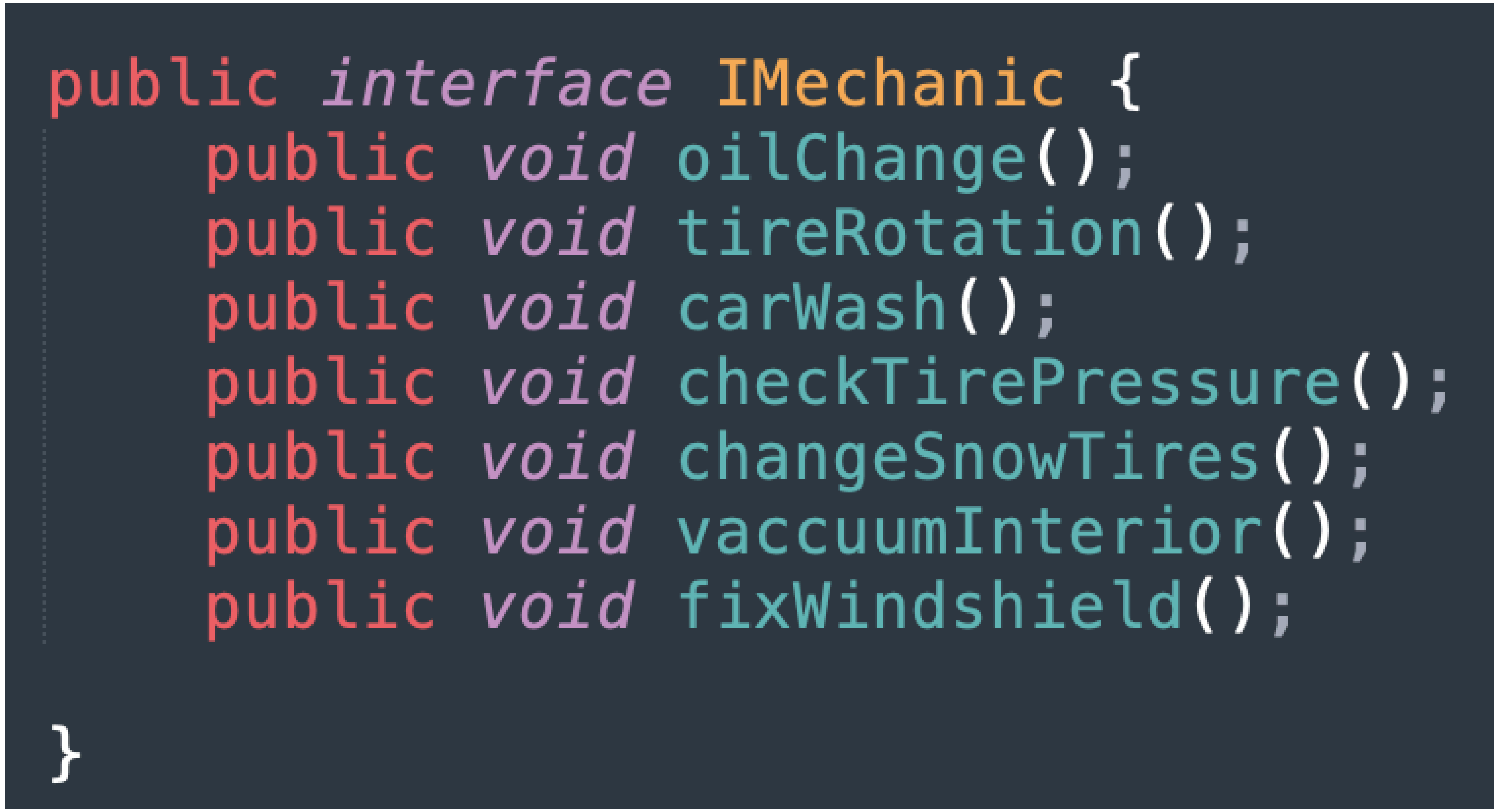
I — INTERFACE SEGREGATION PRINCIPLE (ISP)

Clients shouldn’t be forced to depend on things they don’t use


INTERFACE SEGREGATION PRINCIPLE
Don't force the client to depend on things they don't use.

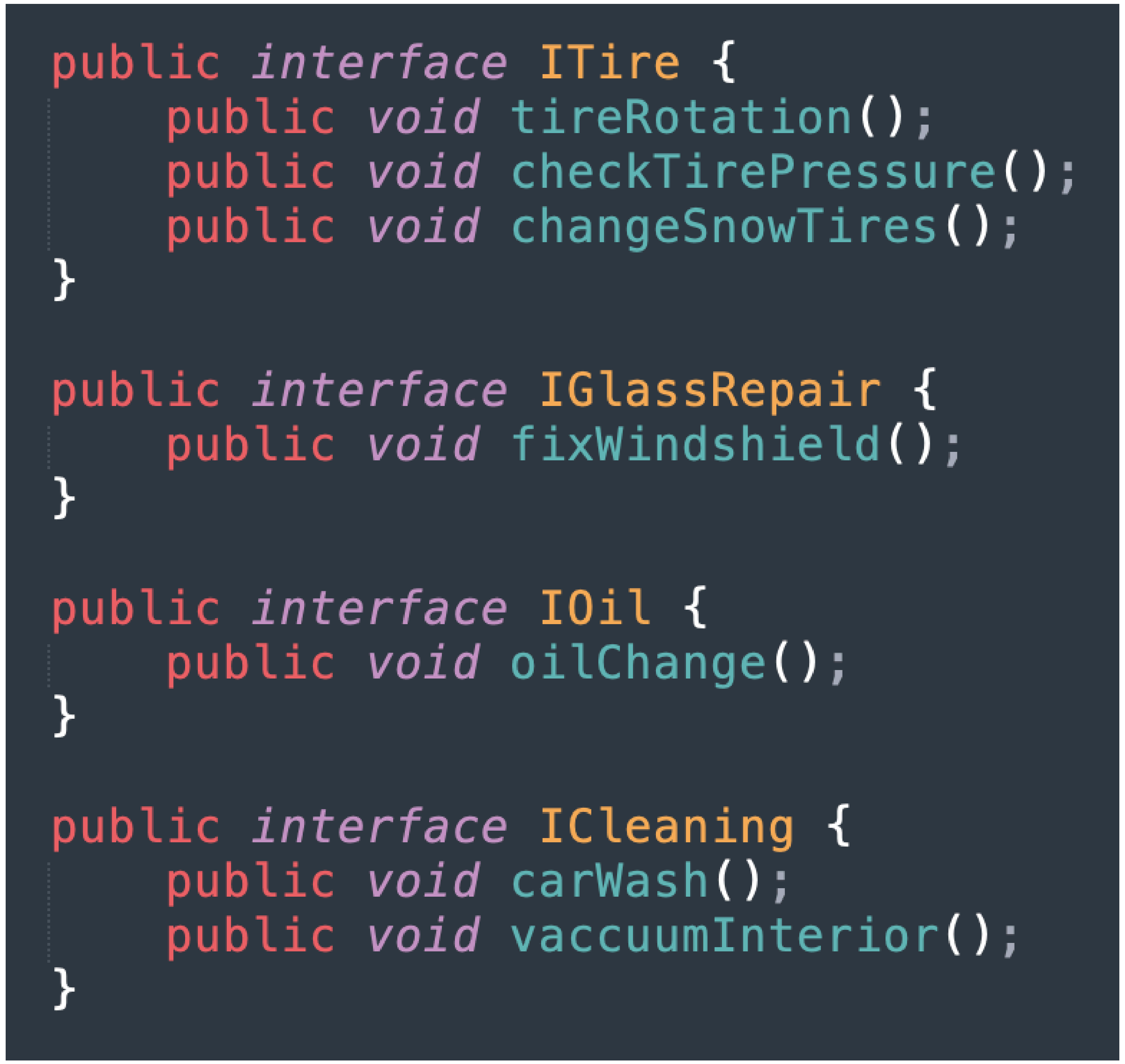
Credit: Sergey Podgornyy

I — INTERFACE SEGREGATION PRINCIPLE (ISP)



```
public interface IMechanic {
    public void oilChange();
    public void tireRotation();
    public void carWash();
    public void checkTirePressure();
    public void changeSnowTires();
    public void vaccuumInterior();
    public void fixWindshield();

}
```

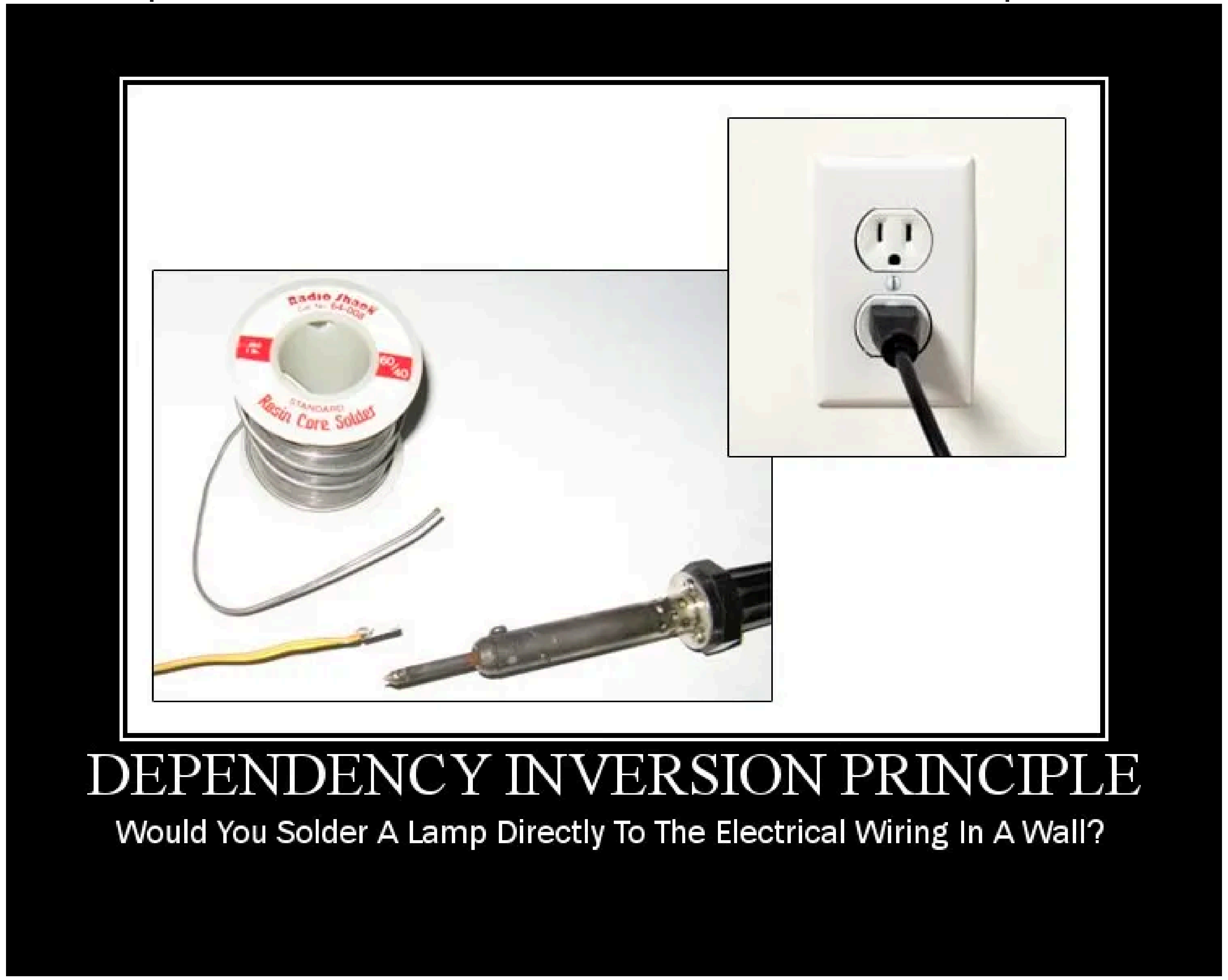
I — INTERFACE SEGREGATION PRINCIPLE (ISP)



```
public interface ITire {
    public void tireRotation();
    public void checkTirePressure();
    public void changeSnowTires();
}

public interface IGlassRepair {
    public void fixWindshield();
}

public interface IOil {
    public void oilChange();
}

public interface ICleaning {
    public void carWash();
    public void vaccuumInterior();
}
```

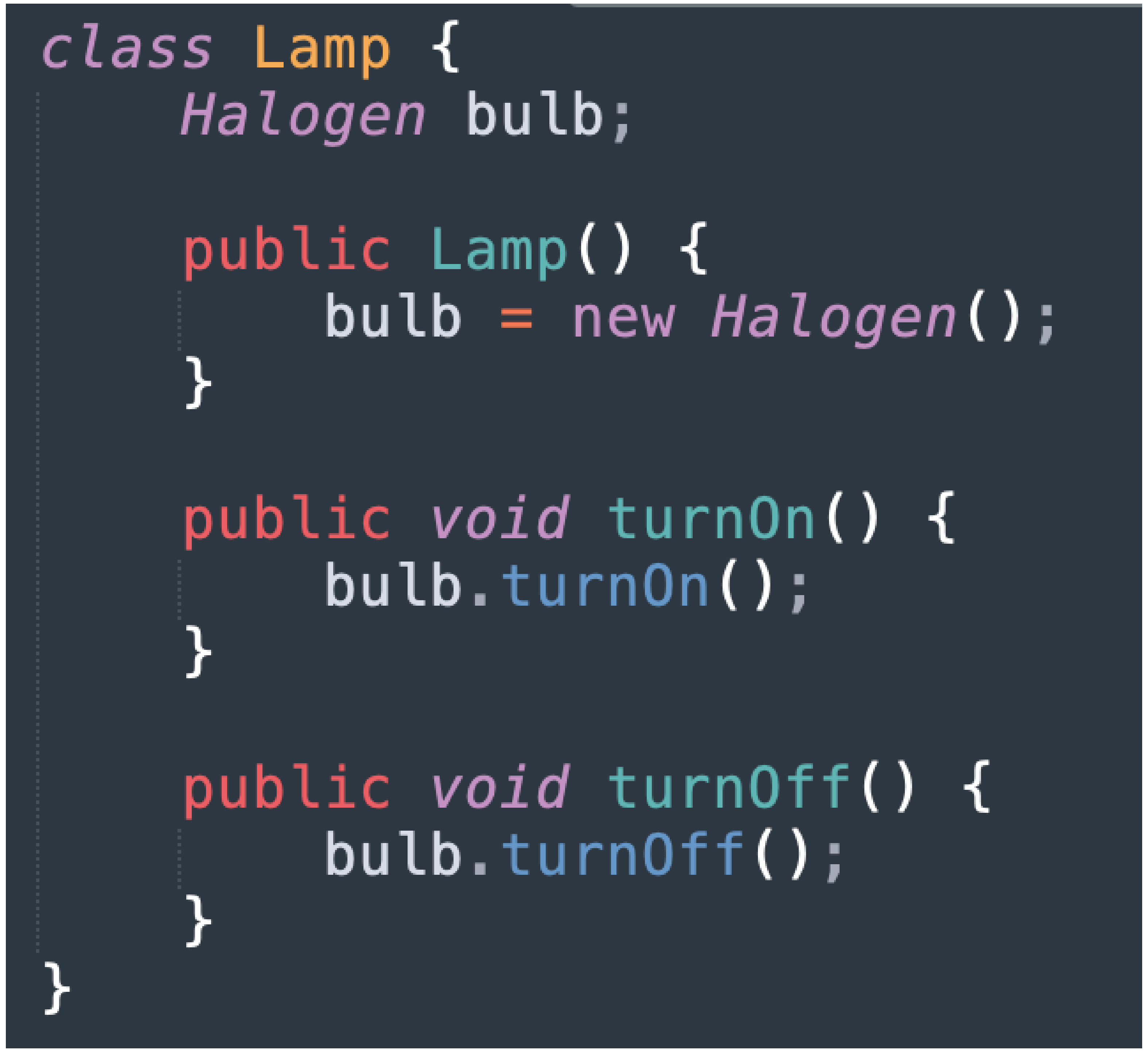
D — DEPENDENCY INVERSION PRINCIPLE (DIP)

High-level modules shouldn't depend on low-level ones. Both should depend on abstractions (interfaces)


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

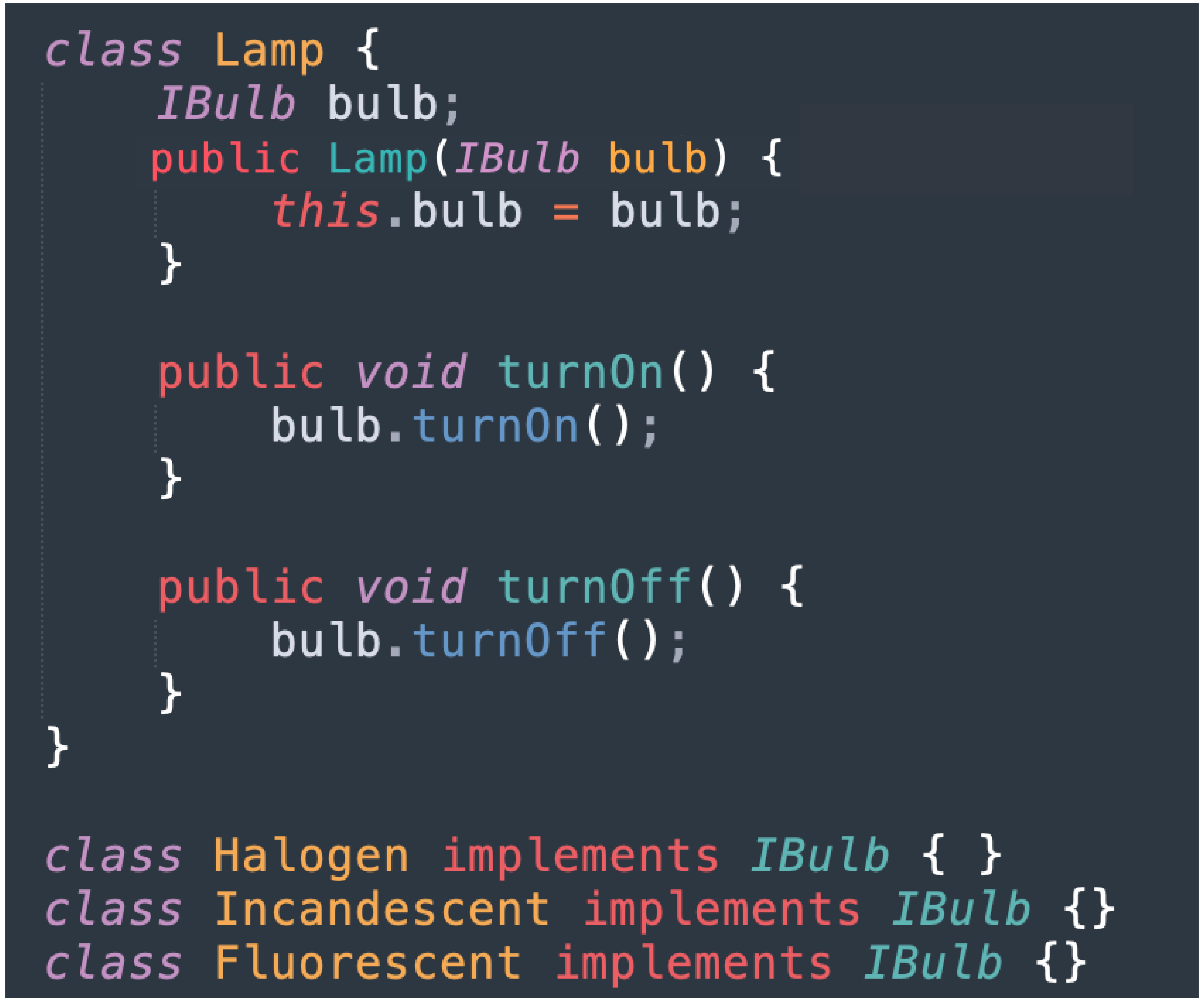
Credit: Abhishek Shukla

D — DEPENDENCY INVERSION PRINCIPLE (DIP)



```java
class Lamp {
    Halogen bulb;

    public Lamp() {
        bulb = new Halogen();
    }

    public void turnOn() {
        bulb.turnOn();
    }

    public void turnOff() {
        bulb.turnOff();
    }
}
```

D — DEPENDENCY INVERSION PRINCIPLE (DIP)



```java
class Lamp {
    IBulb bulb;
    public Lamp(IBulb bulb) {
        this.bulb = bulb;
    }

    public void turnOn() {
        bulb.turnOn();
    }

    public void turnOff() {
        bulb.turnOff();
    }
}

class Halogen implements IBulb { }
class Incandescent implements IBulb {}
class Fluorescent implements IBulb {}
```

WIRING IT TOGETHER

- One of the most common ways to solve the DIP violations is with Dependency injection

Let’s look at one more code example

TYING SOLID TOGETHER

- We’ve now seen SOLID violations in lot’s of different examples.
- Sometimes our code violates MANY principles/patterns but BIG refactors are a nightmare.
    - What do we do?
- Refactoring code to SOLID = small steps
    - Add interfaces
    - split responsibility slowly
    - tests that run before AND after

PAUSE & PROJECT REFLECT

Go an explore your own project.

Consider adding an interface to a business layer class or abstract class to a DSO

Take this new abstraction and use it for **complete** **true** isolated unit test

Let me show you how this might help us