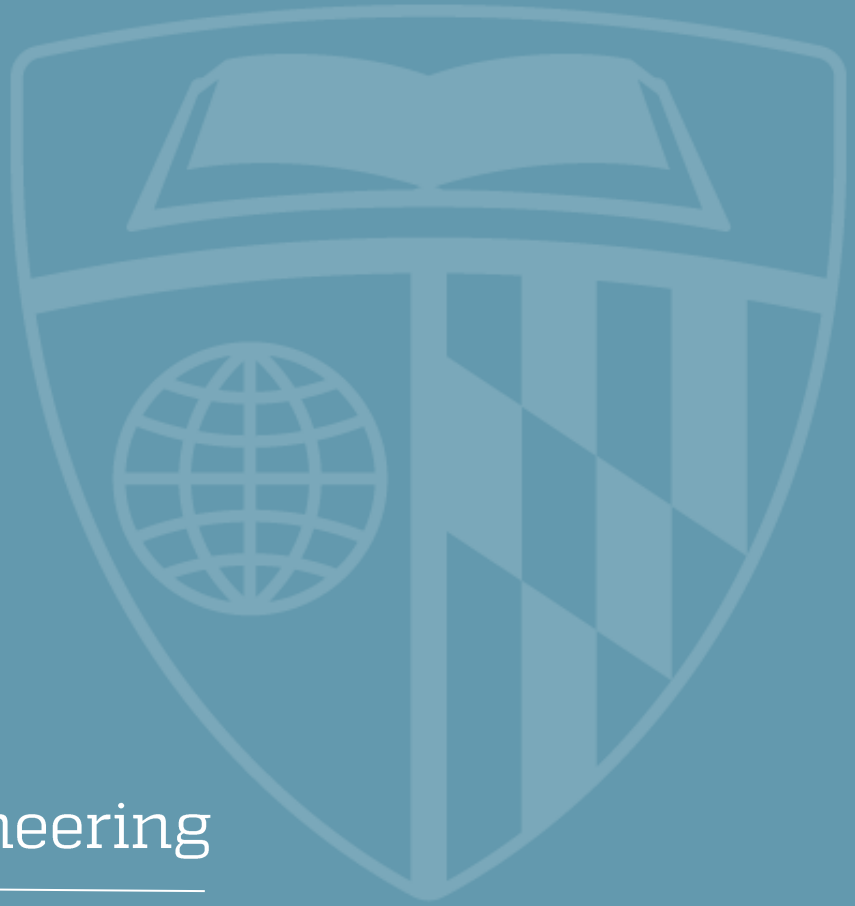


JOHNS HOPKINS
UNIVERSITY

EN.601.421 / EN.601.621

Object Oriented Software Engineering

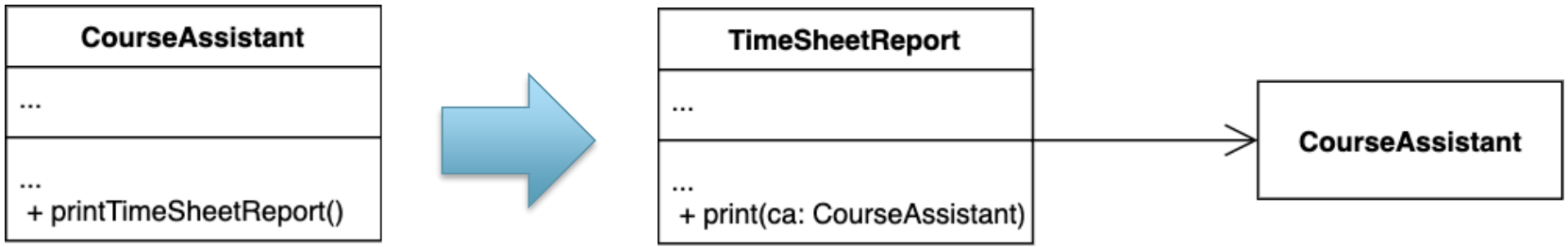
SOFTWARE IS ALWAYS IN FLUX!

Design Principles

- A good software design organizes the code in a way that it is "easy to understand, change, maintain and reuse.“
- SOLID Principles
  - Single Responsibility Principle
  - Open/Closed Principle
  - Liskov Substitution Principle
  - Interface Segregation Principle
  - Dependency Inversion Principle

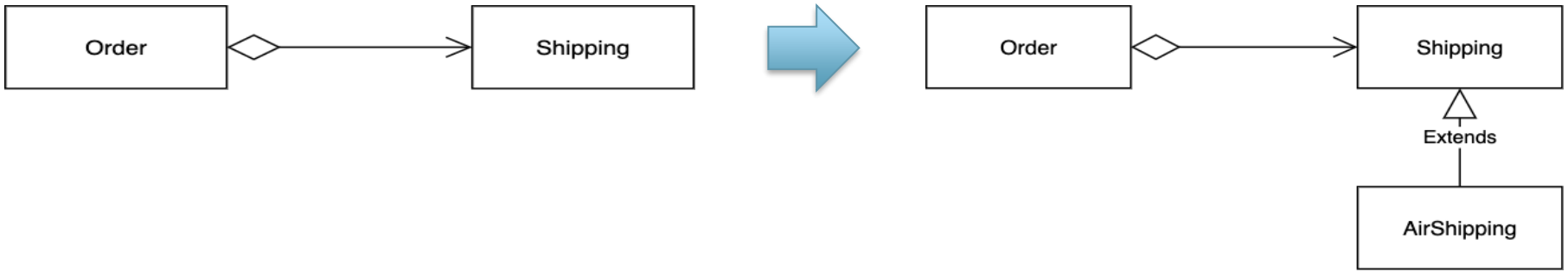
Single Responsibility Principle

- A class should have one, and only one, reason to change.
  - Goes hand-in-hand with “high cohesion”
  - **Applicable at many scales**: variables, methods, classes, software component and services
  - Ask yourself: **"what is the responsibility of this class?“**



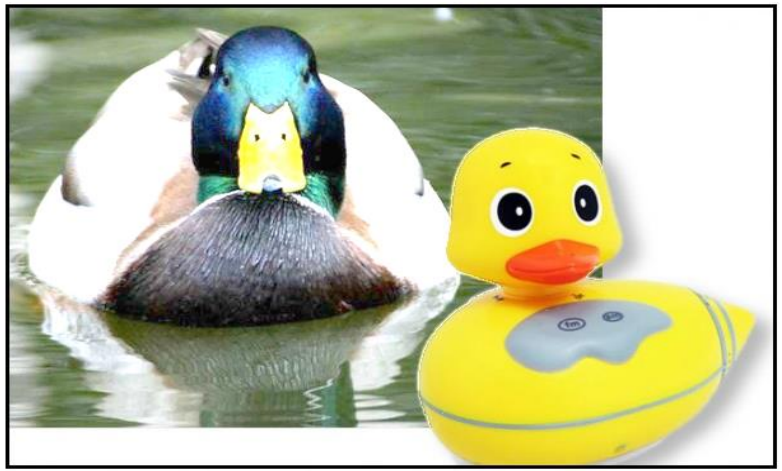
Open-Closed Principle

- Classes should be open for extension but closed for modification.
  - new functionality does not require a rewrite of existing code!



Liskov Substitution Principle

- Subclass (derived) class should be substitutable for their base (parent) class.



Liskov Substitution Principle

```
class Vehicle {
  String name;
  Engine engine;
  double speed;

  String getName() { ... }
  void setName(String n) { ... }
  double getSpeed() { ... }
  void setSpeed(double d) { ... }
  Engine getEngine() { ... }
  void setEngine(Engine e) { ... }
  void startEngine() { ... }
}
```

```
class Car extends Vehicle {
  @Override
  void startEngine() { ... }
}
```

```
class Bicycle extends Vehicle {
  @Override
  void startEngine() /*problem!*/
}
```

Interface Segregation Principle

- A class that implements an interface shouldn't be forced to implement methods it does not use.

Interface segregation principle

```
interface Shape {
    float calculateArea()
    float calculateVolume()
}


class Cube extends Shape {
    float side;
    @Override
    float calculateArea() {
        //Does not apply for Cube
        throw new UnsupportedOperationException();
    }
    @Override
     float calculateVolume() = {side * side * side}
}
```

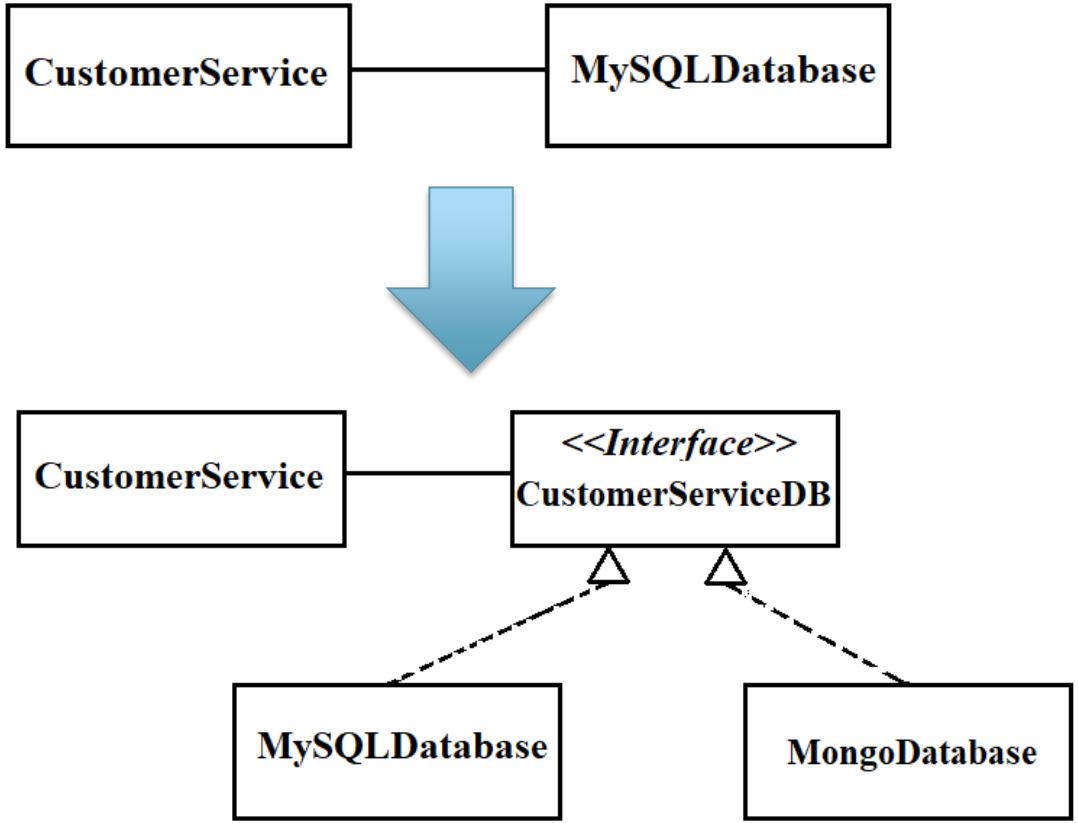
```
class Square extends Shape {
    float side;

    @Override
    float calculateArea() = {side * side}

    @Override
     float calculateVolume() {
        //Does not apply for Square
        throw new UnsupportedOperationException();
     }
}
```

Solution: segregate into *Shape2D* and *Shape3D* interfaces!

Dependency inversion principle

- High-level classes shouldn't have to change because low-level classes change.

Dependency Inversion Principle

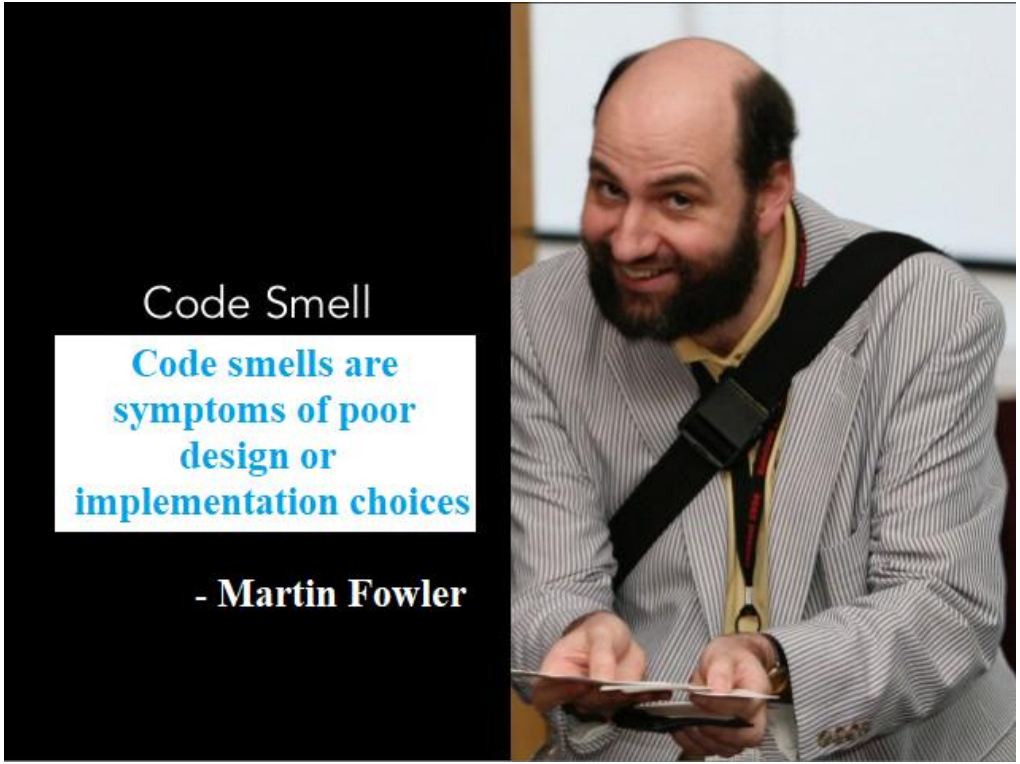


High-level modules should not depend on low-level modules. Both should depend on **abstractions**.

Refactoring

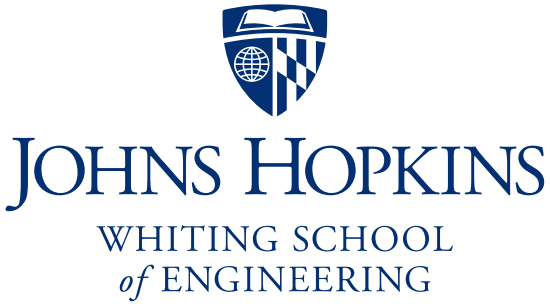
- The process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.

Code Smell


Code Smell
Code smells are symptoms of poor design or implementation choices
- Martin Fowler

Common Code Smells

- Large Class
- Long Method
- Data Clumps
- Duplicate Code
- Primitive Obsession
- Many more . . .


JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING