COMP 3550

6.2 — COMPOSITION & INTERFACE-BASED DESIGN

Week 6: Alternatives to Inheritance & Dependency Injection

COMPOSITION OVER INHERITANCE

Why choose composition?

- **Flexible**: Swap behaviors without rewriting entire hierarchies
- **Testable**: Mock individual parts for unit tests
- **Reusable**: Same behavior can be shared by unrelated classes

COMPOSITION OVER INHERITANCE

Why choose composition?

- **Flexible**: Swap behaviors without rewriting entire hierarchies
- **Testable**: Mock individual parts for unit tests
- **Reusable**: Same behavior can be shared by unrelated classes

```
class CreditCardPayment { ... }
class PayPalPayment extends CreditCardPayment { ... }
```

May force an *is-a* relationship that isn't true

COMPOSITION OVER INHERITANCE

Why choose composition?

- **Flexible**: Swap behaviors without rewriting entire hierarchies
- **Testable**: Mock individual parts for unit tests
- **Reusable**: Same behavior can be shared by unrelated classes

```
interface PaymentMethod {
    void pay(double amount);
}

class Order {
    private PaymentMethod payment;
    Order(PaymentMethod payment) { this.payment = payment; }
    void checkout() { payment.pay(100.0); }
}
```

Now we can test decoupled pieces, make different types of Payment methods, etc.

PUTTING IT TOGETHER: COMPOSITION + INTERFACE SEGREGATION

Many small plugs are better than one giant plug.

This can help us with Composition just as it did with Inheritance hierarchies

ISP:

- Prefer small, focused interfaces
- A client should only need to depend on the methods it actually uses

Goal:

- Give objects only the abilities they need
- Assemble behavior from small, focused parts
- Avoid tangled inheritance and big interfaces

```java
interface Studyable { void study(); }
interface Teachable { void teach(); }

class StudentRole implements Studyable {
    public void study() {
        System.out.println("Studying hard...");
    }
}

class TeacherRole implements Teachable {
    public void teach() {
        System.out.println("Teaching enthusiastically!");
    }
}

class StudentTeacher {
    private Studyable studentRole;
    private Teachable teacherRole;

    StudentTeacher(Studyable s, Teachable t) {
        this.studentRole = s;
        this.teacherRole = t;
    }

    void study() { studentRole.study(); }
    void teach() { teacherRole.teach(); }
}
```

STRATEGY PATTERN AS COMPOSITION

Picking the behavior you need at runtime

Strategy Pattern idea:

- Define a strategy interface for a family of behaviors
- Store a reference to the strategy in the context class
- Swap the behavior at runtime without changing the context class

```java
interface PaymentMethod {
    void pay(double amount);
}

class CreditCardPayment implements PaymentMethod {
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " with Credit Card");
    }
}

class PayPalPayment implements PaymentMethod {
    public void pay(double amount) {
        System.out.println("Paid $" + amount + " with PayPal");
    }
}

class PaymentProcessor {
    private PaymentMethod method;

    PaymentProcessor(PaymentMethod method) {
        this.method = method;
    }

    void process(double amount) {
        method.pay(amount);
    }
}
```

STRATEGY PATTERN AS COMPOSITION

Picking the behavior you need at runtime

Strategy Pattern :

- At runtime:

```
PaymentProcessor processor = new PaymentProcessor(new PayPalPayment());
processor.process(100.0);
```

Other strategies exist! We will look at some in these videos but not all. Nowhere near all:

Check out RefactoringGuru for a full list with descriptions and examples!

https://refactoring.guru/design-patterns/catalog

TESTING BECOMES EASIER WITH STRATEGY

Just swap in a mock, no rewiring required

Why Strategy helps testing:

- You can inject a mock or fake implementation of the strategy
- No need to spin up external systems (e.g., real payment gateways)
- Tests focus on the class's behavior, not the dependency's complexity

```
class MockPayment implements PaymentMethod {
    public void pay(double amount) {
        System.out.println("[TEST] Pretend to pay $" + amount);
    }
}

// In test:
PaymentProcessor processor = new PaymentProcessor(new MockPayment());
processor.process(42.0);
```

PROJECT PAUSE AND REFLECT

It’s easy to overuse this idea and add interfaces and composition **everywhere**

Just like everything else in this course *IT DEPENDS*

Sit down with your group and discuss a few key seams where using interfaces and composition would increase flexibility and testability of your code. Make some tickets and try to implement these refactors.