Composition Over Inheritance

Introduction to Composition Over Inheritance Principle

Composition over Inheritance is a design principle that suggests favoring composition (using objects of other classes) over inheritance (extending classes) when designing systems. The idea is to achieve code reuse and flexibility by composing objects with the desired behaviors, rather than relying on rigid class hierarchies.

Why Composition Over Inheritance is Important

1. **Flexibility:** Composition allows you to change the behavior of a class at runtime by composing it with different objects, whereas inheritance locks you into a fixed behavior defined by the parent class.

2. **Avoids Inheritance Pitfalls:** Inheritance can lead to issues like the fragile base class problem, where changes in the base class can unintentionally affect derived classes.

3. **Promotes Encapsulation:** Composition helps keep the internal details of a class hidden, as the composed objects handle specific responsibilities.

4. **Simplifies Testing:** Composed objects can be tested independently, leading to more modular and maintainable code.

How to Apply Composition Over Inheritance

- **Identify Behaviors:** Break down the desired functionality into smaller, reusable behaviors.

- **Create Separate Classes:** Implement these behaviors in separate classes that can be composed into more complex objects.

- **Delegate Responsibilities:** Use composition to delegate specific responsibilities to different objects, rather than relying on a single inheritance hierarchy.

Example: Without Composition Over Inheritance

```
public class Vehicle {

    public void startEngine() {
        System.out.println("Engine started");

    }

}



public class Car extends Vehicle {

    public void openTrunk() {

        System.out.println("Trunk opened");

    }

}



public class Bicycle extends Vehicle {

    // Bicycle doesn't have an engine, but it inherits the startEngine method

}
```

In this example, the `Bicycle` class inherits from `Vehicle`, but a bicycle doesn?t have an engine, so the `startEngine` method is irrelevant. This is a classic example of where inheritance can lead to awkward or incorrect designs.

Example: With Composition Over Inheritance

```
public interface Engine {

    void start();

}



public class CarEngine implements Engine {

    public void start() {

        System.out.println("Car engine started");

    }
}


public class Bicycle {

    // Bicycle doesn't have an engine, so no need for an Engine object

}


public class Car {

    private Engine engine;


    public Car(Engine engine) {

        this.engine = engine;

    }


    public void startEngine() {

        engine.start();

    }


    public void openTrunk() {

        System.out.println("Trunk opened");

    }

}
```

In this refactored version, the `Car` class composes an `Engine` object, allowing for flexibility. The `Bicycle` class doesn?t need an engine, so it doesn?t include one, avoiding the pitfalls of inheritance.

Key Takeaways

- **Reuse through Composition:** Instead of extending a class to gain behavior, create small, focused classes that can be composed together.

- **Reduce Coupling:** Composition reduces the tight coupling between classes, making your codebase more modular and easier to change.

- **Enhance Flexibility:** With composition, you can easily change the behavior of an object at runtime by changing the composed objects.

When to Use Inheritance

While composition is generally preferred, there are scenarios where inheritance is appropriate:

- **Is-a Relationship:** When there?s a clear "is-a" relationship, inheritance can be used. For example, a `Dog` is an `Animal`.

- **Shared Behavior:** When multiple classes share a lot of common behavior and properties, inheritance can help avoid code duplication.

Would you like to dive deeper into this principle with more complex examples or explore how to balance inheritance and composition in different scenarios?