# SE 350 - Object-Oriented Software Development

## Software Design Patterns: Exam Review

Instructor: Stefan Mitsch

# Topics

- Final exam is comprehensive: all course topics are relevant!
  - OOP basics and principles
  - UML class diagrams
  - SOLID
  - Design Patterns
  - Refactoring (code smells)

# Visibility Modifiers

- Access modifiers: `private`, `protected`, `public`
- Non-access modifiers: `final`, `static`

```
public class Demo {
  private final int x;
  private Demo(Demo other) {
    this.x = other.x;
  }
  public static Demo copy(Demo other) {
    return new Demo(other);
  }
}
```

- Why can the constructor access the `private` field of `other`?
- Why can the method `copy` access the `private` constructor?
- Why can the constructor set the `final` field `x`?

# Nested Classes

```
public class Outer {
  private int x;
  public Outer(int x) { this.x = x; }
  private static int sx;

  public static class StaticNested {
    private int x;
    public StaticNested(int x) { this.x = x; }
    public String toString() {
      return "sx=" + sx + ", x=" + x;
    }
  }

  public class Inner {
    private int x;
    public Inner(int x) { this.x = x;}
    public String toString() {
      "sx=" + sx + ", x=" + x + ", Outer.x=" + Outer.this.x;
    }
  }

  // continued in right column
```

```
public static void main(String[] args) {
  Outer o = new Outer(1);
  StaticNested sn = new Outer.StaticNested(2);
  Inner i = o.new Inner(3);
  System.out.println(sn.toString());
  System.out.println(i.toString());
}
```

- Review: `static` vs. non-`static` nested class
- Instantiation
- Who can access what?

# Open-Closed Principle

```
public class Calculator {
  public double area(List<Rectangle> rectangles) {
    double total = 0;
    for (Rectangle r : rectangles) {
      total += r.height * r.width;
    }
    return total;
  }
}
public class Rectangle {
  public double width;
  public double height;
}
```

- Code on the left/right violates the Open-Closed Principle?
- Why/why not?

```
public class Calculator {
  public double area(List<Object> shapes) {
    double total = 0;
    for (Object s : shapes) {
      if (s instanceOf Rectangle) {
        Rectangle r = (Rectangle)s;
        total += r.height * r.width;
      } else if (s instanceOf Circle) {
        Circle c = (Circle)s;
        total += Math.PI * c.radius * c.radius;
      } else throw new IllegalArgumentException();
    }
    return total;
  }
}
public class Rectangle {
  public double width;
  public double height;
}
public class Circle {
  public double radius;
}
```

# Open-Closed Principle

```
public class Calculator {
  public double area(List<Shape> shapes) {
    double total = 0;
    for (Shape s : shapes) total += area(s);
    return total;
  }
  private double area(Shape s) {
    switch (s.type()) {
      case "Rectangle":
        Rectangle r = (Rectangle)s;
        return r.height * r.width;
      case "Circle":
        Circle c = (Circle)s;
        return Math.PI * c.radius * c.radius;
      default: throw new IllegalArgumentException();
    }
  }
}
public interface Shape {
  String type();
}
public class Rectangle implements Shape {
  public double width;
  public double height;
  public String type() { return "Rectangle"; }
}
public class Circle implements Shape {
  public double radius;
  public String type() { return "Circle"; }
}
```

```
public class Calculator {
  public double area(List<Shape> shapes) {
    double total = 0;
    for (Shape s : shapes) total += s.area();
    return total;
  }
}
public interface Shape {
  double area();
}
public class Rectangle implements Shape {
  public double width;
  public double height;
  public double area() { return height * width; }
}
public class Circle implements Shape {
  public double radius;
  public double area() { return Math.PI * radius * radius; }
}
```

- Code on the left/right violates the Open-Closed Principle?
- Why/why not?

# Liskov Substitution Principle

```java
public class Base {
  // @requires: 0 <= x <= 10
  // @guarantees: 0 <= result <= 20
  public int getValue(int x) {
    if (!(0 <= x && x <= 10)) throw new IllegalArgumentException("Expected 0<=x<=10");
    int result = 2*x;
    assert 0 <= result && result <= 20;
    return result;
  }
}
```

```java
public class Sub extends Base {
  @Override
  public int getValue(int x) {
    // ...
  }
}
```

Given the annotated contract of method `getValue`: which of the following implementations in class `Sub` may violate the Liskov Substitution Principle?

- `throw new UnsupportedOperationException();`?
- `return 0;`?
- `return x;`?
- `return 3*x;`?
- `return 100;`?

- It is ok the weaken the input requirement `@requires: 0 <= x <= 15`?
- It is ok to strengthen the output guarantee `@guarantees: 0 <= result <= 7`?

# Refactoring to SOLID

```
public class HighLevel {
  private LowLevel service;
}
```

```
public interface Service {}

public class HighLevel {
  private Service service;
}

public LowLevel implements Service {}
```

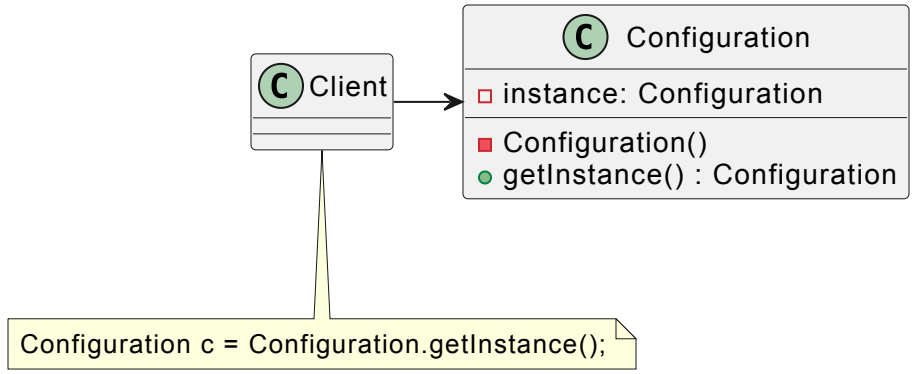
Which of the following statements is true?

- Code left violates the Dependency Inversion Principle?
- Code right violates the Dependency Inversion Principle?
- Code left violates the Open-Closed Principle?
- Code left violates the Interface Segregation Principle?
- Code right violates the Liskov Substitution Principle?

# Design Pattern Use Cases

❓ I want to create an application that uses a single globally accessible configuration. Which of the following design patterns should I use?

- Abstract Factory
- Singleton
- Decorator
- Template Method
- Composite




- Which design pattern is used in this class diagram?

# Design Pattern Use Cases

❓ I want to let my users decide which of several possible sorting algorithms they want to use with my collection implementation. Which of the following design patterns should I use?

- Abstract Factory
- Singleton
- Decorator
- Strategy
- Composite

▶ Class Diagram

# Design Pattern Use Cases

❓ I want to sometimes add behavior to one of my objects.
Which of the following design patterns should I use?

- Abstract Factory
- Singleton
- Decorator
- Strategy
- Composite

▶ Class Diagram

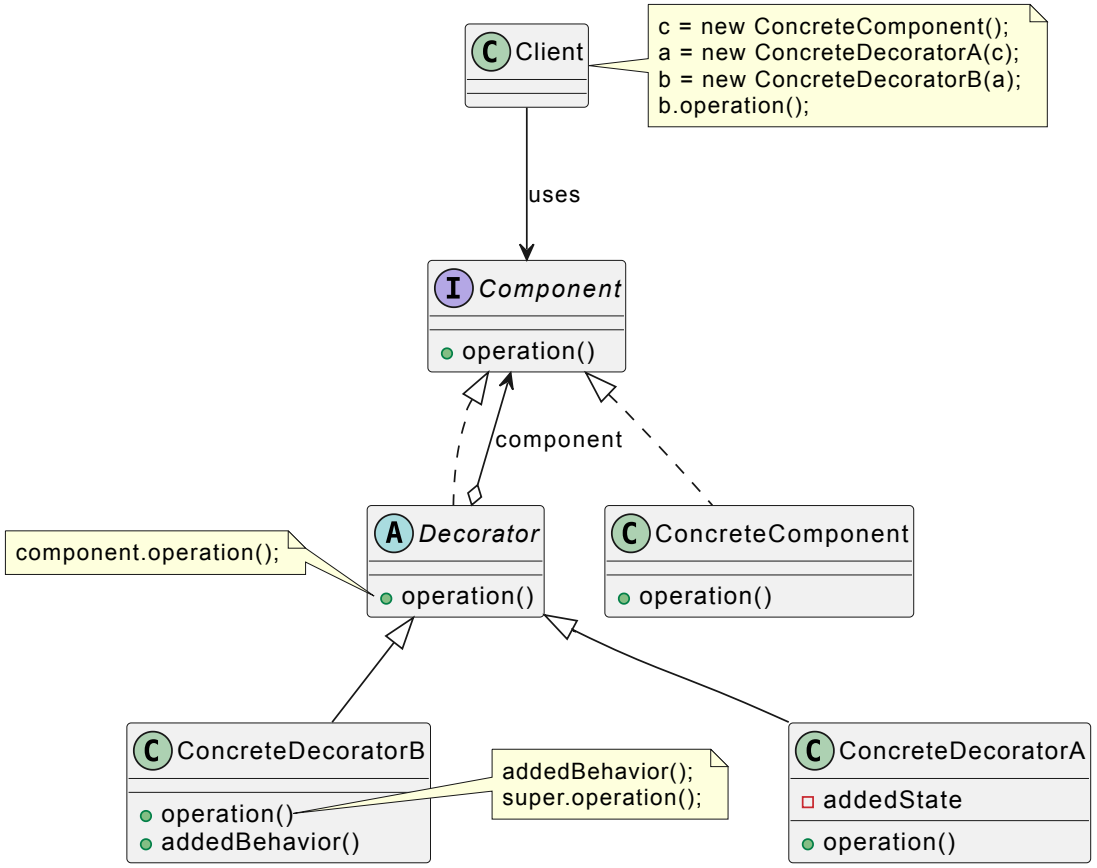
# Template Method Modifiers

```
public /* (1) */ class Demo {
  /* (2) */ int calculate() {
    return 2 * doCalcX();
  }
  /* (3) */ int doCalcX();
}
```

Fill in the blank mutability and visibility modifiers in the template method implementation above.

- (1): `final` or `abstract`?
- (2): `private/protected/public` and/or `final` and/or `abstract`?
- (3): `private/protected/public` and/or `final` and/or `abstract`?

# Decorator Participants




- The `Decorator` abstract base class references the wrapped component?
- The `Decorator` manages a single instance?
- The `Decorator` is a base class for concrete decorators?
- Wrapped components can be other decorators or a concrete component or both?
- We can decorate a decorator?

# Design Pattern Implementation

```
public interface Entity {
  void doSomething();
}

public class Department implements Entity {
  private List<Entity> children;
  public Department(List<Entity> children) {
    this.children = children;
  }
  public void doSomething() {
    for (Entity e : children) e.doSomething();
  }
}

public class Employee implements Entity {
  public void doSomething() { /* something */ }
}
```

The code to the left is an implementation of which design pattern?

- Composite
- Singleton
- Decorator
- Memento
- Command

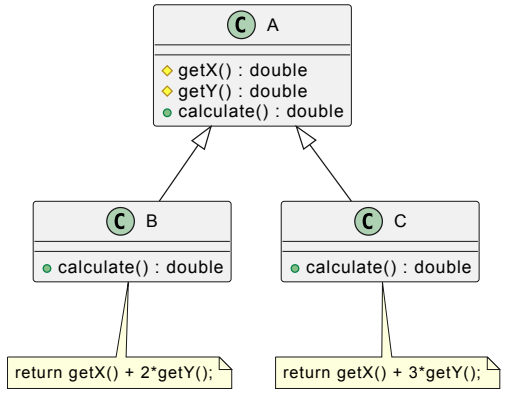
# Design Pattern Implementation

```
public class A {
  private int x;

  public static class B {
    private int x;
    private B(int x) { this.x = x; }
  }

  public B save() { return new B(x); }
  public void restore(B s) { this.x = s.x; }
}
```

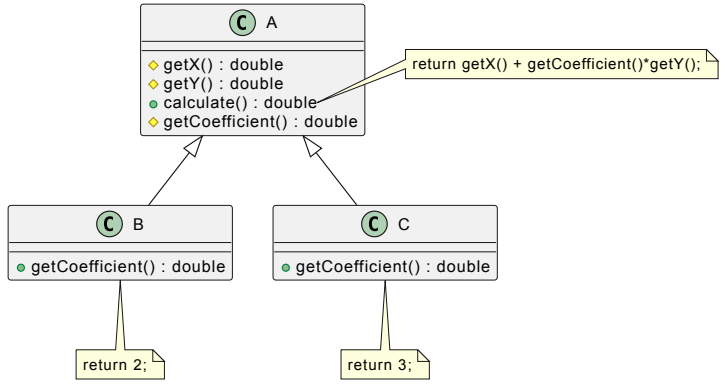
The code to the left is an implementation of which design pattern?

- Composite
- Singleton
- Decorator
- Memento
- Command

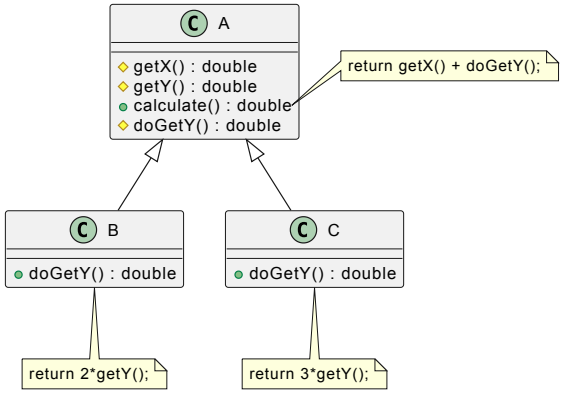
# Design Pattern Refactoring




(1)




(2)




(3)

- The code in column (1) duplicates the structure of an algorithm in its subclasses?
- Column (2) implements the Template Method design pattern?
- Column (3) implements the Template Method design pattern?
- The intent of the Template Method design pattern is to implement an algorithm structure once and let subclasses implement varying parts?
- A concrete subclass in Template Method implements *invariant* steps?

# Code Smells

```
public class Incrementor {
  private Printer p;
  private final int i;

  public Incrementor(int i) { this.i = i; }

  public Printer getPrinter() { return p; }
  public void setPrinter(Printer p) { this.p = p; }

  public int getI() { return i; }
  public void doubleIncrement() { i += i; }
  public void identityIncrement() { i = 0; }

  public int increment(int y) {
    p.print(y + "+" + i + " == " + (y + i));
    return y + i;
  }
}
```

Which of the following code smells is present in the code on the left

- Excessively long class?
- Excessively long method?
- Long parameter list?
- Temporary field used and initialized only sometimes?
- A small number of classes monopolizes processing?
- Parallel inheritance hierarchies?
- Excessively access data of other objects?