Linking Cohesion and Coupling to SOLID

How high cohesion and low coupling leads to code which follows the SOLID principles. And the other way around.

Patrick Ronecker

Agenda

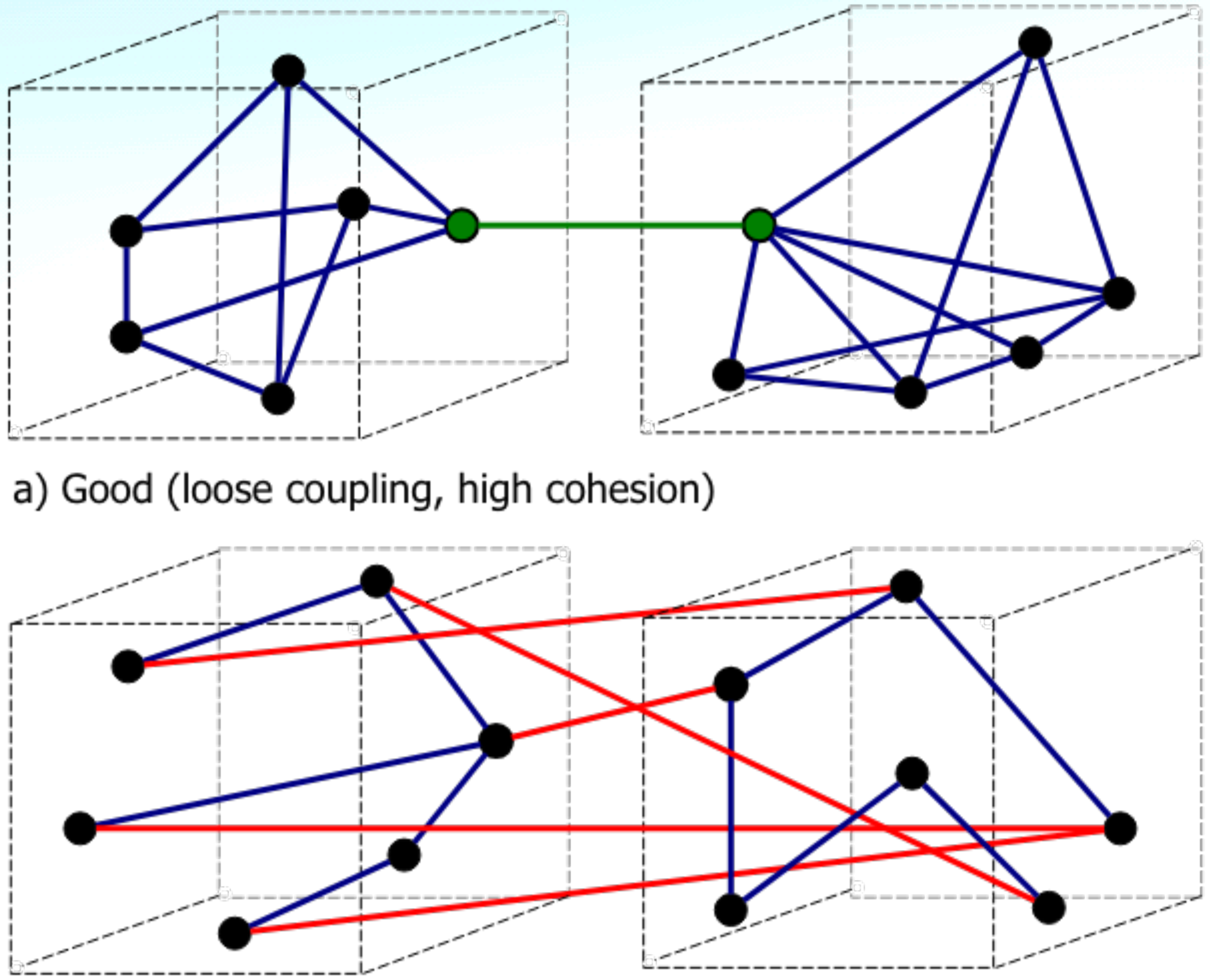
- Theory
  - Coupling & Cohesion
  - SOLID
- Examples
- Final Thoughts

Cohesion & Coupling

says how strongly related and coherent are the responsibilities within modules (classes) of an application

is the degree of interdependence between modules (classes) of an application

HIGH ↑



a) Good (loose coupling, high cohesion)

b) Bad (high coupling, low cohesion)

↓ LOW

Single Responsibility Principle (SRP)

- A class should have only one reason to change
- Focus only on one job or responsibility

↓

- Definition of a highly cohesive class
- High cohesion naturally aligns with the SRP

Open/Closed Principle (OCP)

Software entities should be open for extension but closed for modification.

- Low coupled design gives us flexibility and maintainability
- No tight link between software entities
- Highly cohesive classes are easier to extend
- Extension is possible without modifying existing code

Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

- Helps us with loose coupling
- Reduction of side effects of each component (Goal of LSP)
- High Cohesion and low coupling leads to a design of small and independent components, which are grouped by their functionality

Interface Segregation Principle (ISP)

No client should be forced to depend on methods it does not use.

- Supported by high cohesion in the design of interfaces
- Interfaces focused around a specific set of related functionalities
- Clients only need to know the relevant interfaces
- Leeds to reduced coupling and increased coherence

Dependency Inversion Principle (DIP)

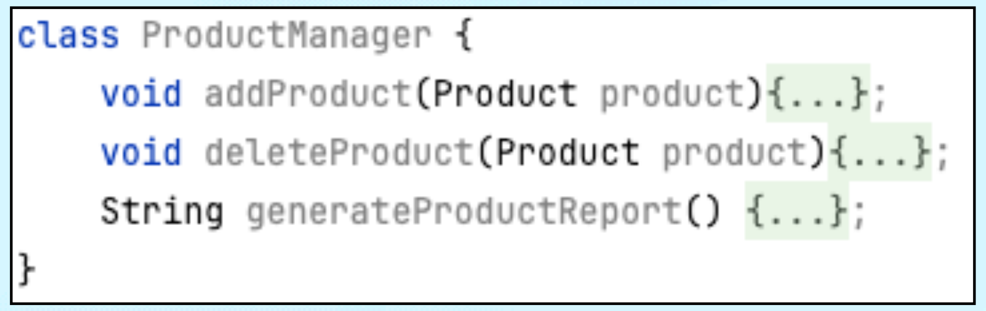
High-level modules should not depend on low-level modules.
Both should depend on abstractions.

Abstractions should not depend upon details, but details should depend upon abstractions.

- Low coupling is a fundamental aspect
- Interaction between classes through abstract interfaces instead of concrete implementations
- Reduction of direct dependencies

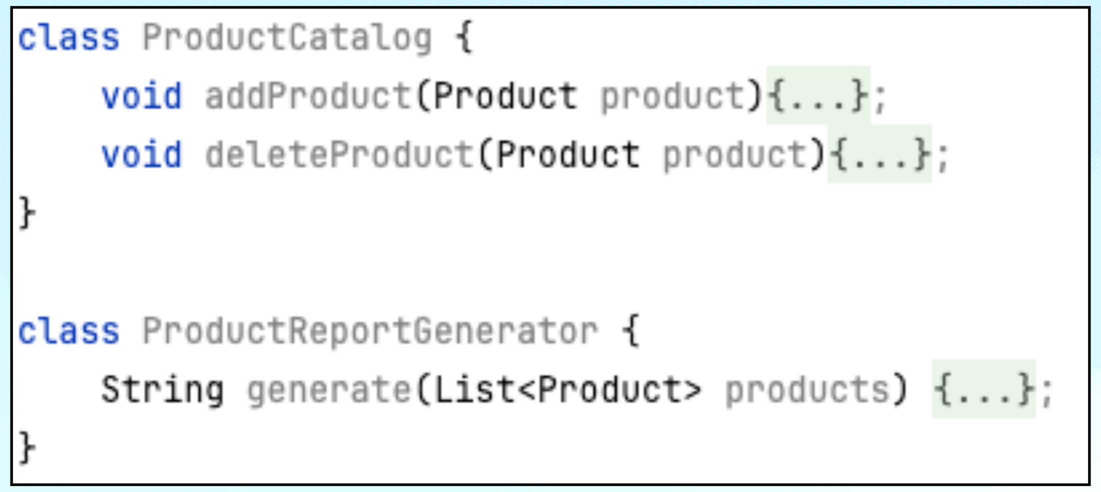
Example 1: SRP & OCP

E-commerce System



```
class ProductManager {
    void addProduct(Product product){...};
    void deleteProduct(Product product){...};
    String generateProductReport() {...};
}
```

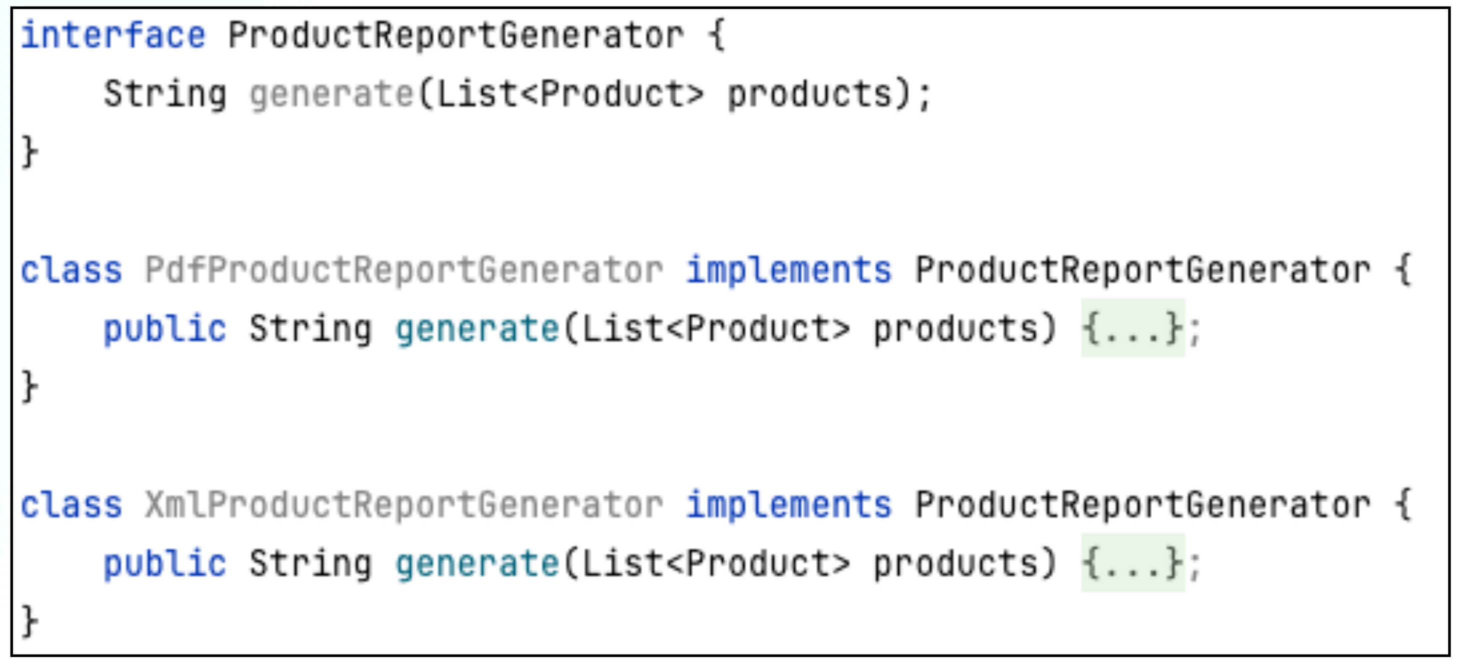
SRP



```
class ProductCatalog {
    void addProduct(Product product){...};
    void deleteProduct(Product product){...};
}

class ProductReportGenerator {
    String generate(List<Product> products) {...};
}
```

Cohesion

OCP

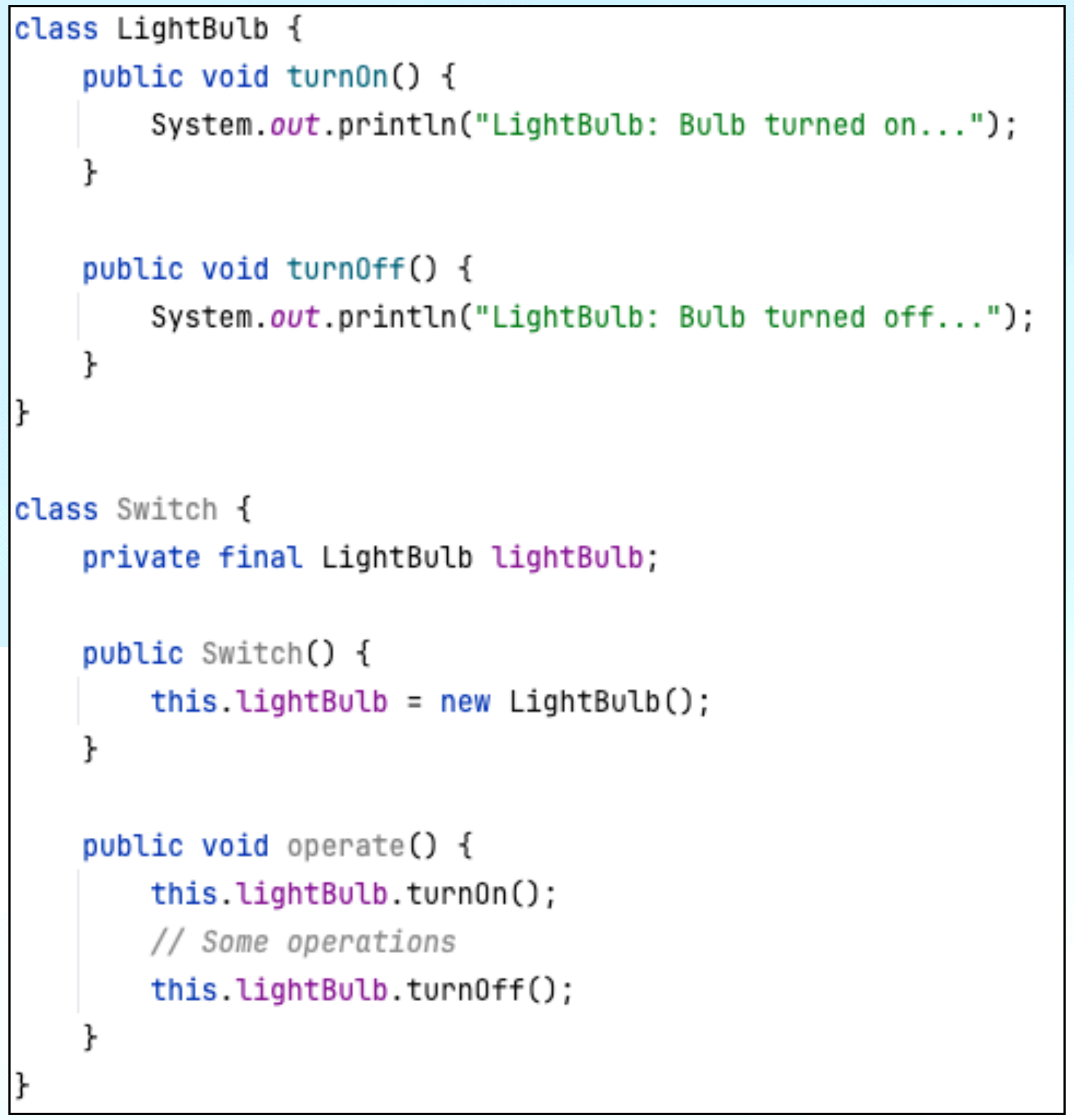


```
interface ProductReportGenerator {
    String generate(List<Product> products);
}

class PdfProductReportGenerator implements ProductReportGenerator {
    public String generate(List<Product> products) {...};
}

class XmlProductReportGenerator implements ProductReportGenerator {
    public String generate(List<Product> products) {...};
}
```

Coupling

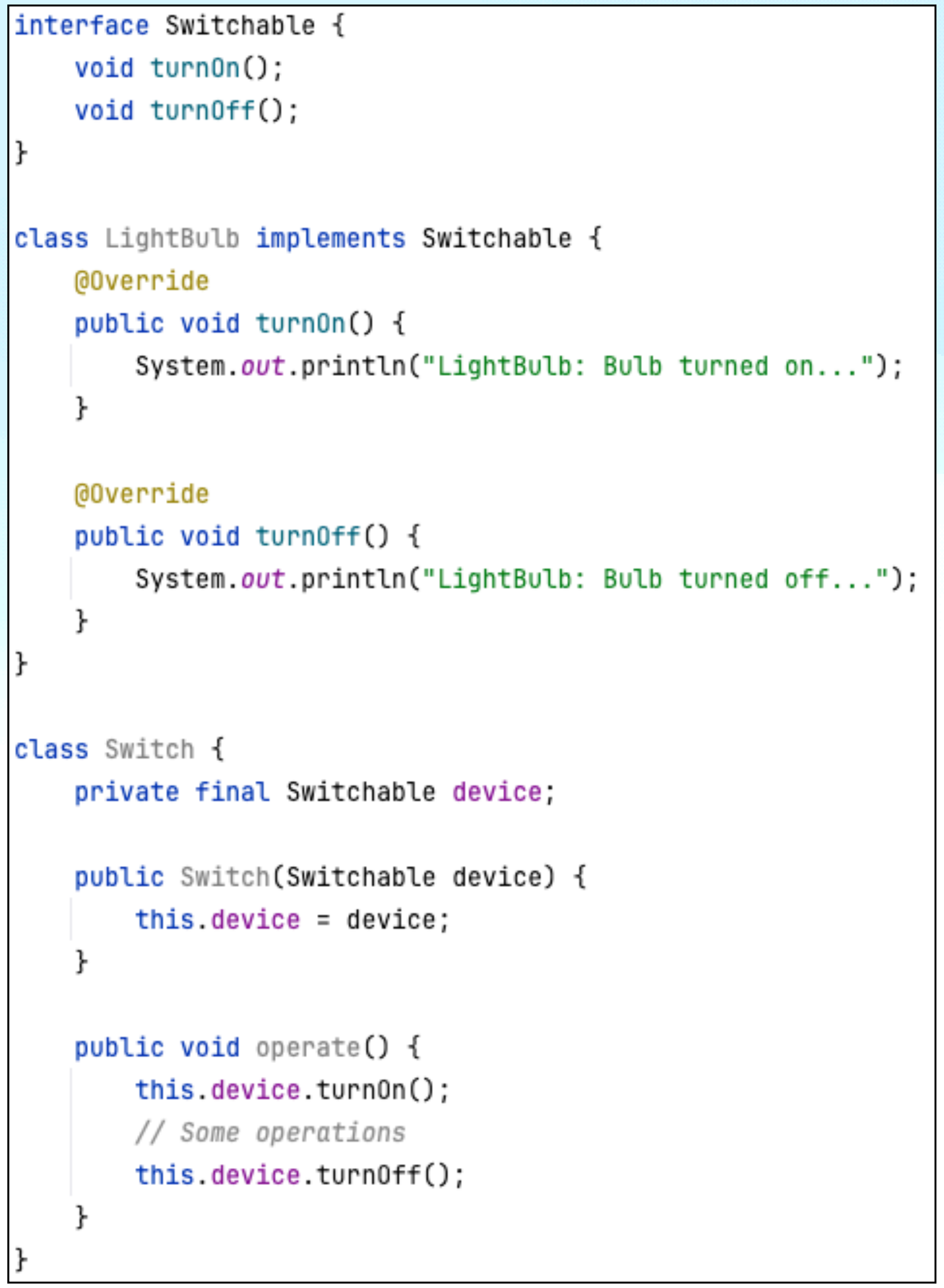
Example 2: DIP

Lightswitch



```java
class LightBulb {
    public void turnOn() {
        System.out.println("LightBulb: Bulb turned on...");
    }

    public void turnOff() {
        System.out.println("LightBulb: Bulb turned off...");
    }
}

class Switch {
    private final LightBulb lightBulb;

    public Switch() {
        this.lightBulb = new LightBulb();
    }

    public void operate() {
        this.lightBulb.turnOn();
        // Some operations
        this.lightBulb.turnOff();
    }
}
```

DIP

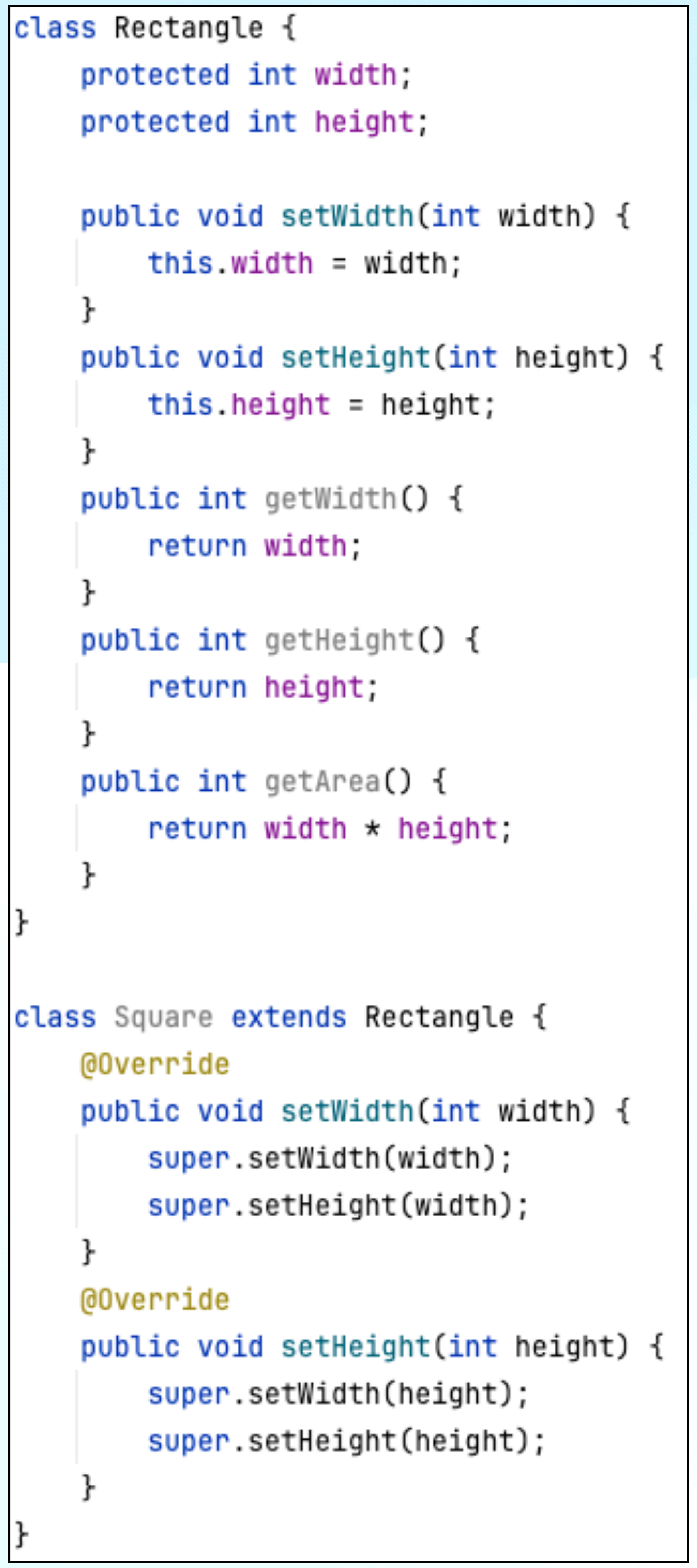


```java
interface Switchable {
    void turnOn();
    void turnOff();
}

class LightBulb implements Switchable {
    @Override
    public void turnOn() {
        System.out.println("LightBulb: Bulb turned on...");
    }

    @Override
    public void turnOff() {
        System.out.println("LightBulb: Bulb turned off...");
    }
}

class Switch {
    private final Switchable device;

    public Switch(Switchable device) {
        this.device = device;
    }

    public void operate() {
        this.device.turnOn();
        // Some operations
        this.device.turnOff();
    }
}
```

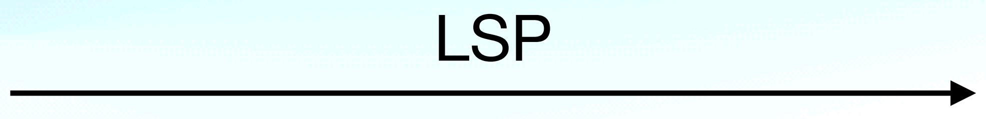
Coupling

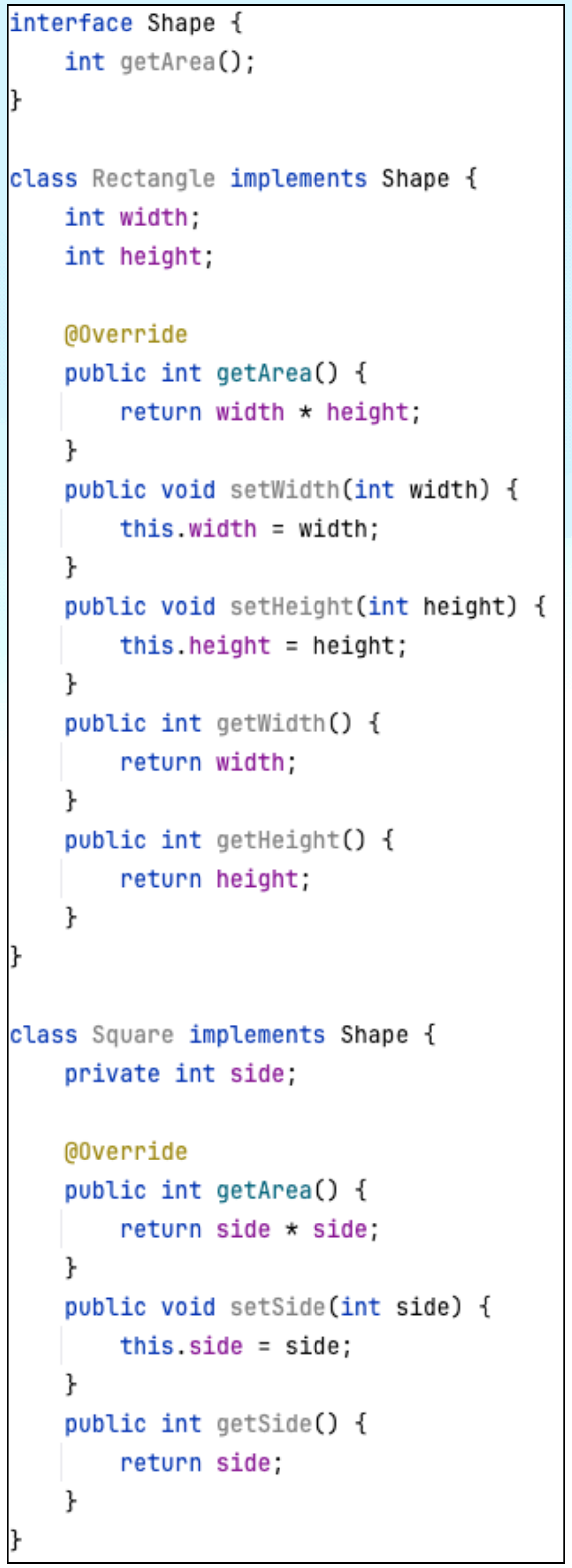
Example 3: LSP

Rectangle & Square

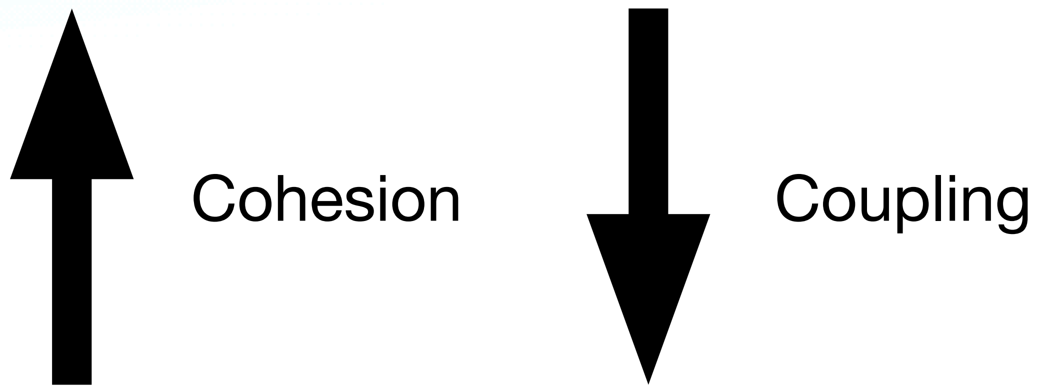


```java
class Rectangle {
    protected int width;
    protected int height;

    public void setWidth(int width) {
        this.width = width;
    }
    public void setHeight(int height) {
        this.height = height;
    }
    public int getWidth() {
        return width;
    }
    public int getHeight() {
        return height;
    }
    public int getArea() {
        return width * height;
    }
}

class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }
    @Override
    public void setHeight(int height) {
        super.setWidth(height);
        super.setHeight(height);
    }
}
```



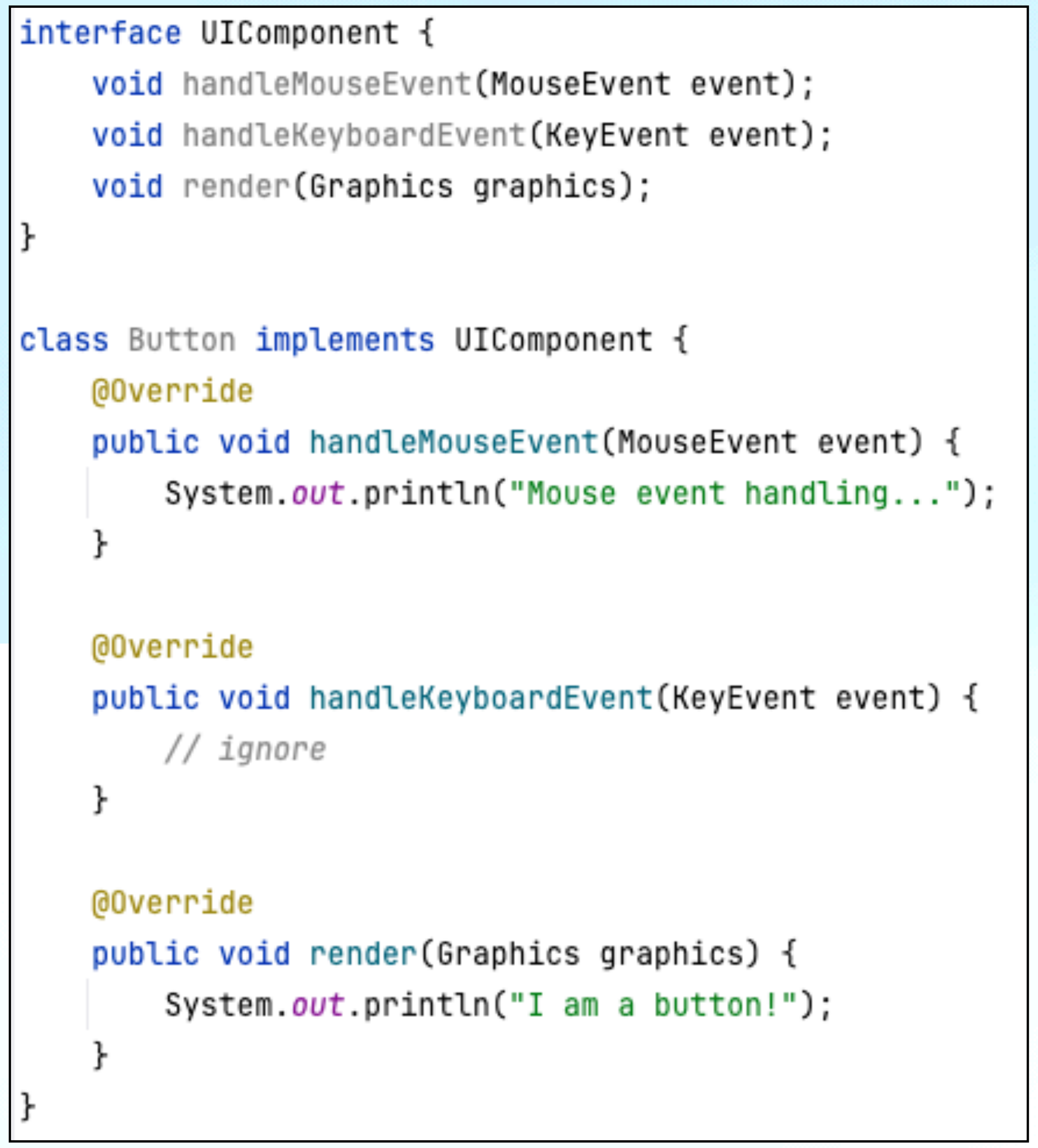


```java
interface Shape {
    int getArea();
}

class Rectangle implements Shape {
    int width;
    int height;

    @Override
    public int getArea() {
        return width * height;
    }
    public void setWidth(int width) {
        this.width = width;
    }
    public void setHeight(int height) {
        this.height = height;
    }
    public int getWidth() {
        return width;
    }
    public int getHeight() {
        return height;
    }
}

class Square implements Shape {
    private int side;

    @Override
    public int getArea() {
        return side * side;
    }
    public void setSide(int side) {
        this.side = side;
    }
    public int getSide() {
        return side;
    }
}
```



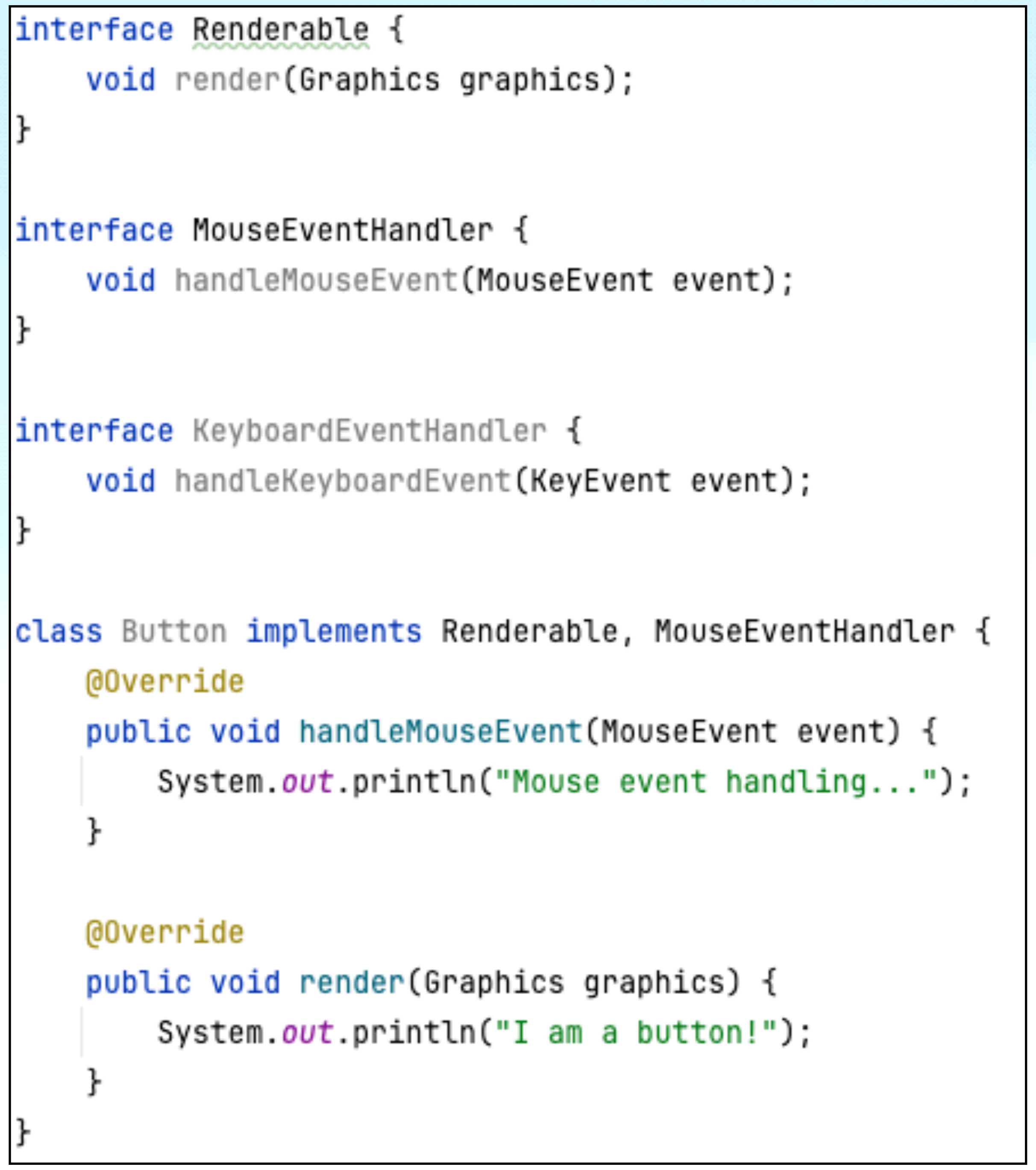
Example 4: ISP

User Interface Component Library



```java
interface UIComponent {
    void handleMouseEvent(MouseEvent event);
    void handleKeyboardEvent(KeyEvent event);
    void render(Graphics graphics);
}

class Button implements UIComponent {
    @Override
    public void handleMouseEvent(MouseEvent event) {
        System.out.println("Mouse event handling...");
    }

    @Override
    public void handleKeyboardEvent(KeyEvent event) {
        // ignore
    }

    @Override
    public void render(Graphics graphics) {
        System.out.println("I am a button!");
    }
}
```

ISP →



```java
interface Renderable {
    void render(Graphics graphics);
}

interface MouseEventHandler {
    void handleMouseEvent(MouseEvent event);
}

interface KeyboardEventHandler {
    void handleKeyboardEvent(KeyEvent event);
}

class Button implements Renderable, MouseEventHandler {
    @Override
    public void handleMouseEvent(MouseEvent event) {
        System.out.println("Mouse event handling...");
    }

    @Override
    public void render(Graphics graphics) {
        System.out.println("I am a button!");
    }
}
```

↑ Cohesion ↓ Coupling

Final Thoughts

- We aim for high cohesion and fight against coupling
- No coupling is not achievable
- It is always a balancing act
- Yin-yang of software-design

Any questions?

Thank you for your attention.

Sources:

- Agile Technical Practices Distilled by Pedro Moreira Santos, Marco Consolaro, Alessandro Di Gioia
- https://en.wikipedia.org/wiki/Coupling_%28computer_programming%29
- https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html
- https://blog.cleancoder.com/uncle-bob/2020/10/18/Solid-Relevance.html

patrick.ronecker@css.ch