
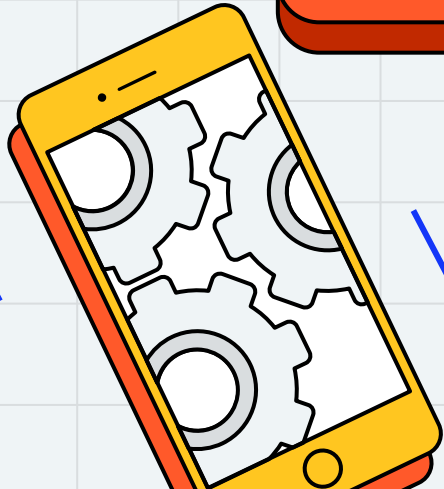




Application

Programming



Hend Alkittawi





SOLID Principles

Applying SOLID Principles In Object
Oriented Design

WHY SOLID PRINCIPLES?

- Watch [this video](#) from the Coursera course *Software Development Processes and Methodologies*
- The content for this lecture is based on a series of papers/book chapters by Robert Martin
- You might find these references useful
 - [Uncle Bob SOLID principles](#)
 - [SOLID Design Principles with Java Examples | Clean Code and Best Practices | Geekific](#)

SYMPTOMS OF ROTTING DESIGNS

- **Rigidity**
 - Difficult to change in even simple ways
 - Changes cause a cascade of issues in dependent modules
- **Immobility**
 - Inability to reuse modules
- **Fragility**
 - Tendency for changes to cause problems in many areas
 - New problems often in areas with no conceptual relationship to the original change
- **Viscosity**
 - Viscosity of design → It is harder to make changes that preserve the original design
 - Viscosity of environment → Development environment is slow and inefficient

CHANGING REQUIREMENTS

- Requirements change in a way that was not anticipated by the original design
- The requirements document is the most volatile document in a project
- We need to make our designs resilient to changes and protect them from rotting
- Tree Comic

DEPENDENCY MANAGEMENT

- Rot is caused by changes that introduce new and unplanned for dependencies
- Rigidity, Fragility, Immobility, and Viscosity are directly or indirectly caused by **improper dependencies** between software modules
- To avoid rot, **dependencies between modules in an application must be managed**
- This management consists of dependency firewalls across which **dependencies do not propagate**

SOLID PRINCIPLES

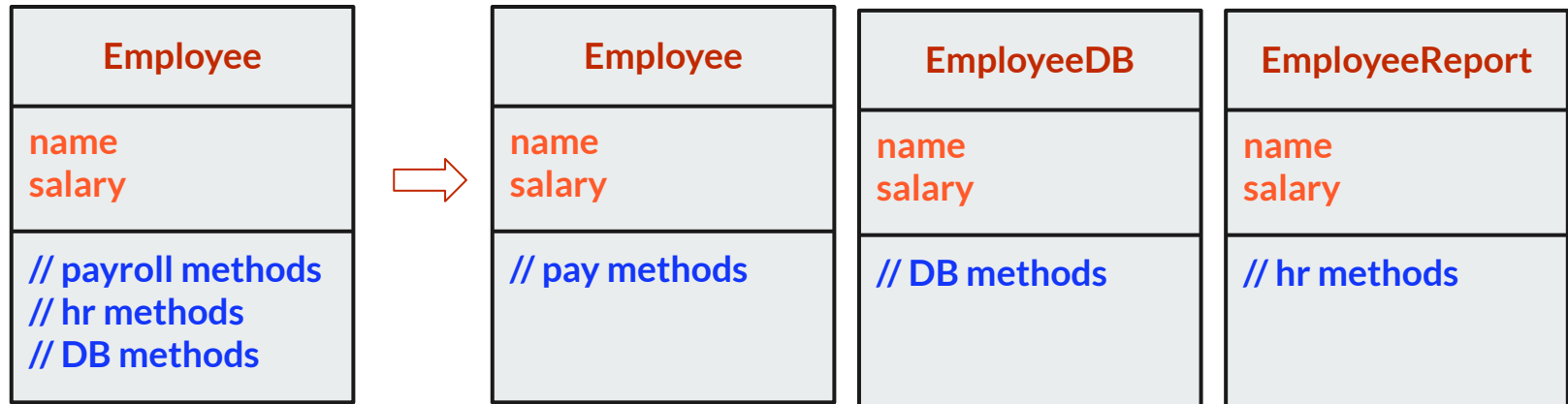
- **SOLID is a way to manage dependencies**
 - The **S**ingle Responsibility Principle (SRP)
 - The **O**pen Closed Principle (OCP)
 - The **L**iskov Substitution Principle (LSP)
 - The **I**nterface Segregation Principle (ISP)
 - The **D**ependency Inversion Principle (DIP)

SOLID PRINCIPLES

- **The Single Responsibility Principle (SRP)**
 - There should never be more than one reason for a class to change
 - If a class has more than one responsibility, then there will be more than one reason for it to change
 - A class should have one responsibility
 - Multiple responsibilities can become coupled
 - Changes to one responsibility can impair the class' ability to meet its other responsibilities

SOLID PRINCIPLES

- **The Single Responsibility Principle (SRP)**
 - An example of how a class that has too many responsibilities can be split into multiple classes
 - If needed, a class can have objects from the other classes



SOLID PRINCIPLES

- **The Open Closed Principle (OCP)**
 - Software entities should be open for extension but closed for modification
 - Design modules that never change
 - When requirements change, extend the behavior of a module by adding new code not by changing code that already works
 - Modules that conform to OCP
 - Have behaviors that can be extended
 - Have source code that does not change

SOLID PRINCIPLES

- The Open Closed Principle (OCP)

- Example for a design that is not closed for modification

```
public abstract class Shape {  
    public abstract String getShapeType();  
}
```

```
public class Circle extends Shape {  
    private String shapeType = "Circle";  
    private double radius;  
    private Point center;  
  
    public String getShapeType(){  
        return shapeType;  
    }  
}
```

```
public class Square extends Shape {  
    private String shapeType = "Square";  
    private double side;  
    private Point topLeftCorner;  
  
    public String getShapeType(){  
        return shapeType;  
    }  
}
```

```
public void drawAllShapes(Shape[] shapes){  
    for(int i = 0; i < shapes.length; i++){  
        Shape shape = shape[i];  
        switch (shape.getShapeType()){  
            case "Circle":  
                drawCircle(shape);  
                break;  
            case "Square":  
                drawSquare(shape);  
                break;  
            default:  
                System.out.println("Unknown shape: "  
                    + shape.getShapeType());  
        }  
    }  
}
```

What if we
add more
shapes?

SOLID PRINCIPLES

- The Open Closed Principle (OCP)

- Example how the previous design can be modified to be closed for modifications

```
public abstract class Shape {
    public abstract void draw();
}

public class Circle extends Shape {
    private double radius;
    private Point center;

    public void draw(){
        // code to draw here;
    }
}

public class Square extends Shape {
    private double side;
    private Point topLeftCorner;

    public void draw(){
        // code to draw here;
    }
}
```

```
public void drawAllShapes(Shape[] shapes){
    for(int i = 0; i < shapes.length; i++){
        Shape shape = shape[i];
        shape.draw();
    }
}
```

What if we
add more
shapes?

SOLID PRINCIPLES

- The Liskov Substitution Principle (LSP)

- Subclasses should be substitutable for their base classes
- Example for a subclass that is not substitutable for its base class

```
public class Rectangle {
    private double width;
    private double height;

    public void setWidth(double width) { this.width = width; }
    public void setHeight(double height) { this.height = height; }
    public double getWidth() { return width; }
    public double getHeight() { return height; }
}
```

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

```
public void calArea(Rectangle r) {
    r.setWidth(5);
    r.setHeight(4);
    double area = r.getWidth() * r.getHeight();
    if (area != 20)
        System.out.println("Unexpected area: " + area);
}
```

What if
calArea()
is passed a
square?

SOLID PRINCIPLES

- The Liskov Substitution Principle (LSP)

- LSP is an important feature of programs that conform to the Open Closed Principle
- When subclasses are completely substitutable for their base class, then methods that use the base class can be substituted with impunity and the subclasses can be changed with impunity
- In our example
 - Geometrically, a square is a rectangle, but a square object is not a rectangle object.
 - Behaviorally, a square is not a rectangle. The behavior of a square object is not consistent with the behavior of a rectangle object.
 - For LSP to hold, subclasses must conform to the behavior that clients expect of the base classes that they use.

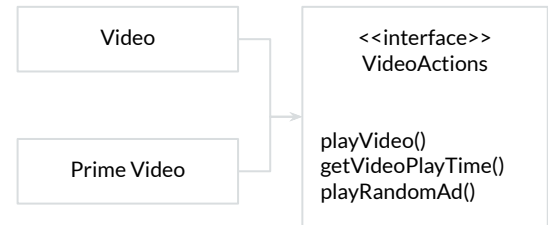
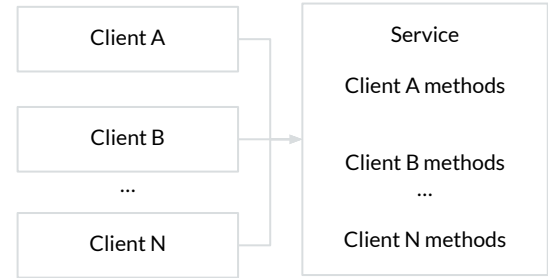
SOLID PRINCIPLES

- **The Interface Segregation Principle (ISP)**
 - Using **many client specific interfaces** is better than using one general purpose interface.
 - If you have a class that has several clients, rather than loading the class with all the methods that the clients need, create specific interfaces for each client.
 - It is okay for a method to appear in more than one interface so that separate clients can use the same method.
 - When interfaces between classes and existing clients change, consider adding new interfaces for existing objects which can reduce recompilation and redeployment.

SOLID PRINCIPLES

- The Interface Segregation Principle (ISP)

- Example for a design that does not follow ISP
- Bloated interfaces can lead to inadvertent couplings between clients that ought to be isolated otherwise
- Bloated interfaces can be segregated to prevent this coupling



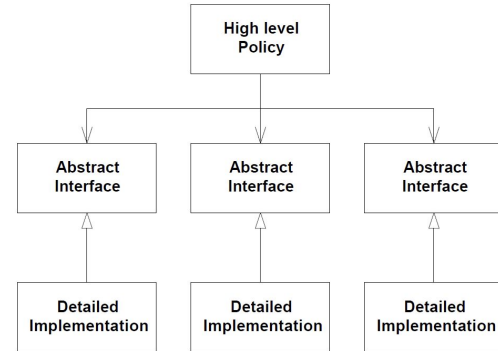
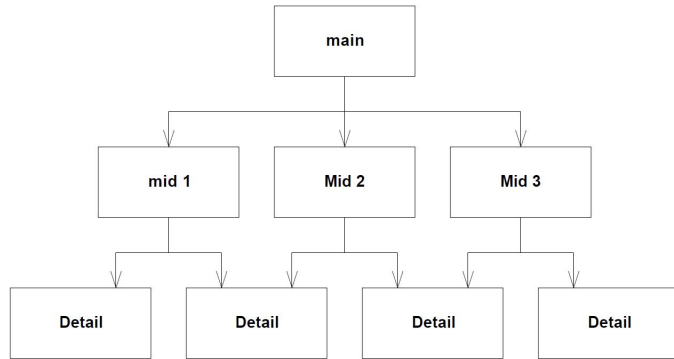
SOLID PRINCIPLES

- **The Dependency Inversion Principle (DIP)**
 - Depend on abstractions. Do not depend on concretions (implementations)
 - High level modules should not depend on low level modules. Both should depend upon abstractions.
 - Abstractions should not depend upon details. Details should depend upon abstractions
 - Every dependency in a design should target an interface or an abstract class. No dependency should target a concrete class

SOLID PRINCIPLES

- The Dependency Inversion Principle (DIP)

- the idea ...



SOLID PRINCIPLES

- The Dependency Inversion Principle (DIP)

- Example for how a copy program can work with any reader and writer that implement the Reader and Writer interfaces. It is no longer dependent on particular lower level modules!

```
public void copy() throws Exception {
    Scanner scnr = new Scanner(System.in);
    PrintStream ps = new PrintStream(new File("myFile"));
    while (scnr.hasNext()) {
        String line = scnr.nextLine();
        ps.println(line);
    }
}
```

```
public interface Reader {
    public boolean hasLine();
    public String getLine() throws Exception;
}

public interface Writer {
    public void putLine (String s) throws Exception;
}

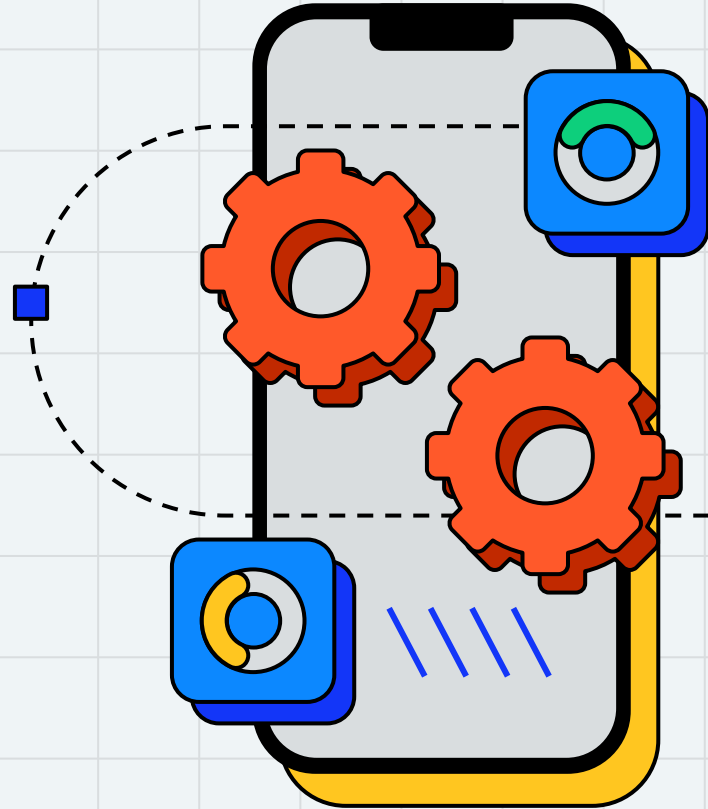
public void copy( Reader input, Writer output) throws Exception{
    while (input.hasLine()) {
        output.putLine(input.getLine());
    }
}
```

SOLID PRINCIPLES

- **The Dependency Inversion Principle (DIP)**
 - Proper application of the Dependency Inversion Principle is necessary for the creation of reusable frameworks.
 - It is important for the construction of code that is resilient to change.
 - When abstraction is isolated from details, code is easier to maintain.

IMPORTANT

In the industry, problem solving often requires interaction among many colleagues. Rarely will you be able to get everyone on a project to agree on the right approach to a solution. Also, rarely will any particular approach be perfect. You'll often compare the relative merits of different approaches.





THANK

YOU!



DO YOU HAVE ANY QUESTIONS?



hend.alkittawi@utsa.edu



By Appointment



Online