Object-Oriented Software Analysis and Design

School of Computer Science
University of Windsor

SOLID

- SOLID is a collection of best-practice, object-oriented design principles which can be applied to your design, allowing you to accomplish various desirable goals such as low-coupling, high cohesion, etc.

SOLID is an acronym for the following principles

S: Single Responsibility Principle (SRP)

- A class should have one, and only one, reason to change.

O: Open-Closed Principle (OCP)

- You should be able to extend a classes behavior, without modifying it.

L: Liskov Substitution Principle (LSP)

- Derived classes must be substitutable for their base classes.

I: Interface Segregation Principle (ISP)

- Make fine grained interfaces that are client specific.

D: Dependency Inversion Principle (DIP)

- Depend on abstractions, not on concretions.

SOLID: Single Responsibility Principle

Single Responsibility Principle (SRP)

- In SRP a reason to change is defined as a responsibility, therefore SRP states

  "*An object should have only one reason to change.*"

- If an object has more than one reason to change then it has more than one responsibility and is in violation of SRP. An object should have one and only one reason to change.

- Consider the following example where I have a `BankAccount` class that has some methods:

Single Responsibility Principle (SRP): Example (contd.)

```
//BankAccount
public class BankAccount {

  private double balance;

  public BankAccount () {...};

  public void setBalance (double newBalance) {
    balance = newBalance;
  }
  public double getBalance () {
    return balance;
  }

  public void deposit(double amount){...}
  public void withdraw(double amount){...}
  public void addInterest(double amount){...}
  public void transferMoney(double fromAcc, double toAcc, double
      amount){...}
}
```

Single Responsibility Principle (SRP): Example (contd.)

- Say we use this `BankAccount` class for a person's `Checking` and `Savings` account. That would cause this class to have more than two reasons to change.
- This is because `Checking` accounts do not have interest added to them and only `Savings` accounts have interest added to them on a monthly basis or however the bank calculates it.
- So, let's refactor this to be more SRP friendly.

Single Responsibility Principle (SRP): Example (contd.)

```
//abstract BankAccount
public abstract class BankAccount {

  private double balance;

  public BankAccount () {...};

  public void setBalance (double newBalance) {
    balance = newBalance;
  }
  public double getBalance () {
    return balance;
  }

  public void deposit(double amount){...}
  public void withdraw(double amount){...}
  public void transferMoney(double fromAcc, double toAcc, double
      amount){...}
}
```

Single Responsibility Principle (SRP): Example (contd.)

```
//ChequingAccount
public class ChequingAccount extends BankAccount {

  public chequingAccount () {...};

  // Some other methods
}


//SavingsAccount
public class SavingsAccount extends BankAccount {

  public SavingsAccount () {...};

  public void addInterest(double amount)
  {
    //calculate interest
  }
}
```

Single Responsibility Principle (SRP): Example (contd.)

- So what we have done is simply create an abstract class out of `BankAccount` and then created a concrete `ChequingAccount` and `SavingsAccount` class so that we can isolate the methods that are causing more than one reason to change.

- SRP is one of the hardest principles to enforce because there is always room for refactoring out one class to multiple; each class has one responsibility.

SO**O**LID: Open-Closed Principle

Open-Closed Principle (OCP)

- The open/closed principle states

  "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*";

  that is, such an entity can allow its behavior to be modified without altering its source code.

- In practice, this means creating software entities whose behavior can be changed without the need to edit and recompile the code itself.

Open-Closed Principle (OCP)

- The simplest way to demonstrate this principle is to consider a method that does one thing.

- The Open-Closed Principle can also be achieved in many other ways, including through the use of inheritance or through compositional design patterns like the Strategy pattern.

Open-Closed Principle (OCP): Example

- Let's say that we've got a `Rectangle` class.

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

    public double getWidth() {
        return width;
    }
    public void setWidth(double width) {
        this.width = width;
    }

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

Open-Closed Principle (OCP): Example (contd.)

- Now we want to build an application that can calculate the total area of a collection of rectangles.

```
import java.util.ArrayList;

public class AreaCalculator {

    public double Area(ArrayList<Rectangle> shapes)
    {
        double area = 0;

        for(Rectangle rect : shapes){
            area += rect.getWidth()*rect.getHeight();
        }
        return area;
    }
}
```

Open-Closed Principle (OCP): Example (contd.)

- Congratulations! You have done a great job! Now we want to extend this application so that it can calculate the area of not only rectangles but also circles as well.

Open-Closed Principle (OCP): Example (contd.)

- And we come up with a solution as given below:

```
import java.util.ArrayList;

public class AreaCalculator {
    public double Area(ArrayList<Object> shapes){
        double area = 0;
        for(Object shape : shapes){
            if (shape instanceof Rectangle){
                Rectangle rectangle = (Rectangle) shape;
                area +=
                    rectangle.getWidth()*rectangle.getHeight();
            }
            else{
                Circle circle = (Circle) shape;
                area += circle.getRadius()*Math.PI;
            }
        }
        return area;
    }
}
```

Open-Closed Principle (OCP): Example (contd.)

- The solution presented in the previous slide works well. But the solution have modified the `AreaCalculator` method. Now we want to calculate the area of triangles.

- To incorporate the area of triangles, again we need to make changes in the `AreaCalculator` method.

- It seems easy to extend the program to calculate the area of triangles. But in a real world scenario where the code base is ten, a hundred or a thousand times larger and modifying the class means redeploying it's assembly/package to five different servers that can be a pretty big problem.

- Oh, and in the real world your software client would have changed the requirements five more times since you read the last sentence.

Open-Closed Principle (OCP): Example (contd.)

- A solution that abides by the Open/Closed Principle. Let's define an abstract method for calculating the area of a shape.

```
public abstract class Shape {
    public abstract double Area();
}
```

Open-Closed Principle (OCP): Example (contd.)

- `Rectangle` inherits from `Shape`

```
public class Rectangle extends Shape{
    private double width;
    private double height;
    public double getWidth() {
        return width;
    }
    public void setWidth(double width) {
        this.width = width;
    }
    public double getHeight() {
        return height;
    }
    public void setHeight(double height) {
        this.height = height;
    }
    public double Area() {
        return width*height;
    }
}
```

Open-Closed Principle (OCP): Example (contd.)

- `Circle` inherits from `Shape`

```
public class Circle extends Shape{
    private double radius;

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    public double Area() {
        return radius*radius*Math.PI;
    }
}
```

Open-Closed Principle (OCP): Example (contd.)

```
import java.util.ArrayList;

public class Area {

    public double Area(ArrayList<Shape> shapes)
    {
        double area = 0;

        for(Shape s : shapes){
            area += s.Area();
        }
        return area;
    }
}
```

Open-Closed Principle (OCP): Example (contd.)

- As we've moved the responsibility of actually calculating the area away from `AreaCalculator`'s `Area` method it is now much simpler and robust as it can handle any type of `Shape` that we throw at it.

- In other words we've closed it for modification by opening it up for extension.

SOLID: Liskov Substitution Principle

Liskov Substitution Principle (LSP)

- The Liskov Substitution Principle (LSP) states that

  "*derived classes must be substitutable for the base class*".

- When this principle is violated, it tends to result in a lot of extra conditional logic scattered throughout the application, checking to see the specific type of an object.

- This duplicate, scattered code becomes a breeding ground for bugs as the application grows.

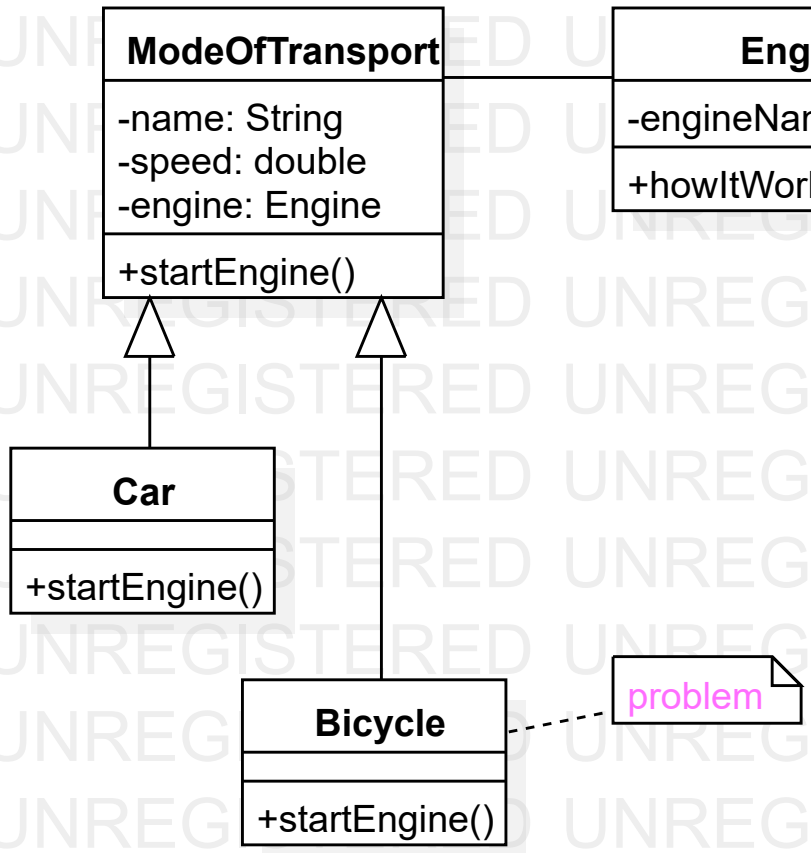
Liskov Substitution Principle (LSP)

- Most introductions to object-oriented development discuss inheritance, and explain that one object can inherit from another if it has an "IS-A" relationship with the inherited object.

- However, this is necessary, but not sufficient. It is more appropriate to say that one object can be designed to inherit from another if it always has an "IS-SUBSTITUTABLE-FOR" relationship with the inherited object.

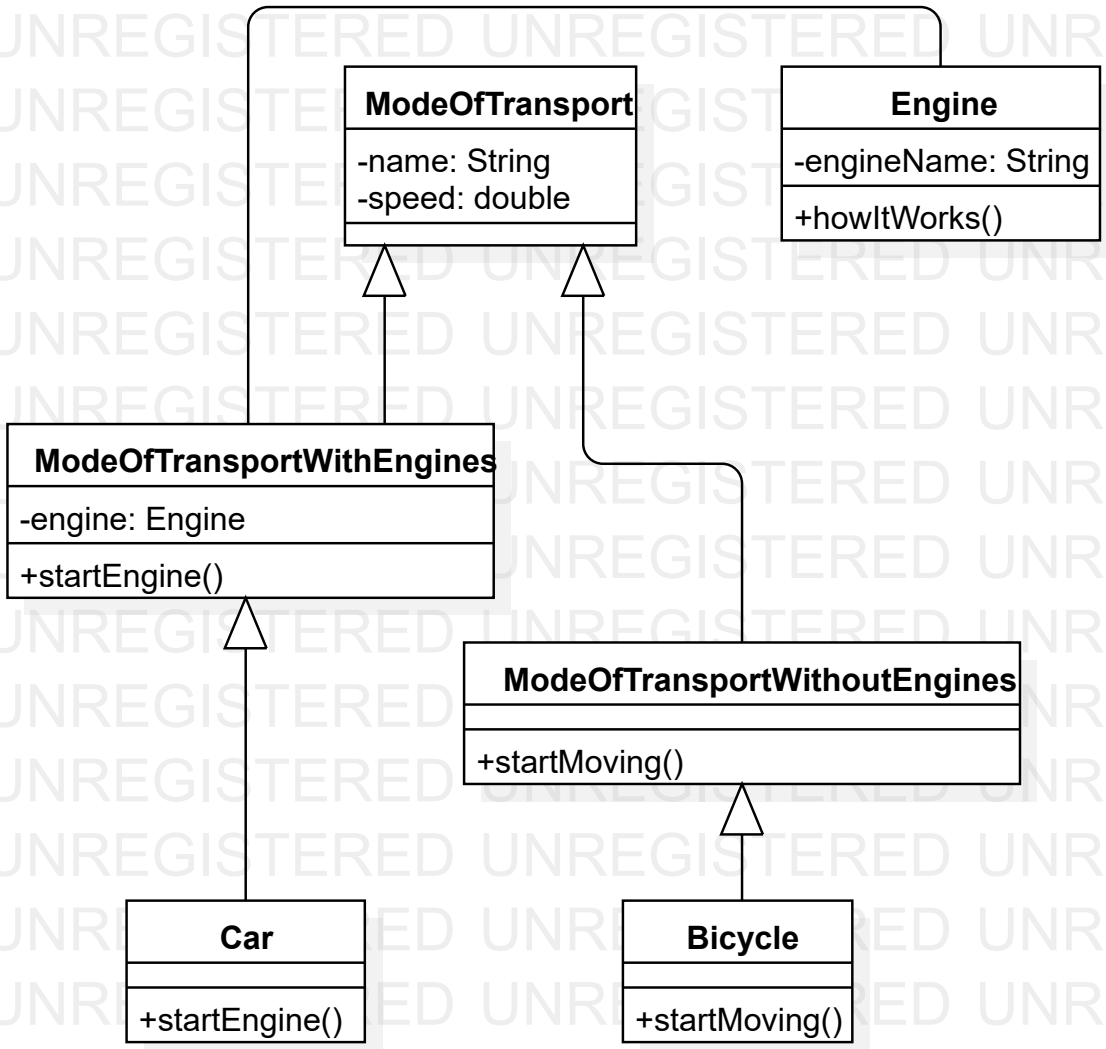
Liskov Substitution Principle (LSP): Example

- Liskov Substitution Principle Violation



Liskov Substitution Principle (LSP): Example (contd.)

- Following the Liskov Substitution Principle



SOLID: Interface Segregation Principle

Interface Segregation Principle (ISP)

- The Interface Segregation Principle (ISP) states that

  "*clients should not be forced to depend on methods that they do not use*".

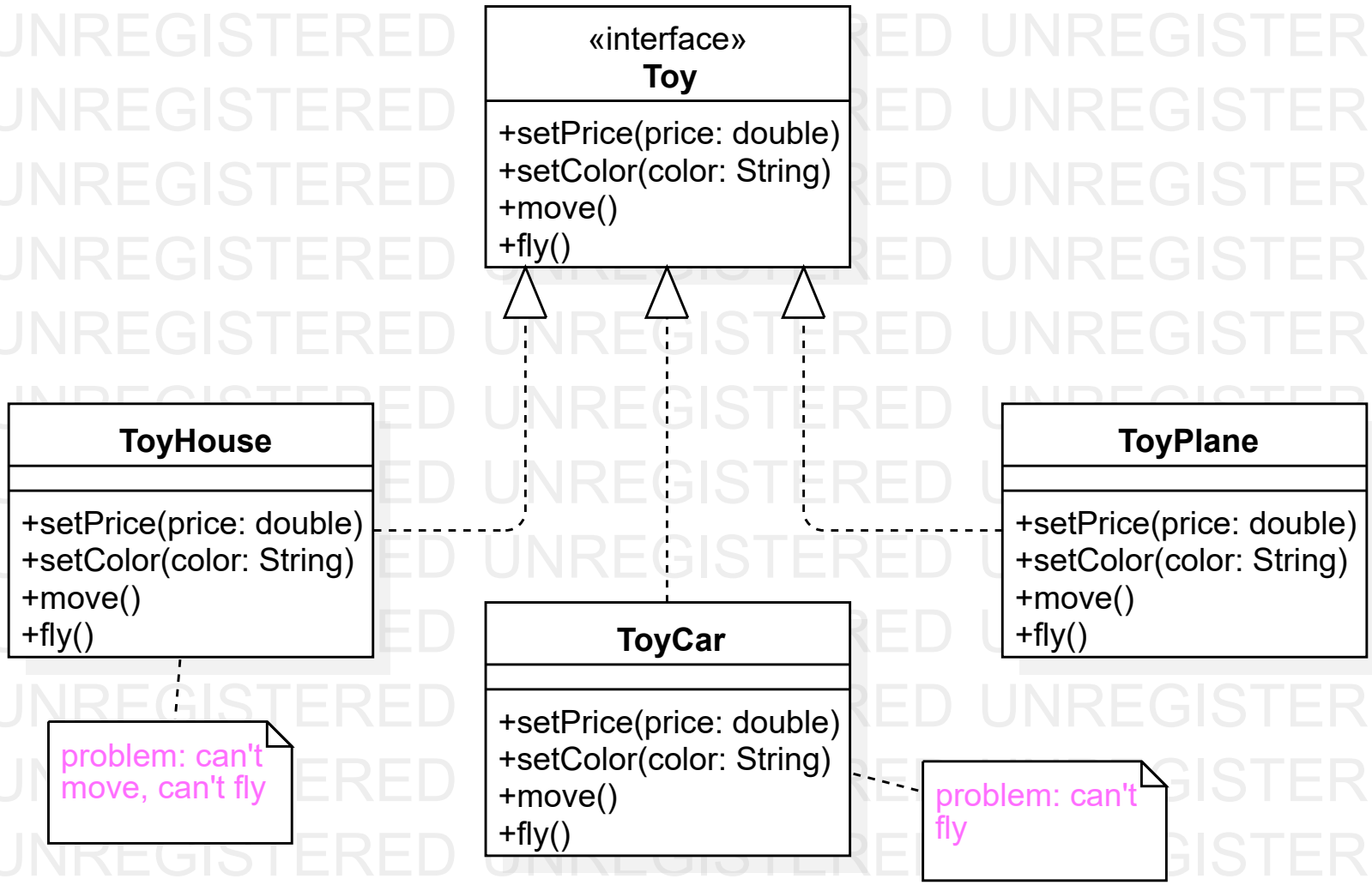
- Application developers should **favor** thin, focused interfaces to "fat" interfaces that offer more functionality than a particular class or method needs.

- Ideally, your thin interfaces should be cohesive, meaning they have groups of operations that logically belong together.

- Another **benefit** of smaller interfaces is that they are easier to implement fully, and thus less likely to break the Liskov Substitution Principle by being only partially implemented.

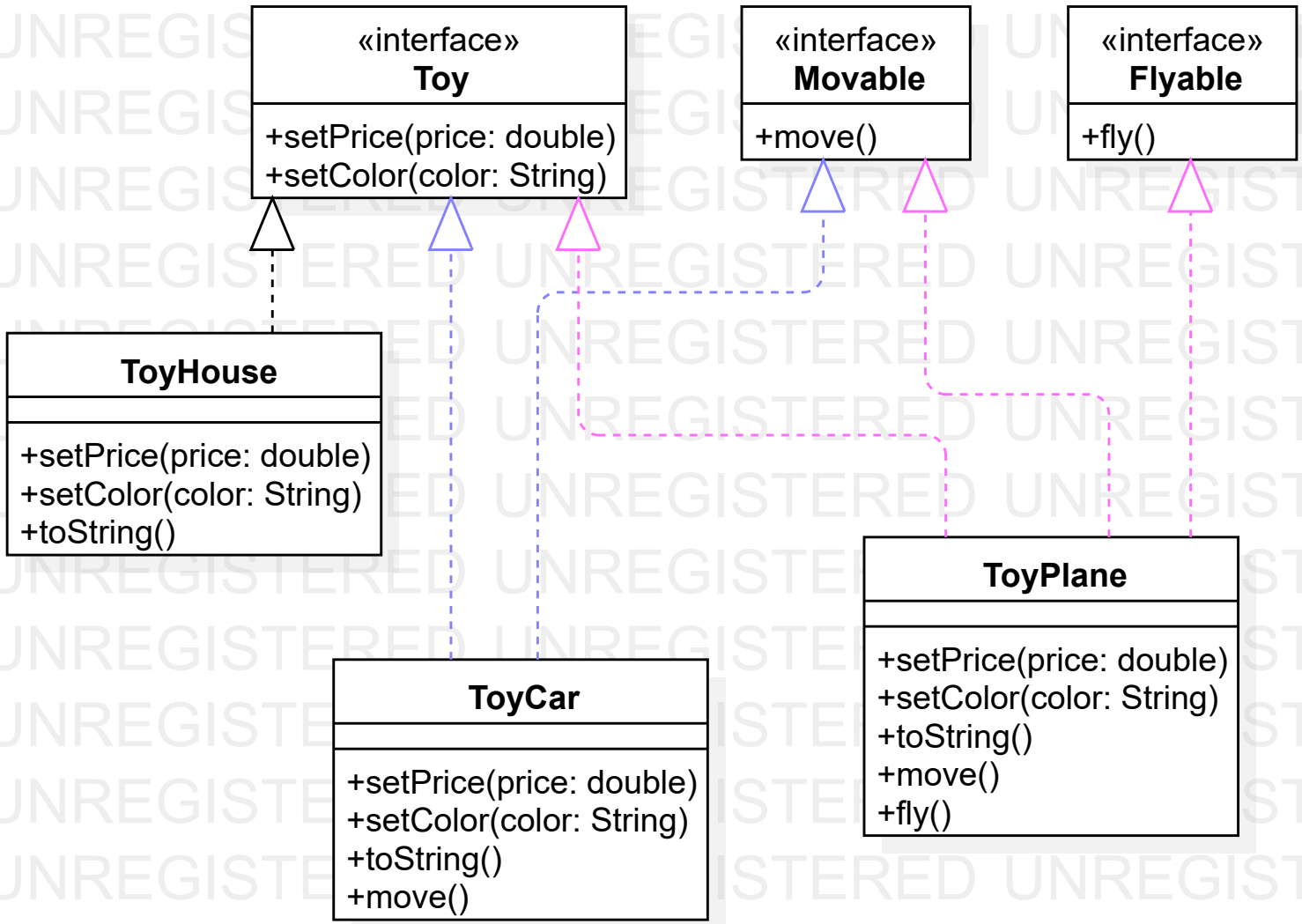
Interface Segregation Principle (ISP): Example

- Interface Segregation Principle Violation



Interface Segregation Principle (ISP): Example (contd.)

- Following the Interface Segregation Principle



SOLID: Dependency Inversion Principle

Dependency Inversion Principle (DIP)

- The Dependency Inversion Principle (DIP) states that

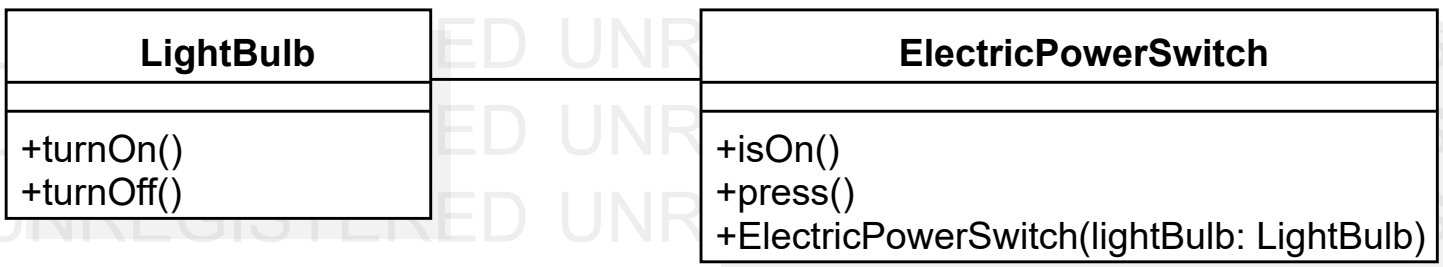
  "*high level modules should not depend on low level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions.*"

- It's extremely common when writing software to implement it such that each module or method specifically refers to its collaborators, which does the same.

- This type of programming typically lacks sufficient layers of abstraction, and results in a very tightly coupled system, since every module is directly referencing lower level modules.

Dependency Inversion Principle (DIP): Example

- Dependency Inversion Principle Violation



Dependency Inversion Principle (DIP): Example (contd.)

- Following the Dependency Inversion Principle