



# SOLID Principles



## S - Single Responsibility

A class should have one and only one reason to change, meaning that a class should have only one job.

*“Do one thing and do it well”*



# S - Single Responsibility

Not using Single Responsibility

```
public class AreaCalculator{  
    public void calculateArea(double height, double width){  
        double area = height * width;  
        saveAsJson(area)  
    }  
}
```



# S - Single Responsibility

Using Single Responsibility

```
public class AreaCalculator{  
    public double calculateArea(double height, double width){  
        double area = height * width;  
        return area  
    }  
}  
  
public class AreaWriter{  
    public void saveAsJson(double area)  
    public void saveAsText(double area)  
}
```



# S - Single Responsibility

Benefits :

1. Fewer and simpler test cases compared to a class with multiple responsibilities
2. Less functionality in a single class will have fewer dependencies
3. Smaller, well-organised classes are easier to understand



## ○ - Open/Closed

Objects or entities should be open for extension, but closed for modification.



## ○ - Open/Closed

Not using the Open/Closed principle

```
public class Post{  
    public void createPost(DataBase database, String post){  
        database.addPost(post)  
    }  
}
```

*~new feature~*

```
public class Post{  
    public void createPost(DataBase database, String post){  
        if(post.startsWith("@"))  
            database.addMention(post)  
        else  
            database.addPost(post)  
    }  
}
```



# O - Open/Closed

Using the Open/Closed principle

```
public class Post{  
    public void createPost(DataBase database, String post){  
        database.addPost(post)  
    }  
}  
  
public class MentionPost extends Post{  
    @Override  
    public void createPost(DataBase database, String post){  
        database.addMention(post)  
    }  
}
```





# O - Open/Closed

Benefits :

1. Prevents bugs occurring in existing code when adding new features
2. Encourages code reuse when extending classes
3. Ensures that system complexity can only grow along designed routes (the bits you've planned for extension)



## L - Liskov Substitution

If **S** is a subtype of **T**, then objects of type **T** may be replaced (or substituted) with objects of type **S**.



# L - Liskov Substitution

Not using Liskov Substitution

```
public class Rectangle{  
    int height;  
    int width;  
  
    public Rectangle(int height, int width){  
        this.height = height;  
        this.width = width;  
    }  
  
    public double area(){  
        return height * width;  
    }  
}
```

```
public class Square extends Rectangle{  
    int height;  
    int width;  
  
    public Square(int height, int width) throws Exception{  
        if(height != width)  
            throw Exception;  
        this.height = height;  
        this.width = width;  
    }  
  
    public double area(){  
        return height * height;  
    }  
}
```



# L - Liskov Substitution

## Using Liskov Substitution

```
public interface Shape{
    public double area();
}

public class Rectangle implements Shape{

    int height;
    int width;

    public Rectangle(int height, int width){
        this.height = height;
        this.width = width;
    }

    @Override
    public double area(){
        return height * width;
    }
}
```

```
public class Square implements Shape{

    int height;

    public Square(int height){
        this.height = height;
    }

    @Override
    public double area(){
        return height * height;
    }
}
```



# L - Liskov Substitution

Benefits :

1. Encourages code reuse
2. Loosely dependent code. Without Liskov Substitution when a subclass can not substitute its parent class the code will need multiple conditional statements to determine the class or type to handle certain cases differently



## I - Interface Segregation

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



# I - Interface Segregation

Not using Interface Segregation

```
public interface Torch {  
    public void on();  
    public void off();  
    public void windUp();  
}  
  
public class BatteryTorch implements Torch{  
    boolean on;  
  
    public BatteryTorch(){  
        this.on = false;  
    }  
  
    public on(){  
        this.on = true;  
    }  
  
    public off(){  
        this.on = false;  
    }  
  
    public void windUp(){}  
}
```

```
public class WindUpTorch implements Torch{  
  
    boolean on;  
    int energy;  
  
    public WindUpTorch(){  
        this.on = false;  
        this.energy = 0;  
    }  
  
    public on(){  
        this.on = true;  
    }  
  
    public off(){  
        this.on = false;  
    }  
  
    public void windUp(){  
        this.energy += 10;  
    }  
}
```

# I - Interface Segregation

## Using Interface Segregation

```
public interface Torch {  
    public void on();  
    public void off();  
}  
  
public interface WindUpBattery{  
    public void windUp();  
}  
  
public class BatteryTorch implements Torch{  
    boolean on;  
  
    public BatteryTorch(){  
        this.on = false;  
    }  
  
    public on(){  
        this.on = true;  
    }  
  
    public off(){  
        this.on = false;  
    }  
}
```

```
public class WindUpTorch implements Torch, WindUpBattery{  
    boolean on;  
    int energy;  
  
    public WindUpTorch(){  
        this.on = false;  
        this.energy = 0;  
    }  
  
    public on(){  
        this.on = true;  
    }  
  
    public off(){  
        this.on = false;  
    }  
  
    public void windUp(){  
        this.energy += 10;  
    }  
}
```





# I - Interface Segregation

Benefits :

1. Keeps code decoupled increasing readability and maintainability of the code
2. Clients that only need a subset of features are able to only use the features they need



## D - Dependency Inversion

1. High level modules should not depend on low level modules; both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend upon abstractions.

If you adhere to the Open/Closed and the Liskov Substitution principles the Dependency Inversion principle will automatically be applied.



# D - Dependency Inversion

Not using Dependency Inversion

```
public class VEightEngine{  
    //Methods to start stop, speed up and slow down  
}  
  
public class Car{  
    VEightEngine engine;  
  
    public Car(){  
        this.engine = new VEightEngine();  
    }  
}
```



# D - Dependency Inversion

Using Dependency Inversion

```
public interface Engine{
    public void start();
    public void stop();
    public void speedUp();
    public void slowDown();
}

public class VEightEngine implements Engine{
    //Methods to start stop, speed up and slow down
}

public class HybridEngine implements Engine{
    //Methods to start stop, speed up and slow down
}
```

```
public class Car{
    Engine engine;

    public Car(Engine engine){
        this.engine = engine;
    }
}
```



# D - Dependency Inversion

Benefits :

1. Reduces coupling between different pieces of code
2. Choose at run-time which implementation is better suited for your particular environment
3. Allows injection of mock/fake instances to facilitate testing