**SOLID**

Table of Contents

[Principles 2](#_Toc83392716)

[Single Responsibility Principle 3](#_Toc83392717)

[Cohesion and Coupling 3](#_Toc83392718)

[Responsibilities 4](#_Toc83392719)

[Summary 5](#_Toc83392720)

[Open/Closed Principle 7](#_Toc83392721)

[Liskov Substitution Principle 11](#_Toc83392722)

[Interface Segregation Principle 15](#_Toc83392723)

[Dependency Inversion Principle 17](#_Toc83392724)

## Principles

SOLID consists of 5 principles:

1. **Single Responsibility Principle** (SRP)
2. **Open/Close Principle** (OCP)
3. **Liskov Substitution Principle** (LSP)
4. **Interface Segregation Principle** (ISP)
5. **Dependency Inversion Principle** (DIP)

## Single Responsibility Principle

The Single Responsibility Principle (SRP) states that every **object** should have a **single responsibility**, and that responsibility must be entirely **encapsulated** by the class.

A single class should never change for **more than one reason**. This is sometimes impossible to achieve, but we should still try.

Before we can get into the details of what any of that means, we need to discuss **cohesion** and **coupling**.

### Cohesion and Coupling

**Cohesion** refers to how **closely related** the different **parts of a module** are to each other, which means the parts are responsible for the same things. The more we **increase** cohesion, the more it indicates that a particular module has a **specific task** and does only that task.

**Coupling** refers to the degree to which a particular module **relies on other modules**. The **less coupling** we have, the **less chances** there are of causing **problems with other modules** if we change one particular module.

It is desirable to **increase cohesion** and **decrease coupling**.

### Responsibilities

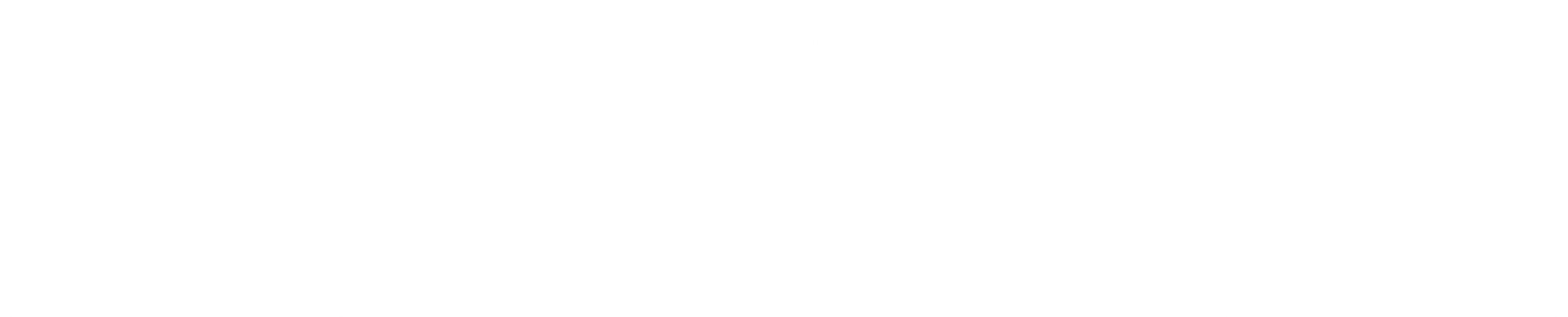
Coming back to SRP, we still haven’t discussed what exactly we mean by responsibilities.

A **responsibility** is a single reason for change. From the user’s perspective, it is a difference in usage scenarios.

Every **user requirement** typically turns into a responsibility. Since user requirements **change**, this means responsibilities can change. If we have lots of responsibilities packed within a **single module**, then that module will **change frequently**. This also means that there is an increased chance of introducing new **errors**.

SRP essentially deals with this. Each class should deal with a **single responsibility** and therefore have a single reason to exist. It will do exactly one job, and therefore have just **one reason to change**. If we have a class that changes a lot for different reasons, we need to break this class into smaller classes, each handling one of them. In this way, if there is an error, it will be very easy to find.

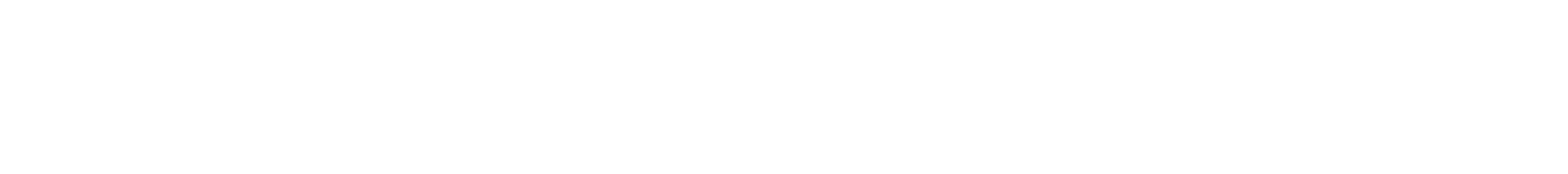
Example



Say we have a Rectangle class that has two methods, area(), which returns the area of the rectangle, and draw(), which draws the rectangle on some user interface.

The problem here is, if there are changes to the user interface, we will need to change the Rectangle class and if there are changes in the computational geometry, then we also need to change the Rectangle class. Thus, this setup does not follow SRP.

We need to divide it into two classes, for example, RectangleGeometry and RectangleView, with the latter being connected to the former.



It will most likely not be possible to identify how the classes need to be divided at first. Once we start working and notice that we have to change a class frequently, then we can decide to **refactor** our code and divide the class.

### Summary

☑ SRP **lowers coupling** and **increases cohesion**.

☑ Having multiple **small classes** with **distinct responsibilities** leads to more **flexible design**.

☑ Splitting responsibilities between smaller methods and classes makes the system **easier to understand**.

☑ Components with a single responsibility are easier to **reuse**.

☑ Most methods with narrow responsibilities should have **no side effects**.

☑ It is easier to write and maintain **tests** for methods and classes with focused, independent responsibilities.

☑ Having methods with single responsibilities helps **quickly find performance issues**.

☒ The **more focused** we make a component, the **more code** we will need to write.

☒ It takes **more time and effort** to create an initial version of an application if we concentrate on finely separating responsibilities instead of just having large components.

## Open/Closed Principle

The **Open/Closed Principle** states that software entities, like classes, modules, functions, etc., should be **open for extension** but **closed for modification**. Essentially, this means that we should be able to **add new features** without having to **change** anything in the **existing code**. The only reason for existing code to change should be to fix bugs. This allows extended functionality via implementation classes without necessitating changes in the base class.

For example, consider the class below:

public class Rectangle  
{  
 public double Width { get; set; }  
 public double Height { get; set; }  
}

C#

We need an application that can calculate the area of a collection of Rectangle objects.

public class AreaCalculator  
{  
 public double Area(Rectangle[] shapes)  
 {  
 double area = 0;  
 foreach (var shape in shapes)  
 {  
 area += shape.Width \* shape.Height;  
 }  
 return area;  
 }  
}

C#

This definitely works. Next, we are told that we also need to calculate the area of Circle objects. This requires us to change our code.

public class AreaCalculator  
{  
 public double Area(Rectangle[] shapes)  
 {  
 double area = 0;  
 foreach (var shape in shapes)  
 {  
 if (shape is Rectangle)  
 {  
 Rectangle rectangle = (Rectangle) shape;  
 area += rectangle.Width \* rectangle.Height;  
 }  
 else  
 {  
 Circle circle = (Circle) shape;  
 area += circle.Radius \* circle.radius \* Math.PI;  
 }  
 }  
 return area;  
 }  
}

C#

Next, we are again told that we also need to calculate the area for Triangle objects. The pattern should become clear by this point. The code we have created is not **closed for modification**. If we want to extend its capabilities, we are forced to modify it. In other words, it is not **open for extension**.

Here, it may seem like a small problem, but in a real program, this could mean modifying hundreds of lines of code and redeploying new packages to several servers around the world.

Instead of doing this, we could create an abstract Shapes class from which all the other classes inherit. In this way, we could force all the subclasses to have a method to calculate their own areas, and then just call those methods for each object in the Area method.

public abstract class Shape  
{  
 public abstract double Area();  
}  
  
public class Rectangle : Shape  
{  
 public double Width { get; set; }  
 public double Height { get; set; }  
 public override double Area()  
 {  
 return Width \* Height;  
 }  
}  
  
public class Circle : Shape  
{  
 public double Radius { get; set; }  
 public override double Area()  
 {  
 return Radius \* Radius \* Math.PI;  
 }  
}  
  
public class AreaCalculator  
{  
 public double Area(Rectangle[] shapes)  
 {  
 double area = 0;  
 foreach (var shape in shapes)  
 {  
 area += shape.Area();  
 }  
 return area;  
 }  
}

C#

This code follows OCP. We do not need to change the existing code to add new shapes to it.

This brings up the question of how we would even know that our code did not initially follow OCP. The answer is, we wouldn’t. If we know from **experience** that a particular set of changes is likely to occur, we can write our code in that way from the beginning. However, if we do not know this, then we go by trial an error. If we find **once** that our code needed to be modified to accommodate changes, we should let it go, but as soon as we need to do this **twice**, we should change our code completely so that it follows OCP.

Remember that OCP adds **complexity**, which is why we wait until the second change to apply it. Also, no design can ever be **completely closed** against all changes.

## Liskov Substitution Principle

The **Liskov Substitution Principle** states that a **child class** should never break a **parent class’s type definitions**. In simple terms, if it looks like a duck and quacks like a duck but needs batteries, you probably have the wrong abstraction.

A **child class** must not **remove base class behaviour** or **violate base class invariants**. Everywhere the code expects an object of the **parent class**, we should be able to use a **child class** object instead without any issues.

In general, OOP describes child classes with an ‘is a’ relationship. LSP suggests we use the ‘is substitutable for’ relationship instead.

Consider the following code:

class Transportation  
{  
 String name;  
 String getName();  
 void setName(String n);  
 double speed;  
 double getSpeed();  
 void setSpeed(double d);  
 Engine engine;  
 Engine getEngine();  
 void setEngine(Engine e);  
 void startEngine();  
}  
  
class Car extends Transportation  
{  
 @Override  
 void startEngine();  
}

JAVA

There is no issue in this code. A Car object is after all sensibly a type of Transportation object, and it overrides the startEngine() method since it has its own engine type.

Now consider this code:

class Bicycle extends Transportation  
{  
 @Override  
 void startEngine();  
}

JAVA

This is a problem. A Bicycle object is still a type of Transportation object, but it does not have an engine. Yet, it still overrides the startEngine() method. In this case, a Bicycle object will be unable to act as a proper substitute for a Transportation object.

Now say we refactor our code:

class Transportation  
{  
 String name;  
 String getName();  
 void setName(String n);  
 double speed;  
 double getSpeed();  
 void setSpeed(double d);  
}

JAVA

Now we can extend the Transportation class for non-motorized vehicles and motorized vehicles without any issues.

class TransportWithoutEngine extends Transportation  
{  
 void startMoving();  
}  
  
class TransportWithEngine extends Transportation  
{  
 Engine engine;  
 Engine getEngine();  
 void setEngine(Engine e);  
 void startEngine();  
}  
  
class Bicycle extends TransportWithoutEngine  
{  
 @Override  
 void startMoving();  
}

JAVA

Consider another example.

public static class Animal  
{  
 public String favoriteFood;  
 public Animal(String favoriteFood)  
 {  
 this.favoriteFood = favoriteFood;  
 }  
}  
  
public static class Dog extends Animal  
{  
 public Dog(String favoriteFood)  
 {  
 super(favoriteFood);  
 }  
}

public static class Cat extends Animal  
{  
 public Cat(String favoriteFood)  
 {  
 super(favoriteFood);  
 }  
}

JAVA

Say we have an Animal class and a Dog and a Cat class that extends the Animal class. We now want to write a method that allows us to give treats to the animals.

public static void GiveTreatTo (Animal animal)  
{  
 String msg = "You fed the " + animal.getClass().getSimpleName()   
 + " some " + animal.favoriteFood;  
 System.out.println(msg);  
}

JAVA

Notice that this method can be used with any objects that extend form the Animal class. We are using LSP to ensure that the child objects can replace the parent object without any issues.

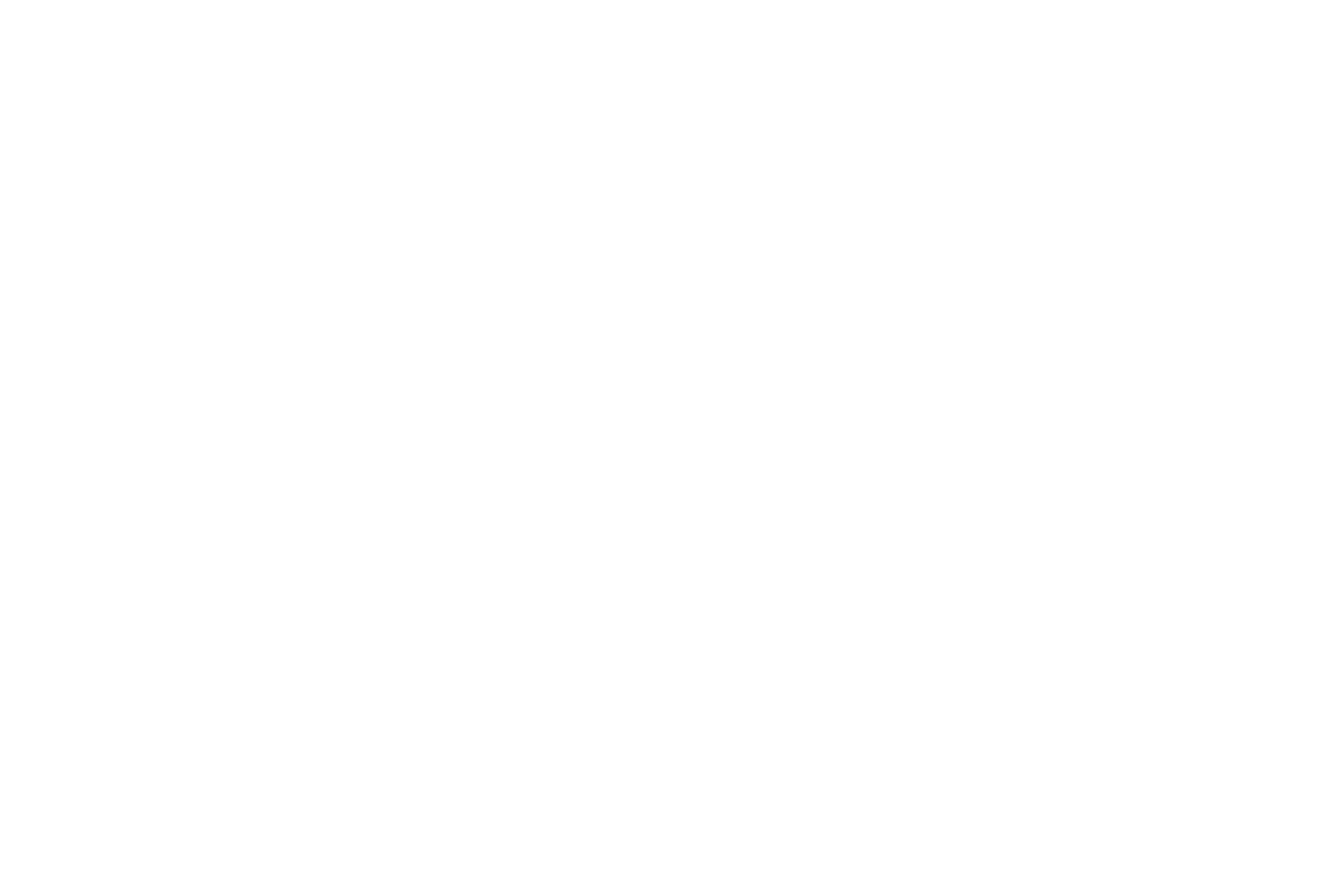
public static void main(String[] args)  
{  
 Dog rover = new Dog("bacon");  
 Cat bingo = new Cat("fish");  
 GiveTreatTo(rover);  
 GiveTreatTo(bingo);  
}

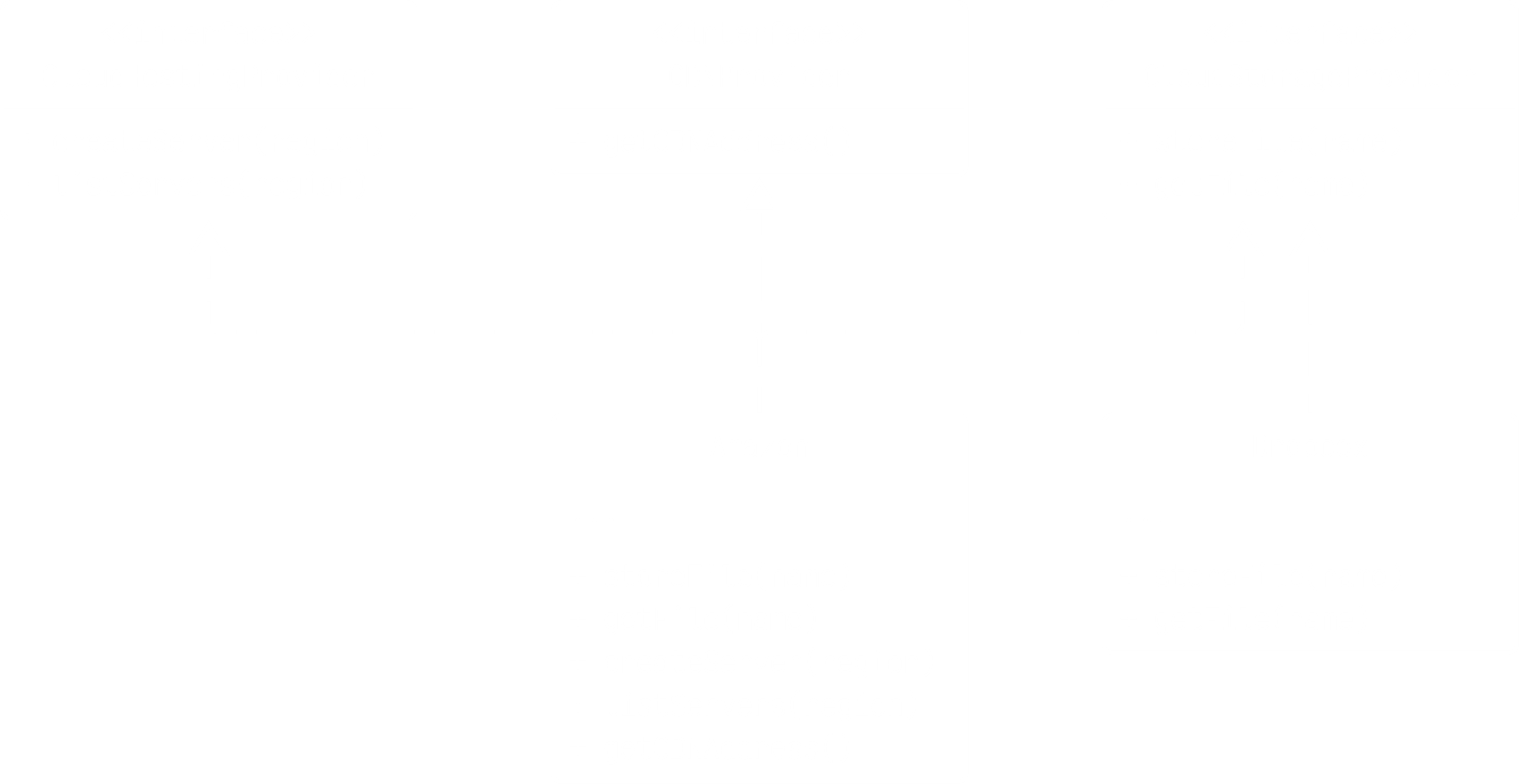
JAVA

## Interface Segregation Principle

In SRP, we learnt that every module should have a single responsibility. The **Interface Segregation Principle** (ISP) tells us something similar. A single interface can have many different **methods** that each serve a different **client**. ISP tells us that the methods should be broken down into **groups** such that each group serves a single client. In this way, the client does not have to deal with the other methods.

Interfaces that haves lots of methods in them serving multiple different clients are called **fat interfaces**. Such interfaces, especially if they are large programs, can be difficult to manage and to change. Another issue is that **changes** to any methods in the original interface will force all the clients to **recompile** if we do not divide the interface. If we do divide it, only the client that is affected by the method change will have to recompile.





Notice from the above example that ISP is actually just SRP.

ISP should be used when we begin to face **issues** with an interface. If we find ourselves depending on a fat interface where we are not using most of the methods, then it is time to implement ISP.

## Dependency Inversion Principle

A **dependency** is a class or value that the class we are creating depends on. For example, if our class has the following line of code:

ConfigurationManager.AppSettings["someSettings"];

JAVA

then ConfigurationManager is a dependency.

A class that has lots of dependencies has **high coupling**. Such a class would be difficult to change, since any changes will most likely require changes to be made to dependent modules as well.

The **Dependency Inversion Principle** (DIP) tells us that modules should depend on **interfaces**. Interfaces are the most common form of **abstraction**, which is an indirect representation of a concrete class or method.

Suppose we have a very complex Manager class which works with another class, the Worker class. Now, we introduce a new class, SuperWorker. In order to introduce this class and get everything working, we now have to change the Manager class. This could be difficult and time consuming, not to mention the current functionality might be affected, which means unit testing has to be redone.

class Worker  
{  
 public void work()  
 {  
 // working  
 }  
}

class Manager  
{  
 Worker worker;  
  
 public void setWorker (Worker w)  
 {  
 worker = w;  
 }  
  
 public void manager()  
 {  
 worker.work();  
 }  
}  
  
class SuperWorker  
{  
 public void work()  
 {  
 // working much more  
 }  
}

JAVA

This also violates OCP.

Instead, if we follow DIP, the Manager class can use an IWorker interface, while the Worker class, the SuperWorker class and any future classes can just implement that interface.

interface IWorker  
{  
 public void work();  
}  
  
class Worker implements IWorker  
{  
 public void work()  
 {  
 // working  
 }  
}

class SuperWorker implements IWorker  
{  
 public void work()  
 {  
 // working much more  
 }  
}  
  
class Manager  
{  
 IWorker worker;  
  
 public void setWorker(IWorker w)  
 {  
 worker = w;  
 }  
  
 public void manage()  
 {  
 worker.work();  
 }  
}

JAVA