12. SOLID Principles

What are SOLID principles?

SOLID is a mnemonic device for 5 design principles of object-oriented programs (OOP) that result in readable, adaptable, and scalable code. SOLID can be applied to any OOP program.

The 5 principles of SOLID are:

- Single-responsibility principle
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

SOLID principles were developed by computer science instructor and author Robert C. Martin (sometimes called “Uncle Bob”) in 2000 and quickly became a staple of modern object-oriented design (OOD). The SOLID acronym became commonplace when these principles gained widespread popularity in the programming world.

https://www.youtube.com/watch?v=zHiWqnTWsn4

S: Single-responsibility principle

“A class should only have a single responsibility, that is, only changes to one part of the software’s specification

should be able to affect the specification of the class.”

The single-responsibility principle (SRP) states that each class, module, or function in your program should only do one job. In other words, each should have full responsibility for a single functionality of the program. The class should contain only variables and methods relevant to its functionality.

Classes can work together to complete larger complex tasks, but each class must complete a function from start to finish before it passes the output to another class.

“a class should have only one reason to change”. Here the “reason” is that we want to change the single functionality this

class pursues. If we do not want this single functionality to change, we will never change this class because all components of the class should relate to that behavior.

Therefore, we could change all but one class in the program without breaking the original class.

SRP makes it easy to follow another well-respected principle of OOP, encapsulation. It is easy to hide data from the user when all data and methods for a job are within the same single-responsibility class.

If you add a getter and setter method to a single-responsibility class, the class meets all criteria of an encapsulated class.

The benefit of programs that follow SRP is that you can change the behavior of a function by editing the single class responsible for it. Also, if a single functionality breaks, you know where the bug will be in the code and can trust that only that class will break.

This factor also helps with readability because you only have to read a class until you determine its functionality.

Implementation

Let’s look at an example of how SRP can be applied to make our RegisterUser class more readable.

```
// does not follow SRP
public class RegisterService
{
public void RegisterUser(string username)
{
if (username == "admin")
throw new InvalidOperationException();

SqlConnection connection = new SqlConnection();
connection.Open();
SqlCommand command = new SqlCommand("INSERT INTO
[...]");//Insert user into database.

SmtpClient client = new SmtpClient("[smtp.myhost.com](http://smtp.myhost.com/)");
client.Send(new MailMessage()); //Send a welcome email.
}
}
```

The program above does not follow SRP because RegisterUser does three different jobs: register a user, connect to the database, and send an email.

This type of class would cause confusion in larger projects, as it is unexpected to have email generation in the same class as the registration.

There are also many things that could cause this code to change like if we make a switch in a database schema or if we adopt a new email API to send emails.

Instead, we need to split the class into three specific classes that each accomplish a single job. Here’s what our same class would look like with all other jobs refactored to separate classes:

This achieves the SRP because RegisterUser only registers a user and the only reason it would change is if more username restrictions are added. All other behavior is maintained in the program but is now achieved with calls to userRepository and emailService.

```
public void RegisterUser(string username)
{
if (username == "admin")
throw new InvalidOperationException();

_userRepository.Insert(...);

_emailService.Send(...);

}
```

O: Open-closed principle

“Software entities ... should be open for extension,
but closed for modification.”

This statement, at first, seems like a contradiction since it asks you to program entities (class/function/module) to be both open and closed. The open-closed principle (OCP) calls for entities that can be widely adapted but also remain unchanged. This leads us to create duplicate entities with specialized behavior through polymorphism.

Through polymorphism, we can extend our parent entity to suit the needs of the child entity while leaving the parent intact.

Our parent entity will serve as an abstract base class that can be reused with added specializations through inheritance. However, the original entity is locked to allow the program to be both open and closed.

The advantage of OCP is that it minimizes program risk when you add new uses for an entity. Instead of reworking the base class to fit a work-in-progress feature, you create a derived class separate from the classes currently present throughout the program.

We can then work on this unique derived class, confident that any changes we make to it will not affect the parent or any other

derived class.

Implementation

OCP implementations often rely on polymorphism and abstraction to code behavior at a class level rather than hard-coding for certain situations. Let’s see how we can correct an area calculator program to follow OSP:

```
// Does not follow OSP
public double Area(object[] shapes)
{
double area = 0;
for (Shape shape : 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;
}
public class AreaCalculator
{
public double Area(Rectangle[] shapes)
{
double area = 0;
for (Shape shape : shapes)
{
area += shape.Width*shape.Height;
}
return area;
}
}
```

This program does not follow OSP because Area() is not open to extension and can only ever handle Rectangle and Circle shapes. If we want to add support for Triangle, we'd have to modify the method, so it is not closed to modification.

We can achieve OSP by adding an abstract class Shape that all types of shapes inherit.

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

public class Rectangle extends Shape
{
//variables, getters, setters
public override double Area()
{
return Width*Height;
}
}
public class Circle extends Shape
{
public double Radius { // method body
 }
public override double Area()
{
return Radius*Radius*Math.PI;
}
}
public double Area(Shape[] shapes)
{
double area = 0;
for(Shape shape : shapes)
{
area += shape.Area();
}
return area;
}
```

Now each subtype of shape handles its own area calculation through polymorphism. This opens the Shape class to extension because a new shape can easily be added with its own area calculation without error.

Further, nothing in the program modifies the original shape, and it will not need to be modified in the future. As a result, the program now achieves the OCP principle.

L: Liskov substitution principle

“Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.”

The Liskov substitution principle (LSP) is a specific definition of a subtyping relation created by Barbara Liskov and Jeannette

Wing. The principle says that any class must be directly replaceable by any of its subclasses without error.

In other words, each subclass must maintain all behavior from the base class along with any new behaviors unique to the subclass. The child class must be able to process all the same requests and complete all the same tasks as its parent class.

In practice, programmers tend to develop classes based on behavior and grow behavioral capabilities as the class becomes more specific. The advantage of LSP is that it speeds up the development of new subclasses as all subclasses of the same type share a consistent use.

You can trust that all newly created subclasses will work with the existing code. If you decide that you need a new subclass, you can create it without reworking the existing code.

Let's do a simple example in Java:

Bad example

```
public class Bird{
    public void fly(){"fly Bird"}
}
public class Duck extends Bird{}
```

The duck can fly because it is a bird, but what about this:

```
public class Ostrich extends Bird{}
```

Ostrich is a bird, but it can't fly, Ostrich class is a subtype of class Bird, but it shouldn't be able to use the fly method, that means we are breaking the LSP principle.

Good example

```
public class Bird{}
public class FlyingBirds extends Bird{
    public void fly(){}
}
public class Duck extends FlyingBirds{}
public class Ostrich extends Bird{}
```

I: Interface segregation principle

“Many client-specific interfaces are better than one general-purpose interface.”

The interface segregation principle (ISP) requires that classes only be able to perform behaviors that are useful to achieve its end functionality. In other words, classes do not include behaviors they do not use.

This relates to our first SOLID principle in that together these two principles strip a class of all variables, methods, or behaviors that do not directly contribute to their role. Methods must contribute to the end goal in their entirety.

Any unused part of the method should be removed or split into a separate method.

The advantage of ISP is that it splits large methods into smaller, more specific methods. This makes the program easier to debug for three reasons:

1. There is less code carried between classes. Less code means fewer bugs.
2. A single method is responsible for a smaller variety of behaviors. If there is a problem with a behavior, you only need to look over the smaller methods.
3. If a general method with multiple behaviors is passed to a class that doesn't support all behaviors (such as calling for a property that the class doesn't have), there will be a bug if the class tries to use the unsupported behavior.

Implementation
To see how the ISP principle looks in code, let's see how a program changes with and without following the ISP principle.

First, the program that doesn't follow ISP:

```
// Not following the Interface Segregation Principle

public interface IWorker
{

string ID { get; set; }
string Name { get; set; }
string Email { get; set; }
float MonthlySalary { get; set; }
float OtherBenefits { get; set; }
float HourlyRate { get; set; }
float HoursInMonth { get; set; }
float CalculateNetSalary();
float CalculateWorkedSalary();
}
```

This program does not follow ISP because the FullTimeEmployee class does not need the CalculateWorkedSalary() function, and the ContractEmployeeclass does not need the CalculateNetSalary().

```
public class FullTimeEmployee : IWorker
{
public string ID { get; set; }
```

```
public string Name { get; set; }
public string Email { get; set; }
public float MonthlySalary { get; set; }
public float OtherBenefits { get; set; }
public float HourlyRate { get; set; }
public float HoursInMonth { get; set; }
public float CalculateNetSalary() => MonthlySalary + OtherBenefits;

public float CalculateWorkedSalary() => throw new
NotImplementedException();
}

public class ContractEmployee : IWorker
{
public string ID { get; set; }
public string Name { get; set; }
public string Email { get; set; }
public float MonthlySalary { get; set; }
public float OtherBenefits { get; set; }
public float HourlyRate { get; set; }
public float HoursInMonth { get; set; }
public float CalculateNetSalary() => throw new
NotImplementedException();
public float CalculateWorkedSalary() => HourlyRate * HoursInMonth;
}
```

Neither of these methods advance the goal of these classes. Instead, they are implemented because they are derived classes of the IWorkerinterface.

Here’s how we could refactor the program to follow the ISP principle:

In this version, we’ve split the general interface IWorker into one base interface, IBaseWorker, and two child interfaces IFullTimeWorkerSalary and IContractWorkerSalary.

```
// Following the Interface Segregation Principle

public interface IBaseWorker
{
string ID { get; set; }
string Name { get; set; }
string Email { get; set; }

}
```

```
public interface IFullTimeWorkerSalary : IBaseWorker
{
float MonthlySalary { get; set; }
float OtherBenefits { get; set; }
float CalculateNetSalary();

}

public interface IContractWorkerSalary : IBaseWorker
{
float HourlyRate { get; set; }
float HoursInMonth { get; set; }
float CalculateWorkedSalary();
}

public class FullTimeEmployeeFixed : IFullTimeWorkerSalary
{
public string ID { get; set; }
public string Name { get; set; }
public string Email { get; set; }
public float MonthlySalary { get; set; }
public float OtherBenefits { get; set; }
public float CalculateNetSalary() => MonthlySalary + OtherBenefits;
}

public class ContractEmployeeFixed : IContractWorkerSalary
{

public string ID { get; set; }
public string Name { get; set; }
public string Email { get; set; }
public float HourlyRate { get; set; }
public float HoursInMonth { get; set; }
public float CalculateWorkedSalary() => HourlyRate *
HoursInMonth;
}
```

The general interface contains methods that all workers share. The child interfaces split up methods by worker type, FullTime with a salary or Contract that gets paid hourly.

Now, our classes can implement the interface for that type of worker to access all methods and properties in the base class and the worker-specific interface.

The end classes now only contain methods and properties that further their goal and thus achieve the ISP principle.

D: Dependency inversion principle

“One should depend upon abstractions, [not] concretions.”

The dependency inversion principle (DIP) has two parts:

1. High-level modules should not depend on low-level modules. Instead, both should depend on abstractions (interfaces)
2. Abstractions should not depend on details. Details (like concrete implementations) should depend on abstractions. The first part of this principle reverses traditional OOP software design. Without DIP, programmers often construct programs to have high-level (less detail, more abstract) components explicitly connected with low-level (specific) components to complete tasks.

DIP decouples high and low-level components and instead connects both to abstractions. High and low-level components can still benefit from each other, but a change in one should not directly break the other.

The advantage of this part of DIP is that decoupled programs require less work to change. Webs of dependencies across your program mean that a single change can affect many separate parts.

If you minimize dependencies, changes will be more localized and require less work to find all affected components.

The second part can be thought of as “the abstraction is not affected if the details are changed”. The abstraction is the user-facing part of the program.

The details are the specific behind-the-scenes implementations that cause program behavior visible to the user. In a DIP program, we could fully overhaul the behind-the-scenes implementation of how the program achieves its behavior without the user’s knowledge.

This process is known as refactoring.

This means you won’t have to hard-code the interface to work solely with the current details (implementation). This keeps our code loosely coupled and allows us the flexibility to refactor our implementations later.

Implementation
We’ll create a general business program with an interface, high-level, low-level, and detailed components.

The principle states that we must use abstraction (abstract classes and interfaces) instead of concrete implementations. High-level modules should not depend on the low-level module but both should depend on the abstraction. Because the abstraction does not depend on detail but the detail depends on abstraction. It decouples the software. Let's understand the principle through an example.

```
1. public class WindowsMachine
2. {
3. public final keyboard;
4. public final monitor;
5. public WindowsMachine()
6. {
7. monitor = new monitor();  //instance of monitor class
8. keyboard = new keyboard(); //instance of keyboard class
9. }
10. }
```

Now we can work on the Windows machine with the help of a keyboard and mouse. But we still face the problem. Because we have tightly coupled the three classes together by using the new keyword. It is hard o test the class windows machine.

To make the code loosely coupled, we decouple the WindowsMachine from the keyboard by using the Keyboard interface and this keyword.

Keyboard.java

```
1. public interface Keyboard
2. {
3. //functionality
```

4. }

WindowsMachine.java

```
1. public class WindowsMachine
2. {
3. private final Keyboard keyboard;
4. private final Monitor monitor;
5. public WindowsMachine(Keyboard keyboard, Monitor monitor)
6. {
7. this.keyboard = keyboard;
8. this.monitor = monitor;
9. }
10. }
```

In the above code, we have used the dependency injection to add the keyboard dependency in the WindowsMachine class. Therefore, we have decoupled the classes.