


SOLID Principles

Contents

- What is SOLID
- Single responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

What is SOLID?

Single Responsibility Principle

Single Responsibility Principle

“A class should have one and only one reason to change”

Tight Coupling

Binds two (or more) details together in a way that’s difficult to change.

Loose Coupling

Offers a modular way to choose which details are involved in a particular operation.

Separation of Concerns

Programs should be separated into distinct sections, each addressing a separate concern, or set of information that affects the program.

Single Responsibility Principle

```
6   public class Employee
7   {
8       public double CalculatePay(Money money)
9       {
10          //business logic for payment here
11      }
12
13      public Employee Save(Employee employee)
14      {
15          //store employee here
16      }
17  }
```

Business logic

Persistence

There are two responsibilities

Single Responsibility Principle

How to solve this?

Single Responsibility Principle

```
21  public class Employee
22  {
23      public double CalculatePay(Money money)
24      {
25          //business logic for payment here
26      }
27  }
28
29  public class EmployeeRepository
30  {
31      public Employee Save(Employee employee)
32      {
33          //store employee here
34      }
35  }
```

Just create two different classes


Open/Closed Principle

Open/Closed Principle

*"Software entities should be **open** for extension, but **closed** for modification."*

Open/Closed Principle

```
40  public enum PaymentType = { Cash, CreditCard };
41
42  public class PaymentManager
43  {
44      public PaymentType PaymentType { get; set; }
45
46      public void Pay(Money money)
47      {
48          if(PaymentType == PaymentType.Cash)
49          {
50              //some code here - pay with cash
51          }
52          else
53          {
54              //some code here - pay with credit card
55          }
56      }
57  }
```

Humm...and if I need to add a new payment type?

You need to modificate this class.

Open/Closed Principle

open for extension

close for modification

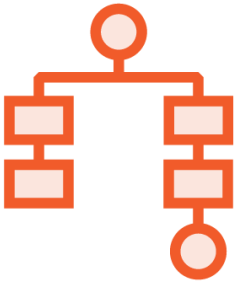
```
60    public class Payment
61    {
62        public virtual void Pay(Money money)
63        {
64            // from base
65        }
66    }
```

```
68    public class CashPayment : Payment
69    {
70        public override void Pay(Money money)
71        {
72            //some code here - pay with cash
73        }
74    }
```

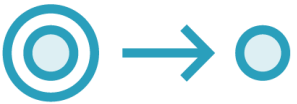
```
76    public class CreditCardPayment : Payment
77    {
78        public override void Pay(Money money)
79        {
80            //some code here - pay with credit card
81        }
82    }
```

Typical Approaches to OCP

Parameters



Inheritance



Composition / Injection

Extremely Concrete

```
public class DoOneThing
{
    public void Execute()
    {
        Console.WriteLine(“Hello world.”);
    }
}
```

Parameter-Based Extension

```
public class DoOneThing
{
    public void Execute(string message)
    {
        Console.WriteLine(message);
    }
}
```

Inheritance-based Extension

```
public class DoOneThing
{
    public virtual void Execute()
    {
        Console.WriteLine("Hello world.");
    }
}
public class DoAnotherThing
{
    public override void Execute()
    {
        Console.WriteLine("Goodbye world!");
    }
}
```

Composition/Injection Extension

```
public class DoOneThing
{
    private readonly MessageService _messageService;
    public DoOneThing(MessageService messageService)
        => _messageService = messageService;

    public void Execute()
    {
        Console.WriteLine(_messageService.GetMessage());
    }
}
```


Liskov Substitution Principle

Liskov Substitution Principle

"A subclass should behave in such a way that it will not cause problems when used instead of the superclass."

Liskov Substitution Principle

```
5   public class Employee
6   {
7       public virtual string GetProjectDetails(int employeeId)
8       {
9           Console.WriteLine("base project details");
10      }
11  }
```

```
13  public class CasualEmployee : Employee
14  {
15      public override string GetProjectDetails(int employeeId)
16      {
17          base.GetProjectDetails(employeeId);
18          Console.WriteLine("casual employee project details");
19      }
20  }
```

```
public class ContractualEmployee : Employee
{
    //broken your base class here
    public override string GetProjectDetails(int employeeId)
    {
        Console.WriteLine("contractual employee project details");
    }
}
```

Liskov Substitution Principle

```
5   public class Employee
6   {
7       public virtual string GetProjectDetails(int employeeId)
8       {
9           Console.WriteLine("base project details");
10      }
11  }
```

```
13  public class CasualEmployee : Employee
14  {
15      public override string GetProjectDetails(int employeeId)
16      {
17          base.GetProjectDetails(employeeId);
18          Console.WriteLine("casual employee project details");
19      }
20  }
```

```
public class ContractualEmployee : Employee
{
    public override string GetProjectDetails(int employeeId)
    {
        base.GetProjectDetails(employeeId);
        Console.WriteLine("contractual employee project details");
    }
}
```

Much better

Rectangle

```
public class Rectangle
{
    public virtual int Height { get; set; }
    public virtual int Width { get; set; }
}
```

Area Calculation Utility

```
public class AreaCalculator
{
    public static int CalculateArea(Rectangle r)
    {
        return r.Height * r.Width;
    }
}
```

Square (a Subtype of Rectangle)

```
public class Square : Rectangle
{
    private int _height;
    public int Height
    {

        get { return _height; }
        set
        {
            _width = value;
            _height = value;
        }
    }
    // Width implemented similarly
}
```

The Problem

```
Rectangle myRect = new Square();
myRect.Width = 4;
myRect.Height = 5;

Assert.Equal(20, AreaCalculator.CalculateArea(myRect));

// Actual Result: 25
```

One Solution

```
public class Rectangle
{
    public int Height { get; set; }
    public int Width { get; set; }

    public bool IsSquare => Height == Width;
}
```

Another Solution

```
public class Rectangle
{
    public int Height { get; set; }
    public int Width { get; set; }
}

public class Square
{
    public int Side { get; set; }
}
```


Interface Segregation Principle

Interface Segregation Principle

“Clients should not be forced to depend upon interfaces that they don't use”

Interface Segregation Principle

```
62  public interface IEmployee
63  {
64      string GetProjectDetails(int employeeId);
65
66      string GetEmployeeDetails(int employeeId);
67  }
68
```

Interface Segregation Principle

```
69  public class CasualEmployee : IEmployee
70  {
71      public string GetProjectDetails(int employeeId)
72      {
73          //code here - return base project details
74      }
75
76      public string GetEmployeeDetails(int employeeId)
77      {
78          //code here - specific casual employee details
79      }
```

WHY?????
I don't need you!!

```
82  public class ContractualEmployee : IEmployee
83  {
84      public string GetProjectDetails(int employeeId)
85      {
86          //code here - specific project details
87      }
88
89      public string GetEmployeeDetails(int employeeId)
90      {
91          throw new System.NotImplementedException();
92      }
93  }
```

Interface Segregation Principle

How to solve this?

Interface Segregation Principle

```
106   public interface IEmployee
107   {
108       string GetEmployeeDetails(int employeeId);
109   }
```

You need to create two interfaces

```
111   public interface IProject
112   {
113       string GetProjectDetails(int employeeId);
114   }
```

Interface Segregation Principle

```
116   public class CasualEmployee : IEmployee, IProject
117   {
118       public string GetEmployeeDetails(int employeeId)
119       {
120           //code here - specific casual employee details
121       }
122
123       public string GetProjectDetails(int employeeId)
124       {
125           //code here - specific contractual employee details
126       }
127   }
```

```
129   public class ContractualEmployee : IProject
130   {
131       public string GetProjectDetails(int employeeId)
132       {
133           //code here - specific project details
134       }
135   }
```


Dependency Inversion Principle

Dependency Inversion Principle

"High-level modules should not depend on low-level modules. Both should depend on abstractions."

"Abstractions should not depend upon details. Details should depend upon abstractions."

Dependency Inversion Principle

```
175  public class Email
176  {
177      public void SendEmail()
178      {
179          // code to send mail
180      }
181  }
```

```
183  public class Notification
184  {
185      private Email _email;
186      public Notification()
187      {
188          _email = new Email();
189      }
190
191      public void PromotionalNotification()
192      {
193          _email.SendEmail();
194      }
195  }
```

And if I need to send a notification by SMS?
You need to change this.

Dependency Inversion Principle

```
199   public interface IMessenger
200   {
201       void SendMessage();
202   }
```

So, I create an interface and now?

```
204   public class Email : IMessenger
205   {
206       public void SendMessage()
207       {
208           // code to send email
209       }
210   }
```

```
212   public class SMS : IMessenger
213   {
214       public void SendMessage()
215       {
216           // code to send SMS
217       }
218   }
```

Dependency Inversion Principle

```
220   public class Notification
221   {
222       private IMessenger _iMessenger;
223       public Notification()
224       {
225           _ iMessenger = new Email();
226       }
227       public void DoNotify()
228       {
229           _ iMessenger.SendMessage();
230       }
231   }
```

Dependency Inversion Principle

Constructor injection:

```
235  public class Notification
236  {
237      private IMessenger _iMessenger;
238      public Notification(Imessenger pMessenger)
239      {
240          _ iMessenger = pMessenger;
241      }
242      public void DoNotify()
243      {
244          _ iMessenger.SendMessage();
245      }
246  }
```

Dependency Inversion Principle

Property injection:

```
248   public class Notification
249   {
250       private IMessenger _iMessenger;
251
252       public IMessenger MessageService
253       {
254           private get;
255           set
256           {
257               _ iMessenger = value;
258           }
259       }
260
261       public void DoNotify()
262       {
263           _ iMessenger.SendMessage();
264       }
265   }
```

Dependency Inversion Principle

Method injection:

```
268  public class Notification
269  {
270      public void DoNotify(IMessenger pMessenger)
271      {
272          pMessenger.SendMessage();
273      }
274  }
```

Depending on Details

```
public interface IOrderDataAccess
{
    SqlDataReader ListOrders(SqlParameterCollection params);
}
```

Abstractions Should Not Depend on Details

```
public interface IOrderDataAccess
{
    List<Order> ListOrders(Dictionary<string,string> params);
}
```

Dependency Injection

Don’t create your own dependencies
- Depend on abstractions
- Request dependencies from client

Client injects dependencies as
- Constructor arguments
- Properties
- Method arguments

Literature

https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design

Thank You!