SOLID Principles

Presented by Mónica Rodrigues

January 24rd, 2017

Contents

Who am I?

Mónica Rodrigues

Degree in Computer Science Engineer at ISEL

Software engineer since 2009

Keywords: Web Applications, ASP.NET MVC, ASP.NET Web API, REST, Javascript, Angular, Best practices, Design patterns, SQL Server and much more..

FARFETCH

GEEK GIRLS PORTUGAL

netponto comunidade

What is SOLID?

S	O	L	I	D
SRP	OCP	LSP	ISP	DIP
Single Responsibility Principle	Open/Closed Principle	Liskovs Substitution Principle	Interface Segregation Principle	Dependency Inversion Principle

Single Responsibility Principle

"A class should have one and only one reason to change"

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

*"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    }
```

Liskov Substitution Principle

"Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T"

What do you say?

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

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

“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  }
196
```

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    }
```

Keep in mind

DRY - Don’t repeat yourself

+

SLAP - Single layer abstraction principle

+

SOLID

BEST DEVELOPER

Enjoy code and

Keep it simple!

Thank you