# SOLID Principles

Presented by Mónica Rodrigues

January 24rd, 2017

# Contents

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

#### 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..









#### What is SOLID?

S

SRP

Single Responsibility Principle 0

OCP

Open/Closed Principle

LSP

Liskovs Substitution Principle

**ISP** 

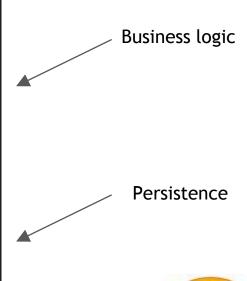
Interface Segregation Principle D

DIP

Dependency Inversion Principle

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

```
public class Employee
  1-1-{
         public double CalculatePay(Money money)
             //business logic for payment here
11
         public Employee Save(Employee employee)
             //store employee here
```





# How to solve this?



```
public class Employee
    public double CalculatePay(Money money)
        //business logic for payment here
public class EmployeeRepository
    public Employee Save(Employee employee)
        //store employee here
```



Just create two different classes

# Open/Closed Principle

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

#### Open/Closed Principle

```
public enum PaymentType = { Cash, CreditCard };
public class PaymentManager
    public PaymentType PaymentType { get; set; }
    public void Pay(Money money)
        if(PaymentType == PaymentType.Cash)
            //some code here - pay with cash
        else
            //some code here - pay with credit card
```



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

You need to modificate this class.

#### Open/Closed Principle

```
open for extension

60 public class Payment

62 public virtual void Pay(Money money)

63 {

64 // from base

65 }
```



```
public class CashPayment : Payment

public override void Pay(Money money)

full {
    //some code here - pay with cash
    }

}
```

```
public class CreditCardPayment : Payment

public override void Pay(Money money)

f

//some code here - pay with credit card

}

}
```

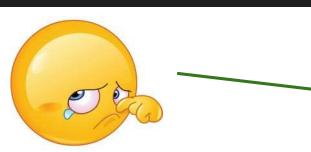
"Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T"

What do you say?



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

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



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

```
public class Employee

function of the public virtual string GetProjectDetails(int employeeId)

function of the public virtual string GetProjectDetails(int employeeId)

function of the public virtual string GetProjectDetails(int employeeId)

function of the public class Employee

function of the public class Employee

function of the public class Employee

function of the public virtual string GetProjectDetails(int employeeId)

fun
```

```
public class CasualEmployee : Employee
   public override string GetProjectDetails(int employeeId)
       base.GetProjectDetails(employeeId);
       Console.WriteLine("casual employee project details");
                                            public class ContractualEmployee : Employee
                                                public override string GetProjectDetails(int employeeId)
                                                    base.GetProjectDetails(employeeId);
                                                    Console.WriteLine("contractual employee project details");
                Much better
```

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

```
public interface IEmployee

find the string GetProjectDetails(int employeeId);

find the string GetEmployeeDetails(int employeeId);
```

```
public class CasualEmployee : IEmployee
     public string GetProjectDetails(int employeeId)
        //code here - return base project details
     public string GetEmployeeDetails(int employeeId)
        //code here - specific casual employee details
                                                   public class ContractualEmployee : IEmployee
                                                       public string GetProjectDetails(int employeeId)
                                                           //code here - specific project details
                                                       public string GetEmployeeDetails(int employeeId)
WHY?????
I don't need you!!
                                                           throw new System.NotImplementedException();
```

# How to solve this?



```
public interface IEmployee

for {
    string GetEmployeeDetails(int employeeId);
}
```



You need to create two interfaces

```
public interface IProject

public interface IProject

string GetProjectDetails(int employeeId);

114
}
```

```
public class CasualEmployee : IEmployee, IProject
{
    public string GetEmployeeDetails(int employeeId)
    {
        //code here - specific casual employee details
}

public string GetProjectDetails(int employeeId)

//code here - specific contractual employee details

//code here - specifi
```

```
public class ContractualEmployee : IProject

public string GetProjectDetails(int employeeId)

//code here - specific project details

//code here - specific project details

}
```

"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."

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



```
public class Notification
   private Email email;
   public .otification()
        email = new Email();
   public void PromotionalNotification()
        email.SendEmail();
```

```
public interface IMessenger

public interface IMessenger

void SendMessage();

public interface IMessenger

void SendMessenger

void SendMess
```

So, I create an interface and now?

```
public class Email : IMessenger

public void SendMessage()

public void SendMessage()

// code to send email

// code to send email
```

```
public class SMS : IMessenger

public void SendMessage()

public void SendMessage()

f

// code to send SMS

// code to send SMS

}
```

```
public class Notification
          private IMessenger iMessenger;
223
          public Notification()
225
                iMessenger = new Email();
          public void DoNotify()
228
                iMessenger.SendMessage();
231
```



#### Constructor injection:

```
public class Notification
    private IMessenger iMessenger;
    public Notification(Imessenger pMessenger)
          iMessenger = pMessenger;
    public void DoNotify()
          iMessenger.SendMessage();
```

Property injection:

```
public class Notification
249 E (
          private IMessenger iMessenger;
          public IMessenger MessageService
             private get;
             set
                   iMessenger = value;
          public void DoNotify()
                iMessenger.SendMessage();
```

Method injection:

```
public class Notification

public void DoNotify(IMessenger pMessenger)

pmessenger.SendMessage();

pmessenger.SendMessage();
```

# Keep in mind

DRY - Don't repeat yourself

+

SLAP - Single layer abstraction principle

+

SOLID

BEST DEVELOPER



Enjoy code and

Keep it simple!



Thank you