



SOLID Principle with C# Example



Subhhajit Bhadury

27 Sep 2015 CPOL

SOLID principle with C# example

Introduction

If you are going to search in Google with this word 'SOLID', you will get tons of articles, videos, tutorials on this topic. My aim is to make this article as simple as possible and provide small but good examples.

And I can provide all those examples in C#. But don't worry, get the concept and keep practicing with the programming language that you are familiar with.

1. About SOLID

SOLID is basically 5 principles, which will help to create a good software architecture. You can see that all design patterns are based on these principles. SOLID is basically an acronym of the following:

- **S** is single responsibility principle (SRP)
- **O** stands for open closed principle (OCP)
- **L** Liskov substitution principle (LSP)
- **I** interface segregation principle (ISP)
- **D** Dependency injection principle (DIP)

I believe that with pictures, with examples, an article will be more approachable and understandable.

1.1 Single responsibility principle (SRP)

A class should take one responsibility and there should be one reason to change that class. Now what does that mean? I want to share one picture to give a clear idea about this.

Now see this tool is a combination of so many different tools like knife, nail cutter, screw driver, etc. So will you want to buy this tool? I don't think so. Because there is a problem with this tool, if you want to add any other tool to it, then you need to change the base and that is not good. This is a bad architecture to introduce into any system. It will be better if nail cutter can only be used to cut the nail or knife can only be used to cut vegetables.

Now I want to give one C# example on this principle:

```
namespace SRP
{
    public class Employee
    {
        public int Employee_Id { get; set; }
        public string Employee_Name { get; set; }

        /// <summary>
        /// This method used to insert into employee table
        /// </summary>
    }
}
```

```

    /// <param name="em">Employee object</param>
    /// <returns>Successfully inserted or not</returns>
    public bool InsertIntoEmployeeTable(Employee em)
    {
        // Insert into employee table.
        return true;
    }
    /// <summary>
    /// Method to generate report
    /// </summary>
    /// <param name="em"></param>
    public void GenerateReport(Employee em)
    {
        // Report generation with employee data using crystal report.
    }
}

```

'Employee' class is taking 2 responsibilities, one is to take responsibility of employee database operation and another one is to generate employee report. **Employee** class should not take the report generation responsibility because suppose some days after your customer asked you to give a facility to generate the report in Excel or any other reporting format, then this class will need to be changed and that is not good.

So according to SRP, one class should take one responsibility so we should write one different class for report generation, so that any change in report generation should not affect the '**Employee**' class.

```

public class ReportGeneration
{
    /// <summary>
    /// Method to generate report
    /// </summary>
    /// <param name="em"></param>
    public void GenerateReport(Employee em)
    {
        // Report reneration with employee data.
    }
}

```

2.2 Open closed principle (OCP)

Now take the same '**ReportGeneration**' class as an example of this principle. Can you guess what is the problem with the below class!!

```

public class ReportGeneration
{
    /// <summary>
    /// Report type
    /// </summary>
    public string ReportType { get; set; }

    /// <summary>
    /// Method to generate report
    /// </summary>
    /// <param name="em"></param>
    public void GenerateReport(Employee em)
    {
        if (ReportType == "CRS")
        {
            // Report generation with employee data in Crystal Report.
        }
        if (ReportType == "PDF")
        {
            // Report generation with employee data in PDF.
        }
    }
}

```

Brilliant!! Yes you are right, too much 'If' clauses are there and if we want to introduce another new report type like 'Excel', then you need to write another 'if'. This class should be open for extension but closed for modification. But how to do that!!

```
public class IReportGeneration
{
    /// <summary>
    /// Method to generate report
    /// </summary>
    /// <param name="em"></param>
    public virtual void GenerateReport(Employee em)
    {
        // From base
    }
}
/// <summary>
/// Class to generate Crystal report
/// </summary>
public class CrystalReportGeneraion : IReportGeneration
{
    public override void GenerateReport(Employee em)
    {
        // Generate crystal report.
    }
}
/// <summary>
/// Class to generate PDF report
/// </summary>
public class PDFReportGeneraion : IReportGeneration
{
    public override void GenerateReport(Employee em)
    {
        // Generate PDF report.
    }
}
```

So if you want to introduce a new report type, then just inherit from **IReportGeneration**. So **IReportGeneration** is open for extension but closed for modification.

2.3 Liskov substitution principle (LSP)

This principle is simple but very important to understand. Child class should not break parent class's type definition and behavior. Now what is the meaning of this!! Ok let me take the same **employee** example to make you understand this principle. Check the below picture. **Employee** is a parent class and **Casual** and **Contractual employee** are the child classes, inhering from **employee** class.

Now see the below code:

```
public abstract class Employee
{
    public virtual string GetProjectDetails(int employeeId)
```

```

    {
        return "Base Project";
    }
    public virtual string GetEmployeeDetails(int employeeId)
    {
        return "Base Employee";
    }
}
public class CasualEmployee : Employee
{
    public override string GetProjectDetails(int employeeId)
    {
        return "Child Project";
    }
    // May be for contractual employee we do not need to store the details into database.
    public override string GetEmployeeDetails(int employeeId)
    {
        return "Child Employee";
    }
}
public class ContractualEmployee : Employee
{
    public override string GetProjectDetails(int employeeId)
    {
        return "Child Project";
    }
    // May be for contractual employee we do not need to store the details into database.
    public override string GetEmployeeDetails(int employeeId)
    {
        throw new NotImplementedException();
    }
}

```

Up to this is fine right? Now, check the below code and it will violate the LSP principle.

```

List<Employee> employeeList = new List<Employee>();
employeeList.Add(new ContractualEmployee());
employeeList.Add(new CasualEmployee());
foreach (Employee e in employeeList)
{
    e.GetEmployeeDetails(1245);
}

```

Now I guess you got the problem. Yes right, for **contractual employee**, you will get not implemented exception and that is violating LSP. Then what is the solution? Break the whole thing in 2 different interfaces, 1. **IProject** 2. **IEmployee** and implement according to **employee** type.

```

public interface IEmployee
{
    string GetEmployeeDetails(int employeeId);
}

public interface IProject
{
    string GetProjectDetails(int employeeId);
}

```

Now, **contractual employee** will implement **IEmployee** not **IProject**. This will maintain this principle.

2.4 Interface segregation principle (ISP)

This principle states that any client should not be forced to use an interface which is irrelevant to it. Now what does this mean, suppose there is one database for storing data of all types of employees (i.e. Permanent, non-permanent), now what will be the best approach for our interface?

```
public interface IEmployee
{
    bool AddEmployeeDetails();
}
```

And all types of **employee** class will inherit this interface for saving data. This is fine right? Now suppose that company one day told to you that they want to read only data of permanent employees. What you will do, just add one method to this interface?

```
public interface IEmployeeDatabase
{
    bool AddEmployeeDetails();
    bool ShowEmployeeDetails(int employeeId);
}
```

But now we are breaking something. We are forcing non-permanent **employee** class to show their details from database. So, the solution is to give this responsibility to another interface.

```
public interface IAddOperation
{
    bool AddEmployeeDetails();
}
public interface IGetOperation
{
    bool ShowEmployeeDetails(int employeeId);
}
```

And non-permanent **employee** will implement only **IAddOperation** and permanent **employee** will implement both the interface.

2.5 Dependency inversion principle (DIP)

This principle tells you not to write any tightly coupled code because that is a nightmare to maintain when the application is growing bigger and bigger. If a class depends on another class, then we need to change one class if something changes in that dependent class. We should always try to write loosely coupled class.

Suppose there is one notification system after saving some details into database.

```
public class Email
{
    public void SendEmail()
    {
        // code to send mail
    }
}

public class Notification
{
    private Email _email;
    public Notification()
    {
        _email = new Email();
    }

    public void PromotionalNotification()
    {
        _email.SendEmail();
    }
}
```

Now **Notification** class totally depends on **Email** class, because it only sends one type of notification. If we want to introduce any other like SMS then? We need to change the notification system also. And this is called tightly coupled. What can we do to make it loosely coupled? Ok, check the following implementation.

```

public interface IMessenger
{
    void SendMessage();
}
public class Email : IMessenger
{
    public void SendMessage()
    {
        // code to send email
    }
}

public class SMS : IMessenger
{
    public void SendMessage()
    {
        // code to send SMS
    }
}

public class Notification
{
    private IMessenger _iMessenger;
    public Notification()
    {
        _ iMessenger = new Email();
    }
    public void DoNotify()
    {
        _ iMessenger.SendMessage();
    }
}

```

Still **Notification** class depends on **Email** class. Now, we can use dependency injection so that we can make it loosely coupled. There are 3 types to DI, Constructor injection, Property injection and method injection.

Constructor Injection

```

public class Notification
{
    private IMessenger _iMessenger;
    public Notification(IMessenger pMessenger)
    {
        _ iMessenger = pMessenger;
    }
    public void DoNotify()
    {
        _ iMessenger.SendMessage();
    }
}

```

Property Injection

```

public class Notification
{
    private IMessenger _iMessenger;

    public Notification()
    {
    }
    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();  
    }  
}
```

So, SOLID principle will help us to write loosely coupled code which is highly maintainable and less error prone.

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author




Subhhajit Bhadury

India 

No Biography provided

Comments and Discussions

 **33 messages** have been posted for this article Visit <https://www.codeproject.com/Tips/1033646/SOLID-Principle-with-Csharp-Example> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#)
[Advertise](#)
[Privacy](#)
[Cookies](#)
[Terms of Use](#)

Article Copyright 2015 by Subhhajit Bhadury
Everything else Copyright © [CodeProject](#), 1999-2020

Web02 2.8.200113.1