Contents

SOLID Principles

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

Single Responsibility Principle (SRP)

- A class should have a single responsibility
- Separation of Concerns (*SoC*)

Open Closed Principle (OCP)

- Avoid too many “if”, “switch-case” statements
- Class should be ***open for extension*** but ***closed for modification***
- Example:

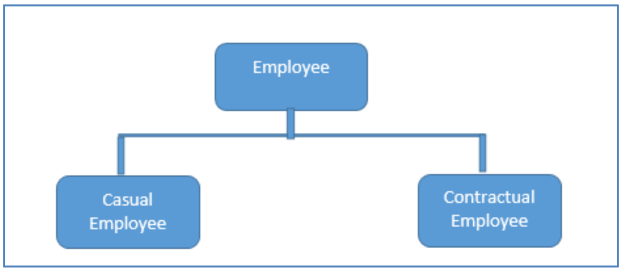
```
public class ReportGeneration
{
    /// <summary>
    /// Report type
    /// </summary>
    public string ReportType { get; set; }

    /// <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.
        }
     }
 }
```

```
    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.
        }
    }
```

Liskov Substitution Principle (LSP)

- Child class should not break parent class's type definition and behaviour
- Example:



```
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();
    }
}
```

- Now, based on the above class hierarchy, the following code will violate the LSP:

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

- For `contractual employee`, you will get not implemented exception and that is violating LSP
- Solution? :

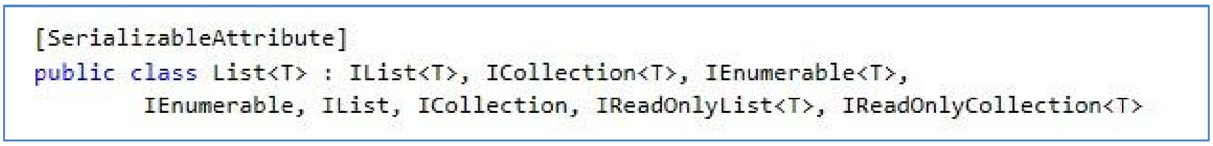
```
public interface IEmployee
{
    string GetEmployeeDetails(int employeeId);
}

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

- Now, `contractual employee` will implement `IEmployee` not `IProject`.

Interface Segregation Principle (ISP)

- Any client should not be forced to use an interface which is irrelevant to it
- For e.g.; `List` inherits from eight different interfaces



```
[SerializableAttribute]
public class List<T> : IList<T>, ICollection<T>, IEnumerable<T>,
        IEnumerable, IList, ICollection, IReadOnlyList<T>, IReadOnlyCollection<T>
```

- Example:

```
public interface IMessage
{
    IList<string> SendToAddress { get; set; }
    string Subject { get; set; }
    string MessageText { get; set; }
    bool Send();
}

public class EmailMessage : IMessage
{
    IList<string> SendToAddress { get; set; }
    string Subject { get; set; }
    string MessageText { get; set; }

    bool Send()
    {
        // Contact SMTP server and send message
    }
}
```

- The team now needs to also send SMS or text messages and decides to leverage the existing interface

```
public class SMSMessage : IMessage
{
    IList<string> SendToAddress { get; set; }
    string MessageText { get; set; }
    string Subject
    {
        get { throw new NotImplementedException(); }
        set { throw new NotImplementedException(); }
    }

    bool Send()
    {
        // Contact SMS server and send message
    }
}
```

- Because SMS doesn’t have a Subject, an exception is thrown. You can’t simply take out Subject because it’s required by the interface. It can get worse if the team decides to add CCToAddress

```
public interface IMessage
{
    IList<string> SendToAddress { get; set; }
    IList<string> CCToAddress { get; set; }
    string Subject { get; set; }
    string MessageText { get; set; }
    bool Send();
}

public class SMSMessage : IMessage
{
    IList<string> SendToAddress { get; set; }
    string MessageText { get; set; }
    string Subject
    {
        get { throw new NotImplementedException(); }
        set { throw new NotImplementedException(); }
    }

    string CCToAddress
    {
        get { throw new NotImplementedException(); }
        set { throw new NotImplementedException(); }
    }

    bool Send()
    {
        // Contact SMS server and send message
    }
}
```

- It would get even worse with BCCToAddress and email attachments

Applying Interface Segregation Principle

- A better way is to put the interface on a diet and have it comply with the Interface Segregation Principle

```
public interface IMessage
{
    IList<string> SendTo { get; set; }
    string MessageText { get; set; }
    bool Send();
}

public interface IEmailMessage
{
    IList<string> CCTo { get; set; }
    IList<string> BCCTo { get; set; }
    IList<string> AttachementFilePaths { get; set; }
    string Subject { get; set; }
}

public class EmailMessage : IMessage, IEmailMessage
{
    IList<string> SendTo { get; set; }
    IList<string> CCTo { get; set; }
    IList<string> BCCTo { get; set; }
    IList<string> AttachementFilePaths { get; set; }
    string Subject { get; set; }
    string MessageText { get; set; }

    bool Send()
    {
        // Contact SMTP server and send message
    }
}

public class SMSMessage : IMessage
{
    IList<string> SendTo { get; set; }
    string MessageText { get; set; }

    bool Send()
    {
        // Contact SMS server and send message
    }
}
```

- **So, put your interfaces on a diet**

Dependency Inversion Principle (DIP)

- Repository example OR Messenger example as follows:

```
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
- Make it loosely coupled. How? Use ctor injection

```
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 pMessenger)
    {
        _ iMessenger = pMessenger;
    }
    public void DoNotify()
    {
        _ iMessenger.SendMessage();
    }
}
```

- And how to use it?

```
public static void Main(string[] args)
{
   // Send an Email.
   Email emailMessage = new EmailMessage();
   Notification notifyByEmail = new Notification(emailMessage);
   notifyByEmail.DoNotify();

   // Send an SMS.
   SMS smsMessage = new SMS();
   Notification notifyBySMS = new Notification(smsMessage);
   notifyBySMS.DoNotify();
}
```