SOLID PRINCIPLE

What is Solid Principle?

Developers start building applications with good and tidy designs using their knowledge and experience. But over time, applications might develop bugs. The application design must be altered for every change request or new feature request. After some time we might need to put in a lot of effort, even for simple tasks, which might require full working knowledge of the entire system. But we can't blame change or new feature requests. They are part of software development. We can't stop them or refuse them either. So who is the culprit here? Obviously, it is the design of the application.

The following are the design flaws that cause damage to software, mostly.

1. Putting more stress on classes by assigning more responsibilities to them. (A lot of functionality not related to a class.)
2. Forcing the classes to depend on each other. If classes are dependent on each other (in other words tightly coupled), then a change in one will affect the other.
3. Spreading duplicate code in the system/application.

SOLID principles are the design principles that enable us to manage most software design problems. Robert C. Martin compiled these principles in the 1990s. These principles provide us with ways to move from tightly coupled code and little encapsulation to the desired results of loosely coupled and encapsulated real needs of a business properly. SOLID is an acronym for the following.

- S: Single Responsibility Principle (SRP)
- O: Open-closed Principle (OCP)
- L: Liskov substitution Principle (LSP)
- I: Interface Segregation Principle (ISP)
- D: Dependency Inversion Principle (DIP)

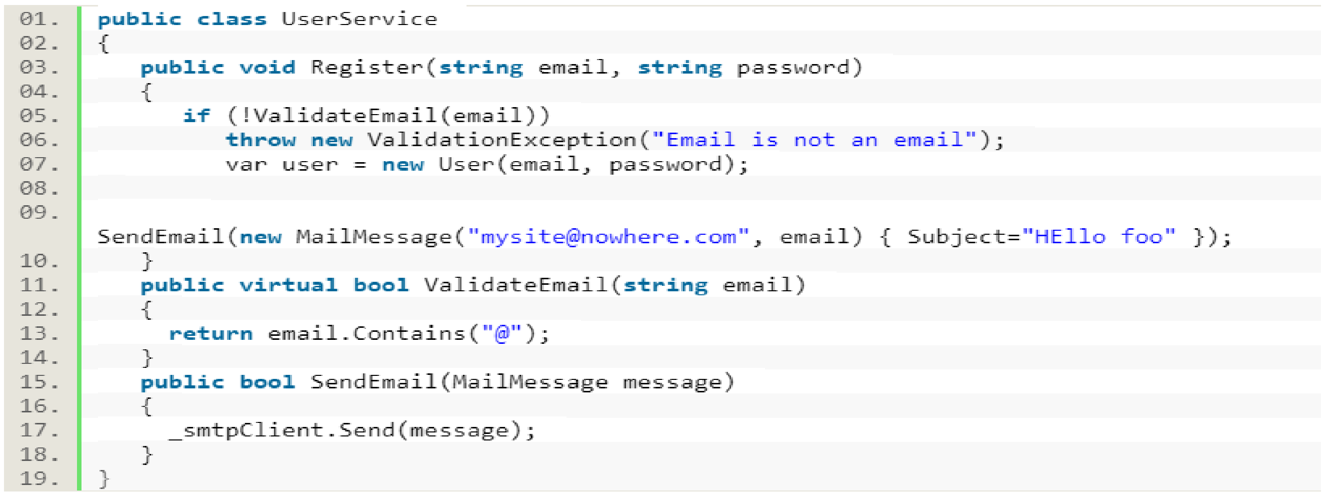
Single Responsibility Principle (SRP):

SRP says "Every software module should have only one reason to change".

SINGLE RESPONSIBILITY PRINCIPLE
Just Because You Can, Doesn't Mean You Should

This means that every class, or similar structure, in your code should have only one job to do. Everything in that class should be related to a single purpose. Our class should not be like a Swiss knife wherein if one of them needs to be changed then the entire tool needs to be altered. It does not mean that your classes should only contain one method or property. There may be many members as long as they relate to a single responsibility.

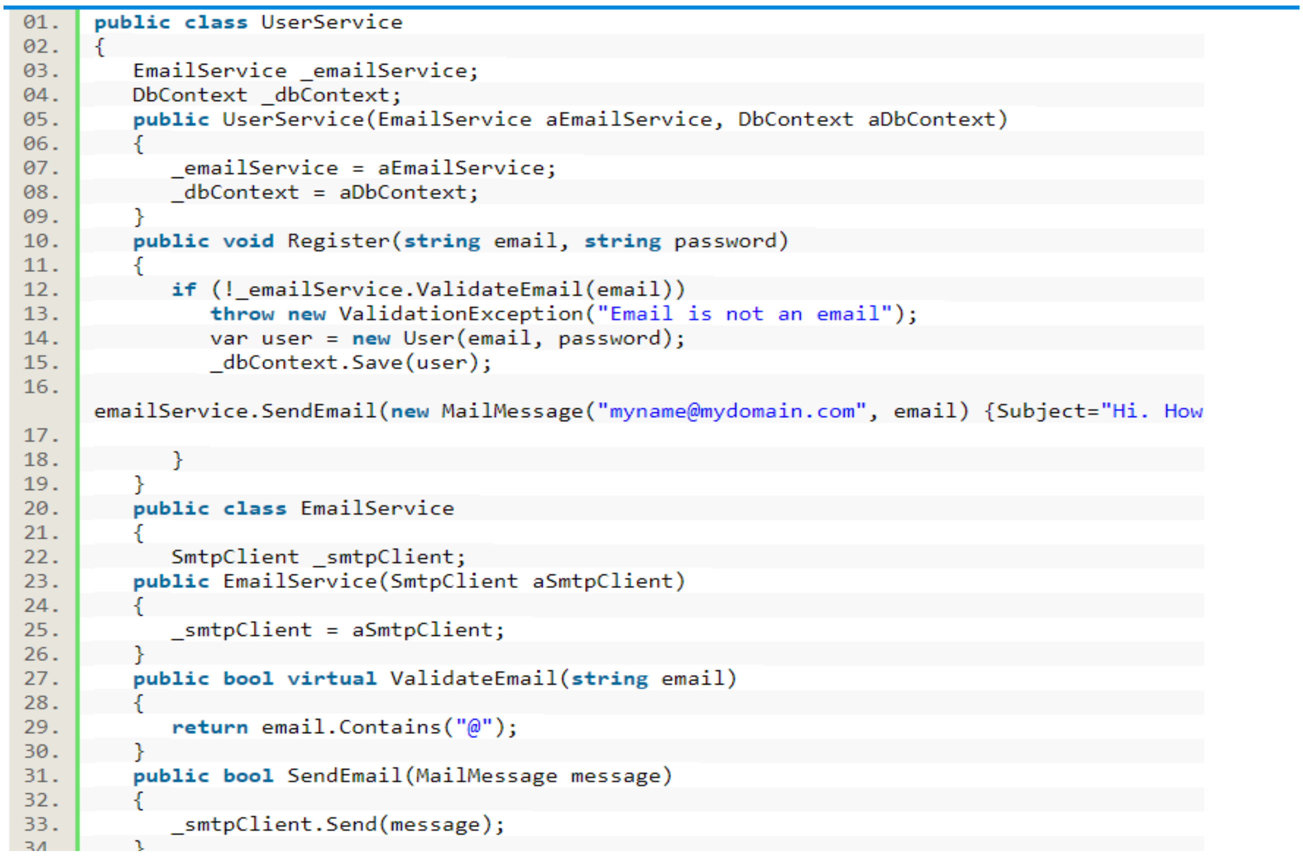
The Single Responsibility Principle gives us a good way of identifying classes at the design phase of an application and it makes you think of all the ways a class can change. A good separation of responsibilities is done only when we have the full picture of how the application should work. Let us check this with an example. In simple terms, a module or class should have a very small piece of responsibility in the entire application. Or as it states, a class/module should have no more than one reason to change.



```
public class UserService
{
   public void Register(string email, string password)
   {
      if (!ValidateEmail(email))
         throw new ValidationException("Email is not an email");
         var user = new User(email, password);


SendEmail(new MailMessage("mysite@nowhere.com", email) { Subject="HEllo foo" });
   }
   public virtual bool ValidateEmail(string email)
   {
     return email.Contains("@");
   }
   public bool SendEmail(MailMessage message)
   {
     _smtpClient.Send(message);
   }
}
```

It looks fine, but it is not following SRP. The SendEmail and ValidateEmail methods have nothing to do with the UserService class. Let's refract it.



```
public class UserService
{
   EmailService _emailService;
   DbContext _dbContext;
   public UserService(EmailService aEmailService, DbContext aDbContext)
   {
      _emailService = aEmailService;
      _dbContext = aDbContext;
   }
   public void Register(string email, string password)
   {
      if (!_emailService.ValidateEmail(email))
         throw new ValidationException("Email is not an email");
         var user = new User(email, password);
         _dbContext.Save(user);

emailService.SendEmail(new MailMessage("myname@mydomain.com", email) {Subject="Hi. How

      }
   }
   public class EmailService
   {
      SmtpClient _smtpClient;
   public EmailService(SmtpClient aSmtpClient)
   {
      _smtpClient = aSmtpClient;
   }
   public bool virtual ValidateEmail(string email)
   {
      return email.Contains("@");
   }
   public bool SendEmail(MailMessage message)
   {
      _smtpClient.Send(message);
```

Open/Closed Principle

The Open/closed Principle says "A software module/class is open for extension and closed for modification". Bertrand Meyer is generally credited for having

originated the definition of open/closed principle in his book Object-Oriented Software Construction.
Here "Open for extension" means, we need to design our module/class in such a way that the new functionality can be added only when new requirements are generated. "Closed for modification" means we have already developed a class and it has gone through unit testing. We should then not alter it until we find bugs. This principle suggests that the class should be easily extended but there is no need to change its core implementations. The application or software should be flexible to change. How change management is implemented in a system has a significant impact on the success of that application/ software. The OCP states that the behaviors of the system can be extended without having to modify its existing implementation. New features should be implemented using the new code, but not by changing existing code. The main benefit of adhering to OCP is that it potentially streamlines code maintenance and reduces the risk of breaking the existing implementation.
As it says, a class should be open for extensions, we can use inheritance to do this. Okay, let's dive into an example.

Suppose we have a Rectangle class with the properties Height and Width.

```
public class Rectangle{
   public double Height {get;set;}
   public double Wight {get;set; }
}
```

Our app needs the ability to calculate the total area of a collection of Rectangles. Since we already learned the Single Responsibility Principle (SRP), we don't need to put the total area calculation code inside the rectangle. So here I created another class for area calculation.

```
public class AreaCalculator {
   public double TotalArea(Rectangle[] arrRectangles)
   {
      double area;
      foreach(var objRectangle in arrRectangles)
      {
         area += objRectangle.Height * objRectangle.Width;
      }
      return area;
   }
}
```

We made our app without violating SRP. No issues for now. But can we extend our app so that it could calculate the area of not only Rectangles but also the area of Circles as well? Now we have an issue with the area calculation issue because the way to do circle area calculation is different. Hmm. Not a big deal. We can change the TotalArea method a bit so that it can accept an array of objects as an argument. We check the object type in the loop and do area calculations based on the object type.

```
public class Rectangle{
   public double Height {get;set;}
   public double Wight {get;set; }
}
public class Circle{
   public double Radius {get;set;}
}
public class AreaCalculator
{
   public double TotalArea(object[] arrObjects)
   {
      double area = 0;
      Rectangle objRectangle;
      Circle objCircle;
      foreach(var obj in arrObjects)
      {
         if(obj is Rectangle)
         {
            area += obj.Height * obj.Width;
         }
         else
         {
            objCircle = (Circle)obj;
            area += objCircle.Radius * objCircle.Radius * Math.PI;
         }
      }
      return area;
   }
}
```

Wow. We are done with the change. Here we successfully introduced Circle into our app. We can add a Triangle and calculate its area by adding one more "if" block in the TotalArea method of AreaCalculator. But every time we introduce a new shape we need to alter the TotalArea method. So the AreaCalculator class is not closed for modification. How can we make our design to avoid this situation? Generally, we can do this by referring to abstractions for dependencies, such as interfaces or abstract classes, rather than using concrete classes. Such interfaces can be fixed once developed so the classes that depend upon them can rely upon unchanging abstractions. Functionality can be added by creating new classes that implement the interfaces. So let's refract our code using an interface.

```
public abstract class Shape
{
   public abstract double Area();
}
```

Inheriting from Shape, the Rectangle and Circle classes now look like this:

```
public class Rectangle: Shape
{
    public double Height {get;set;}
    public double Width {get;set;}
    public override double Area()
    {
        return Height * Width;
    }
}
public class Circle: Shape
{
    public double Radius {get;set;}
    public override double Area()
    {
        return Radius * Radus * Math.PI;
    }
}
```

Every shape contains its area with its own way of calculation functionality and our AreaCalculator class will become simpler than before.

```
public class AreaCalculator
{
    public double TotalArea(Shape[] arrShapes)
    {
        double area=0;
        foreach(var objShape in arrShapes)
        {
            area += objShape.Area();
        }
        return area;
    }
}
```

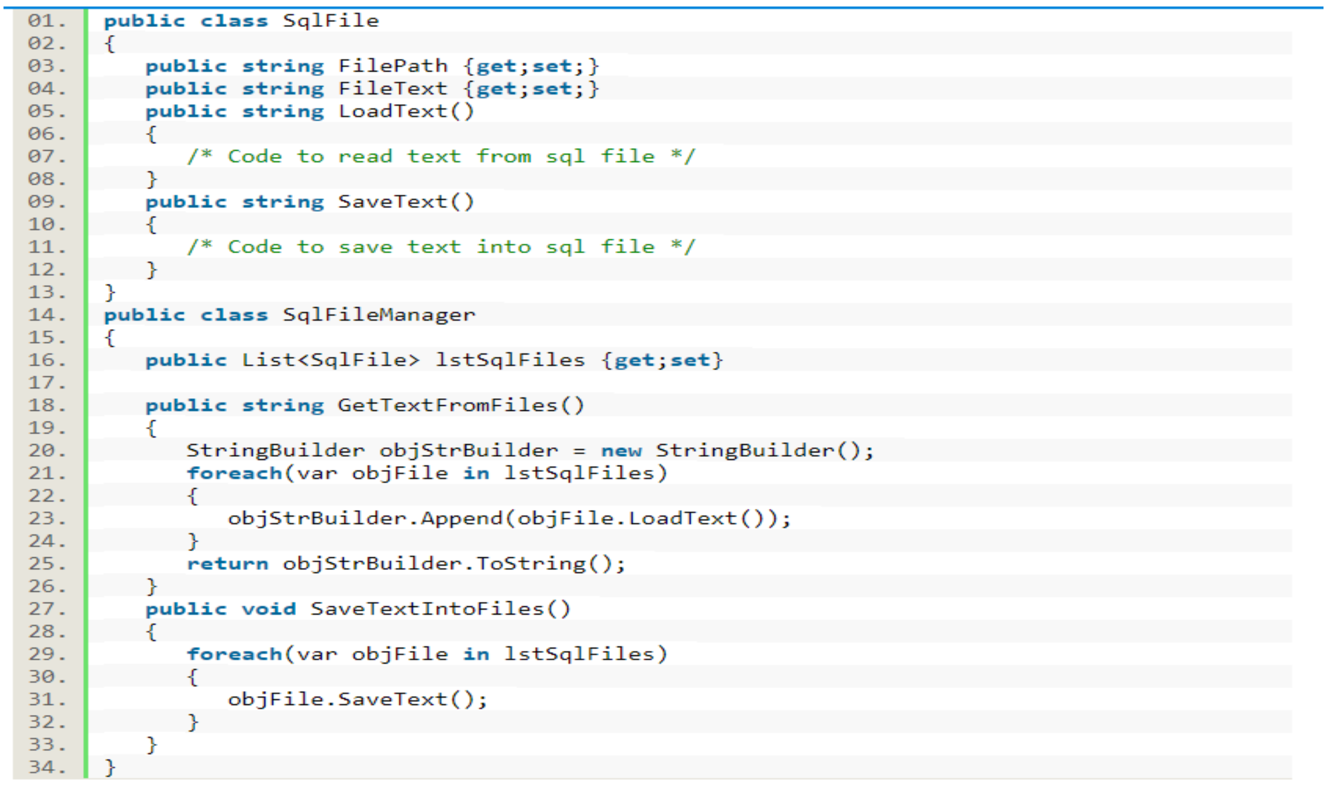
Now our code is following SRP and OCP both. Whenever you introduce a new shape by deriving from the "Shape" abstract class, you need not change the "AreaCalculator" class.

Liskov Substitution Principle

The Liskov Substitution Principle (LSP) states that "you should be able to use any derived class instead of a parent class and have it behave in the same manner without modification". It ensures that a derived class does not affect the behavior of the parent class, in other words,, that a derived class must be substitutable for its base class.

This principle is just an extension of the Open Closed Principle and it means that we must ensure that new derived classes extend the base classes without changing their behavior. I will explain this with a real-world example that violates LSP.

A father is a doctor whereas his son wants to become a cricketer. So here the son can't replace his father even though they both belong to the same family hierarchy.
Now jump into an example to learn how a design can violate LSP. Suppose we need to build an app to manage data using a group of SQL files text. Here we need to write functionality to load and save the text of a group of SQL files in the application directory. So we need a class that manages the load and saves the text of group of SQL files along with the SqlFile Class.



```
public class SqlFile
{
    public string FilePath {get;set;}
    public string FileText {get;set;}
    public string LoadText()
    {
        /* Code to read text from sql file */
    }
    public string SaveText()
    {
        /* Code to save text into sql file */
    }
}
public class SqlFileManager
{
    public List<SqlFile> lstSqlFiles {get;set}

    public string GetTextFromFiles()
    {
        StringBuilder objStrBuilder = new StringBuilder();
        foreach(var objFile in lstSqlFiles)
        {
            objStrBuilder.Append(objFile.LoadText());
        }
        return objStrBuilder.ToString();
    }
    public void SaveTextIntoFiles()
    {
        foreach(var objFile in lstSqlFiles)
        {
            objFile.SaveText();
        }
    }
}
```

After some time our leaders might tell us that we may have a few read-only files in the application folder, so we need to restrict the flow whenever it tries to do a save on them. We can do that by creating a "ReadOnlySqlFile" class that inherits the "SqlFile" class and we need to alter the SaveTextIntoFiles() method by introducing a condition to prevent calling the SaveText() method on ReadOnlySqlFile instances.

```
public class SqlFile
{
    public string LoadText()
    {
    /* Code to read text from sql file */
    }
    public void SaveText()
    {
        /* Code to save text into sql file */
    }
}
public class ReadOnlySqlFile: SqlFile
{
    public string FilePath {get;set;}
    public string FileText {get;set;}
    public string LoadText()
    {
        /* Code to read text from sql file */
    }
    public void SaveText()
    {
        /* Throw an exception when app flow tries to do save. */
        throw new IOException("Can't Save");
    }
}
```

To avoid an exception we need to modify "SqlFileManager" by adding one condition to the loop.

```
public class SqlFileManager
{
    public List<SqlFile? lstSqlFiles {get;set}
    public string GetTextFromFiles()
    {
        StringBuilder objStrBuilder = new StringBuilder();
        foreach(var objFile in lstSqlFiles)
        {
            objStrBuilder.Append(objFile.LoadText());
        }
        return objStrBuilder.ToString();
    }
    public void SaveTextIntoFiles()
    {
        foreach(var objFile in lstSqlFiles)
        {
            //Check whether the current file object is read-only
or not.If yes, skip calling it's
            // SaveText() method to skip the exception.

            if(! objFile is ReadOnlySqlFile)
            objFile.SaveText();
        }
    }
}
```

Here we altered the SaveTextIntoFiles() method in the SqlFileManager class to determine whether or not the instance is of ReadOnlySqlFile to avoid the exception. We can't use this ReadOnlySqlFile class as a substitute for its parent without altering SqlFileManager code. So we can say that this design is not following LSP. Let's make this design follow the LSP. Here we will introduce interfaces to make the SqlFileManager class independent from the rest of the blocks.

```
public interface IReadableSqlFile
{
   string LoadText();
}
public interface IWritableSqlFile
{
   void SaveText();
}
```

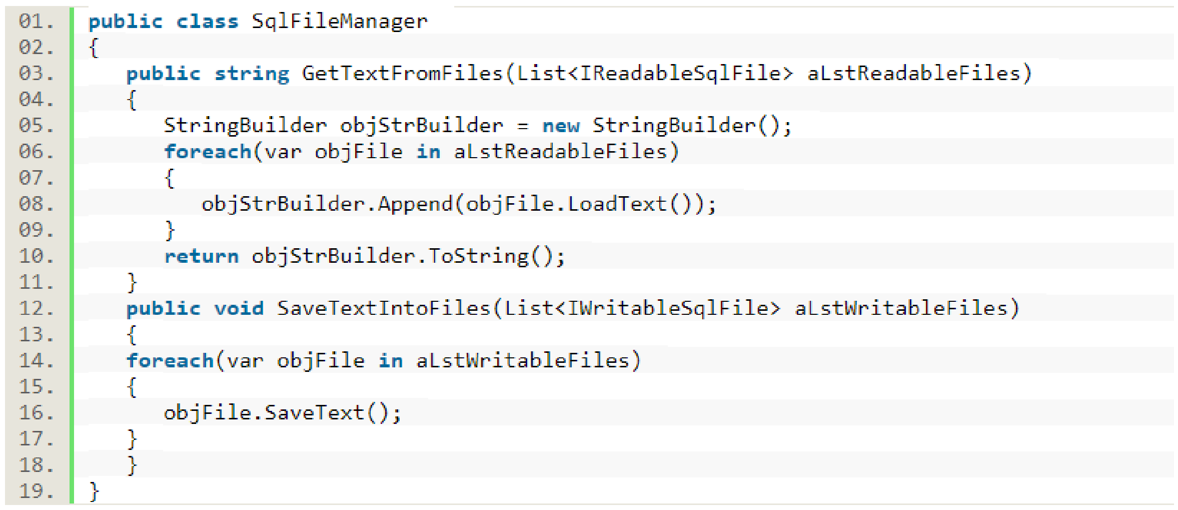
Now we implement IReadableSqlFile through the ReadOnlySqlFile class that reads only the text from read-only files.

```
public class ReadOnlySqlFile: IReadableSqlFile
{
   public string FilePath {get;set;}
   public string FileText {get;set;}
   public string LoadText()
   {
      /* Code to read text from sql file */
   }
}
```

Here we implement both IWritableSqlFile and IReadableSqlFile in a SqlFile class by which we can read and write files.

```
public class SqlFile: IWritableSqlFile,IReadableSqlFile
{
   public string FilePath {get;set;}
   public string FileText {get;set;}
   public string LoadText()
   {
      /* Code to read text from sql file */
   }
   public void SaveText()
   {
      /* Code to save text into sql file */
   }
}
```

Now the design of the SqlFileManager class becomes like this:



```
public class SqlFileManager
{
   public string GetTextFromFiles(List<IReadableSqlFile> aLstReadableFiles)
   {
      StringBuilder objStrBuilder = new StringBuilder();
      foreach(var objFile in aLstReadableFiles)
      {
         objStrBuilder.Append(objFile.LoadText());
      }
      return objStrBuilder.ToString();
   }
   public void SaveTextIntoFiles(List<IWritableSqlFile> aLstWritableFiles)
   {
   foreach(var objFile in aLstWritableFiles)
   {
      objFile.SaveText();
   }
   }
}
```

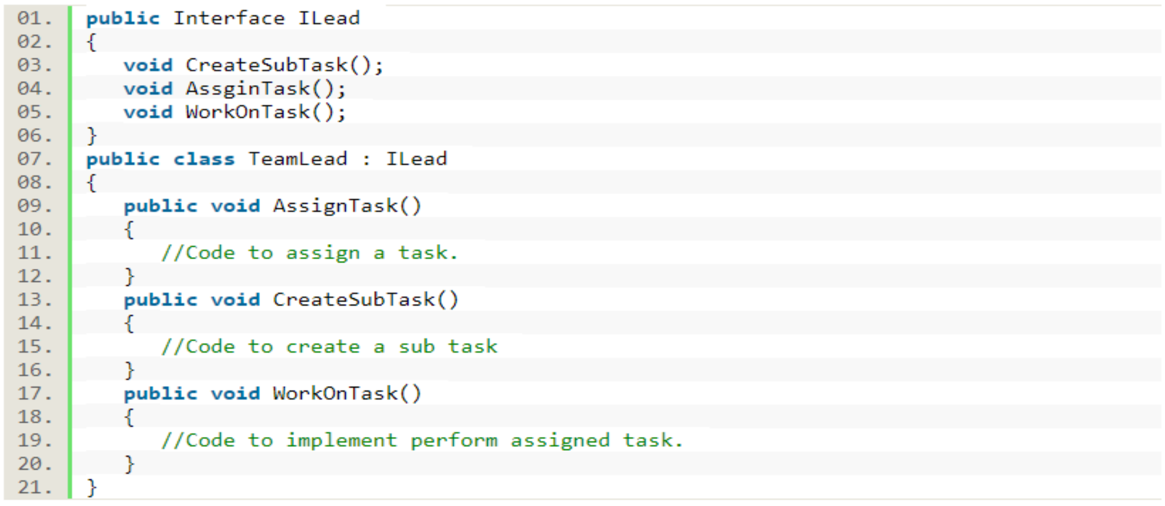
Here the GetTextFromFiles() method gets only the list of instances of classes that implement the IReadOnlySqlFile interface. That means the SqlFile and ReadOnlySqlFile class instances. And the SaveTextIntoFiles() method gets only the list instances of the class that implements the IWritableSqlFiles interface, in other words, SqlFile instances in this case. Now we can say our design is following the LSP. And we fixed the problem using the Interface segregation principle by (ISP) identifying the abstraction and the responsibility separation method.

Interface Segregation Principle (ISP)

No client should be forced to implement methods which it does not use, and the contracts should be broken down to thin ones. The ISP was first used and formulated by Robert C. Martin while consulting for Xerox.

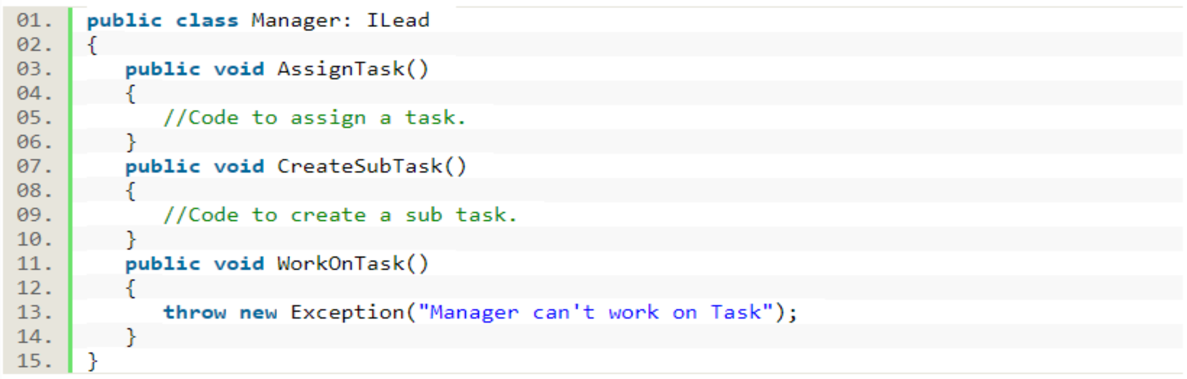
Interface segregation principle is required to solve the design problem of the application. When all the tasks are done by a single class or in other words, one class is used in almost all the application classes then it has become a fat class with overburden. Inheriting such class will results in having sharing methods which are not relevant to derived classes but its there in the base class so that will inherit in the derived class.

Using ISP, we can create separate interfaces for each operation or requirement rather than having a single class to do the same work. Let's start with an example that breaks the ISP. Suppose we need to build a system for an IT firm that contains roles like TeamLead and Programmer where TeamLead divides a huge task into smaller tasks and assigns them to his/her programmers or can directly work on them. Based on specifications, we need to create an interface and a TeamLead class to implement it.



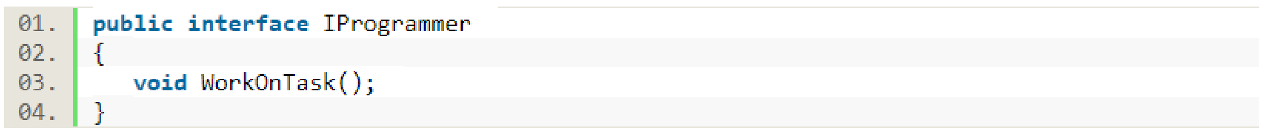
```
public Interface ILead
{
    void CreateSubTask();
    void AssginTask();
    void WorkOnTask();
}
public class TeamLead : ILead
{
    public void AssignTask()
    {
        //Code to assign a task.
    }
    public void CreateSubTask()
    {
        //Code to create a sub task
    }
    public void WorkOnTask()
    {
        //Code to implement perform assigned task.
    }
}
```

Later another role like Manager, who assigns tasks to TeamLead and will not work on the tasks, is introduced into the system. Can we directly implement an ILead interface in the Manager class, like the following?

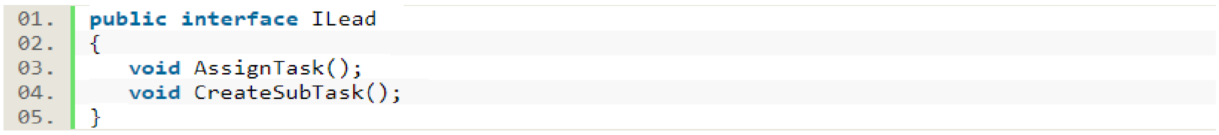


```
public class Manager: ILead
{
    public void AssignTask()
    {
        //Code to assign a task.
    }
    public void CreateSubTask()
    {
        //Code to create a sub task.
    }
    public void WorkOnTask()
    {
        throw new Exception("Manager can't work on Task");
    }
}
```

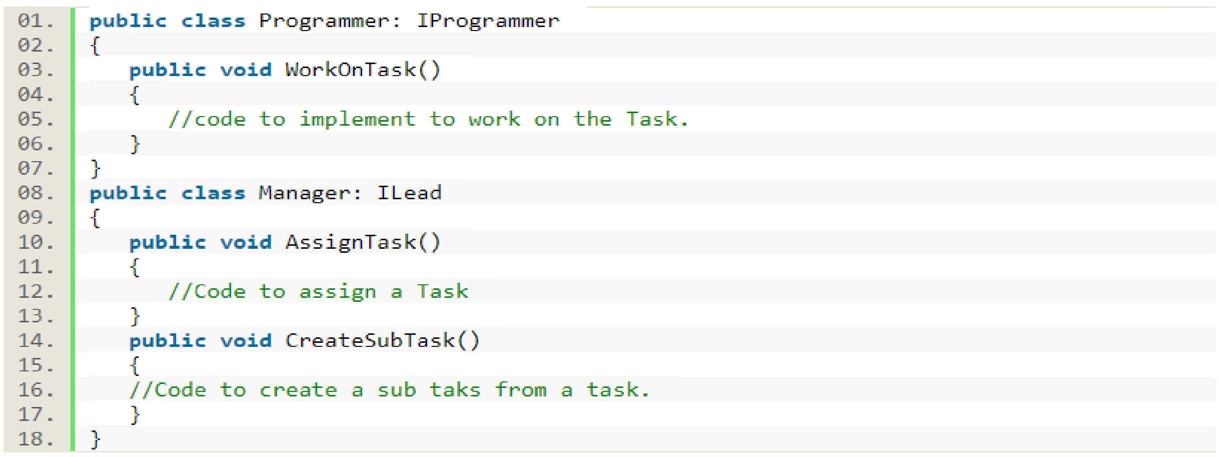
Since the Manager can't work on a task and at the same time no one can assign tasks to the Manager, this WorkOnTask() should not be in the Manager class. But we are implementing this class from the ILead interface, we need to provide a concrete Method. Here we are forcing the Manager class to implement a WorkOnTask() method without a purpose. This is wrong. The design violates ISP. Let's correct the design. Since we have three roles, 1. Manager, that can only divide and assign the tasks, 2. TeamLead that can divide and assign the tasks and can work on them as well, 3. The programmer that can only work on tasks, we need to divide the responsibilities by segregating the ILead interface. An interface that provides a contract for WorkOnTask().


```
public interface IProgrammer
{
    void WorkOnTask();
}
```

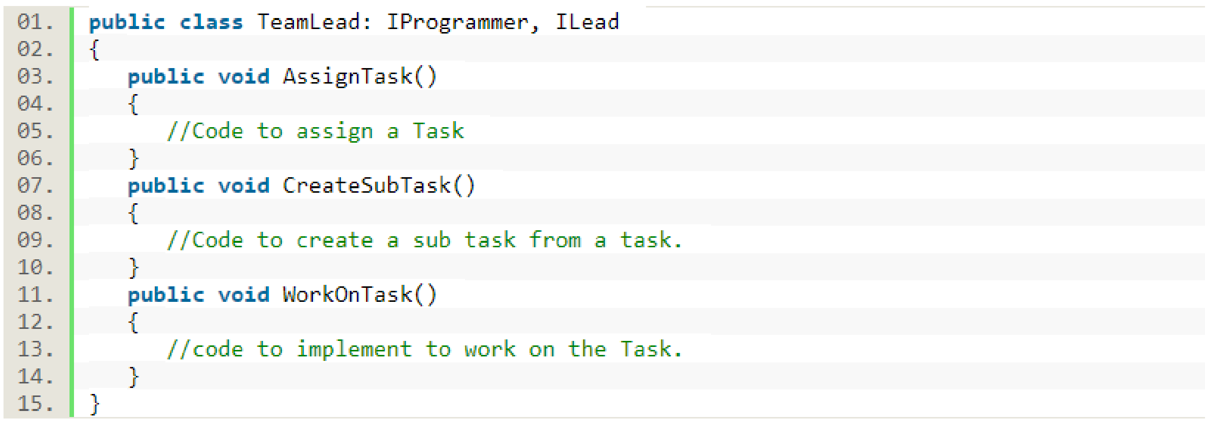
An interface that provides contracts to manage the tasks:


```
public interface ILead
{
    void AssignTask();
    void CreateSubTask();
}
```

Then the implementation becomes:


```
public class Programmer: IProgrammer
{
    public void WorkOnTask()
    {
        //code to implement to work on the Task.
    }
}
public class Manager: ILead
{
    public void AssignTask()
    {
        //Code to assign a Task
    }
    public void CreateSubTask()
    {
    //Code to create a sub taks from a task.
    }
}
```

TeamLead can manage tasks and can work on them if needed. Then the TeamLead class should implement both of the IProgrammer and ILead interfaces.


```
public class TeamLead: IProgrammer, ILead
{
    public void AssignTask()
    {
        //Code to assign a Task
    }
    public void CreateSubTask()
    {
        //Code to create a sub task from a task.
    }
    public void WorkOnTask()
    {
        //code to implement to work on the Task.
    }
}
```

Here we separated responsibilities/purposes and distributed them on multiple interfaces and provided a good level of abstraction too.

Dependency Inversion Principle

This principle is about dependencies among components. The definition of DIP is given by Robert C. Martin is as follows:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

The principle says that high-level modules should depend on abstraction, not on the details, of low-level modules. In simple words, the principle says that there should not be a tight coupling among components of software and to avoid that, the components should depend on abstraction.

The terms Dependency Injection (DI) and Inversion of Control (IoC) are generally used as interchangeably to express the same design pattern. The pattern was initially called IoC, but Martin Fowler (known for designing the enterprise software) anticipated the name as DI because all frameworks or runtime invert the control in some way and he wanted to know which aspect of control was being inverted.

Inversion of Control (IoC) is a technique to implement the Dependency Inversion Principle in C#. Inversion of control can be implemented using either an abstract class or interface. The rule is that the lower level entities should join the contract to a single interface and the higher-level entities will use only entities that are implementing the interface. This technique removes the dependency between the entities.

NOTE:

In below implementation, I have used interface as a reference, but you can use abstract class or interface as per your requirement.

In below code, we have implemented DIP using IoC using injection constructor. There are different ways to implement Dependency injection. Here, I have use injection thru constructor but you inject the dependency into class's constructor (Constructor Injection), set property (Setter Injection), method (Method Injection), events, index properties, fields and basically any members of the class which are public.

```
public interface IAutomobile

{

 void Ignition();

 void Stop();

}



public class Jeep : IAutomobile

{

 #region IAutomobile Members

 public void Ignition()

 {

 Console.WriteLine("Jeep start");

 }



 public void Stop()

 {

 Console.WriteLine("Jeep stopped.");

 }

 #endregion

}



public class SUV : IAutomobile

{

 #region IAutomobile Members

 public void Ignition()
 {

 Console.WriteLine("SUV start");

 }



 public void Stop()

 {

 Console.WriteLine("SUV stopped.");

 }

 #endregion

}




public class AutomobileController

{

 IAutomobile m_Automobile;



 public AutomobileController(IAutomobile automobile)

 {

 this.m_Automobile = automobile;

 }



 public void Ignition()

 {

 m_Automobile.Ignition();

 }
 public void Stop()

 {

 m_Automobile.Stop();

 }

}



class Program

{

 static void Main(string[] args)

 {

 IAutomobile automobile = new Jeep();

 //IAutomobile automobile = new SUV();

 AutomobileController automobileController = new AutomobileController(automobile);

 automobile.Ignition();

 automobile.Stop();



 Console.Read();

 }

}
```

In the above code, IAutomobile interface is in an abstraction layer and AutomobileController as the higher-level module. Here, we have integrated all in a single code but in real-world, each abstraction layer is a separate class with additional functionality. Here products are completely decoupled from the consumer using IAutomobile interface. The object is injected into the constructor of the AutomobileController class in reference to the interface IAutomobile. The constructor where the object gets injected is called injection constructor.

DI is a software design pattern that allows us to develop loosely coupled code. Using DI, we can reduce tight coupling between software components. DI also allows us to better accomplish future changes and other difficulties in our software. The purpose of DI is to make code sustainable.