SOLID Principles In C#

Damodhar Naidu Updated date Mar 16, 2021 1.9m 162 126

SOLID design principles in C# are basic design principles. SOLID stands for Single Responsibility Principle (SRP), Open closed Principle (OSP), Liskov substitution Principle (LSP), Interface Segregation Principle (ISP), and Dependency Inversion Principle (DIP).

Here let's learn basics of SOLID design principles using C# and .NET.

1. The reasons behind most unsuccessful applications
2. Solutions
3. Intro to SOLID principles
4. SRP
5. OCP
6. LSP
7. ISP

8. DIP

The reason behind most unsuccessful applications

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 and it 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 in 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.

Solution

1. Choosing the correct architecture (in other words MVC, 3-tier, Layered, MVP, MVVP and so on).
2. Following Design Principles.
3. Choosing correct Design Patterns to build the software based on its specifications.

Now we go through the Design Principles first and will cover the rest soon.

Intro to SOLID principles

SOLID principles are the design principles that enable us to manage most of the 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 of 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)

S: 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 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.

```
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 within 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 are you!"});

      }
   }
   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);
   }
}
```

O: Open/Closed Principle

The Open/closed Principle says "A software module/class is open for extension and closed for modification".

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. 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;
    }
}
```

Hey, we did it. 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 calculation 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 it's the 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. Awesome. Isn't it?

L: Liskov Substitution Principle

LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

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();
       }
    }
}
```

OK. We are done with our part. The functionality looks good for now. 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.

OK. 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:

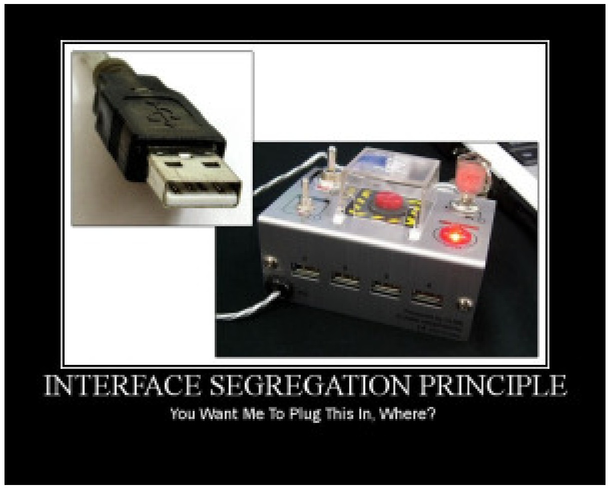
```
public class SqlFileManager
{
   public string GetTextFromFiles(List<IReadableSqlFile> aLstReadableFiles)
   {
      StringBuilder objStrBuilder = new StringBuilder();
      foreach(var objFile in aLstReadableFiles)
```

```
        {
           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.

I: Interface Segregation Principle (ISP)

The Interface Segregation Principle states "that clients should not be forced to implement interfaces they don't use. Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one submodule.".


INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

We can define it in another way. An interface should be more closely related to the code that uses it than code that implements it. So the methods on the interface are defined by which methods the client code needs rather than which methods the class implements. So clients should not be forced to depend upon interfaces that they don't use.

Like classes, each interface should have a specific purpose/responsibility (refer to SRP). You shouldn't be forced to implement an interface when your object doesn't share that purpose. The larger the interface, the more likely it includes methods that not all implementers can do. That's the essence of the Interface Segregation Principle. 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.
    }
}
```

OK. The design looks fine for now. 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.
   }
}
```

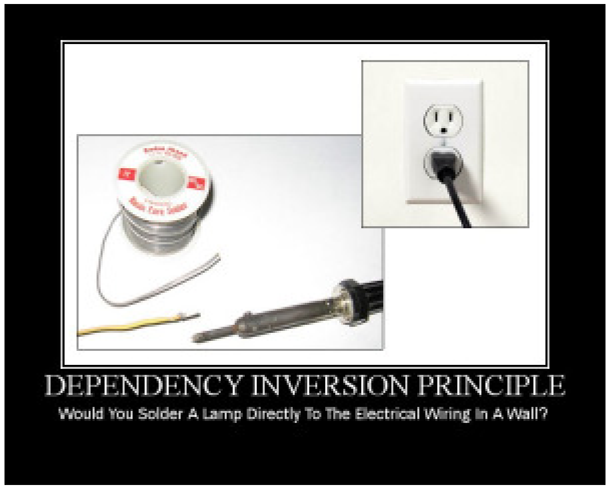
TeamLead can manage tasks and can work on them if needed. Then the TeamLead class should implement both of the IProgrammer and ILead interfaces.

```
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.
    }
}
```

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

D: Dependency Inversion Principle

The Dependency Inversion Principle (DIP) states that high-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions. Secondly, abstractions should not depend upon details. Details should depend upon abstractions.


DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

High-level modules/classes implement business rules or logic in a system (application). Low-level modules/classes deal with more detailed operations; in other words they may deal with writing information to databases or passing messages to the operating system or services.

A high-level module/class that has a dependency on low-level modules/classes or some other class and knows a lot about the other classes it interacts with is said to be tightly coupled. When a class knows explicitly about the design and implementation of another class, it raises the risk that changes to one class will break the other class. So we must keep these high-level and low-level modules/classes loosely coupled as much as we can. To do that, we need to make both of them dependent on abstractions instead of knowing each other. Let's start with an example.

Suppose we need to work on an error logging module that logs exception stack traces into a file. Simple, isn't it? The following are the classes that provide the functionality to log a stack trace into a file.

```
public class FileLogger
{
   public void LogMessage(string aStackTrace)
   {
      //code to log stack trace into a file.
   }
}
public class ExceptionLogger
{
   public void LogIntoFile(Exception aException)
   {
      FileLogger objFileLogger = new FileLogger();
      objFileLogger.LogMessage(GetUserReadableMessage(aException));
   }
   private GetUserReadableMessage(Exception ex)
   {
      string strMessage = string. Empty;
      //code to convert Exception's stack trace and message to user readable format.
      ....
      ....
      return strMessage;
   }
}
```

A client class exports data from many files to a database.

```
public class DataExporter
{
   public void ExportDataFromFile()
   {
   try {
      //code to export data from files to database.
   }
   catch(Exception ex)
   {
      new ExceptionLogger().LogIntoFile(ex);
   }
}
}
```

Looks good. We sent our application to the client. But our client wants to store this stack trace in a database if an IO exception occurs. Hmm... okay, no problem. We can implement that too. Here we need to add one more class that provides the functionality to log the stack trace into the database and an extra method in ExceptionLogger to interact with our new class to log the stack trace.

```
public class DbLogger
{
   public void LogMessage(string aMessage)
   {
      //Code to write message in database.
   }
}
public class FileLogger
{
   public void LogMessage(string aStackTrace)
   {
      //code to log stack trace into a file.
   }
}
public class ExceptionLogger
{
   public void LogIntoFile(Exception aException)
   {
      FileLogger objFileLogger = new FileLogger();
      objFileLogger.LogMessage(GetUserReadableMessage(aException));
   }
```

```
    public void LogIntoDataBase(Exception aException)
    {
       DbLogger objDbLogger = new DbLogger();
       objDbLogger.LogMessage(GetUserReadableMessage(aException));
    }
    private string GetUserReadableMessage(Exception ex)
    {
       string strMessage = string.Empty;
       //code to convert Exception's stack trace and message to user readable format.
       ....
       ....
       return strMessage;
    }
 }
 public class DataExporter
 {
    public void ExportDataFromFile()
    {
       try {
          //code to export data from files to database.
       }
       catch(IOException ex)
       {
          new ExceptionLogger().LogIntoDataBase(ex);
       }
       catch(Exception ex)
       {
          new ExceptionLogger().LogIntoFile(ex);
       }
    }
 }
```

Looks fine for now. But whenever the client wants to introduce a new logger, we need to alter ExceptionLogger by adding a new method. If we continue doing this after some time then we will see a fat ExceptionLogger class with a large set of methods that provide the functionality to log a message into various targets. Why does this issue occur? Because ExceptionLogger directly contacts the low-level classes FileLogger and DbLogger to log the exception. We need to alter the design so that this ExceptionLogger class can be loosely coupled with those classes. To do that we need to introduce an abstraction between them so that ExcetpionLogger can contact the abstraction to log the exception instead of depending on the low-level classes directly.

```
public interface ILogger
{
   void LogMessage(string aString);
}
```

Now our low-level classes need to implement this interface.

```
public class DbLogger: ILogger
{
   public void LogMessage(string aMessage)
   {
      //Code to write message in database.
   }
}
public class FileLogger: ILogger
{
   public void LogMessage(string aStackTrace)
   {
      //code to log stack trace into a file.
   }
}
```

Now, we move to the low-level class's initiation from the ExcetpionLogger class to the DataExporter class to make ExceptionLogger loosely coupled with the low-level classes FileLogger and EventLogger. And by doing that we are giving provision to DataExporter class to decide what kind of Logger should be called based on the exception that occurs.

```
public class ExceptionLogger
{
   private ILogger _logger;
   public ExceptionLogger(ILogger aLogger)
   {
      this._logger = aLogger;
   }
   public void LogException(Exception aException)
   {
      string strMessage = GetUserReadableMessage(aException);
      this._logger.LogMessage(strMessage);
   }
   private string GetUserReadableMessage(Exception aException)
```

```
    {
        string strMessage = string.Empty;
        //code to convert Exception's stack trace and message to user readable format.
        ....
        ....
        return strMessage;
    }
}
public class DataExporter
{
    public void ExportDataFromFile()
    {
        ExceptionLogger _exceptionLogger;
        try {
            //code to export data from files to database.
        }
        catch(IOException ex)
        {
            _exceptionLogger = new ExceptionLogger(new DbLogger());
            _exceptionLogger.LogException(ex);
        }
        catch(Exception ex)
        {
            _exceptionLogger = new ExceptionLogger(new FileLogger());
            _exceptionLogger.LogException(ex);
        }
    }
}
```

We successfully removed the dependency on low-level classes. This ExceptionLogger doesn't depend on the FileLogger and EventLogger classes to log the stack trace. We don't need to change the ExceptionLogger's code anymore for any new logging functionality. We need to create a new logging class that implements the ILogger interface and must add another catch block to the DataExporter class's ExportDataFromFile method.

```
public class EventLogger: ILogger
{
    public void LogMessage(string aMessage)
    {
        //Code to write message in system's event viewer.
    }
}
```

And we need to add a condition in the DataExporter class as in the following:

```
public class DataExporter
{
   public void ExportDataFromFile()
   {
      ExceptionLogger _exceptionLogger;
      try {
         //code to export data from files to database.
      }
      catch(IOException ex)
      {
         _exceptionLogger = new ExceptionLogger(new DbLogger());
         _exceptionLogger.LogException(ex);
      }
      catch(SqlException ex)
      {
         _exceptionLogger = new ExceptionLogger(new EventLogger());
         _exceptionLogger.LogException(ex);
      }
      catch(Exception ex)
      {
         _exceptionLogger = new ExceptionLogger(new FileLogger());
         _exceptionLogger.LogException(ex);
      }
   }
}
```

Looks good. But we introduced the dependency here in the DataExporter class's catch blocks. Yeah, someone must take the responsibility to provide the necessary objects to the ExceptionLogger to get the work done.

Let me explain it with a real-world example. Suppose we want to have a wooden chair with specific measurements and the kind of wood to be used to make that chair. Then we can't leave the decision making on measurements and the wood to the carpenter. Here his job is to make a chair based on our requirements with his tools and we provide the specifications to him to make a good chair.

So what is the benefit we get by the design? Yes, we definitely have a benefit with it. We need to modify both the DataExporter class and ExceptionLogger class whenever we need to introduce a new logging functionality. But in the updated design we need to add only another catch block for the new exception logging feature. Coupling is not inherently evil. If you don't have some amount of coupling, your software will not do anything for you. The only thing we need to do is understand the system, requirements, and environment properly and find areas where DIP should be followed.

Great, we have gone through all five SOLID principles successfully. And we can conclude that using these principles we can build an application with tidy, readable and easily maintainable code.

Here you may have some doubt. Yes, about the quantity of code. Because of these principles, the code might become larger in our applications. But my dear friends, you need to compare it with the quality that we get by following these principles. Hmm, but anyway 27 lines are much fewer than 200 lines.

This is my little effort to share the uses of SOLID principles. I hope you enjoyed this article.

Images courtesy :https://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Thank you,

Damu

C# SOLID MVC MVVP SOLID in C# SOLID principles

Next Recommended Article

Liskov Substitution Principle in C#

Damodhar Naidu TOP 1000

My name is Damodara Naidu, and I'm a 32 year old Software Engineer. Everyone calls me Damu. I have done my graduation (Bsc computers) and post graduation (MCA) from Andhra university.I spend my work days writing ... Read more

https://damodaranaidu.wordpress.com/about

958 2m

View Previous Comments

126 162

Type your comment here and press Enter Key (Minimum 10 characters)

Good explanation thanks but have a doubt ,it is OSP or OCP i.e Open Closed Principle?

ravi kumar Mar 15, 2021

1972 22 0 1 1 Reply

Thank you Ravi Kumar for finding the typo. I will correct it soon.

Damodhar Naidu Mar 15, 2021

958 1.4k 2m 0

Thank you so much, this helped me to understand SOLID.

Joel Chuca Mar 13, 2021

1991 3 0 1 1 Reply

Welcome Joel Chuca.

Damodhar Naidu

958 1.4k 2m

Mar 15, 2021

0

Thanks for the clear explanation

Eyayu Tefera

1987 8 0

Feb 22, 2021

1 1 Reply

Welcome Eyayu Tefera

Damodhar Naidu

958 1.4k 2m

Mar 01, 2021

0

Very nicely explained with Examples ... Appreciated

Mahi D

1964 30 0

Jan 16, 2021

1 1 Reply

Thank you Mahi D

Damodhar Naidu

958 1.4k 2m

Jan 18, 2021

0

Very good examples

Rohith Mantena

1879 116 0

Nov 12, 2020

1 1 Reply

Thank you Rohit Mantena

Damodhar Naidu

958 1.4k 2m

Nov 12, 2020

0

Nice work Damu. Do you have blog on Design patterns and Architecture (mvc, mvvp etc).

Shashank Kadge

1990 4 0

Oct 12, 2020

1 1 Reply

Thank you Shashank. I don't have currently and will create one soon.

Damodhar Naidu

958 1.4k 2m

Oct 12, 2020

0

Nice and easy..good explained with great examples..

Naresh Pandey

1983 11 0

Aug 29, 2020

1 1 Reply

Thank you Naresh Pandey

Damodhar Naidu

958 1.4k 2m

Aug 31, 2020

0

Excellent article Damu. Thanks a lot.

IMAD AYOUB

1812 182 0

Jun 29, 2020

1 1 Reply

Thank you IMAD AYOUB

Damodhar Naidu

958 1.4k 2m

Jun 30, 2020

0

Nice and easily understandable

Sabarinath A

1966 28 0

May 27, 2020

1 1 Reply

Thank you Sabarinath A

Damodhar Naidu

958 1.4k 2m

May 28, 2020

0

Nice Article..!!

Yogesh Khurpe

1895 100 3.2k

May 13, 2020

1 1 Reply

Thank you Yogesh Khurpe

Damodhar Naidu

958 1.4k 2m

May 13, 2020

0

FEATURED ARTICLES

CRUD Operation In ASP.NET Core 5 Web API

What is Microsoft Mesh

How To Create SQL Server Database Project With Visual Studio

Angular 11 New Features

What Is Azure Functions? How to get started with Azure Functions?

View All

TRENDING UP

01 How To Set Background Color Of A Selected Row Based On Checking/Unchecking Checkbox In Angular 10

02 Build Restful API's With Node.js - Express - MySQL

03 Happy 155 Years Of Business Intelligence

04 Top 10 Hottest And Tricky T-SQL Questions For Beginners

05 Upload And Download Multiple Files Using Web API

Getting Started With .NET 6

06

07 Onion Architecture In .Net 5

08 Get Microsoft Exam With Free Second Shot And Free Practice Test

09 Simplifying Use Of Static

10 Pattern Matching In C#

View All