Automation Testing: Myths and Solutions

By: Prakash Sharma | August 11, 2017

I have often observed that a lot of QA professionals get confused when it comes to what should be the automation coverage for an application. Automation is meant to reduce cycle ...

Read More

igital
Thoughts

SOLID architecture principle using C# with simple C# example

By: Rakesh Girase | May 6, 2015

Follow Follow Follow

Introduction

In this blog, I am going to explain you the SOLID architecture principle using simple C# code. This would help to build application with layer architecture with readable and easily maintainable code.

What is SOLID

SOLID principles are the design principles that enable us to manage with most of the software design problems. These principles provide us 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.

Below are the acronym of an SOLID.

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

Let's go walk through each of them below.

S: Single Responsibility Principle (SRP)

"There should never be more than one reason for a class to change"

Simple Translation: A class should concentrate on doing one thing The SRP says a class should focus on doing one thing, or have one responsibility. This doesn't mean it should only have one method, but instead all the methods should relate to a single purpose (i.e. should be cohesive).

For example, an **Invoice** class might have the responsibility of calculating various amounts based on its data. In that case it probably shouldn't know about how to retrieve this data from a database, or how to format an invoice for print or display or logging, sending Email etc.
A class that adheres to the SRP should be easier to change than those with multiple responsibilities. If we have calculation logic and database logic and display logic all mixed up within one class it can be difficult to change one part without breaking others.
Mixing responsibilities also makes the class harder to understand, harder to test, and increases the risk of duplicating logic in other parts of the design

Violations of the SRP

```
public class Invoice
{
    public long Amount { get; set; }
    public DateTime InvoiceDate { get; set; }
    public void Add()
    {
        try
        {
            // Code for adding invoice

           // Once Invoice has been added , send mail
            MailMessage mailMessage = new MailMessage("MailAddressFrom","MailAddressTo","MailSubject","MailBody");
            this.SendEmail(mailMessage);
        }
        catch (Exception ex)
        {
             System.IO.File.WriteAllText(@"c:\Error.txt", ex.ToString());
        }
    }

    public void Delete()
    {
        try
        {
            // Code for Delete invoice
        }
        catch (Exception ex)
        {
            System.IO.File.WriteAllText(@"c:\Error.txt", ex.ToString());
        }
    }

    public void SendEmail(MailMessage mailMessage)
    {
        try
        {
            // Code for getting Email setting and send invoice mail
        }
        catch (Exception ex)
        {
            System.IO.File.WriteAllText(@"c:\Error.txt", ex.ToString());
        }
    }
}
```

This Invoice class violating SRP, as It has his own responsibility i.e. Add, Delete invoice and also has extra activity like logging and Sending email as well.

Solution, lets refactor it.

```
public class Invoice
{
    public long Amount { get; set; }
    public DateTime InvoiceDate { get; set; }
    private FileLogger fileLogger;
    private MailSender mailSender;
    public Invoice()
    {
        fileLogger = new FileLogger();
        mailSender = new MailSender();
    }
    public void Add()
    {
        try
        {
            fileLogger.Info("Add method Start");
            // Code for adding invoice
            // Once Invoice has been added , send mail
            mailSender.From = "rakesh.girase@thedigitalgroup.net";
            mailSender.To = "customers@digitalgroup.com";
            mailSender.Subject = "TestMail";
            mailSender.Body = "This is a text mail";
            mailSender.SendEmail();
        }
        catch (Exception ex)
        {
            fileLogger.Error("Error while Adding Invoice", ex);
        }
    }

    public void Delete()
    {
        try
        {
            fileLogger.Info("Add Delete Start");
            // Code for Delete invoice
        }
        catch (Exception ex)
        {
            fileLogger.Error("Error while Deleting Invoice", ex);
        }
    }

}

public interface ILogger
{
    void Info(string info);
    void Debug(string info);
    void Error(string message, Exception ex);
}

public class FileLogger : ILogger
{

    public FileLogger()
    {
        // Code for initialization i.e. Creating Log file with specified
        // details
    }
    public void Info(string info)
    {
        // Code for writing details into text file
    }
    public void Debug(string info)
    {
        // Code for writing debug information into text file
    }
    public void Error(string message, Exception ex)
    {
        // Code for writing Error with message and exception detail
    }
}

public class MailSender
{
    public string From { get; set; }
    public string To { get; set; }
    public string Subject { get; set; }
    public string Body { get; set; }

    public void SendEmail()
    {
        // Code for sending mail
    }
}
```

Now Invoice class can happily delegate the logging activity to the "FileLogger" class and Sending mail activity to "MailSender" class. This way Invoice class can concentrate on Invoice related activities.

O: Open closed Principle (OSP)

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."

Simple Translation: Change a class' behavior using inheritance and composition.

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.

Let’s continue with our same Invoice class example. I have added a simple Invoice type property to the class. This property decided if this is a “Final” Or “Proposed” invoice.

Depending on the same it calculates discount. Have a look at the “GetDiscount” function which returns discount accordingly.

Violation of OSP

```
public enum InvoiceType
{
    Final,Proposed
};

public class Invoice
{
    public InvoiceType InvoiceType { get; set; }

    public double GetDiscount(double amount,InvoiceType invoiceType)
    {
        double finalAmount = 0;
        if (invoiceType == InvoiceType.Final)
        {
            finalAmount = amount - 100;
        }
        else if(invoiceType == InvoiceType.Proposed)
        {
            finalAmount = amount - 50;
        }
        return finalAmount;
    }

}
```

The problem is if we add a new invoice type, we need to go and add one more “IF” condition in the “GetDiscount” function, in other words we need to change the invoice class.

If we are changing the Invoice class again and again, we need to ensure that the previous conditions with new one’s are tested again , existing client’s which are referencing this class are working properly as before.

In other words we are “MODIFYING” the current invoice code for every change and every time we modify we need to ensure that all the previous functionality and connected client are working as before.

How about rather than “MODIFYING” , we go for “EXTENSION”. In other words every time a new invoice type needs to be added we create a new class as shown in the below. So whatever is the current code they are untouched and we just need to test and check the new classes.

Solution, let’s refactor it

```
namespace SOLID_Principles_OSP_S
{
    public enum InvoiceType
    {
        Final, Proposed
    };

    public class Invoice
    {
        public InvoiceType InvoiceType { get; set; }

        public virtual double GetDiscount(double amount)
        {
            double finalAmount = 300;
            return finalAmount;
        }
    }

    public class FinalInvoice : Invoice
    {
        public override double GetDiscount(double amount)
        {
            return base.GetDiscount(amount) - 100;
        }
    }

    public class ProposedInvoice : Invoice
    {
        public override double GetDiscount(double amount)
        {
            return base.GetDiscount(amount) - 50;
        }
    }

    public class RecurringInvoice : Invoice
    {
        public override double GetDiscount(double amount)
        {
            return base.GetDiscount(amount) - 200;
        }
    }
}
```

Putting in simple words the "Invoice" class is now closed for any new modification but it's open for extensions when new Invoice types are added to the project.

L: Liskov substitution Principle (LSP)

"Objects in a program should be replaceable with instances of their sub types without altering the correctness of that program"

Simple Translation: We must make sure that new derived classes are extending the base classes without changing their behavior

LSP states that the derived classes should be perfectly substitutable for their base classes. If class D is derived from A then D should be substitutable for A.

Look at the following C# code sample where the LSP is broken. Simply, an Orange cannot substitute an Apple, which results in printing the color of apple as Orange.

Violation of LSP

```
namespace SOLID_Principles_LSP_V
{

    class Program
    {
        static void Main(string[] args)
        {
            Apple apple = new Orange();
            Console.WriteLine(apple.GetColor());
        }
    }

    public class Apple
    {
        public virtual string GetColor()
        {
            return "Red";
        }
    }

    public class Orange : Apple
    {
        public override string GetColor()
        {
            return "Orange";
        }
    }
}
```

Solution, refactor

Now let us re-factor and make it comply with LSP by having a generic base class for both Apple and Orange.

```
namespace SOLID_Principles_LSP_S
{

    class Program
    {
        static void Main(string[] args)
        {
            Fruit fruit = new Orange();
            Console.WriteLine(fruit.GetColor());
            fruit = new Apple();
            Console.WriteLine(fruit.GetColor());
        }
    }

    public abstract class Fruit
    {
        public abstract string GetColor();
    }

    public class Apple : Fruit
    {
        public override string GetColor()
        {
            return "Red";
        }
    }

    public class Orange : Apple
    {
        public override string GetColor()
        {
            return "Orange";
        }
    }
}
```

I: Interface Segregation Principle (ISP)

"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 sub module."

Simple Translation: "No client consuming an interface should be forced to depend on methods it does not use"

Let's start with an example that breaks 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.

```
namespace SOLID_Principles_ISP_V
{

    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.
        }
    }
}
```

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. 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().

Solution, lets refactor it

```
namespace SOLID_Principles_ISP_S
{

    public interface IProgrammer
    {
        void WorkOnTask();
    }
    public interface ILead
    {
        void AssignTask();
        void CreateSubTask();
    }

    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.
        }
    }

    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.

D: Dependency Inversion Principle (DIP)

"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."

Simple Translation: High level module and Low level module keep as loosely couple as much as we can.

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/class 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 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 static class ExceptionLogger
{
    public static void LogIntoFile(Exception aException)
    {
        FileLogger objFileLogger = new FileLogger();
        objFileLogger.LogMessage(GetUserReadableMessage(aException));
    }
    private static 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 (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.

```
namespace SOLID_Principles
{
    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 and DbLogger to log the exception. We need to alter the design so that this ExceptionLogger class can be loosely coupled with those class. 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.

Solution: Lets refactor it

Now, we move to the low-level class's intitiation 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.

```
namespace SOLID_Principles_DIP_S
{

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

    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.
        }
    }

    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);
            }
        }
    }
}
```

Now the high level (FileLogger and DBLogger) and low level (DataExporter) models are loosely couple.

Conclusion

We have gone through all the five SOLID principles successfully with simple C# example.

SOLID principles of object oriented programming allow us to write structured and neat code that is easy to extend and maintain

Please post your reply , Happy coding .

This post has been viewed 5,845 times

9 thoughts on “SOLID architecture principle using C# with simple C# example”

realhacks24.com
October 31, 2015 at 9:40 pm

There’s significantly a pack to understand about that.
I suppose you made certain good things in features also.

Bhushan
January 15, 2016 at 4:09 am

The Best post on SOILD ever i have seen

SATHISH
February 23, 2016 at 11:43 pm

Excellent one, Provide ‘download pdf’ version

PavanKumar Nallamalli
February 26, 2016 at 6:50 am

The way SOLID principles explained with examples are Awesome!!! Its very clearly explained.. Thanks very much for post….

Keep post valuable information related and so would be easier for others to learn…

AugustBiara
February 11, 2017 at 7:06 am

Very well explained with example.

Pingback: Homepage

GwA2yDkJRvV
February 28, 2017 at 2:10 pm

Great style and design

Marisa
July 13, 2017 at 12:02 pm

Fairly! This was a truly excellent post. Thank you for your provided advice

Blake
July 14, 2017 at 6:46 pm

Rather! This was a truly amazing post. Thank you for your
provided advice