C# Object Oriented Programming Standard & best practices

TOLLPLUS

Contents

Acknowledgement

The best practices and/or coding standard mentioned here are based on own experiences, team experiences and information available in books or on the internet. There is always chance of improvements in whatever we do and this document is no exception. This should not be considered as full and complete.

Before and after implementing this, do check yourself and/or with your team lead for suitability and/or performance.

You may find some spelling or grammatical mistakes in this document, please feel free to let your Team Leader know about it.

Please feel free to write comment, feedback or suggestions at snarayan@tollplus.com or let your Team Leader know; to further improve this document.

1. OOP for beginners

Introduction

According to Wikipedia, Object-**oriented programming (OOP)** is a programming paradigm that uses "objects" and their interactions to design applications and computer programs. Programming techniques may include features such as **abstraction, encapsulation, inheritance, and polymorphism**. In this article I will try to describe each one of them.

In order to understand this article, you should know basics concepts of an object, class. Let's see what are they.

Object

Basically an object is anything that is identifiable as a single material item. You can see around and find many objects like Camera, Monitor, and Laptop etc. In OOP perspective, an object is nothing but an instance of a class that contains real values instead of variables

Class

A class is a template definition of the methods and variables for a particular kind of object. In other words, class is the blue print from which an individual objects is created.

Now let's see basic features of Object Oriented Programming one by one.

Abstraction

Abstraction is a process of identifying the relevant qualities and behaviors an object should possess. Lets take an example to understand abstraction. A Laptop consists of many things such as processor, motherboard, RAM, keyboard, LCD screen, wireless antenna, web camera, usb ports, battery, speakers etc. To use it, you don't need to know how internally LCD screens, keyboard, web camera, battery, wireless antena, speaker's works. You just need to know how to operate the laptop by switching it on. The intrinsic details are invisible. Think about if you would have to call to the engineer who knows all internal details of the laptop before operating it. This would have highly expensive as well as not easy to use everywhere by everyone. So here the Laptop is an object that is designed to hide its complexity.

Think If you need to write a piece of software to track the students details of a school, you may probably need to create Students objects. People comes in all different backgrounds, educational qualifications, locations, hobbies, ages and have multiple religion, language but in terms of application, an student is just a name, age, class and roll number, while the other qualities are not relevant to the application. Determining what other qualities (background, qualifications, location, hobbies etc) are in terms of this application is abstraction.

In object-oriented software, complexity is managed by using abstraction. ***Abstraction means showing only those details to the user that he needs and eliminating/hiding irrelevant and complex details.*** A well thought-out abstraction is usually simple, and easy to use in the perspective of the user, the person who is using your object.

Encapsulation

Encapsulation is a method for protecting data from unwanted access or alteration by packaging it in an object where it is only accessible through the object's interface. Encapsulation is often referred to as information hiding. But both are different. In fact *information hiding* is actually the result of Encapsulation. Encapsulation makes it possible to separate an object's implementation from its original behavior - to restrict access of its internal data. This restriction facilitates certain details of an object's behavior to be hidden. This allows protecting an object's internal state from corruption by its user.

It is the mechanism by which Abstraction is implemented. In other words you can say that Abstraction is the result of the Encapsulation. For example, the Laptop is an object that encapsulates many technologies/hardware that might not be understood clearly by most people who use it.

Inheritance

Inheritance is the ability to define a new class or object that inherits the behavior and its functionality of an existing class. The new class or object is called a child or subclass or derived class while the original class is called parent or base class. For example, in a software company Software Engineers, Sr. Software Engineers, Module Lead, Technical Lead, Project Lead, Project Manager, Program Manager, Directors all are the employees of the company but their work, perks, roles, responsibilities differs. So in OOP, the Employee base class would provide the common behaviors of all types/level of employee and also some behaviors properties that all employee must have for that company. The particular sub class or child class of the employee would implement behaviors specific to that level of the employee. So by above example you can notice that the ***main concept behind inheritance are extensibility and code reuse*** (in this case you are extending the Employee class and using its code into sub class or derived class).

Polymorphism

As name suggests, ***Polymorphism means an ability to assume different forms at different places.*** In OOP, it is a language's ability to handle objects differently based on their run time type and use. Polymorphism is briefly described as "one interface, many implementations". Polymorphism is a characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form.

There are two types of polymorphism.

1. Compile time polymorphism - It is achieved by overloading functions and operators
2. Run time polymorphism - It is achieved by overriding virtual functions

Let's say you have a class that have many Load methods having different parameters, this is called Compile time polymorphism. Let's take another example where you have a virtual method in the base class called Load with one parameter and you have redefined its functionality in your sub class/ child class by overriding base class Load method, this is called Run time polymorphism.

Source: http://www.dotnetfunda.com/articles/show/127/oops-for-beginners-features

2. Attributes and Access Modifiers

Attributes provides additional context to a class, its like an adjective.

```
[Serializable]
class Manager
{
    public virtual string GetName()
    {
        return "Ram";
    }
}
```

Access modifiers control the visibility of the class, its methods, variables and properties.
The default access modifiers for the class are **internal** and for members are **private**.

Access modifiers keyword	Description
public	Class or member is visible in current as well as referencing assemblies
protected	Access is limited to the containing class or types

	derived the containing class
internal	Access is limited to current assembly (.dll)
protected internal	Access is limited to the current assembly or types derived from the containing class
private	Access is limited to the containing type
abstract	Indicates that a class is intended to be a base or parent class of other class. It can't be instantiated
const	Indicates that the value of the field can not be modified
override	Indicates new implementation of a virtual member from the base class
partial	Indicates partial class (more than one file may contain the same class definition and implementations)
readonly	Indicates a field that can only be assigned at the time of declaration or in the constructor of the same class
sealed	Indicates that a class cannot be inherited
static	Indicates that a member belongs to the type, not the instance of the type
virtual	Indicates that a method implementation can be changed by an overriding member in derived class

1. Members, Parameters, Variables should be named in such a way that
    a) It is easily readable and understandable
    b) Grammatically correct
2. Use following casing
    a) Class, Struct, Property, Interface, Enumeration, Events, Constant & Static fields, Method Names, – PascalCase
    b) Private field, variables, parameters – camelCase
3. Avoid declaring global variables unless the variable if of static/constant in nature and is never changing.

3. SOLID Design Principles

Just by writing code in Object Oriented Programming language doesn't guarantee object oriented code and gets all the benefits of object oriented programming features.

SOLID is five Object Oriented Programming principles that ensures that code that follow this principles are truly object oriented code and gets all the benefits of object oriented programming.

SOLID stands for

1. S - Single Responsibility Principle (SRP)

2. O – Open Closed Principles (OCP)
3. L – Liskov Substitution Principle (LSP)
4. I – Interface Segregation Principle (ISP)
5. D – Dependency Inversion Principle (DIP)

Single Responsibility Principle (SRP)

Single Responsibility Principle suggests following

- A class should have only one responsibility based on separation of concern
- Each class should be designed to do only one thing. This makes the class size small, maintainable and easy to understand.

```
// Not a good design based on SRP
    public class Statement
    {
        public long amount { get; set; }
        public DateTime statementDate  { get; set; }

        public void Add()
        {
            try
            {
                // Code for adding statement

                // Once statement has been added , notify to the user
                MailMessage mailMessage = new MailMessage("MailAddressFrom",
"MailAddressTo", "MailSubject", "MailBody");
                this.SendEmail(mailMessage);
            }
            catch (Exception ex)
            {
                System.IO.File.WriteAllText(@"c:\ErrorLog.txt", ex.ToString());
            }
        }

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

        public void SendEmail(MailMessage mailMessage)
```

```
        {
            try
            {
                // Code for getting Email setting and send statement notification
            }
            catch (Exception ex)
            {
                System.IO.File.WriteAllText(@"c:\ErrorLog.txt", ex.ToString());
            }
        }
    }
```

In above code, the Statement class does not follow SRP as it add, delete the invoice apart from that, it also logs the error, and send email.

This problem can be resolved using below design.

```
    // Good design based on SRP
    public class Statement
    {
        public long amount { get; set; }
        public DateTime statementDate { get; set; }
        private ErrorLogger errorLogger;
        private MailSender mailSender;
        public Statement()
        {
            errorLogger = new ErrorLogger();
            mailSender = new MailSender();
        }

        public void Add()
        {
            try
            {
                // Code for adding statement

                // Once statement has been added , notify to the user
                mailSender.Body = "body";
                mailSender.From = "fromemail";
                mailSender.To = "toeamil";
                mailSender.Subject = "subject of email";
                mailSender.SendEmail();
            }
            catch (Exception ex)
            {
                errorLogger.LogError("Error occured while adding statement", ex);
            }
        }

        public void Delete()
        {
            try
            {
                // Code for Delete statement
            }
            catch (Exception ex)
            {
```

```
                errorLogger.LogError("Error occured while deleting statement", ex);
            }
        }
    }

    public class ErrorLogger
    {
        public ErrorLogger()
        {
            // Code for initialization i.e. Creating Error Log file with specified
            // details
        }
        public void LogError(string message, Exception ex)
        {
            // Code for writing Error Log file 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
        }
    }
```

Open Closed Principle (OCP)

Open close principle helps in achieving a class open for new functionality without modifying the existing code. It means open for extensibility but closed for modifications. The only reason to change a class should be to fix a bug.

```
// Not a good design based on OCP
    public class Statement
    {
        public long amount { get; set; }
        public DateTime statementDate { get; set; }
        public string DeliveryMethod { get; set; }

        public void Send()
        {
            if (DeliveryMethod == "Email")
            {
                // send via email
            }
            else if (DeliveryMethod == "SMS")
            {
                // send via sms
            }
        }
```

```
    }
```

In above code, if we need to add one more DeliveryMethod, we need to add one more else block and while testing we need to test all different conditions to ensure that we have not broken existing functionality. Also this class is not closed for modifications.

```
// Good design based on OCP
    class Program
    {
        public static void Main()
        {
            new HowtoUse().SendStatement();

            Delivery[] deliveries = {new EmailDelivery(), new SMSDelivery()};
            new HowtoUse().SendToAllDeliveryMethod(deliveries);

            Console.Read();
        }
    }


    public class Delivery
    {
        public virtual void Send()
        {
            // send phyiscal delivery
            Console.WriteLine("Phyiscal delivery.");
        }
    }

    public class EmailDelivery : Delivery
    {
        public override void Send()
        {
            // send via email
            Console.WriteLine("Email delivery.");
        }
    }

    public class SMSDelivery : Delivery
    {
        public override void Send()
        {
            // send via SMS
            Console.WriteLine("SMS delivery.");
        }
    }

    // Use
    public class HowtoUse
    {
        public void SendStatement()
        {
            Delivery delivery = new Delivery();
            delivery.Send();
```

```
            Delivery sms = new SMSDelivery();
            sms.Send();

            Delivery email = new EmailDelivery();
            email.Send();

            Console.WriteLine("================================");

        }

        public void SendToAllDeliveryMethod(Delivery[] deliveries)
        {
            foreach(var delivery in deliveries)
            {
                delivery.Send();
            }

            Console.WriteLine("================================");
        }
    }
```

In the above design, you can see that one class is responsible for only one thing. If we need to add a new DeliveryMethod, we just need to add one more class that inherits Delivery class and implements Send method. Remaining things will be taken care at runtime.

Liskov Substitution Principle (LSP)

This principle is named after Barbara Liskov in 70's.'

This states that derived class should be completely substitutable for their base class ie. derived class should be extending base classes without changing their behavior.

```
// Not good design based on LSP
class Program
    {
        public static void Main()
        {
            EmailDelivery email = new EmailDelivery();
            // print Email delivery
            email.Send();

            EmailDelivery sms = new SMSDelivery();
            // print SMS delivery
            sms.Send();

            Console.Read();
        }
    }

    public class EmailDelivery
    {
        public virtual void Send()
        {
            // send via Email
            Console.WriteLine("Email delivery");
```

```
        }
    }

    public class SMSDelivery : EmailDelivery
    {
        public override void Send()
        {
            // send via SMS
            Console.WriteLine("SMS delivery");
        }
    }
```

In above code, sms.Send() prints “SMS Delivery” even if this is of EmailDelivery type variable. So this line of code replaces the EmailDelivery object behavior. This is violation of LSP.

```
// Good design based on LSP
    class Program
    {
        public static void Main()
        {
            Delivery delivery = new EmailDelivery();
            // print Email delivery
            delivery.Send();

            delivery = new SMSDelivery();
            // print Email delivery
            delivery.Send();

            Console.Read();
        }
    }

    public abstract class Delivery
    {
        public abstract void Send();
    }

    // Not good design based on LSP
    public class EmailDelivery : Delivery
    {
        public override void Send()
        {
            // send via Email
            Console.WriteLine("Email delivery");
        }
    }

    public class SMSDelivery : Delivery
    {
        public override void Send()
        {
            // send via SMS
            Console.WriteLine("SMS delivery");
        }
    }
```

Interface Segregation Principle (ISP)

This states that Client should not be forced to implement interface they do not use. So instead of having a single interface covering entire functionality, there can be many small interfaces for group of methods.

```
// Not good based on ISP
public interface IDelivery
{
    void Send(string numberOfEmailId);
    void SendToPrintQueue(string addressAndCity, string country, string pin);
}

public class EmailDelivery : IDelivery
{
    public void Send(string emailId)
    {
        // send via email
    }

    public void SendToPrintQueue(string  addressAndCity, string country, string pin)
    {
        throw new NotImplementedException();
    }
}

public class SMSDelivery : IDelivery
{
    public void Send(string number)
    {
        // send via SMS
    }

    public void SendToPrintQueue(string addressAndCity, string country, string pin)
    {
        throw new NotImplementedException();
    }
}

   public class PhysicalDelivery : IDelivery
{
    public void Send(string number)
    {
        throw new NotImplementedException();
    }

    public void SendToPrintQueue(string addressAndCity, string country, string pin)
    {
        // Add it into the print queue
    }
}

public interface IStatement
{
    string NumberOrEmailId { get; set; }
    void SendStatement(IDelivery delivery);
}
```

```
    public class Statement : IStatement
    {
        public string NumberOrEmailId { get; set; }
        public void SendStatement(IDelivery delivery)
        {
            delivery.Send(NumberOrEmailId);
        }
    }

    // Use
    public class HowtoUse
    {
        public void SendStatement()
        {
            Statement smsStatementDelivery = new Statement();
            smsStatementDelivery.NumberOrEmailId = "1234566";
            smsStatementDelivery.SendStatement(new SMSDelivery());

            Statement emailStatementDelivery = new Statement();
            emailStatementDelivery.NumberOrEmailId = "you@you.you";
            emailStatementDelivery.SendStatement(new EmailDelivery());
        }
    }
```

In above code, everything looks fine however IElectronicDelivery interface is implementing SendToPrintQueue method just because we have added that method into IDelivery interface. SendToPrintQueue method does nothing. To solve this problem, we need to segregate the interface.

```
// Good based on ISP
    public interface ISMSDelivery
    {
        void SendSMS(string number);
    }

    public interface IEmailDelivery
    {
        void SendEmail(string emailId);
    }

    public interface IPhysicalDelivery
    {
        void SendToPrintQueue(string addressAndCity, string country, string pin);
    }

    public class EmailDelivery : IEmailDelivery
    {
        public void SendEmail(string emailId)
        {
            // send via email
        }
    }

    public class SMSDelivery : ISMSDelivery
    {
        public void SendSMS(string number)
        {
            // send via SMS
```

```
        }
    }

    public class PhysicalDelivery : IPhysicalDelivery
    {
        public void SendToPrintQueue(string addressAndCity, string country, string pin)
        {
            // Add it into the print queue
        }
    }

    public class Statement : IEmailDelivery, ISMSDelivery, IPhysicalDelivery
    {
        public void SendSMS(string number)
        {
            // send to sms delivery
        }

        public void SendEmail(string emailId)
        {
            // send to email delivery
        }

        public void SendToPrintQueue(string addressAndCity, string country, string pin)
        {
            // send to print queue
        }
    }
```

Dependency Inversion Principle (DIP)

This principle states that high level module should not depend on low level module. Both should depend on abstraction. Abstraction should not depend on details, details should depend on abstraction.

In other words, keep high level module and low level module loosely coupled as much as possible.

```
    // Not good based on DIP
class Program
    {
        public static void Main()
        {
            Statement statement = new Statement();
            statement.SendStatement();

            Console.Read();
        }
    }


    public class EmailDelivery
    {
        public void Send()
        {
            // send via email
            Console.WriteLine("Email Delivery.");
```

```
    }
}


public interface IStatement
{
    void SendStatement();
}

public class Statement : IStatement
{
    public void SendStatement()
    {
        new EmailDelivery().Send();
    }
}
```

In the above code if we want to add another delivery method, we will have to add another method into the Statement class. This problem occurs because Statement class directly contacts the low-level EmailDelivery class.

If we alter this code so that Statement class is loosely coupled with low-level classes then this problem will be resolved.

```
// Good based on ISP

    class Program
    {
        public static void Main()
        {
            Statement email = new Statement(new EmailDelivery());
            email.SendStatement();

            Statement sms = new Statement(new SMSDelivery());
            sms.SendStatement();

            Console.Read();
        }
    }


    public interface IDelivery
    {
        void Send();
    }

    public class EmailDelivery : IDelivery
    {
        public void Send()
        {
            // send via email
            Console.WriteLine("Email Delivery.");
```

```
    }
}

public class SMSDelivery : IDelivery
{
    public void Send()
    {
        // send via SMS
        Console.WriteLine("SMS Delivery.");
    }
}

public interface IStatement
{
    void SendStatement();
}

public class Statement : IStatement
{
    IDelivery _delivery;
    public Statement(IDelivery delivery)
    {
        _delivery = delivery;
    }

    public void SendStatement()
    {
        _delivery.Send();
    }
}
```

In the above code, you can see that Statement class (high-level) is not directly in contact with EmailDelivery or SMSDelivery class (low-level).