Coding Standards

https://en.wikipedia.org/wiki/SOLID

SOLID

Single responsibility

- Maintainability: When classes have a single, well-defined responsibility, they're easier to understand and modify.
- Testability: It's easier to write unit tests for classes with a single focus.
- Flexibility: Changes to one responsibility don't affect unrelated parts of the system

Open–closed

- Extensibility: New features can be added without modifying existing code.
- Stability: Reduces the risk of introducing bugs when making changes.
- Flexibility: Adapts to changing requirements more easily.

Liskov substitution

- Polymorphism: Enables the use of polymorphic behavior, making code more flexible and reusable.
- Reliability: Ensures that subclasses adhere to the contract defined by the superclass.
- Predictability: Guarantees that replacing a superclass object with a subclass object won't break the program.

Interface segregation

- Decoupling: Reduces dependencies between classes, making the code more modular and maintainable.
- Flexibility: Allows for more targeted implementations of interfaces.
- Avoids unnecessary dependencies: Clients don't have to depend on methods they don't use.

Dependency inversion

- Loose coupling: Reduces dependencies between modules, making the code more flexible and easier to test.
- Flexibility: Enables changes to implementations without affecting clients.
- Maintainability: Makes code easier to understand and modify.

Coding Conventions

https://www.geeksforgeeks.org/c-sharp-coding-standards/

```
public class Employee
{
    public Employee GetDetails()
    {
        //...
    }
    public double GetBonus()
    {
        //...
    }
}
```

Method argument and Local variables should always be in Camel Case

```
public class Employee
{
    public void PrintDetails(int employeeId, String firstName)
    {
        int totalSalary = 2000;
        // ...
    }
}
```

Avoid the use of underscore while naming identifiers

```
// Correct
public DateTime fromDate;
public String firstName;


// Avoid
public DateTime from_Date;
public String first_Name;
```

Always prefix an interface with letter I.

```
// Correct
public interface IEmployee
{
```

```
}
public interface IShape
{

}
public interface IAnimal
{

}

// Avoid
public interface Employee
{

}
public interface Shape
{

}
public interface Animal
{

}
```

For better code indentation and readability always align the curly braces vertically.

```
// Correct
class Employee
{
    static void PrintDetails()
    {

    }
}


// Avoid
class Employee
    {
    static void PrintDetails()
    {
      }
}
```

Always use the using keyword when working with disposable types. It automatically disposes the object when program flow leaves the scope.

```
using(var conn = new SqlConnection(connectionString))
{
    // use the connection and the stream
    using (var dr = cmd.ExecuteReader())
    {
     //
    }
}
```

Always declare the variables as close as possible to their use.

```
// Correct
String firstName = "Shubham";
Console.WriteLine(firstName);
//--------------------------


// Avoid
String firstName = "Shubham";
//--------------------------
//--------------------------
//--------------------------
Console.WriteLine(firstName);
```

Always declare the properties as private so as to achieve Encapsulation and ensure data hiding.

```
// Correct
private int employeeId { get; set; }

// Avoid
public int employeeId { get; set; }
```

Always separate the methods, different sections of the program by one space.

```
// Correct
class Employee
{
    private int employeeId { get; set; }

    public void PrintDetails()
    {
    //------------
    }
}

// Avoid
```

```
class Employee
{

   private int employeeId { get; set; }




   public void PrintDetails()
   {
   //------------
   }

}
```

Constants should always be declared in UPPER_CASE.

```
// Correct
public const int MIN_AGE = 18;
public const int MAX_AGE = 60;

// Avoid
public const int Min_Age = 18;
public const int Max_Age = 60;
```