Every successful application depends on

Architecture – e.g. MVC, WebApi, MVVM etc.

Design Principles – SOLID, DRY, KISS, YAGNI

Design Patterns – Singleton, Factory etc.

**Different Types of Software Design Principles**

1. **SOILD :**

**-** Enable us to manage most of the software design problems.

**- SOLID** is an acronym for five design principles intended to make software designs more understandable, flexible and maintainable.

**It helps us to:**

* Achieve reduction in complexity of code.
* Increase readability, extensibility and maintainability.
* Reduce error and implement reusability.
* Better testability.
* Reduce tight coupling.

**S-** Single Responsibility Principle (SRP)

* + **A class should have only one reason to change.**

Consider an interface which breaks the SRP because LogError and SendEmail has nothing to do with IUser (Same example can be used for the ISP)

interface IUser

{

bool Login();

bool Register();

void LogError();

bool SendEmail();

}

After applying SRP

interface IUser

{

bool Login();

bool Register();

}

interface ILogger

{

void LogError();

}

interface IEmail

{

bool SendEmail();

}

**O-** Open/Closed Principle (OCP)

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

**L-** Liskov Substitution Principle (LSP)

* + S is a subtype of T, then objects of type T may be replaced with objects of type S.
  + Derived types must be completely substitutable for their base types.

**Guidelines**

* + 1. No new exception can be thrown by the subtype.
    2. Clients should not know which specific subtype they are calling.
    3. New derived class just extend without replacing the functionality of old class.

**I-** Interface Segregation Principle (ISP)

* + Many client-specific interfaces are better than one general-purpose interface.

**D-** Dependency Inversion Principle (DIP)

* + High level modules should not depend on Low level modules. Both should depends on abstractions.
  + Abstraction should not depend on details. Details should depend on abstractions.

1. **DRY (Don’t Repeat Yourself)**

This principle states that each small pieces of knowledge (code) may only occur exactly once in the entire system. This helps us to write scalable, maintainable and reusable code.

**Example –** Asp.Net MVC framework works on this principle.

1. **KISS (Keep it simple, Stupid!)**

This principle states that try to keep each small piece of software simple and unnecessary complexity should be avoided. This helps us to write easy maintainable code.

1. **YAGNI (You ain't gonna need it)**

This principle states that always implement things when you actually need them never implements things before you need them.

**What is Dependency Injection?**

The Dependency Injection pattern is a particular implementation of Inversion of Control.

**Inversion of Control (IoC)** It is a *design pattern which states that objects should not create objects on which they depend to perform some activity. Instead, they should get those objects from an outside service or a container.*

**Dependency Injection (DI)** *means that this is done without the object intervention, usually by a framework component that passes constructor parameters and set properties.*

* Dependency injection **eliminates tight coupling between objects** to make both the objects and applications more flexible, reusable, and easier to test that use them,
* It **facilitates the creation of loosely coupled objects and their dependencies**.
* The basic idea behind Dependency Injection is that you should isolate the implementation of an object from the construction of other objects on which it depends.

**Advantages of Dependency Injection**

The advantages of using Dependency Injection pattern and Inversion of Control are the following:

* Reduces class coupling
* Increases code reusing
* Centralized configuration
* Improves code maintainability
* Improves application testing (Code becomes more testable because it abstracts and isolates class dependencies.)

**Disadvantages of Dependency Injection**

However, the primary drawback of dependency injection is that *wiring instances together can become a nightmare if there are too many instances and many dependencies that need to be addressed.*

**Use Dependency with**

DI with Autofac

DI with Unity

DI with Dagger 2

DI with Structure Map (StructureMap.dll.)

**Different Types of Injections are**

* Constructor Injection
* Setter Injection
* Interface Injection
* Service Locator

**In MVC 6**

// Scoped is a single instance for the duration of the scoped request, which means per HTTP request in ASP.NET.

*AddScoped<IGreeting, Greeting>();*

// Singleton is a single instance for the lifetime of the application domain.

*AddSingleton <IGreeting, Greeting>();*

// Transient is a single instance per code request.

*AddTransient<IGreeting, Greeting>();*

**Static Class:-**

1. You cannot create the instance of static class.
2. Loaded automatically by the .NET Framework common language runtime (CLR) when the program or namespace containing the class is loaded.
3. Static Class cannot have constructor.
4. We cannot pass the static class to method.
5. We cannot inherit Static class to another Static class in C#.
6. A class having all static methods.
7. Better performance (static methods are bonded on compile time)

**Singleton:-**

1. You can create one instance of the object and reuse it.
2. Singleton instance is created for the first time when the user requested.
3. Singleton class can have constructor.
4. You can create the object of singleton class and pass it to method.
5. Singleton class does not say any restriction of Inheritance.
6. We can dispose the objects of a singleton class but not of static class.
7. Methods can be overridden.
8. Can be lazy loaded when need (static classes are always loaded).
9. We can implement interface (static class cannot implement interface).
10. We can use Singleton in :
    1. Logging,
    2. Cache Management
    3. Printer Spooling
    4. Configuration Management
    5. Session based shopping cart
    6. Managing a connection or pool of connections to database.
    7. File Management

public sealed class Singleton

{

private static int cnt = 0;

private static readonly Lazy<Singleton> instance = new Lazy<Singleton>(() => new Singleton());

public static Singleton GetInstance

{

get

{

return instance.Value;

}

}

private Singleton()

{

Console.WriteLine("Counter Value : " + ++cnt);

}

public void PrintDetails(string message)

{

Console.WriteLine(message);

}

}

**Eager Initialization (CLR takes care of the variable initialization and it’s thread safety)**

public sealed class Singleton

{

private Singleton()

{ }

private static readonly Singleton instance = new Singleton();

public static Singleton GetInstance

{

get

{

return instance;

}

}

}

**Lazy Initialization**

public sealed class Singleton

{

private Singleton()

{ }

private static readonly Lazy<Singleton> instance = new Lazy<Singleton>(() => new Singleton());

public static Singleton GetInstance

{

get

{

return instance.Value;

}

}

}

**Parallel Call**

Parallel.Invoke(

()=> studetails(),

()=> empdetails(),

);

**Double Check locking**

class Singleton

{

private static Singleton instance = null;

private static readonly object obj = new object();

public static Singleton GetInstance

{

get

{

if (instance == null)

{

lock (obj)

{

if (instance == null)

instance = new Singleton();

}

}

return instance;

}

}

}