**Design Patterns**

Design Pattern are evolved from the features of Object-Oriented Programming.  
It provides reusable solutions to the problems that we encounter every day of programming  
It Acts as templates which can be applied to the real-world programming programs.

Design Patterns are categorized into 3 types and they are:

1. Creational Design Patterns.
2. Structural Design Patterns.
3. Behavioural Design Patterns.

**What are Creational Design Patterns?**

These patterns deal with the process of objects creation.

* [**Singleton pattern**](https://en.wikipedia.org/wiki/Singleton_pattern) restricts object creation for a class to only one instance.
* [**Prototype pattern**](https://en.wikipedia.org/wiki/Prototype_pattern)creates objects by **cloning** an existing object.
* [**Factory method pattern**](https://en.wikipedia.org/wiki/Factory_method_pattern) creates objects **without** specifying the exact class to create.
* [**Abstract factory pattern**](https://en.wikipedia.org/wiki/Abstract_factory_pattern)groups object factories that have a **common theme**.
* [**Builder pattern**](https://en.wikipedia.org/wiki/Builder_pattern) constructs complex objects by **separating construction and representation.**

**What are Structural Design Patterns?**

These patterns deal with the composition of objects structures.

* **Adapter** pattern works as a bridge between two **incompatible** interfaces.
* [**Bridge**](https://en.wikipedia.org/wiki/Bridge_pattern) decouples an abstraction from its implementation so that the two can vary independently.
* [**Composite**](https://en.wikipedia.org/wiki/Composite_pattern) composes zero-or-more similar objects so that they can be manipulated as one object.
* **Decorator** allows a user to add new functionality to an existing object without altering its structure.
* [**Facade**](https://en.wikipedia.org/wiki/Facade_pattern)provides a simplified interface to a large body of code.
* [**Flyweight**](https://en.wikipedia.org/wiki/Flyweight_pattern) reduces the cost of creating and manipulating many similar objects.
* In **proxy** pattern, a class represents functionality of another class

**What are Behavioural Design Patterns?**

These patterns deal with the process of communication, managing relationships, and responsibilities between objects.

* [**Chain of responsibility**](https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern)delegates commands to a chain of processing objects.
* [**Command**](https://en.wikipedia.org/wiki/Command_pattern)creates objects which encapsulate actions and parameters.
* [**Interpreter**](https://en.wikipedia.org/wiki/Interpreter_pattern)implements a specialized language.
* [**Iterator**](https://en.wikipedia.org/wiki/Iterator_pattern) accesses the elements of an object sequentially without exposing its underlying representation.
* [**Mediator**](https://en.wikipedia.org/wiki/Mediator_pattern)allows [loose coupling](https://en.wikipedia.org/wiki/Loose_coupling) between classes by being the only class that has detailed knowledge of their methods.
* [**Memento**](https://en.wikipedia.org/wiki/Memento_pattern)provides the ability to restore an object to its previous state (undo).
* [**Observer**](https://en.wikipedia.org/wiki/Observer_pattern) is a publish/subscribe pattern which allows several observer objects to see an event.
* [**State**](https://en.wikipedia.org/wiki/State_pattern)allows an object to alter its behavior when its internal state changes.
* [**Strategy**](https://en.wikipedia.org/wiki/Strategy_pattern)allows one of a family of algorithms to be selected on-the-fly at runtime.
* [**Template method**](https://en.wikipedia.org/wiki/Template_method_pattern) defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
* [**Visitor**](https://en.wikipedia.org/wiki/Visitor_pattern)separates an algorithm from an object structure by moving the hierarchy of methods into one object.

**Solid Principles**

SOLID principles are the **design principles** that enable us to **manage** most of the software **design problems**

**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)

**If we don’t follow SOLID Principles, we**1. End up with tight or strong coupling of the code with many other modules/applications   
2. Tight coupling causes time to implement any new requirement, features or any bug fixes and sometimes it creates unknown issues   
3. End up with a code which is not testable   
4. End up with duplication of code   
5. End up creating new bugs by fixing another bug   
6. End up with many unknown issues in the application development cycle

**Following SOLID Principles helps us to**   
1. Achieve reduction in complexity of code   
2. Increase readability, extensibility and maintenance   
3. Reduce error and implement Reusability   
4. Achieve Better testability   
5. Reduce tight coupling

**Single Responsibility Principle**

1. Robert C. Martin expresses the principle as, "A class should have only one reason to change”

2.Every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class

**Liskov Substitution Principle**

1. Introduced by Barbara Liskov state that “objects in a program should be replaceable with instances of their sub-types without altering the correctness of that program”  
2. If a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module  
3. We can also state that Derived types must be substitutable for their base types

**Open/Closed Principle**

1. “Software entities should be open for extension but closed for modification”  
2. The design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code   
3. The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged

**Interface Segregation Principle**

1. “Many client-specific interfaces are better than one general-purpose interface”  
2. We should not enforce clients to implement interfaces that they don't use. Instead of creating one big interface we can break down it to smaller interfaces

**Dependency Inversion Principle**

1. One should “depend upon abstractions, [not] concretions"  
2. Abstractions should not depend on the details whereas the details should depend on abstractions  
3. High-level modules should not depend on low level modules

1. class Singleton
2. {
3. private static Singleton \_instance;
4. // Constructor is 'protected'
5. protected Singleton()
6. {
7. }
8. public static Singleton Instance()
9. {
10. // Uses lazy initialization.
11. // Note: this is not thread safe.
12. if (\_instance == null)
13. {
14. \_instance = new Singleton();
15. }
16. return \_instance;
17. }
18. }
19. static void Main()
20. {
21. // Constructor is protected -- cannot use new
22. Singleton s1 = Singleton.Instance();
23. Singleton s2 = Singleton.Instance();
25. // Test for same instance
26. if (s1 == s2)
27. {
28. Console.WriteLine("Objects are the same instance");
29. }
31. // Wait for user
32. Console.ReadKey();
33. }