**Design patterns using C# and .Net core**

**Solid principles:**

**Solid:** A group of 5 object oriented functions

S = Single Responsibility

O = Open / closed

L = Liskov substitution

I = Interface segregation

D = Dependency inversion

**Single Responsibility:** A class should be responsible for a single part of the functionality It should only do one certain thing.

**Open/Closed:** Open to extension and closed to modification

**Liskov Substitution:** It is known as “Substitutability”. We should be able to use a subclass in place of its parent class

* A subclass should be used wherever its base class can be used

**Interface Segregation:** A classshouldnot depend on methods that it does not need to implement.

**Dependency Inversion:** Your classes and modules should depend on abstractions instead of concrete implementations.

**New functionality:** It is a part of new class

**Interface segregation principle:**

* A class should not have to implement methods it does not need
* Keep classes and interface small compact
* Segregate= Split
* Interface segregation is splitting your interface into smaller once.

**The dependency inversion principle:** these provides abstractions that we need

* Abstractions = Interfaces
* Common interface = USB
* Common interface = An entity with two buttons and a pointer
* Implementation does not matter

**Loose Coupling:** It is a powerful programmingand it allows classes to swap without breaking our application.

* Serializing need to be done
* We need to inject the services to post
* The one binds the post service and the Mock post service is a common interface

**Creational design patterns:**

**Design pattern:** Generalized, reusable solutions to common design issues in software engineering.

**Group of design patterns:**

* Creational
* Structural
* Behavioral

**The singleton pattern:**

**Singleton:** The project should have only a single instance of a specific class throughout the entire application.

**Benefits of singleton:**

* Can be used to shared state
* To avoid long initialization
* Cross-class communication
* Perfectly represents unique item

**Key implementation points:**

* Private constructor
* Read only instance
* Static instance
* Static accessor
* The class should be sealed
* Instance initialized in a static constructor
* We use static constructor to initialize the instance this has several benefits:

**1). Lazy loading:** We get to initialize the instance to remember

**2).** **Thread safety:** This static constructor runs once in a whole application

**3).** **Immutable instance**

**4).** **Dependency injection:** It violates the single responsibility principle by having extra responsibilities.

**The factory pattern:** It is a combination of the single responsibility and open/closed principles

**Loose coupling**: It is all about interfaces, software that communicate with each other have little to no knowledge of each other’s actual implementation

**Black boxes:** Software component that are loosely coupled often code “black boxes”

**Benefits of loose coupling:**

* Easier to work with large projects
* Swap implementation
* Testability
* Components grow independently

**The object pool:** It is a collection of Pre-initialization objects of specific time usually these objects take a bit long to initialize

* A pool of pre-initialized objects whose initialization is heavyweight. every time we need such an object we take one from the pool and return it back to the pool
* The object pool pattern is used in situations where the cost of initializing a class in instance is very high

**Dependency injection:** The use of new keyword is not allowed

* An OOP cannot exist without new

**DI Container:**

* The registration of the class is supported by interfaces
* Uses will be provided by interface in the type concrete one
* The container will be responsible for actual invoking the constructor passing the requirement dependencies

DI container welcome to operations: 1).Register 2).Resolve

**Structured design patterns:**

**The Adaptor pattern:** Converts an interface of a class to one expected by the consumer

**The Façade pattern:** The goal of the façade pattern is to simplify an interface.

**Principle of least interest (Law of Demeter):** Every component should have little knowledge of how other components work and only communicate with a few specific close friends

**The composite pattern:** Used to create part-whole collections in the form tree-like structures that can contain both individual items and collections as well.