**Design Patterns**

Design patterns provide solutions to the common problems, occur in the Software Design. The 23 Gang of Four (GoF) patterns are generally considered the foundation for all other patterns.

Design patterns will prevent major issues to come in future and also helps the other architects to easily understand your code. They are categorized in three groups.

1. **Creational –** *these patterns deals mainly with creation of objects and classes.*
2. **Structural –** *these patterns deals with Class and Object Composition.*
3. **Behavioral –** *these patterns mainly deals with Class – Object Communication.*

**Gangs of Four**

1. **Erich Gamma**
2. **Richard Helm**
3. **John Vlissides**
4. **Ralph Johnson**

**1. Creational Design Pattern**

**a. Singleton Pattern** – creates a class which can have a single object throughout the application, so that whenever any other object tries to access the object of the class, it will access the same object always.

**Real-time-example –** the singleton pattern can be used for anything that you don’t want to repeat. Singleton patterns are used for logging, driver’s objects, caching and thread pool.

**2. Factory Pattern**

Factory Pattern deals with the instantiation of object without exposing the instantiation logic.

3. Factory Method Pattern

Abstract the process of object creation and allows the object to be created at runtime when it is required.

Real-time-example –

Abstract Design Pattern

Acts a super factory which creates other factories. Interface is responsible for creating a set of related objects, or dependent object without specifying their concrete classes.

1. Abstract Factory – this is an interface which is used to create abstract product.
2. Concrete Factory – this is class which is implements the Abstract Factory Interface.
3. Abstract Product – this is an interface which declares a type of product.
4. Concrete Product – this is class which implements the Abstract Product interface.
5. Client – this is class which use Abstract Factory and Abstract Product interfaces.

**SOLID**

1. **Single Responsibility Principle** – this principle states that there should never be more than one reason for a class to change.

**Example: Savings and Current Account in bank**.

1. **Open/Close Principle** – this principle states that software entities should be open for extension but closed for modification.

**Example**: **A Payment Gateway.**

1. **Liskov Substitution Principle** - This principle states that functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

**Example**: **Inheritance hierarchy with Person and Student.**

1. **Interface Segregation Principle** - This principle states that Clients should not be forced to depend upon interfaces that they don’t use.
2. **Dependency Inversion Principle** – Dependency Injection Pattern