Abstraction

- Abstraction is the process of simplifying complex systems by focusing on essential aspects while hiding irrelevant details.
- In Java, abstraction is achieved through abstract classes and interfaces, where only method signatures are provided, and implementation details are left to concrete subclasses or implementing classes.

Association

- Association represents a relationship between two or more classes, where objects of one class are related to objects of another class.
- It can be a one-to-one, one-to-many, or many-to-many relationship.
- Associations are typically represented by lines connecting the related classes, often with multiplicities indicating the number of objects involved.

Aggregation

- Aggregation is a specialized form of association where objects of one class "own" or "contain" objects of another class.
- It implies a whole-part relationship, where the contained objects can exist independently of the container object.
- Aggregation is represented by a diamond-headed line connecting the container class to the contained class.

Composition

- Composition is a stronger form of aggregation, where the lifetime of the contained objects is tightly coupled with the lifetime of the container object.
- It implies a stronger whole-part relationship, where the contained objects cannot exist without the container object.
- Composition is represented by a solid diamond-headed line connecting the container class to the contained class.

Generalization

- Generalization represents an "is-a" relationship between classes, where one class (subclass) inherits properties and behaviors from another class (superclass).
- It allows for code reuse and promotes polymorphism.
- Generalization is represented by an arrow-headed line pointing from the subclass to the superclass.

Specialization

- Specialization is the reverse of generalization, representing a more specific subclass inheriting from a more general superclass.
- It allows for refining the behavior and properties of the superclass in specialized subclasses.
- Specialization is represented by an arrow-headed line pointing from the subclass to the superclass.

Realization

- Realization represents the implementation of an interface by a class.
- It signifies that the implementing class agrees to provide concrete implementations for all methods declared in the interface.
- Realization is represented by a dashed line with a triangle-headed arrow pointing from the implementing class to the interface.

Design Principles

SOLID

DESIGN PRINCIPLES

SOLID Introduction

- SOLID principles are the design principles that enable us manage most of the software design problems
- The term SOLID is an acronym for five design principles intended to make software designs more understandable, flexible and maintainable
- The principles are a subset of many principles promoted by Robert C. Martin
- The SOLID acronym was first introduced by Michael Feathers

SOLID Acronym

- 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)

Single Responsibility Principle

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

- 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”
- 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
- We can also state that Derived types must be substitutable for their base types

Open/Closed Principle

- “Software entities should be open for extension, but closed for modification”
- 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
- 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

- “Many client-specific interfaces are better than one general-purpose interface”
- 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

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

If we don’t follow SOLID Principles we

- End up with tight or strong coupling of the code with many other modules/applications
- Tight coupling causes time to implement any new requirement, features or any bug fixes and some times it creates unknown issues
- End up with a code which is not testable
- End up with duplication of code
- End up creating new bugs by fixing another bug
- End up with many unknown issues in the application development cycle

Following SOLID Principles helps us to

- Achieve reduction in complexity of code
- Increase readability, extensibility and maintenance
- Reduce error and implement Reusability
- Achieve Better testability
- Reduce tight coupling

Solution to develop a successful application depends on

- Architecture : choosing an architecture is the first step in designing application based on the requirements. Example : MVC, WEBAPI.. Etc
- Design Principles : Application development process need to follow the design principles
- Design Patterns : We need to choose correct design patterns to build the software