**DATE: - 4-11-2024**

The day I joined snad developers

**SOLID PRICIPLES AND DESIGN PRICIPLES**

Design Patterns, SOLID principles, DBMS, OOPS are essential tools for creating, maintaining and scalable software.

**SOLID PRINCIPLES**

**Single responsibility principle**

A class should have only one reason to change.

Each class should have one sole purpose, and not filled with excessive functionality.

Every module/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.

**Open closed principle**

Software entities should be open of extension but closed for modification.

The design and the writing of the code should be in such a way that new functionality should be added with the minimum changes in the existing code.

**Liskov substitution principle**

This means that every subclass or derived class should be substitutable for their parent or base class.

Objects in a program should be replaceable with the instances of their subtypes without altering the correctness of the program.

Which means if program module is using base class, then the reference to the base class can be replaced with the derived class without affecting the functionality of the program module

We can also states that derived types must be substitutable for their base types.

**Interface segregation**

Interface should not force classes to implement what they can’t do.

A large interface should be divided into smaller ones.

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 into smaller interfaces

**Dependency inversion**

Components should depend on abstractions not on concretions.

Abstractions should not depend on the details whereas the details should be dependent on the abstractions.

High level modules should not depend on the low-level modules.

**DESIGN PATTERNS**

Design patterns are tried and tested solutions.

Design Patterns are standardized, reusable solutions to common problems encountered in software design. They provide a template for how to solve a problem that can be used in many different situations at code level.

**What is a design pattern and why it is important in software industry?**

Industry standard approach to solve a recurring problem in software is called design pattern

Design pattern promotes reusability that leads to more robust and highly maintainable code

It makes our code easy to understand and debug. It helps with faster development and new members of the team will understand easily.

**Types of design pattens in java**?

* Creational design patterns
* Structural design patterns
* Behavioral design patterns

**Creational design patterns:**

Create an object in a base possible way or loosely coupled manner.

Provides object creational mechanism that increases flexibility and reuse of existing code.

* Singleton design pattern
* Factory design pattern
* Abstract factory pattern
* Builder pattern
* Prototype

**Structural design pattern: -**

Explain how to assemble objects into larger structures, while keeping these structures flexible and efficient.

Defines how objects and classes can be combined and form large structures.

* Adapter pattern
* Bridge pattern
* Decorator pattern
* Facade pattern
* Composite pattern
* Proxy pattern
* Flyweight pattern

**Behavioral design pattern: -**

Take care of effective communication and the assignment of responsibility of objects.

Provides solution for the data interaction with the objects and how to provide loose coupling and flexibility. when two classes want to communicate with each other in loosely coupled manner.

* Chain of responsibility pattern
* Template method pattern
* Observer pattern
* Strategy pattern
* Mediator pattern
* Command pattern
* State pattern
* Visitor pattern
* Interpreter pattern
* Iteration pattern

**Singleton design pattern**

It ensures that only one class exists in the JVM.

Make sure the singleton class must provide a global access point to get the instance

Implementations

Eager initialization

Static block initialization

Lazy initialization

Thread safe initialization

Bill pugh implementation

Using reflection to destroy singleton pattern

Enum singleton pattern

Using clone to destroy/prevent pattern

Using serialization destroy/prevent pattern

Steps to create singleton pattern

Create private constructor to restrict instantiation of the class from other class.

Create private static variable of the same class that is the only instance of the class.

Create public static method that returns instance of the class and this is global access point for outer world to get the instance of the single class.

**Factory design pattern**

**When do you use factory design pattern?**

When we have a super class with multiple subclasses and based on input, we need to return one of the sub objects

This pattern takes responsibility for the instantiation of a class from the client program to the factory class.

**Factory design pattern advantage?**

It is a way to code for interface rather than implementation.

Factory patterns make our code more robust, less coupled and easy to extend so we can easily change our base class implementation because the client program is unaware of this.

**Abstract factory design pattern**

Abstract factory design patterns are almost same as factory design pattern except the fact that it is factory of factories

Abstract factory design patterns are considered as another layer of abstraction over factory pattern.

When we need another level of abstraction over a group of factories , you should could consider abstract factory pattern.

**Builder design pattern**

It solves problems when objects contain a lot of attributes.

**Prototype design pattern**

Prototype design pattern is used when object creation is costly and requires a lot of time and resources and you have a similar object already existing.

Prototype pattern provides a mechanism to copy the original object to a new object and then modify it according to our needs. Prototype pattern uses java cloning to clone the object.

**Adapter design pattern(structural)**

It is used so two unrelated interface can work together.

The object that joins this unrelated interface is called an adapter.

**Composite design pattern**

It is used when we must represent a part out of the whole hierarchy

When we want to create a structure in a way that the object in the structure has to be treated the same way then we can apply composite design pattern.

We can break the pattern down into:

1.component: is the base interface for all the objects in the composition. It should be either an interface or an abstract class with the common methods to manage the child composites.

2.leaf : implements the default behavior of the base component . It doesn’t contain a reference to the other objects.

3.composite: it has leaf elements. It implements the base component methods and defines child-related operations.

4.client: it had the access to the composition elements by using the base component object.

**Proxy design pattern**

Proxy means in place of, representing or on behalf of are literal meanings of the proxy design pattern.

Consider a heavy java object like jdbc connection or session factory that requires some initial configuration. We only want such objects to be initiated on demand, and once they are we want to reuse them for all cells.

**Façade design pattern**

It is used to help client applications to easily interact with the system.

This pattern provides a unified interface to a set of interface in a system or subsystem. It defines a higher-level interface that makes the subsystem easier to use.

It is more helpful for client applications, it doesn’t hide subsystem interface from the client. Whether to use it or not it is completely dependent on client code.

It can be applied at any point of development, usually when the number of interface grow and system gets complex.

Subsystem interface are not aware of façade and they shouldn’t have any reference of the facade interface.

**Decorator design pattern**

It is used to modify the functionality of the object at the runtime. At the same time other instances of the same class will not be affected by this, so individual objects gets the modified behavior.

This pattern provides wrapper to the existing class.

This pattern uses abstract classes or interfaces with the composition to implement the wrapper.

It is helpful in providing runtime modification abilities and hence more flexibility. It is easy to extend and maintain when the number of choices are more.

**Bridge design pattern**

When we have interface hierarchies in both interface and implementations then bridge design pattern Is used to decouple the interface from implementation and hiding the implementation details from the client programs.

This design pattern decouple an abstraction from its implementation so that the two can vary independently.

The implementation of bridge design pattern follows the notion to refer composition over inheritance.

It can be used when abstraction and implementation can have different hierarchies independently and we want to hide the implementation from the client application.

**Template method design pattern (Behavioral design pattern)**

This pattern used to create a methos stub and that defers some of the steps of the implementation to the subclasses.

It defines the steps to execute an algorithm and it can provide default implementation that might be common for all or some of the subclasses.

Template method should consists of certain steps whose order is fixed and for some of the methods, implementation differs from base class to subclass. If we want some of the methods not be overridden by subclasses, then we make those template methods final

**Mediator design pattern**

It is used to provide centralized communication medium between different objects in a system.

It is helpful in enterprise applications where multiple objects are interacting with each other. If the objects interact with each other directly, the system components are tightly coupled with each other that makes higher maintainability cost and not hard to extend. Mediator pattern focuses on provide a mediator between objects for communication and help in implementing lose coupling between objects.

The system objects that communicate with each other are called colleagues. Usually, we have an interface or abstract class that provides the contract for communication and then we have the concrete implementation of the mediators.

Mediator design patterns are useful when the communication between the objects is complex we can have a central point of communication that takes care of communication logic.

**Chain of responsibility design pattern**

It is used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them. Then the objects in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not.

**Strategy design pattern**

Strategy pattern is used when we have multiple algorithms for a specific task and client decides actual implementation to be used at the runtime.

One of the best examples of strategy pattern is collections.sort() method that takes comparator parameter. Based on the different implementations of the comparator interface the objects are getting sorted in different ways.

Example: lambda expressions

**State design pattern**

It is used when an object change its behaviour based on its internal state.

The benefit of using state pattern to implement polymorphic behaviour. Thus, making our code more robust, easily maintainable and flexible.

Example: suppose we want to implement an AC remote with a simple button to perform action. If the state is ON, it will turn on the AC, if it is off it will turn of the AC.

**Visitor design pattern**It is used when we must perform an operation on a group of similar kind of objects. With the help of visitor pattern, we can move operational logic from the object to another class.

The benefit of the pattern is that if the logic of operation changes, then we need to make change in the visitor implementation rather than doing it in the classes.

And adding a new class to the system is easy, it will require change only in visitor interface and implementation and existing classes will not be affected.

**Observer design pattern**

It is used when we are interested in the state of the object and want to get notified whenever there is a change in the state of the object.

The object that watch on the state of the another object are called observer and the object that is being watched is called subject.

In this we define one to many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically.

Subject contains a list of observers to notify of any change in its state, so it should provide methods using which observers can register and unregister themselves. Subject also contain a method to notify all the observers of any change and it either can send the update while notifying the observer or it can provide another method to get the update.

observer should have a method to set the object to watch and another method that will be used by subject to notify them of any updates.

**Iterator design pattern**

Iterator pattern provides a standard way to traverse through list of objects. Iterator is widely used in java collections framework

Iterator interface provides methods to traverse through the collection.

This pattern provides a way to access the elements of an aggregate object without exposing its underlying representation.

Iterator pattern is not only about traversing through the collection, we can provide different kind of iterator based on our requirements.

Iterator design pattern hides the actual implementation of traversal through the collection and client program just use iterator methods.

It is useful when you want to provide a standard way to iterator over a collection and hide the implementation logic from the client program.

Example: all implementations of java.util.iterator

**Interpreter design pattern**

It is used to define grammatical representations for a language and provide an interpreter to deal with the grammar.

Example:

java compiler that interprets the java source code into byte code that is understandable to JVM.

Google translator the input can be any language we can get out interpreted in another language.

Java.util.pattern and subclasses of java.text.format are some examples of interpreter pattern used in jdk.

**Command design pattern**

A request is wrapped under an object as a command and passes the command to the corresponding object which executes the command.

**Why Were Design Patterns Introduced?**

Before the introduction of design patterns, developers often faced recurring design challenges without a common language or proven solutions to address them. This led to:

* Inconsistent Solution
* Reinventing the Wheel
* Lack of Communication

Design patterns were introduced to:

* Provide Reusable Solutions
* Facilitate Communication
* Enhance Software Quality

**Problems Solved by Design Patterns**

* Enhancing Communication
* Code Reusability and Maintainability
* Complexity Management
* Flexibility and Scalability
* Decoupling Components

Port number: -5432

Pass: - root