Behavioral

SOLID Design Principles

S: Single Responsibility Principle.

* The single class should take just one primary responsibility.

O: Open-Close principle (OCP)

* Open for extension and closed for modification.

L: Liskov Substitution Principle

* A base class can be substituted with child class without breaking the functionality. This require that all the subclass behaves in same way as parent class. Model your classes based on behaviors not properties; model your data based on properties and not on behaviors. LSP applied on interfaces and contracts. All the subclass should honor the contract. A square is rectangle is true. But rectangle can have different height and width, but square can’t. So, Rectangle contract states that height and width can be different but it is not true for square so they can’t have IS-A relationship. It violates LSP.

I: Interface segregation principle.

* An interface should only implement behavior once. The interface should not contain behavior which a client doesn’t need or isn’t relevant to it.

D: Interface Segregation Principle.

* High-level modules should not depend on low-level modules. Both should depend on abstractions.<https://refactoring.guru/>
* Abstractions should not depend on details. Details should depend on abstraction.
* The high-level and low-level module should be separated with abstraction.
* Open-Close Principle – Abstraction (Interface) should be closed for modification.
* Liskov Substitution Principle – High-level module should be able to access multiple implementation of low-level module via abstraction.

Design Patterns are common architectural approaches. Design patterns are universally relevant.

Creational

* Deal with creations of the object.
* Explicit (constructor) vs implicit (DI, reflection etc.)
* Wholesale (Single live) vs piecewise (step-by-step)

Structural

* Concerned with structure.
* Many Patterns are wrappers that mimic the underlaying class interface.
* Stress the importance of good API design.

Behavioral

* The are all different, no clear pattern.
* They solve individual problem with each pattern.

**Creational Patterns**

**Builder Design Pattern**

When creation of object is bit complicated, or the creation of object requires multiple arguments then it should be created step-by-step and builder design pattern helps to provide guidelines for creating such objects. When piecewise object construction is complicated, provide an API for doing it succinctly.

StringBuilder is java provided example of builder design pattern.

StringBuilder is also example of Fluent Builder pattern. The StringBuilder method append() returns StringBuilder so

chain of StringBuilder objects can be created fluently.

**Factory Design Pattern**

* Object creation becomes too complex.
* Constructors are not descriptive. The name of the constructor is name of the class.
* Constructors can’t be differentiated based on name of the parameters.
* To achieve different object creation ways, the class is filled with multiple constructors. It is very difficult

to differentiate between purpose of each constructor as name of the constructors are same.

* Factory pattern is used to create wholesale object creation, not piecewise.

**Abstract Factory Pattern**

* Abstract Factory patterns work around a super-factory which creates other factories.
* Abstract Factory pattern provides another level of abstraction over factory pattern.
* Abstract Factory provides interfaces for creating families of related or dependent objects without specifying their concrete classes.

**Prototype Design Pattern**

* It is also a creational pattern.
* It helps to create object from existing objects.
* It is used when creating an object is quite complex and expensive.
* It is better to create prototype from existing objects and modify the required fields rather than creating whole new objects.
* Cloneable
* Copy Constructor
* Serializable

**Singleton Design Pattern**

* A component which is instantiated only once.
* Singleton pattern restricts the instantiation of a class and ensures that only one instance of the class exists in the Java Virtual Machine.
* Singleton pattern is used for logging, drivers objects, caching, and thread pool.
* Singleton design pattern is also used in other design patterns like Abstract Factory, Builder, Prototype, Facade, etc.
* Object creation is expensive.
* Prevent clients from creating multiple instances as it contains state data which should be consistent and should be available to all.
* Lazy initialization or thread-safty.
* Multiton Design Pattern: The multiton design pattern generalize Singleton design pattern. The multiton design pattern allows control creation of multiple instances, which it manages using a map.

**c**

**Adapter Design Pattern**

* Allow objects with incompatible interfaces to collaborate.
* Adapter is a special object that converts the interface of the object so that another object can understand it.

**Bridge Design Pattern**

* It solves cartesian product problem in programming world.
* The Bridge pattern attempts to solve problem by switching from inheritance to the object composition. What this means is that you extract one of the dimensions into a separate class hierarchy, so that the original classes will reference an object of the new hierarchy, instead of having all its state and behaviors within one class.

**Composite Design Pattern**

* It lets you compose objects into tree structures and then work with these structures as if they were individual objects.
* The client would work with single interface, the interface can be an object or composition of objects.
* The single call over interface will traverse over all the nodes, like traversing a tree.

**Decorator Design Pattern**

* Adding new behaviors to objects dynamically by placing them inside special wrapper objects called, decorators.
* The object and decorators follow the same interface.
* Decorator can be recognized by creation methods or constructors that accept objects of the same class or interface as a current class.

**Façade Design Pattern**

* Provides a simple, easy to understand/user interface over a large and sophisticated body of code.

**Flyweight Design Pattern**

* Flyweightdesign pattern allows application to fit more objects into the available amount of memory by sharing common parts of state between multiple objects instead of keeping all the data in each object.
* The object that stores only constant data is called flyweight. It is referenced from all objects sharing same state.
* Saving name and formatting text as example.

**Proxy Design Pattern**

* Provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.
* Proxy is an interface for accessing a particular resource.
* Proxy provides you same interface but entirely different behavior.

**Behavior Design Pattern**

**Chain of Responsibility**

* Pass requests along a chain of handlers. Each handler decides either to process the request or pass it to next handler in chain.
* Authentication + Authorization + Validation + Caching
* Authentication Handler => Authorization Handler => Validation Handler => Caching Handler.
* setNext(h: Handler) and handle(request)

**Command Design Pattern**

* Command design pattern turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method argument, delay or queue a request’s execution and support undoable operations.
* Command object contains all the information to perform action on the request. It contains request details, action to be performed, action recorder etc.

**Interpreter Design Pattern**

* Grammatical representation for a language. (Define the rules)
* Interpreter to interpret the grammar. (understand rules and converts language to object-oriented design.)
* **Lexing:** The process of converting a sequence of characters into sequence of lexical token. Each token has defined meaning(comes from grammer).
* **Parsing:** Interpreting sequences of said token(parsing) and converting them back to language.

**Iterator Design Pattern**

* An iterator is a class that facilitates the traversal.
* Keep a reference to the current element.
* Knows how to move to a different element.
* The main idea of the Iterator pattern is to extract the traversal behavior of a collection into a separate object called an *iterator*.

**Mediator Design Pattern**

* The pattern restricts direct communications between the objects and force them to collaborate only via a mediator objects.
* The objects communicating not necessarily have reference to each other and only aware about mediator object.
* Pilots approaching airport don’t communicate with each other but with ATC which acts as mediator.