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.
* 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