**Java design patterns**

1. Creational patterns – related to object creation
2. Structural pattern – related to class structure and integration related structure problems
3. Behavioral patterns – related to link between class and object - algorithm and run related issues solving

**Creational Pattern:**

1. **Singleton class** –
2. **Builder Pattern** - such as instantiating and initializing the components that make up the object, are kept within the object, often as part of its constructor.

Eg)

* + java.lang.StringBuilder#append() (unsynchronized)
  + java.lang.StringBuffer#append() (synchronized)
  + java.nio.ByteBuffer#put() (also on CharBuffer, ShortBuffer, IntBuffer, LongBuffer, FloatBuffer and DoubleBuffer)
  + javax.swing.GroupLayout.Group#addComponent()
  + All implementations of java.lang.Appendable

1. **Factory Method** -

Factory design simplifies the object creation for different kinds of mode. The Factory Method Pattern give us a way to encapsulate the instantiations of concrete types. The Factory Method pattern encapsulates the functionality required to select and instantiate an appropriate class, inside a designated method referred to as a factory method. The Factory Method selects an appropriate class from a class hierarchy based on the application context and other influencing factors. It then instantiates the selected class and returns it as an instance of the parent class type. The advantage of this approach is that the application objects can make use of the factory method to get access to the appropriate class instance. This eliminates the need for an application object to deal with the varying class selection criteria.

Eg)

The following are the usage(s) of the Factory Method Pattern in JDK.

* java.util.Calendar#getInstance()
* java.util.ResourceBundle#getBundle()
* java.text.NumberFormat#getInstance()
* java.nio.charset.Charset#forName()
* java.net.URLStreamHandlerFactory#createURLStreamHandler(String) (Returns singleton object perProtocol)

1. **Abstract Factory Design** –

It is an abstracted version of factory method and there is no specified concreate object creation here.

Eg )Abstract Factory Pattern in JDK

• java.util.Calendar#getInstance()

• java.util.Arrays#asList()

• java.util.ResourceBundle#getBundle()

• java.sql.DriverManager#getConnection()

• java.sql.Connection#createStatement()

• java.sql.Statement#executeQuery()

• java.text.NumberFormat#getInstance()

• javax.xml.transform.TransformerFactory#newInstance()

• java.sql.Statement#executeQuery()

• java.text.NumberFormat#getInstance()

• javax.xml.transform.TransformerFactory#newInstance()

1. **Prototype Design Pattern** – If you want to create a complex object dynamically then we can clone the particular object and change only few states instead of creating the new object.

Prototype Pattern in JDK

• java.lang.Object#clone()

• java.lang.Cloneable

**Structural patterns:**

1. **Composite pattern** – make composite all the object into the class – XML pull parsing or HTML tag in java class

**Real Time:** There can be lots of practical examples of the Composite Pattern. A file directory system, an html representation in java, an XML parser all are well managed composites and all can easily be represented using the Composite Pattern. But before digging into the details of an example, let’s see some more details regarding the Composite Pattern.

1. **Bridge design pattern** – if the requirement keep on growing better split the growing pat as interface and pass the interface as parameter to implement dynamically

**Real Time:** A design with too many subclasses is not flexible and is hard to maintain. An Inheritance also binds an implementation to the Abstraction permanently, which makes it difficult to modify, extend, and reuse the abstraction and implementation independently.

Please note that, the car and the product should vary independently in order to make the software system easy to extend and reusable.

1. **Proxy Pattern** – Used to solve the problem access issue. It is a controller or place holder of another object.

The Proxy Pattern is used to create a representative object that controls access to another object, which may be remote, expensive to create or in need of being secured. One reason for controlling access to an object is to defer the full cost of its creation and initialization until we actually need to use it. Another reason could be to act as a local representative for an object that lives in a different JVM. The Proxy can be very useful in controlling the access to the original object, especially when objects should have different access rights.

In the Proxy Pattern, a client does not directly talk to the original object, it delegates it calls to the proxy object which calls the methods of the original object. The important point is that the client does not know about the proxy, the proxy acts as an original object for the client.

* 1. Remote Proxy
  2. Production control
  3. Virtual proxy – lazy loading proxy

1. **Flyweight Design** - to avoid creating multiple objects we can store those object and retrieve from the storage from when we required.

The Flyweight Pattern is designed to control such kind of object creation and provides you with a basic caching mechanism. It allows you to create one object per type (the type here differs by a property of that object), and if you ask for an object with the same property (already created), it will return you the same object instead of creating a new one.

Eg) java.lang.Integer#valueOf(int) (also on Boolean, Byte, Character, Short and Long)

1. **Adaptor pattern** – to adapt to particular interface without changing the previous interface

**Real Time :** A software developer, Max, has worked on an e-commerce website. The website allows users to shop and pay online. The site

is integrated with a 3rd party payment gateway, through which users can pay their bills using their credit card. Everything was going well, until his manager called him for a change in the project. The manager told him that they are planning to change the payment gateway vendor, and he has to implement that in the code.

1. **Façade pattern** - simplify the complexity of the APIs. Make all the complexity in the subclass and the parent class will provide simple calls for the client

The Facade Pattern makes a complex interface easier to use, using a Facade class. The Facade Pattern provides a unified interface to a set of interface in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

**Real Time:** Everything was going fine, until the clients started complaining about starting and stopping the process of the server. They said, although the server is working great, the initializing and the shutting down processes are very complex and they want an easy way to do that. The server has exposed a complex interface to the clients which looks a bit hectic to them. We need to provide an easy way to start and stop the server.

A complex interface to the client is already considered as a fault in the design of the current system. But fortunately or unfortunately, we cannot start the designing and the coding from scratch. We need a way to resolve this problem and make the interface easy to access. A Facade Pattern can help us to resolve this design problem. But before that, let us see about the Facade Pattern.

1. **Decorator Design Pattern**

The intent of the Decorator Design Pattern is to attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to sub-classing for extending functionality. The Decorator Pattern is used to extend the functionality of an object dynamically without having to change the original class source or using inheritance. This is accomplished by creating an object wrapper referred to as a Decorator around the actual object.

To understand the Decorator Design Pattern, let’s help a pizza company make an extra topping calculator. A user can ask to add extra topping to a pizza and our job is to add toppings and increase its price using the system. This is something like adding an extra responsibility to our pizza object at runtime and the Decorator Design Pattern is suitable

for this type of requirement. But before that, let us know more about this beautiful pattern.

**Decorator Design Pattern in Java**

• java.io.BufferedInputStream(InputStream)

• java.io.DataInputStream(InputStream)

• java.io.BufferedOutputStream(OutputStream)

• java.util.zip.ZipOutputStream(OutputStream)

• java.util.Collections#checked[List|Map|Set|SortedSet|SortedMap]()

**Behavioral patterns:**

1. **Observable design patter** – android gps module is example for this design

The other way to understand the Observer Pattern is the way Publisher-Subscriber relationship works. Let’s assume for example that you subscribe to a magazine for your favorite sports or fashion magazine. Whenever a new issue is published, it gets delivered to you. If you unsubscribe from it when you don’t want the magazine anymore, it will not get delivered to you. But the publisher continues to work as before, since there are other people who are also subscribed to that particular magazine.

**Use the Observer pattern in any of the following situations:**

• When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.

• When a change to one object requires changing others, and you don’t know how many objects need to be changed?

• When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don’t want these objects tightly coupled.

1. **Mediator pattern** - Act as mediator which will talk to all module

Instead of making loop communication between the modules

Design Patterns are used almost everywhere in JDK. The following are the usages of the Mediator Pattern in JDK

• java.util.concurrent.ScheduledExecutorService (all scheduleXXX() methods)

• java.util.concurrent.ExecutorService (the invokeXXX() and submit() methods)

• java.util.concurrent.Executor#execute()

• java.util.Timer (all scheduleXXX() methods)

• java.lang.reflect.Method#invoke()

1. **Chain of sequence pattern** – group of objects is chained together in a sequence and a responsibility (a request) is provided in order to be handled by the group. If an object in the group can process the particular request, it does so and returns the corresponding response. Otherwise, it forwards the request to the subsequent object in the group.

Use Chain of Responsibility when

• More than one objects may handle a request, and the handler isn’t known a priori. The handler should be ascertained automatically.

• You want to issue a request to one of several objects without specifying the receiver explicitly.

• The set of objects that can handle a request should be specified dynamically

Chain of Responsibility in JDK

The following are the usages of the Chain of Responsibility Pattern in Java.

• java.util.logging.Logger#log()

• javax.servlet.Filter#doFilter().

1. **State Design Pattern –**

Allows an object to alter its behavior when its internal state changes. The object will appear to change its class. The state of an object can be defined as its exact condition at any given point of time, depending on the values of its properties or attributes. The set of methods implemented by a class constitutes the behavior of its instances. Whenever there is a change in the values of its attributes, we say that the state of an object has changed. The State pattern is useful in designing an efficient structure for a class, a typical instance of which can exist in many different states and exhibit different behavior depending on the state it is in. In other words, in the case of an object of such a class, some or all of its behavior is completely influenced by its current state. In the State design pattern terminology, such a class is referred to as a Context class. A Context object can alter its behavior when there is a change in its internal state and is also referred

as a Stateful object.

Use the State pattern in either of the following cases:

• An object’s behavior depends on its state, and it must change its behavior at run-time depending on that state.

• Operations have large, multipart conditional statements that depend on the object’s state. This state is usually represented by one or more enumerated constants. Often, several operations will contain this same conditional structure. The State pattern

puts each branch of the conditional in a separate class. This lets you treat the object’s state as an object in its own right that can

vary independently from other objects.

State Design Pattern in Java

• javax.faces.lifecycle.LifeCycle#execute()

1. **Strategy Design Pattern** – it is an very easy and most using design pattern

We should pass the specific object to the strategy class to do operation.

Use the Strategy pattern when:

• Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.

• You need different variants of an algorithm. For example, you might define algorithms reflecting different space/time trade-offs. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.

• An algorithm uses data that clients shouldn’t know about. Use the Strategy pattern to avoid exposing complex, algorithm specific data structures.

• A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Strategy Pattern in JDK

• java.util.Comparator#compare()

• javax.servlet.http.HttpServlet

• javax.servlet.Filter#doFilter()

1. **Command Design Pattern**–

The intent of the Command Design Pattern is to encapsulate a request as an object, thereby letting the developer to parameterize clients with different requests, queue or log requests, and support undoable operations.

The Command Design Pattern is a behavioral design pattern and helps to decouple the invoker from the receiver of a request.

Command Design Pattern in JDK

• java.lang.Runnable

• javax.swing.Action

1. **Interpreter Design Pattern** –

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language. In general, languages are made up of a set of grammar rules. Different sentences can be constructed by following this grammar rules. Sometimes an application may need to process repeated occurrences of similar requests that are a combination of a set of grammar rules. These requests are distinct but are similar in the sense that they are all composed using the same set of rules. A simple example of this would be the set of different arithmetic expressions submitted to a calculator program. Though each such expression is different, they are all constructed using the basic rules that make up the grammar for the language of arithmetic expressions.

Interpreter Design Pattern in JDK

• java.util.Pattern

• java.text.Normalizer

• java.text.Format

1. **Iterator Design Pattern**

The intent of the Iterator Design Pattern is to provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

The Iterator pattern allows a client object to access the contents of a container in a sequential manner, without having any knowledge about the internal representation of its contents. The term container, used above, can simply be defined as a collection of data or objects. The objects within the container could in turn be collections, making it a collection of collections.

An aggregate object, such as a list, should give you a way to access its elements without exposing its internal structure. Moreover, you might want to traverse the list in different ways, depending on what you want to accomplish. But you probably don’t want to

bloat the List interface with operations for different traversals, even if you could anticipate the ones you will need. You might also need to have more than one traversal pending on the same list. The Iterator pattern lets you do all this. The key idea in this pattern is to take the responsibility for access and traversal out of the list object and put it into an iterator object. The Iterator class defines an interface for accessing the list’s elements.

An iterator object is responsible for keeping track of the current element; that is, it knows which elements have been traversed already.

Iterator Pattern in JDK

• java.util.Iterator

• java.util.Enumeration

1. **Visitor Design Pattern –**

The Visitor Design Pattern provides you with a way to add new operations on the objects without changing the classes of the elements, especially when the operations change quite often.

The Visitor pattern is useful when designing an operation across a heterogeneous collection of objects of a class hierarchy. The Visitor pattern allows the operation to be defined without changing the class of any of the objects in the collection. To accomplish this, the Visitor pattern suggests defining the operation in a separate class referred to as a visitor class. This separates the operation from the object collection that it operates on. For every new operation to be defined, a new visitor class is created. Since the operation is to be performed across a set of objects, the visitor needs a way of accessing the public members of these objects.

This requirement can be addressed by implementing the following two design ideas.

Visitor Design Pattern in JDK

• javax.lang.model.element.Element and javax.lang.model.element.ElementVisitor

• javax.lang.model.type.TypeMirror and javax.lang.model.type.TypeVisitor

1. **Memento Pattern –** To store the object state with in the object itself we can use this mechanism.

Memento Pattern in JDK

• java.util.Date

• java.io.Serializable

1. **Template Pattern -** defines the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses to redefine certain steps of an algorithm without changing the algorithm’s structure.

Template Pattern in JDK

• java.util.Collections#sort()

• java.io.InputStream#skip()

• java.io.InputStream#read()

• java.util.AbstractList#indexOf()