Design Patterns:
Design patterns act as a description or template for solving a common problem in software designing process, and thus, provide a repeatable solution, that can speed up the development process by providing tested, proven development paradigms.
Design patterns help avoid dealing with issues, that may emerge later in the software development process.

Creational Design Pattern: These design patterns are concerned with the creation of objects.

(1) Singleton Pattern: This is one of the simplest design patterns, and provides the easiest way to create an object. Singleton pattern should be used when there must only be a single instance of a class and it must be available throughout the code. It provides the access to its only object without the need to instantiate the class object. Special care has to be taken when multiple threads have to access the same resource through the same singleton object.

A singleton pattern can be used in situations like:
- Logger classes.
- Configuration classes.
- Shared mode resource sharing.

While using this design pattern, we make the class of the single instance object take care of the creation, initialization and access of the singleton object. One can implement it in Java by declaring the instance as a private static data member and encapsulating all the initialization code and providing access to the instance.

Advantages: Provider controls access to the single and only instance. Singleton pattern is extensible to support application specific number of instances. Avoids polluting the namespace with global variables.

(2) Abstract Factory Pattern: This design pattern provides the interface to create a family of related objects, without specifying their classes explicitly. It can be used when a system needs to be independent of its products, namely, from the tasks of creation, composition, and representation, or products of the same family are to be used together and products from different families should never be used together. It makes sure that the implementation remains hidden and only the product interfaces are revealed.

The following classes are involved in the implementation of Abstract Factory:
- AbstractFactory: Declares interface for operations to create abstract products.
- ConcreteFactory: Implements the operations to create those abstract products.
- AbstractProduct: Declares interface for the product type.
- Product: Implements the AbstractProduct interface.
- Client: uses the interfaces declared.

Thus, the Abstract Factory provides interface to create families of related objects, without any need for specifying concrete classes.

Advantages:
It keeps the client from knowing anything about the actual creation of objects. When new concrete types are needed, only thing to be done is to modify the client code and make it use a different

factory rather than instantiating a new type, that would have required changing the code when new object is created.

Structural Design Patterns:
These design Patterns ease the designing process by identifying a simple way to realize relationships between entities involved. The following are a few types:

(1) Bridge Pattern: The Bridge pattern is used to decouple an abstraction from its implementation so that the two can vary independently. The motive behind this design pattern is to publish interface in an inheritance hierarchy, and bury implementation in its own inheritance hierarchy.

This pattern can be useful when:

- User wants run-time binding of the implementation
- There is a proliferation of classes as a result of coupled interface and numerous implementations
- User wants to share an implementation among multiple objects.

The following classes participate in its implementation:

- Abstraction: Defines interface for abstraction
- AbstractionImplmentation: Implements the abstraction interface.
- Implementor: Defines interface for implementation classes
- ConcreteImplementor: Implements implementor interface.

This helps in hiding details from the clients and helps improve extensibility. It is the proper choice when one wants to avoid any sort of permanent binding between abstraction and implementation and thus each of them can vary independently of each other.

(2) Adapter Pattern: Sometimes the user needs to use an old component, but cannot move forward due to the incompatibility with the new system being currently developed. The adapter design pattern can help in such situations. It wraps an existing class with a new interface. In other words, it can convert the interface of a class into interface of another class, and thus impedance matches an old component to a new system.
The following classes/objects participate in this design pattern:
Target: Defines interface that client uses.
Client: Collaborates to the objects that conform to the Target.
Adapter: Defines existing interface that needs adapting.
Adaptee: Adapts the Adaptee interface to Target interface

Adapter is about creating an intermediary abstraction that translates, or maps, the old component to the new system. Clients call methods on the Adapter object which redirects them into calls to the legacy component. This strategy can be implemented either with inheritance or with aggregation.

Adapter functions as a wrapper or modifier of an existing class. It provides a different or translated view of that class. It allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients.

Behavioural Design pattern:
Behavioral design patterns identify common communication patterns between objects and realize these patterns, thereby making the communication flexible. The following are a few types:

(1) Observer Pattern: A large monolithic design does not scale very well as and when new requirements are added. Observer pattern tries to overcome this by defining a one-to-many dependency between the objects.Thus, when one object changes its state, all its dependents are notified and are automatically updated. Thus it is used only when: a change in state of one object is to be reflected in another object without keeping tight coupling between objects; there is a scope of future enhancement involving adding new observers with minimal changes. It specifies a pull interaction model.

Example: MVC Pattern.

It can be described as follows:

- Encapsulating the core components of the data model in a Subject abstraction.
- Encapsulating the variable components in an Observer hierarchy, that consists of decoupled and distinct Observer objects.
- Each Observer registers with the Subject, once they are created.
- The Subject broadcasts to all the registered Observers when a change occurs.
- Thus, the number and type of “view” objects can be configured dynamically and not statically at compile time.
- Each Observer is responsible for "pulling" required information from the Subject.

(2) Command Pattern:
The most used application of this design pattern is in implementation of the undo feature in most of the applications.
It allows:

- Encapsulation of request in an object.
- Parametrization of clients with various kinds of requests.
- Saving these requests in a queue.

The design pattern can be implemented with the following participating classes:

- Command: Declaring an interface for execution of an operation.
- Invoker: Asks the Command to carry out requests.
- Receiver: Knows how to perform operations.
- Client: Creates a ConcreteCommand object and sets the receiver.
- ConcreteCommand: Extending Command interface and implementing the Execute method.

Client asks for execution of a command. Invoker takes command. Encapsulates it. Places it in queue and ConcreteCommand in charge of the requested command. Sends the results to Receiver.
The implementation should be interpreted as follows: Client that executes a command is not the same as the client that creates that command. This provides for flexibility in the sequencing of commands as per requirement. The client that creates these command “tokens” knows what needs to be done, and passes it to the other client who knows what to execute and has the resources for the same.