**Java Design Patterns**

1.Creational Design Pattern

1. Factory Pattern
2. Abstract Factory Pattern
3. Singleton Pattern
4. Prototype Pattern
5. Builder Pattern.

2. Structural Design Pattern

1. Adapter Pattern
2. Bridge Pattern
3. Composite Pattern
4. Decorator Pattern
5. Facade Pattern
6. Flyweight Pattern
7. Proxy Pattern

3. Behavioral Design Pattern

1. Chain Of Responsibility Pattern
2. Command Pattern
3. Interpreter Pattern
4. Iterator Pattern
5. Mediator Pattern
6. Memento Pattern
7. Observer Pattern
8. State Pattern
9. Strategy Pattern
10. Template Pattern
11. Visitor Pattern

**CREATIONAL DESIGN PATTERN**

Factory Method Pattern

Factory Method Pattern allows the sub-classes to choose the type of objects to create.

**Defines an interface or abstract class for creating an object but let the subclasses decide which class to instantiate.** Subclasses are responsible to create the instance of the class.

An abstract class or interface that declares the factory method.

The Factory Method Pattern is also known as **Virtual Constructor.**

**Uses:**

* Factory Method Pattern allows the sub-classes to choose the type of objects to create.
* It promotes the loose-coupling by eliminating the need to bind application-specific classes into the code.

**When it is required?**

* When a class doesn't know what sub-classes will be required to create
* When a class wants that its sub-classes specify the objects to be created.
* When the parent classes want to delegate the creation of objects to its sub-classes.

# Abstract Factory Pattern

**Defines an interface or abstract class for creating families of related objects but without specifying their concrete sub-classes.**

That means Abstract Factory lets a class returns a factory of classes.

So, this is the reason that Abstract Factory Pattern is one level higher than the Factory Pattern.

An Abstract Factory Pattern is also known as **Kit.**

**Uses:**

* Abstract Factory Pattern isolates the client code from concrete (implementation) classes.
* It eases the exchanging of object families.
* It promotes consistency among objects.

**When it is required?**

* When the system needs to be independent of how its object are created, composed, and represented.
* When the family of related objects has to be used together, then this constraint needs to be enforced.
* When you want to provide a library of objects that does not show implementations and only reveals interfaces.
* When the system needs to be configured with one of a multiple family of objects.

# Singleton design Pattern

**Defines a class that has only one instance and provides a global point of access to it.**

A class must ensure that only single instance should be created and single object can be used by all other classes.

There are two forms of singleton design pattern

* **Early Instantiation:** creation of instance at load time.
* **Lazy Instantiation:** creation of instance when required.

**Uses:**

* Saves memory because object is not created at each request. Only single instance is reused again and again.

**When it is Required?**

* Singleton pattern is mostly used in multi-threaded and database applications. It is used in logging, caching, thread pools, configuration settings etc.

**How to create?**

To create the singleton class, we need to have static member of class, private constructor and static factory method

* **Static member:** It gets memory only once because of static, it contains the instance of the Singleton class.
* **Private constructor:** It will prevent to instantiate the Singleton class from outside the class.
* **Static factory method:** This provides the global point of access to the Singleton object and returns the instance to the caller.

# Prototype Design Pattern

**Cloning of an existing object instead of creating new one and can also be customized as per the requirement**. **when object creation is a time-consuming, and costly operation, so we create objects with the existing object itself.**

**Uses:**

* It reduces the need of sub-classing.
* It hides complexities of creating objects.
* The clients can get new objects without knowing which type of object it will be.
* It lets you add or remove objects at runtime.

**When it is Required?**

* **When the classes are instantiated at runtime.**
* **When the client application needs to be unaware of object creation and representation.**

# Builder Design Pattern

**construct a complex object from simple objects using step-by-step approach.**

**Uses:**

mostly used when object can't be created in single step like in the de-serialization of a complex object.

**When it is Required?**

* It provides clear separation between the construction and representation of an object.
* It provides better control over construction process.
* It supports to change the internal representation of objects.

# Object Pool Pattern

**An Object pool is a container which contains a specified amount of objects. When an object is taken from the pool, it is not available in the pool until it is put back. Objects in the pool have a lifecycle: creation, validation and destroy.**

**Uses:**

* It boosts the performance of the application significantly.
* It is most effective in a situation where the rate of initializing a class instance is high.
* It manages the connections and provides a way to reuse and share them.
* It can also provide the limit for the maximum number of objects that can be created.

**When it is Required?**

* When an application requires objects which are expensive to create. Eg: there is a need of opening too many connections for the database then it takes too longer to create a new one and the database server will be overloaded.
* When there are several clients who need the same resource at different times.
* Object pool design pattern is essentially used in Web Container of the server for creating thread pools and data source pools to process the requests.

**STRUCTURAL DESIGN PATTERN**

Adapter Pattern

Provide the interface according to client requirement while using the services of a class with a different interface. Converts the interface of a class into another interface that a client wants.

The Adapter Pattern is also known as **Wrapper.**

Target Interface: This is the desired interface class which will be used by the clients.

Adapter class: This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class.

Adaptee class: This is the class which is used by the Adapter class to reuse the existing functionality and modify them for desired use.

Client: This class will interact with the Adapter class.

**Uses:**

* It allows two or more previously incompatible objects to interact.
* It allows reusability of existing functionality.

**When it is Required?**

* When an object needs to utilize an existing class with an incompatible interface.
* When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.
* When you want to create a reusable class that cooperates with classes which don't have compatible interfaces.

# Bridge Pattern

Decouple the functional abstraction from the implementation so that the two can vary independently. The Bridge Pattern is also known as Handle or Body

**Uses:**

* It enables the separation of implementation from the interface.
* It improves the extensibility.
* It allows the hiding of implementation details from the client.

**When it is Required?**

* When you don't want a permanent binding between the functional abstraction and its implementation.
* When both the functional abstraction and its implementation need to extended using sub-classes.
* It is mostly used in those places where changes are made in the implementation does not affect the clients.

# Composite Pattern

Allow clients to operate in generic manner on objects that may or may not represent a hierarchy of objects.

**Uses:**

* It defines class hierarchies that contain primitive and complex objects.
* It makes easier to you to add new kinds of components.
* It provides flexibility of structure with manageable class or interface.

**When it is Required?**

* When you want to represent a full or partial hierarchy of objects.
* When the responsibilities are needed to be added dynamically to the individual objects without affecting other objects. Where the responsibility of object may vary from time to time.

# Decorator Pattern

The Decorator Pattern uses composition instead of inheritance to extend the functionality of an object at runtime. Attach a flexible additional responsibilities to an object dynamically.

The Decorator Pattern is also known as **Wrapper.**

**Uses:**

* It provides greater flexibility than static inheritance.
* It enhances the extensibility of the object, because changes are made by coding new classes.
* It simplifies the coding by allowing you to develop a series of functionality from targeted classes instead of coding all of the behavior into the object.

**When it is Required?**

* When you want to transparently and dynamically add responsibilities to objects without affecting other objects.
* When you want to add responsibilities to an object that you may want to change in future.
* Extending functionality by sub-classing is no longer practical.

# Facade Pattern

Facade Pattern describes a higher-level interface that makes the sub-system easier to use. Just provide a unified and simplified interface to a set of interfaces in a subsystem, therefore it hides the complexities of the subsystem from the client. Every Abstract Factory is a type of Facade.

**Uses:**

* It shields the clients from the complexities of the sub-system components.
* It promotes loose coupling between subsystems and its clients.

**When it is Required?**

* When you want to provide simple interface to a complex sub-system.
* When several dependencies exist between clients and the implementation classes of an abstraction.

# Flyweight Pattern

To reuse already existing similar kind of objects by storing them and create new object when no matching object is found.

**Uses:**

* It reduces the number of objects.
* It reduces the amount of memory and storage devices required if the objects are persisted

**When it is Required?**

* When an application uses number of objects
* When the storage cost is high because of the quantity of objects.
* When the application does not depend on object identity.

# Proxy Pattern

Proxy means an object representing another object. provides the control for accessing the original object. we can perform many operations like hiding the information of original object, on demand loading etc. Proxy pattern is also known as **Surrogate or Placeholder.**

RMI API uses proxy design pattern. Stub and Skeleton are two proxy objects used in RMI.

**Uses:**

* It provides the protection to the original object from the outside world.
* It can be used in **Virtual Proxy** scenario---Consider a situation where there is multiple database call to extract huge size image. Since this is an expensive operation so here we can use the proxy pattern which would create multiple proxies and point to the huge size memory consuming object for further processing. The real object gets created only when a client first requests/accesses the object and after that we can just refer to the proxy to reuse the object. This avoids duplication of the object and hence saving memory.
* It can be used in **Protective Proxy** scenario---It acts as an authorization layer to verify that whether the actual user has access the appropriate content or not. For example, a proxy server which provides restriction on internet access in office. Only the websites and contents which are valid will be allowed and the remaining ones will be blocked.
* It can be used in **Remote Proxy** scenario---A remote proxy can be thought about the stub in the RPC call. The remote proxy provides a local representation of the object which is present in the different address location. Another example can be providing interface for remote resources such as web service or REST resources.
* It can be used in **Smart Proxy** scenario---A smart proxy provides additional layer of security by interposing specific actions when the object is accessed. For example, to check whether the real object is locked or not before accessing it so that no other objects can change it.

# Behavioral Design Patterns

# Chain Of Responsibility Pattern

# In chain of responsibility, sender sends a request to a chain of objects. The request can be handled by any object in the chain.

A Chain of Responsibility Pattern says that just **"avoid coupling the sender of a request to its receiver by giving multiple objects a chance to handle the request".** For example, an ATM uses the Chain of Responsibility design pattern in money giving process.

we can say that normally each receiver contains reference of another receiver. If one object cannot handle the request then it passes the same to the next receiver and so on.

**Uses:**

* It reduces the coupling. It adds flexibility while assigning the responsibilities to objects.
* It allows a set of classes to act as one; events produced in one class can be sent to other handler classes with the help of composition.

**When it is Required?**

* When more than one object can handle a request and the handler is unknown.
* When the group of objects that can handle the request must be specified in dynamic way.

# Command Pattern

A Command Pattern says that "*encapsulate a request under an object as a command and pass it to invoker object. Invoker object looks for the appropriate object which can handle this command and pass the command to the corresponding object and that object executes the command*".

It is also known as **Action or Transaction.**

* Command This is an interface for executing an operation.
* ConcreteCommand This class extends the Command interface and implements the execute method. This class creates a binding between the action and the receiver.
* Client This class creates the ConcreteCommand class and associates it with the receiver.
* Invoker This class asks the command to carry out the request.
* Receiver This class knows to perform the operation.

**Uses:**

* It separates the object that invokes the operation from the object that actually performs the operation.
* It makes easy to add new commands, because existing classes remain unchanged.

**When it is Required?**

* When you need parameterize objects according to an action perform.
* When you need to create and execute requests at different times.
* When you need to support rollback, logging or transaction functionality.

# Interpreter Pattern

An Interpreter Pattern says that **"to define a representation of grammar of a given language, along with an interpreter that uses this representation to interpret sentences in the language".**

This pattern can applied for parsing the expressions defined in simple grammars and sometimes in simple rule engines.

**Uses:**

* It is easier to change and extend the grammar.
* Implementing the grammar is straightforward.

**When it is Required?**

* When the grammar of the language is not complicated.
* When the efficiency is not a priority.

# Iterator Pattern

**To access the elements of an aggregate object sequentially without exposing its underlying implementation.**

The Iterator pattern is also known as **Cursor.**

In collection framework, we are now using Iterator that is preferred over Enumeration.

java.util.Iterator interface uses Iterator Design Pattern.

**Uses:**

* It supports variations in the traversal of a collection.
* It simplifies the interface to the collection.

**When it is Required?**

* When you want to access a collection of objects without exposing its internal representation.
* When there are multiple traversals of objects need to be supported in the collection.

# Mediator Pattern

A Mediator Pattern says that "to define an object that encapsulates how a set of objects interact".

When we begin with development, we have a few classes and these classes interact with each other producing results. Now, consider slowly, the logic becomes more complex when functionality increases. Then what happens? We add more classes and they still interact with each other but it gets really difficult to maintain this code now. So, Mediator pattern takes care of this problem.

Mediator pattern is used to reduce communication complexity between multiple objects or classes. This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintainability of the code by loose coupling.

**Uses:**

* It decouples the number of classes.
* It simplifies object protocols.
* It centralizes the control.
* The individual components become simpler and much easier to deal with because they don't need to pass messages to one another.

**When it Required?**

* It is commonly used in message-based systems likewise chat applications.
* When the set of objects communicate in complex but in well-defined ways.

# Memento Pattern

A Memento Pattern says that "to restore the state of an object to its previous state". But it must do this without violating Encapsulation. Such case is useful in case of error or failure.

The Memento pattern is also known as **Token**.

Undo or backspace or ctrl+z is one of the most used operation in an editor. Memento design pattern is used to implement the undo operation. This is done by saving the current state of the object as it changes state.

**Uses:**

* It preserves encapsulation boundaries.
* It simplifies the originator.

**When it is Required?**

* It is used in Undo and Redo operations in most software.
* It is also used in database transactions.

# Observer Pattern

An Observer Pattern says that "just define a one-to-one dependency so that when one object changes state, all its dependents are notified and updated automatically".

The observer pattern is also known as Dependents or Publish-Subscribe.

**Uses:**

* It describes the coupling between the objects and the observer.
* It provides the support for broadcast-type communication.

**When it is Required?**

* When the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
* When the framework we writes and needs to be enhanced in future with new observers with minimal chamges.

# State Pattern

A State Pattern says that "the class behavior changes based on its state". In State Pattern, we create objects which represent various states and a context object whose behavior varies as its state object changes. The State Pattern is also known as Objects for States.

**Uses:**

* It keeps the state-specific behavior.
* It makes any state transitions explicit.

**When it is Required?**

* When the behavior of object depends on its state and it must be able to change its behavior at runtime according to the new state.
* It is used when the operations have large, multipart conditional statements that depend on the state of an object.

# Strategy Pattern

A Strategy Pattern says that "defines a family of functionality, encapsulate each one, and make them interchangeable". The Strategy Pattern is also known as Policy.

**Uses:**

* It provides a substitute to subclassing.
* It defines each behavior within its own class, eliminating the need for conditional statements.
* It makes it easier to extend and incorporate new behavior without changing the application.

**When it is Required?**

* When the multiple classes differ only in their behaviors.e.g. Servlet API.
* It is used when you need different variations of an algorithm.

# Template Pattern

A Template Pattern says that "just define the skeleton of a function in an operation, deferring some steps to its subclasses".

**Uses:**

* Reusing the code.

**When it is Required?**

* when the common behavior among sub-classes should be moved to a single common class by avoiding the duplication.