Core 2 J2EE Patterns

Presentation Tier

- Intercepting Filter
- Front Controller
- Application Controller
- Context Object
- View Helper
- Service To Worker
- Dispatcher View
- Composite View

Business Tier

- Business Delegate
- Service Locator
- Application Service
- Session Facade
- Business Object
- Transfer Object
- Value List Handler
- Composite Entity
- Transfer Object Assembler
- Transfer Object

Integration Tier

- Data Access Object
- Domain Store
- Service Activator
- Web Service Broker

Design Pattern Terminology

Term	Description
Pattern Name	Describes the essence of the pattern in a short but expressive name.
Intent	Describes what the pattern does.
Also known As	List any synonyms for the pattern
Motivation	provides an example of a problem and how the pattern solves that problem.
Applicability	Lists the situations where the pattern is applicable
Structure	Set of diagrams of the classes and objects that depict the pattern.
participants	Describes the classes and objects that participate in the design pattern and their responsibilities.
Collaborations	Describes how the participants collaborate to carry out their responsibilities.
Consequences	Describes the forces that exist with the pattern and the benefits, trade-offs, and the variable that is isolated by the pattern.

JAVA DESIGN PATTERNS

Creational Patterns

Factory Pattern

Abstract Factory Patterns

Singleton Pattern

Builder pattern

Prototype Pattern

Factory Pattern:- when we have a Superclass with

Structural Patterns

Adapter pattern

Bridge Pattern

Composite Pattern

Decorator Pattern

Facade Pattern

Flyweight Pattern

Proxy Pattern

Behavioral Patterns

Chain of Responsibility Pattern

Command Pattern

Interpreter Pattern

Mediator Pattern

Memento Pattern

Observer Pattern

State Pattern

Strategy Pattern

Template Pattern

Visitor Pattern

Java Design Patterns

Creational Design Pattern → Singleton DP

Singleton Pattern :- Singleton pattern restrict the instance of a class and ensures that only one instance of the class exists in the JVM.

→ The Singleton class must provide a global access point to get the instance of the class.

exist (eg. exist)

→ Singleton pattern is used for logging, drivers objects, caching and Thread Pool.

→ This Pattern is also used in other Pattern like AbstractFactory, Builder, Prototype, Facade etc

→ Singleton design pattern is used in Core java classes also.

Ex: Java.lang.Runtime, java.awt.Desktop.

Rules for Implementing Singleton pattern

→ private Constructor to restrict instantiation of the class from other class.

→ Private static variable of the same class that is the only instance of the class.

→ public static method that returns the instance of the class. This is the global access Point for outer world to get the instance of the Singleton class.

Ex: Thread Safe Singleton

```
public class ThreadSingleton {
① private static ThreadSingleton instance = null;
② private ThreadSingleton() {}
③ public static Synchronized getThreadSafeSing() {
      if (instance == null) {
          instance = new ThreadSingleton();
      }
      return instance;
```

Lazy Initialization

Eager initialization

```
Ex: ThreadSingleton instance = new ThreadSingleton();
```

Static block initialization :

```
class A {
    private A a = null;
    private A() {}
    static {
        if (a == null) {
            a = new A();
        }
    }
```

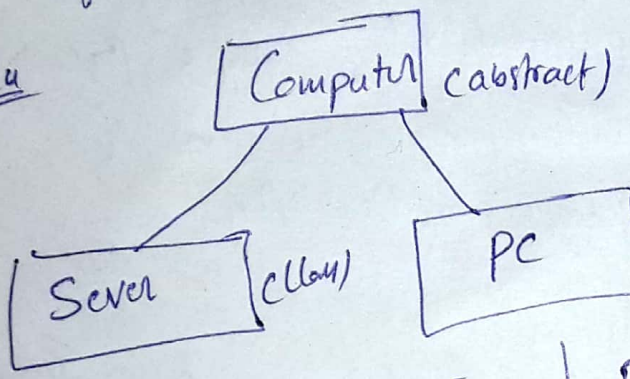
② Factory Design Pattern

Java → Creational Design Pattern → Factory Design Pattern

→ This Pattern Can be widely used in JDK as well as frameworks like Spring and Struts.

→ Factory design pattern is used when we have a Super class with multiple sub-classes and based on input we need to return one of the sub-class. This pattern is responsibility of Instantiation of a class from client program to the factory class.

Eg



Computer Factory (class)

Eg
```
public abstract Computer {
    public abstract String getRAM();
    public abstract String getHDD();
    public abstract Str getCPU();
    override
    Public String toStr() {
        return "RAM" + getRAM() +
               "HDD" + getHDD() +
               "CPU" + getCPU();
    }
```

PC – Subclass
```
public class PC extends Computer {
    private String ram;
    private Str hdd;
    private Str cpu;
    Set/get
    PC(String ram,
       Str hdd,
       Str cpu) {
        this.ram = ram;
        this.hdd = hdd;
        this.cpu = cpu;}
```

Server class
```
public class Server extends Computer {
    private String ram;
    private Str hdd;
    private Str cpu;
    Server(String cpu, String hdd
           Str ram) {
        this.ram = ram;
        this.hdd = hdd;
        this.cpu = cpu;
    }
    Set/get
```

Computer Factory class :-
```
public class ComputerFactory {
    public static Computer getComputer(String type, String ram, Str hdd, Str cpu) {
        if("PC".equals(type)) { return new PC(ram, hdd, cpu); }
        else if("SERVER".equals(type)) { new Server(ram, hdd, cpu); }
        return null; } }
```

③ PROTOTYPE PATTERN

Java | Creational Pattern → Prototype Design Pattern

prototype pattern says that cloning of an existing object instead of creating new one and can also be customized as per the requirement.

This pattern should be followed, if the cost of creating a new object is expensive and resource intensive.

intensive – [illegible] requiring @

advan :- It reduces need of subclassing
- → It hides complexities of creating objects
- → the clients can get new objects without knowing which type of obj it will be
- → It lets you add @ removes objects at runtime.

Usage of prototype Pattern → when the classes are instantiated at runtime.
- → when the cost of creating an object is expensive @ complicated.
- → when you want to keep the number of classes in an application minimum.
- → when the client application needs to be unaware of object creation and representation.

Ex:- an object is to be created after a costly DB operations. We can cache the object, returns its clone on next request and update the DB as and when needed thus reducing DB calls.

Ex

```
public class Employees implements Cloneable{
  private List<String> empList;
  public Employees() { empList = new ArrayList<String>(); }
  public Employees(List<String> list) {
        this.empList = list; }
                      // read all employees from DB and keep list
  public void loadData() { empList.add("RJ");
                      empList.add("SIVRaRyu");
  public Object clone() throws CloneNotSupportedException {
     List<String> temp = new ArrayList<String>();
        temp.addAll(this.getEmpList());
     return new Employees(temp);
  }
```

```
public class PrototypePatternTest {
   psvm(String args[]) throws CloneNot
                          SupportedException {
   Employees emp = new Employees();
           emp.loadData();
   // Use the clone method
   Employees empNew = (Employees)
           emps.clone();

   Employees empNew2 = (Employees)
           emps.clone();
   }
}
```

Abstract Factory Pattern :-

Java

Creational Design pattern

Abstract Factory pattern

Abstract Factory pattern provides that just define an interface @ abstract class for creating related (@dependent) objects without specifying their Concrete sub-classes.

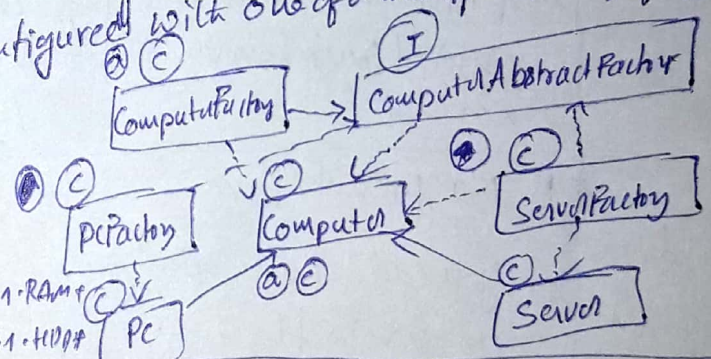
That means AbstractFactory pattern lets a class returns a factory of classes.

It is one level higher than the Factory pattern

Adv → AbstractFactory pattern isolates the client code from Concrete (implementation) class.
→ It can the exchanging of dependent objects.
→ It promotes Consistency among objects.

When to use AbstractFactory Pattern

→ When the System needs to be independent of how its objects 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.



Eg ① public abstract class Computer {
p abstract String getRAM();
get HDD();
get CPU();
@override
p String toString() { return "RAM" + this.RAM + "HDD" + this.HDD + "CPU" + this.CPU; }

② p class PC extends Computer {
prString ram, hdd, cpu;
PC(String ram, String hdd, String cpu) {
this.ram = ram;
this.hdd = hdd;
this.cpu = cpu; }
Set / get

③ p class Server extends Computer {
prString ram, hdd, cpu;
Server(String ram, String hdd, String cpu) {
this.ram = ram;
this.hdd = hdd;
this.cpu = cpu; }
Set / get

public Computer CreateComputer() {
return new Pc(ram, hdd, cpu); }

Factory class for each SubClasses

Step① Create AbstractFactory interface (or) abstract class

p interface ComputerAbstractFactory {
p Computer CreateComputer();

p class PcFactory implements ComputerAbstractFactory {
pr String ram, hdd, cpu; } Set/get
PcFactory(Str ram, Str hdd, Str cpu) { this.ram = ram

```
package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;
import com.journaldev.design.model.Server;

public class ServerFactory implements ComputerAbstractFactory {

	private String ram;
	private String hdd;
	private String cpu;

	public ServerFactory(String ram, String hdd, String cpu){
		this.ram=ram;
		this.hdd=hdd;
		this.cpu=cpu;
	}

	@Override
	public Computer createComputer() {
		return new Server(ram,hdd,cpu);
	}

}
```

```
package com.journaldev.design.abstractfactory;

import com.journaldev.design.model.Computer;

public class ComputerFactory {

	public static Computer getComputer(ComputerAbstractFactory factory){
		return factory.createComputer();
	}
}
```

TestDesignPatterns.java

```
package com.journaldev.design.test;

import com.journaldev.design.abstractfactory.PCFactory;
import com.journaldev.design.abstractfactory.ServerFactory;
import com.journaldev.design.factory.ComputerFactory;
import com.journaldev.design.model.Computer;

public class TestDesignPatterns {

	public static void main(String[] args) {
		testAbstractFactory();
	}

	private static void testAbstractFactory() {
		Computer pc =
com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new PCFactory("2
GB","500 GB","2.4 GHz"));
		Computer server =
com.journaldev.design.abstractfactory.ComputerFactory.getComputer(new
ServerFactory("16 GB","1 TB","2.9 GHz"));
		System.out.println("AbstractFactory PC Config::"+pc);
		System.out.println("AbstractFactory Server Config::"+server);
	}
}
```

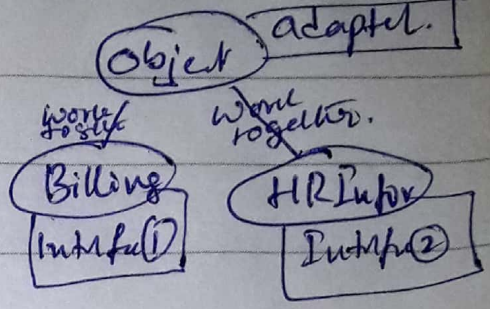
Output of the above program will be

```
AbstractFactory PC Config::RAM= 2 GB, HDD=500 GB, CPU=2.4 GHz
AbstractFactory Server Config::RAM= 16 GB, HDD=1 TB, CPU=2.9 GHz
```

Adapter Pattern – is a Structural design pattern.

→ it is used to two unrelated interfaces can work together.

→ the object that joins these unrelated interfaces is called an Adapter.



→ To implement Adapter Pattern there are two ways ① Class Adapter
② Object Adapter.

① class Adapter → this form uses java inheritance and extends the Source Interface.

② Object Adapter → this form uses Java Composition and adapter contains the Source Object.

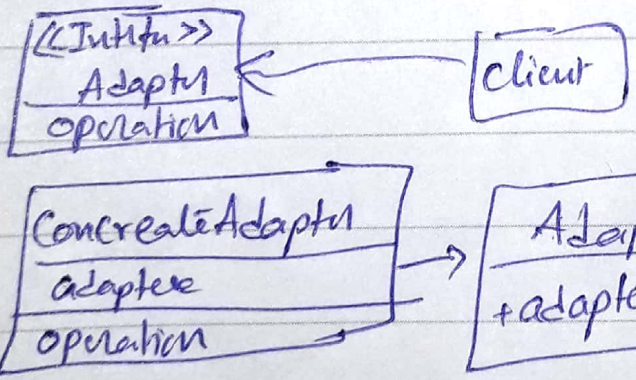
Adapter Design Pattern in JDK

① java.util.Arrays#asList()

② java.io.InputStreamReader(InputStream) returns a Reader

③ java.io.OutputStreamWriter(OutputStream) returns a writer)

Adapter class and Object Structural



When to Use

→ When an object needs to utilize an existing class with an incompatible interface.

→ When you want to create a reusable class that cooperates with which don't have compatible interfaces.

Advantage of Adapter Pattern

→ It allows two or more previously incompatible objects to interact

→ It allows reusability of existing functionality

HCL

Behavioral Design patterns →

- → Chain of Responsibility Pattern
- → Command Pattern
- → Interpreter Pattern
- → Iterator pattern
- → Mediator Pattern
- → Memento Pattern
- → Observer Pattern
- → State Pattern
- → Strategy Pattern
- → Template Pattern
- → Visitor pattern
- → Null Object

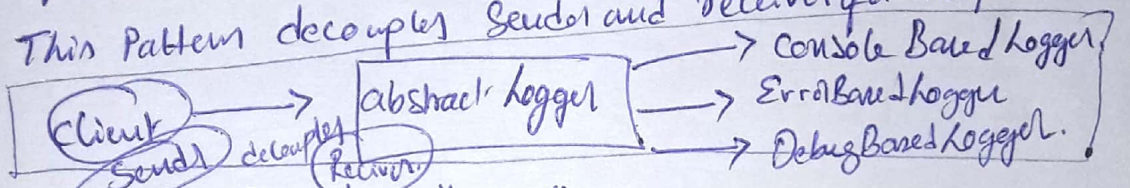
Chain of Responsibility Pattern :-

→ A series of handler objects are chained together to handle a request made by a client object.

In chain of responsibility, Sender Sends a request to a chain of objects. The request can be handled by any object in the chain.

This Pattern decouples Sender and receiver of a request based on type.

Eg



Advantages :-

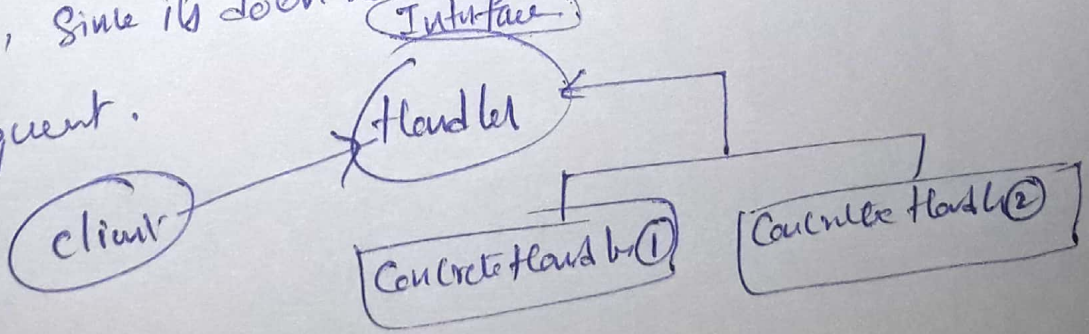
→ It reduces the coupling.

→ It adds flexibility while assigning the responsibility of object.

→ It allows a set of classes to act as one; events produced in one class in one class can be sent to other handler classes with help of composition.

Note: In this pattern, the client is decoupled from the actual handling of the request, since it does not know which class will actually handle the request.

UML



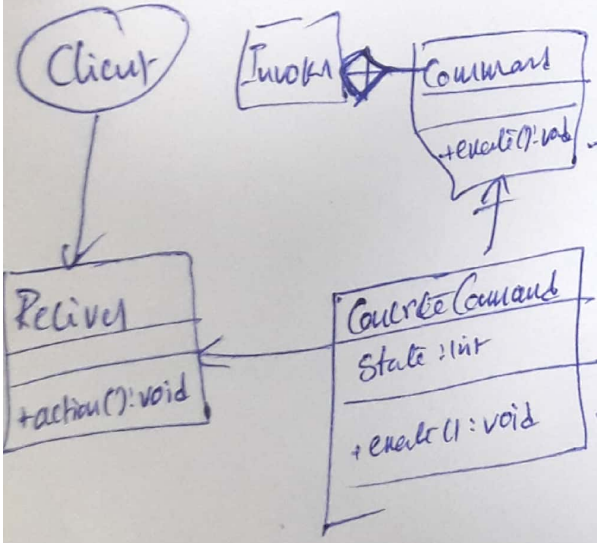
Command Pattern

" 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 passes the Command to the corresponding object which executes the Command.

Adv:-
- → It Encapsulates a request in an Object
- → allows the parameterization of clients with different requests
- → allows saving the requests in a queue.

Adv2:-
- → It Separates the objects 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 to use Command Pattern → When we need to create and execute requests at different times.
- → When we need to support rollback, logging @ transaction functionality.
- → When we need parameterize objects according to an action perform.



Client → creates a ConcreteCommand Object and sets it to reciver;

Reciver - knows how to perform the operations.

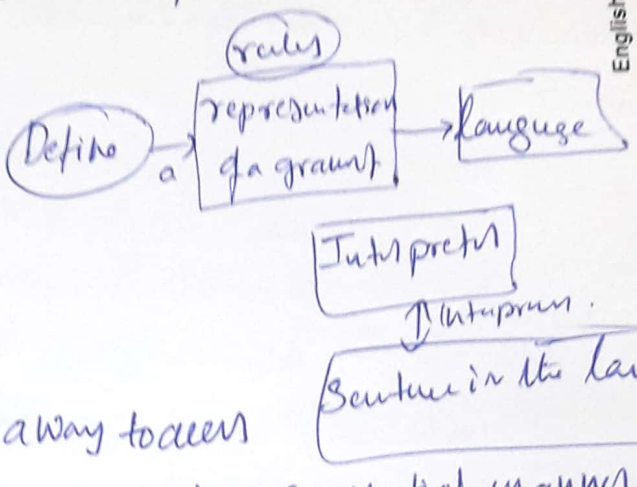
Command - declares an interface for executing an operation.

ConcreteCommand - it extends the Command interface, implementing the Execute method by invoking the corresponding operations on Reciver. it defines a linkage b/w the Reciver and the action.

Invoker → asks the Command to carry out the request;

Interpreter pattern

Define a representation of grammar of a given language, along with an interpreter that uses this representation to interpret sentences in the language.



Iterator pattern :-

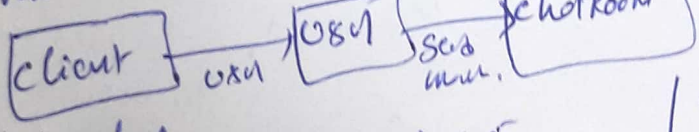
This pattern is used to get away to access the elements of a collection object in sequential manner without any need to know its underlying representation.

Mediator pattern :-

By using this patten to reduce communication complexity b/w multiple objects & classes.

→ this pattern provides a mediator class which normally handles all the communications b/w diffent classes and supports easy maintenance of the code by loose coupling.

Ex chatroom where multiple users chat its message that need to display all.



mediator to display message.

Create mediator class

ChatRoom.java

```
P C ChatRoom {
P static void showMessage (User user,
                           Str msg)
{ S.o.p (user.getUserName())
    + message.
```

```
P C User {
pr String name;
set/get
P V sendMessage (Str msg) {
(ChatRoom.showMessage
    (this, message);
}
```

client class

```
P C MediatorDemo
P S V M (Str args[]) {
User robot = new User("R");
User john = new User("John");
robot.SendMessage("Hi John")
john.SendMessage("Hi Rob");
```

Behavioral patterns :-

① Chain of Responsibility Pattern :-

A Series of Handler objects are chained together to handle a request made by a client object.

If the first handler can't handle the request, the request is forwarded to the next handler, and it is passed down the chain until the request reaches a handler that can handle the request @ the chain ends.

Adv :- This pattern decouples sender and receiver of a request based on type of request.
→ It allows a set of classes to act as one; events produced in one class can be sent to another handler classes with the help of Composition.

WHEN to Use :- → When more than one object can handle a request and the handler is unknown.
→ that must be specified in dynamic way.

Examples in JDK
java.util.logging.Logger#log()
javax.servlet.Filter#doFilter()

② Command pattern :- A Request is wrapped under an object as command and passed to invoker object.

→ Invoker object looks for the appropriate object which can handle this Command and passes the Command to the corresponding object which execute the Command.

Adv :- 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 to Use :- → When we need to support rollback, logging @ transaction functionality. → When we need to create and execute requests at different time.
→ When we need parameterize objects according to an action perform.

Examples in JDK :- java.lang.Runnable (Runnable Interface)
Swing Action (javax.swing.Action).

Template Method Pattern :- An abstract class exposes defined way(s) to execute its methods. Its subclasses can override the template(s) method implementation as per need but the invocation is to be in the same way as defined by an abstract class.

Advantages:- ① No Code duplication
② Code reuse happens with the T.M-Pattern as it uses Inheritance
③ Flexibility lets Subclasses decide how to implement steps in an algorithm.

Examples :- All non-abstract methods of java.util.AbstractList, AbstractSet, AbstractMap.
Servlets doGet and doPost methods
Struts Action class
Spring data util classes.

WHEN to Use :- → When Subclasses should be able to extend the base algorithm without altering its structure.
→ When you have several classes that do similar things with only minor differences. When you alter one of the classes, you have to change others as well.

Observer Pattern :- This pattern is used when there is one-to-many relationship b/w objects such as if one object is modified, its dependent objects are to be notified automatically. (one object changes state)

Advantages:- → It supports the principle of loose coupling b/w objects that interact with each other. → It allows sending data to other objects Effectively without any change in the Subject & Observer classes.
→ Observers can be added/removed at any point in time.

WHEN To Use ⇒ the change of a state in one object must be reflected in another object without keeping the objects tight coupled.

Examples:- MVC pattern the observer pattern is used in the model view controller (arch.)
In MVC, Pattern is used to decouple the model from the view.
View - the observer, model is the Observable object.

Mediator-pattern :- This Pattern reduce Communication Complexity b/w multiple objects @ classes. Mediator D.P. is used to provide a centralized communication medium b/w different objects in a system.

Advantages :- It limits subclassing. A mediator localizes behaviour that otherwise would be distributed among several objects.

→ Allows reusing individual components

→ Centralizes the communications b/w various components.

WHEN to use :- a behaviour that's distributed b/w several classes should be customizable without a lot of subclassing.

→ a set of objects communicate in well-defined but complex ways. the resulting interdependencies are unstructured and difficult to understand.

Examples :- Chat application, ATC (Air traffic Controle)

Example in JDK :- java.util.Timer class schedule() method.

→ Java Concurrency Executor execute() method, java.lang.reflect.Method invoke() method.

Strategy-pattern :- Strategy pattern is used when we have multiple algorithm for a specific task and client decides the actual impl. to be used at runtime. this pattern is also known as policy pattern.

Advantages :- → main benefit of using this is flexibility. the client can choose any algorithm at runtime and can easily add new algorithm without changing existing classes which use strategies.

→ this pattern based on open closed design principle.

Example :- Collections.sort() method and Comparator interface. which is a strategy interface and defines a strategy for comparing objects. Because of this pattern, we don't need to modify sort() method (closed for modification) to compare any object, at the same time we can implement Comparator interface to define new comparing strategy (open for extension).

WHEN to use :- Strategy pattern is quite useful for implementing set of related algorithms e.g. Compression algorithms, filtering strategies. Strategy design pattern allows you to create context classes, which uses strategy implementation classes for applying business rules.

~~Structural~~ Observer - Design pattern :-

Define a one-to-many dependency b/w objects so that when one obj changes state, all its dependents are notified and updated automatically.

→ Observer pattern is based upon notification, there are two kinds of object Subject and Observer. Whenever there is change on Subject's state Observer will receive notification.

Advantages :- → it supports the principle of loose coupling b/w objects that interact with each other.

→ it allows sending data to other objects effectively without any change in the Subject & Observer class.

→ Observers can be added / removed at any point in time.

WHEN to USE :- the change of a state in one object must be reflected in another object without keeping the objects tight coupled.

→ the f/w we are writing needs to be enhanced in future with new observers with minimal changes.

Examples :- Mvc pattern, is used to decouple the model from the view. View represents the observer and model is the observable object.

→ Event Mangent → Swing | java.util.EventListener in swing.
javax.servlet.http.HttpSessionBindingListner.
HttpSessionAttributeListner.

Eg News Agency.

State :- Pattern :- This pattern is used when an object change its behaviour based on its internal state.

Advantages :- to implement polymorphic behaviour is clearly visible.
→ The chances of error are less and its very easy to add more states for additional behavior.
→ Thus making our code more robust. Easily maintainable and flexible.
→ This pattern helped in avoiding if-else (or) Switch-Case conditional logic in this scenario.

WHEN TO USE :- This pattern should use the State pattern when the behavior of an object should be influenced by its state, and when complex conditionals tie object behaviour to its state.

Visitor - Pattern :- This pattern is used when we have to perform an operation on a group of similar kind of objects. With the help of visitor pattern, we can move the operation logic from the objects to another class.

Eg: Shopping cart where we can add different type of items (Elements). When we click on checkout button. It calculates the total amount to be paid. Now we can have the calculation logic in item classes (or) we can move out this logic to another class using visitor pattern.

Advantages :- The benifit of this pattern is that if the logic of operation changes then we need to make change only in the visitor implementation rather than doing it in all the item classes.
→ adding of new item to the system is easy. it will require change only in visitor interface and implementation and existing item classes will not be affected.

Disadvantage :- The drawback of this pattern is that we should know the return type of visit() methods at the time of design otherwise we will have to change the interface and all its implementations. If there are too many implementation of visitor interface. it makes it hard to extend.

Interpreter Pattern :- provides a way to evaluate language grammar or expression. this pattern involves implementing an expression interface which tells to interpret a particular context.
This pattern used in SQL parsing, Symbol processing engine etc.

Advantages :- It's easy to change and extends the grammar.

WHEN to use :- 1. the Interpreter pattern is used excessively in defining grammars to tokenize input and store it.
-> the Interpreter pattern can be used to add functionality to the Composite pattern.

Example :- Rule validator. | Java Compiler | Google Translator |

Iterator Design Pattern :-

this pattern provides a standard way to traverse through a group of objects. this pattern is widely used in java Collection Framework.
"provides a way to access the elements of an aggregate object without exposing its underlying representation".

Example :-

Memento Pattern this pattern is used when we want to save the state of an object so that we can restore later on (restoring the object into initial state when needed).

Example :- it can be used in Database Transactions. rollback uses the memento pattern.

Disadvantage :- uses a lot of memory.