**DESIGN PATTERN**

Design pattern are typical solutions to common problems in software design.

Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.

Attempt made by a group of four people, famously called Gang of Four, to come up with set of common problems and solutions for them, in the given context.

**Advantages**

1. To provide the standard terminology that everybody understands.

2. Not to repeat the same mistakes over and over again.

**Creational Pattern (ABFPS)**

- Creational Patterns deals with creation of objects.

- Creational Pattern increases the flexibility and reuse of existing code.

**Structural Pattern (ABCDFFP)**

- Structural Pattern deals with composition of objects.

- It deals with question such as

- What does a class contain?

- What is the relationship of a class with other classes? Is it inheritance or Composition?

**Behavioural Pattern (SSMMCCIOTV)**

- Behavioural Pattern focuses more on behaviour of objects or more precisely interaction between objects.

- It takes care of effective communication and the assignment of responsibilities between object.

**Design Principles**

**1. Program to an interface, not an implementation.**

Client refers to an interface and are independent of an implementation. That means, an implementation can vary independently from (without having to change) existing clients.

Client separate interface and implementation.

**2. Favour Object Composition over class inheritance**

use class inheritance to change the behaviour statically at compile time.

use object composition to change the behaviour dynamically at run time.

|  |  |  |
| --- | --- | --- |
| Abstract Factory | Adapter | Strategy |
| Builder | Bridge | State |
| Factory | Composite | Mediator |
| Prototype | Decorator | Memento |
| Singleton | Facade | Chain of Responsibility |
|  | Flyweight | Command Pattern |
|  | Proxy | Iterator |
|  |  | Observer |
|  |  | Template |
|  |  | Visitor |

**proxy pattern**

A proxy is an object that represents another object.

A Proxy hides the complexity involved in communicating with real object.

1. A proxy may avoid instantiation of an object until the object is needed.

2. This can be useful if the object requires a lot of time or resource to create.

- Proxy controlling the way an object is accessed.

- proxy providing the additional functionality when accessing an object.

Employee emp = session.load(Student.class, id2);

//Here load method is lazy loading and return the employee proxy object

emp.getName ();

//employee object is initialized after the method called.

**public** **interface** RealObject {

**public** **void** doSomething();

}

**public** **class** RealObjectImpl **implements** RealObject{

**public** **void** doSomething() {

System.***out***.println("Performing work in real object");

}

}

**public** **class** RealObjectProxy **extends** RealObjectImpl{

**public** **void** doSomething() {

System.***out***.println("Delegating real object");

**super**.doSomething();

}

}

**public** **class** Client {

**public** **static** **void** main(String[] args) {

RealObjectProxy rop = **new** RealObjectProxy();

rop.doSomething();

}

}

**Output:**

Delegating real object

Performing work in real object

**Decorator**

- Decorator is designed to let you add responsibilities to objects without subclassing

- Decorator pattern attaches additional responsibilities to an object dynamically. It acts as wrapper to existing classes

- To add responsibilities to individual objects dynamically and transparently, that is, without affecting other objects.

- Decorator provides an enhanced interface. It enhances an object responsibility.

- Adding responsibilities to an object dynamically.

- Extending the functionality of an object dynamically.

- **Java IO Streams are well known example of Decorator Pattern.**

- This pattern creates decorator class which wraps original class and provides the additional functionality keeping class method signature intact

FileInputStream fis = new FileInputStream("file1.txt)

BufferedInputStream bis = new BufferedInputStream(fis);

int i;

while((i=bin.read())!=-1){

system.out.print((char)i);

}

bis.close();

fis.close();

**public** **interface** Component {

**public** **void** add();

}

**public** **class** TextField **implements** Component {

**public** **void** add() {

System.***out***.println("Text Field is added");

}

}

**public** **class** BorderDecorator **implements** Component {

Component comp;

**public** BorderDecorator(Component comp) {

**this**.comp = comp;

}

**public** **void** add() {

comp.add();

System.***out***.println("Border Style added");

}

}

**public** **class** DecoratorPattern {

**public** **static** **void** main(String[] args) {

Component add = **new** BorderDecorator(**new** TextField());

add.add();

}

}

**Output:**

Text Field is added

Border Style added

**Facade**

- Facade is a single class that represents an entire sub system.

- Facade design pattern is used to define a simplified interface to a more complex sub system.

- The main benefit with the facade pattern is that we can combine very complex method calls and code blocks into single method that performs a complex and recurring task.

-Facade pattern can be used to make software library easier to use and understand, since the facade has convenient methods for common task.

- Facade hides the complexities of the system and provides an interface to the client using which the client can access the system.

- Facade provides a simple interface to a complex system.

- Facade provides a unified interface for already existing objects in a sub system.

- Facade making complex subsystem easier to use.

- Facade defines a higher level interface that makes the subsystem easier to use.

- Loosely coupling between client and sub system

Working through facade. Client only refers to and know about(depend on) the simple facade interface, which makes client easier to implement, change, test and reuse.

- Decouples clients from a subsystem

Clients are decoupled from a subsystem by working through a Facade object.

Clients only refer to and know about the simple facade interface and are independentof the complex sub system(loosely coupling). this makes client easier to implement, change, test and reuse.

- Decouples subsystem

When layering a complex system, facade can define single entry point for each subsystem. Subsystem collaborate with each other through facades, which reduces and simplifies dependency between sub sytem.

HashMap uses put and get method. It internally uses hash alogorithm to store and retrieve the value.

HashMap hm = new HashMap();

hm.put("1","guru");

**Adapter Pattern**

Adapter pattern provides an alternative interface for an (already existing) class or object.

- Adapter pattern works as bridge between two incompatiable interfaces.

- This type of design pattern comes under structural pattern as this pattern combines the capability of two indepedent interfaces.

- This pattern involves a single class which is responsible to join the functionalities of indepedent or incompatiable interfaces.

**public** **interface** MediaPlayer {

**public** **void** play(String fileName);

}

**public** **class** MP4Player **implements** MediaPlayer {

**public** **void** play(String fileName) {

System.***out***.println("Playing MP4 Player - "+fileName);

}

}

**public** **class** VLCPlayer **implements** MediaPlayer {

**public** **void** play(String fileName) {

run(fileName);

}

**public** **void** run(String fileName) {

System.***out***.println("Playing VLCPlayer file -"+fileName);

}

}

**public** **interface** MediaPlayerAdapter {

**public** **void** play(String audioType,String fileName);

}

**public** **class** MediaAdapter **implements** MediaPlayerAdapter {

MediaPlayer mediaPlayer;

**public** **void** play(String audioType, String fileName) {

**if** (audioType.equalsIgnoreCase("VLC")) {

mediaPlayer = **new** VLCPlayer();

mediaPlayer.play(fileName);

} **else** **if** (audioType.equalsIgnoreCase("MP4") || audioType.equalsIgnoreCase("MP3")) {

mediaPlayer = **new** MP4Player();

mediaPlayer.play(fileName);

}

}

**public** **static** **void** main(String args[]) {

MediaAdapter mAdapter = **new** MediaAdapter();

mAdapter.play("VLC", "farfaraway.vlc");

mAdapter.play("MP4", "mind me.mp3");

mAdapter.play("MP3", "beyond the horizon.mp4");

}

}

**Output:**

Playing VLCPlayer file -farfaraway.vlc

Playing MP4 Player - mind me.mp3

Playing MP4 Player - beyond the horizon.mp4

**FlyWeight Pattern**

- A fine grained instance used for efficient sharing.

- Creation of objects takes a lot of time and involves multiple instances.

Each instance of an object occupies a lot of memory.

- Some objects might be used several times across the same application with the same values. In these scenarios, you might not want to create a new instance every time it is needed. (i.e) ConnectionPooling, SessionFactory and String.

- Flyweight Pattern is primarily used to reduce the number of objecs created and to decrease the memory footprint and increase performance.

- Flyweight Pattern tries to reuse already existing similar kind objects by storing them and creates new object when no matching object is found.

**public** **interface** Flyweight {

**public** **void** doMath(**int** a, **int** b);

}

**public** **class** FlyweightAdd **implements** Flyweight{

**public** **void** doMath(**int** a, **int** b) {

System.***out***.println("FlyweightAdd a+b:"+(a+b));

}

}

**public** **class** FlyweightMultiply **implements** Flyweight{

**public** **void** doMath(**int** a, **int** b) {

System.***out***.println("Adding"+a+" a\*b ="+a\*b);

}

}

**import** java.util.HashMap;

**import** java.util.Map;

**public** **class** FlyweightFactory {

**private** **static** FlyweightFactory *flyweightFactory*;

Map<String,Flyweight> flyweightPool;

**private** FlyweightFactory() {

flyweightPool = **new** HashMap<>();

}

**public** **static** FlyweightFactory getInstance() {

**if**(*flyweightFactory*==**null**) {

*flyweightFactory* = **new** FlyweightFactory();

}

**return** *flyweightFactory*;

}

**public** Flyweight getFlyweight(String key) {

**if**(flyweightPool.containsKey(key))

**return** flyweightPool.get(key);

**else** {

Flyweight flyweight;

**if**("add".equals(key))

flyweight = **new** FlyweightAdd();

**else**

flyweight = **new** FlyweightMultiply();

flyweightPool.put(key, flyweight);

**return** flyweight;

}

}

}

**public** **class** FlyWeightDemo {

**public** **static** **void** main(String[] args) {

FlyweightFactory fwf = FlyweightFactory.*getInstance*();

**for**(**int** i=0;i<5;i++) {

Flyweight fw = fwf.getFlyweight("add");

fw.doMath(i, i);

}

**for**(**int** i=0;i<5;i++) {

Flyweight fw = fwf.getFlyweight("multiply");

fw.doMath(i, i);

}

}

}

**Output:**

FlyweightAdd a+b:0

FlyweightAdd a+b:2

FlyweightAdd a+b:4

FlyweightAdd a+b:6

FlyweightAdd a+b:8

Adding0 a\*b =0

Adding1 a\*b =1

Adding2 a\*b =4

Adding3 a\*b =9

Adding4 a\*b =16

**Bridge Design pattern**

- Decouple an abstraction from its implementation so that two can vary independently.

- Bridge also known as handle/body idiom. This is design mechanism that encapsulates an implementation class inside of an interface class.

The former is the body and the latter is handle.

The handle is viewed by the user as the actual class, but the work is done in the body.

**public** **interface** Color {

**public** **void** applyColor();

}

**public** **class** PinkColor **implements** Color {

**public** **void** applyColor() {

System.***out***.println("Applying Pink Color");

}

}

**public** **class** RoseColor **implements** Color {

**public** **void** applyColor() {

System.***out***.println("Applying Rose Color");

}

}

**public** **abstract** **class** Shape {

**protected** Color color;

**public** Shape(Color c) {

**this**.color = c;

}

**abstract** **void** applyColor();

}

**public** **class** Rectangle **extends** Shape{

**public** Rectangle(Color c) {

**super**(c);

}

**public** **void** applyColor() {

System.***out***.println("Applying the Color");

color.applyColor();

}

}

**public** **class** BridgePatternTest {

**public** **static** **void** main(String args[]) {

//Color col = new RoseColor();

//col.applyColor();

Shape shape = **new** Rectangle(**new** PinkColor());

shape.applyColor();

}

}

**Output:**

Applying the Color

Applying Pink Color

**Composite Design Pattern**

- It compose objects into tree structures to represent part-whole hierarchies.

- Composite lets client treat individual objects and composition of objects uniformly.

- The Composite design pattern allows to perform operations on the composites and leaves that make up tree structure via common interface.

- Composite pattern compose objects in terms of tree structure to represent part as well as whole hierarchy.

- This type of design pattern comes under structural pattern as this pattern creates a tree structure of a group of objects.

- Composite pattern is used where we need to treat a group of objects in similarly way as a single object.

- This pattern creates a class that contains a group of its own objects. This class provides ways to modify its group of same objects.

We have an class Employee which act as composite pattern actor class. It adds departement level hierarchy and print the employees.

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Employee {

String name;

String dept;

List<Employee> subordinates;

**public** Employee(String name, String dept) {

**this**.name= name;

**this**.dept=dept;

subordinates = **new** ArrayList<>();

}

**public** **void** add(Employee e) {

subordinates.add(e);

}

**public** **void** remove(Employee e) {

subordinates.remove(e);

}

**public** List<Employee> getSubordinates(){

**return** subordinates;

}

@Override

**public** String toString() {

**return** "Employee [name=" + name + ", dept=" + dept + "]";

}

}

**public** **class** CompositePatternDemo {

**public** **static** **void** main(String[] args) {

Employee ceo = **new** Employee("a","CEO");

Employee headSales = **new** Employee("Robert","HeadSales");

Employee headMarketing = **new** Employee("Bob","HeadMarketing");

Employee clerk1 = **new** Employee("Laura","clerk");

Employee clerk2 = **new** Employee("James","clerk");

Employee salesExecutive1 = **new** Employee("sam","salesExecutive");

Employee salesExecutive2 = **new** Employee("saraha","salesExecutive");

ceo.add(headSales);

ceo.add(headMarketing);

headSales.add(salesExecutive1);

headSales.add(salesExecutive2);

headMarketing.add(clerk1);

headMarketing.add(clerk2);

headMarketing = **new** Employee("test","HeadMarketing");

System.***out***.println("CEO"+ceo);

System.***out***.println("HeadSales"+headSales);

System.***out***.println("HeadMarketing"+headMarketing);

}

}

**Output:**

CEOEmployee [name=a, dept=CEO]

HeadSalesEmployee [name=Robert, dept=HeadSales]

HeadMarketingEmployee [name=test, dept=HeadMarketing]

**Strategy pattern**

- Strategy allows us to dynamically swap out algorithm (Application logic) at run time.

- Strategy pattern is using different algorithms.

- Selecting and changing which algorithm to use dynamically.

- Adding new algorithm and changing the behaviour of existing ones independently.

- In Strategy pattern, a class behaviour or its algorithm can be changed at runtime. This type of design pattern comes under behaviour pattern.

- In Strategy pattern, we create objects which represent various strategies and a context object whose behaviour varies as per its strategy object.

- The Strategy object changes the executing algorithm of the context object.

- You can alter part of objects behaviours by supplying it with different strategies that corresponds to behaviour. Strategy works based on the object level, letting you switch behaviours at run time.

- Client can choose any algorithm at runtime without changing the context class, which uses strategy object.

context.setStrategy(new RoadStrategy());

strategy.buildRoute("chennai","bangalore");

context.sestStrategy(new WalkingStrategy());

strategy.buildRoute("chennai","bangalore");

**public** **interface** RouteStrategy {

**public** **void** buildRoute(String source, String destination);

}

**public** **class** RoadStrategy **implements** RouteStrategy{

**public** **void** buildRoute(String source, String destination) {

System.***out***.println("Road Route Shows between "+source+" and "+destination+".");

}

}

**public** **class** WalkingStrategy **implements** RouteStrategy{

**public** **void** buildRoute(String source, String destination) {

System.***out***.println("Cycle Route Shows between "+source+" and "+destination+".");

}

}

**public** **class** PublicTransportStrategy **implements** RouteStrategy{

**public** **void** buildRoute(String source, String destination) {

System.***out***.println("Public Transport Route shows between "+source+" and "+destination+".");

}

}

**public** **class** Navigator {

RouteStrategy routeStrategy;

**public** **void** setRouteStrategy(RouteStrategy routeStrategy) {

**this**.routeStrategy = routeStrategy;

}

**public** **void** buildRoute(String source, String destination) {

routeStrategy.buildRoute(source, destination);

}

}

**public** **class** StrategyPattern {

**public** **static** **void** main(String[] args) {

Navigator strategy = **new** Navigator();

strategy.setRouteStrategy(**new** RoadStrategy());

strategy.buildRoute("Chennai", "Bangalore");

}

}

**Output:**

Road Route Shows between Chennai and Bangalore.

**State pattern**

- Changing the behaviour of an object when its internal state changes.

- Adding new states and changing the behaviour of existing ones independently.

- State pattern is used to alter an object behaviour when it states changes.

- Class behaviour changes based on its state. This type of design pattern comes under behaviour pattern.

- In state pattern, we create objects which represent various states and context object whose behaviour varies as its state object changes.

- State pattern allows an object to behave differently at different state.

stateContext.setState(new Silent());

stateContext.alert();

stateContext.setState(new Vibration());

stateContext.alert();

**public** **interface** MobileAlertState {

**public** **void** alert();

}

**public** **class** Silent **implements** MobileAlertState {

**public** **void** alert() {

System.***out***.println("Silent Alert");

}

}

**public** **class** Vibration **implements** MobileAlertState {

**public** **void** alert() {

System.***out***.println("Vibration Alert");

}

}

**public** **class** AlertStateContext {

**private** MobileAlertState currentState;

**public** AlertStateContext() {

currentState = **new** Vibration();

}

**public** **void** setState(MobileAlertState state) {

currentState = state;

}

//call the alert based on the setState

**public** **void** alert() {

currentState.alert();

}

}

**public** **class** StatePattern {

**public** **static** **void** main(String[] args) {

AlertStateContext stateContext = **new** AlertStateContext();

stateContext.setState(**new** Silent());

stateContext.alert();

stateContext.alert();

stateContext.setState(**new** Vibration());

stateContext.alert();

stateContext.alert();

}

}

**Output:**

Silent Alert

Silent Alert

Vibration Alert

Vibration Alert

**Memento Pattern**

- Storing and restoring an objects internal state without violating encapsulation.

- Memento are used to restore the state of an object to a previous state.

- Memento pattern falls under behavioural pattern

- Memento pattern contains state of an object to be restored.

- Originator creates and stores states in Memento objects.

- Caretaker object is responsible to restore object state from memento.

//originator - object whose state we want to save

**public** **class** Orginator {

String personName;

**int** dayNumber;

**int** weight;

**public** Orginator(String personName, **int** dayNumber, **int** weight) {

**this**.personName = personName;

**this**.dayNumber = dayNumber;

**this**.weight = weight;

}

@Override

**public** String toString() {

**return** "Orginator [personName=" + personName + ", dayNumber=" + dayNumber + ", weight=" + weight + "]";

}

**public** **void** setDayNumberAndWeight(**int** dayNumber, **int** weight) {

**this**.dayNumber = dayNumber;

**this**.weight = weight;

}

// memento - object that stores the saved state of the originator

**private** **class** Memento {

String mementoPersonName;

**int** mementoDayNumber;

**int** mementoWeight;

**public** Memento(String mementoPersonName, **int** mementoDayNumber, **int** mementoWeight) {

**this**.mementoPersonName = mementoPersonName;

**this**.mementoDayNumber = mementoDayNumber;

**this**.mementoWeight = mementoWeight;

}

}

**public** Memento save() {

**return** **new** Memento(personName,dayNumber,weight);

}

**public** **void** restore(Object objMemento) {

Memento memento =(Memento) objMemento;

personName = memento.mementoPersonName;

dayNumber = memento.mementoDayNumber;

weight = memento.mementoWeight;

}

}

**public** **class** CareTaker {

Object objMemento;

**public** **void** saveState(Orginator orginator) {

objMemento = orginator.save();

}

**public** **void** restoreState(Orginator orginator) {

orginator.restore(objMemento);

}

}

**public** **class** TestMemento {

**public** **static** **void** main(String[] args) {

CareTaker careTaker = **new** CareTaker();

Orginator orginator = **new** Orginator("guru",1,74);

System.***out***.println(orginator);

orginator.setDayNumberAndWeight(2, 73);

System.***out***.println(orginator);

System.***out***.println("Saving the state");

careTaker.saveState(orginator);

System.***out***.println(orginator);

orginator.setDayNumberAndWeight(3, 72);

System.***out***.println(orginator);

orginator.setDayNumberAndWeight(4, 71);

System.***out***.println(orginator);

System.***out***.println("Restoring the state");

careTaker.restoreState(orginator);

System.***out***.println(orginator);

}

}

**Output:**

Orginator [personName=guru, dayNumber=1, weight=74]

Orginator [personName=guru, dayNumber=2, weight=73]

Saving the state

Orginator [personName=guru, dayNumber=2, weight=73]

Orginator [personName=guru, dayNumber=3, weight=72]

Orginator [personName=guru, dayNumber=4, weight=71]

Restoring the state

Orginator [personName=guru, dayNumber=2, weight=73]

**Template Pattern**

- Defining behaviour so that subclasses can change certain parts of the behaviour without changing the behaviour structure.

- It defines the skeleton of algorithm in the super class but let subclasses override the specific set of algorithms without changing its structure.

- In Template pattern, an abstract class defines ways/templates to execute its methods. Its subclass can override the method implementation as per need.

- Examples - Non abstract methods of InputStream, OuputStream, Reader, Writer from Java I/O.

**public** **abstract** **class** Game {

**abstract** **void** initialize();

**abstract** **void** startGame();

**abstract** **void** endGame();

**public** **final** **void** play() {

initialize();

startGame();

endGame();

}

}

**public** **class** Cricket **extends** Game {

**public** **void** initialize() {

System.***out***.println("Cricket - Initialize");

}

**public** **void** startGame() {

System.***out***.println("Cricket - StartGame");

}

**public** **void** endGame() {

System.***out***.println("Cricket - endGame");

}

}

**public** **class** Football **extends** Game {

**public** **void** initialize() {

System.***out***.println("Football - Initialize");

}

**public** **void** startGame() {

System.***out***.println("Football - StartGame");

}

**public** **void** endGame() {

System.***out***.println("Football - endGame");

}

}

**public** **class** TemplatePattern {

**public** **static** **void** main(String[] args) {

Game game = **new** Cricket();

game.play();

System.***out***.println();

game = **new** Football();

game.play();

}

}

**Output:**

Cricket - Initialize

Cricket - StartGame

Cricket - endGame

Football - Initialize

Football - StartGame

Football - endGame

**Chain Of Responsibility**

- Chain Of Responsibility pattern decouples sender and receiver of a request based on type of request.

- Chain of Responsibility pattern creates a chain of receiver objects for a request.

- In this pattern, normally each receiver contains the reference to another receiver. If an object cannot handle the request, it passes the same to the next receiver and so on.

- Chain of Responsibility passes a sender request along a chain of potential receivers.

- One such classic example is ServletFilter in Java that allows multiple filter to process an HttpRequest, Though in that case, each filter invokes the chain instead of next filter.

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException{

chain.doFilter(request, response);

}

**Where and When Chain of Responsibility pattern is applicable** :

- When you want to decouple a request’s sender and receiver

- Multiple objects, determined at runtime, are candidates to handle a request

- When you don’t want to specify handlers explicitly in your code

- When you want to issue a request to one of several objects without specifying the receiver explicitly.

- This pattern is recommended when multiple objects can handle a request and the handler doesn’t have to be a specific object. Also, the handler is determined at runtime.

**Drawbacks of the chain of responsibility pattern**:

- Execution of the request isn’t guaranteed; it may fall off the end of the chain if no object handles it.

- It can be hard to observe and debug the runtime characteristics.

Chain Of Responsibility, command, mediator and observer pattern address how you can decouple sender and receiver, but with different trade off.

**public** **abstract** **class** AbstractLogger {

**public** **static** **int** *INFO*=1;

**public** **static** **int** *DEBUG*=2;

**public** **static** **int** *ERROR*=3;

**protected** **int** level;

**protected** AbstractLogger nextLogger;

**public** **void** setNextLogger(AbstractLogger nextLogger) {

**this**.nextLogger=nextLogger;

}

**public** **void** logMessage(**int** level, String message) {

**if**(**this**.level<=level) {

write(message);

}

}

**abstract** **protected** **void** write(String message);

}

**public** **class** ErrorLogger **extends** AbstractLogger {

**public** ErrorLogger(**int** level) {

**this**.level=level;

}

**protected** **void** write(String message) {

System.***out***.println("Error Console::Logger"+message);

}

}

**public** **class** FileLogger **extends** AbstractLogger {

**public** FileLogger(**int** level) {

**this**.level=level;

}

**protected** **void** write(String message) {

System.***out***.println("File::Logger"+message);

}

}

**public** **class** ConsoleLogger **extends** AbstractLogger {

**public** ConsoleLogger(**int** level) {

**this**.level=level;

}

**protected** **void** write(String message) {

System.***out***.println("Standard Console::Logger"+message);

}

}

**public** **class** ChainOfResponsibility {

**public** **static** AbstractLogger getChainOfLoggers() {

AbstractLogger errorLogger = **new** ErrorLogger(AbstractLogger.*ERROR*);

AbstractLogger fileLogger = **new** FileLogger(AbstractLogger.*INFO*);

AbstractLogger consoleLogger = **new** ConsoleLogger(AbstractLogger.*DEBUG*);

errorLogger.setNextLogger(fileLogger);

fileLogger.setNextLogger(consoleLogger);

**return** errorLogger;

}

**public** **static** **void** main(String[] args) {

AbstractLogger loggerChain = *getChainOfLoggers*();

loggerChain.logMessage(AbstractLogger.*INFO*, "This is the information - FILE");

loggerChain.logMessage(AbstractLogger.*ERROR*, "This is error information - ERROR");

loggerChain.logMessage(AbstractLogger.*DEBUG*, "this is debug information - CONSOLE");

}

}

Output:

Error Console::LoggerThis is error information - ERROR

**Command Pattern**

- It avoids tight coupling the sender of a request to its receiver.

- Configuring the sender of a request with a request.

- Command pattern encapsulates a command request as an object.

- There are four terms always associated with command pattern are command, receiver, invoker and client.

- A request is wrapper under an object as command and passed to invoker object. Invoker object looks for appropriate object which can handle his command and passes the command to the corresponding object which executes it.

**public** **interface** Receiver {

**public** **void** execute();

}

**public** **class** AC **implements** Receiver{

**public** **void** execute() {

System.***out***.println("AC Execute");

}

}

**public** **class** Fan **implements** Receiver{

**public** **void** execute() {

System.***out***.println("Fan Execute");

}

}

**public** **class** Command **implements** Runnable {

**private** Receiver receiver;

**public** Command(Receiver receiver) {

**this**.receiver=receiver;

}

**public** **void** run() {

receiver.execute();

}

}

**public** **class** ThreadCommand {

//ThreadCommand is Client

//Runnable interface is Command

//MyRunnable is ConcreteCommmand

//Thread is Invoker with start() method calling ConcreteCommand implementaiton ( which calls run() method)

**public** **static** **void** main(String[] args) {

Receiver receiver = **new** AC();

Thread t = **new** Thread(**new** Command(receiver));

t.start();

}

}

**Output:**

AC Execute

**Mediator Pattern**

- Avoiding tight coupling between set of interacting objects.

- Changing the interaction behavior independently.

- The mediator pattern centralizes communication between objects into a mediator object.

- Mediator pattern is used to reduce communication complexity between multiple objects or classes.

- This pattern provides a mediator class which normally handles all the communication between different classes and supports easy maintenance of the code by loosely coupling.

- Mediator eliminates direct communication between senders and receivers, forcing them to communicate indirectly via mediator object.

**public** **class** User {

**private** String name;

**public** User(String name) {

**super**();

**this**.name = name;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **void** sendMessage(String message) {

ChatRoom.*showMessage*(**this**, message);

}

}

**import** java.util.Date;

**public** **class** ChatRoom {

**public** **static** **void** showMessage(User user, String message) {

System.***out***.println(**new** Date().toString()+"["+user.getName()+"] :"+message);

}

}

**public** **class** MediatorPattern {

**public** **static** **void** main(String[] args) {

User usr1 = **new** User("Kumaresh");

User usr2 = **new** User("Nikhil");

usr1.sendMessage("Hello");

usr2.sendMessage("Hai");

}

}

**Output:**

Sat Mar 14 19:20:39 IST 2020[Kumaresh] :Hello

Sat Mar 14 19:20:39 IST 2020[Nikhil] :Hai

**Observer Pattern**

- Observer Pattern is fall under behavioural pattern.

- Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically.

- Observer pattern is a way of notifying the change to number of classes.

Example : javax.servlet.http.HttpSessionBindingListener.

**public** **interface** Observer {

**public** **void** update(String news);

}

**public** **class** Client **implements** Observer{

String news;

**public** **void** setNews(String news) {

**this**.news = news;

}

**public** **void** update(String news) {

**this**.setNews(news);

System.***out***.println("updated to NewsChannel -"+news);

}

}

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Subject {

**private** String news;

**private** List<Observer> channels = **new** ArrayList<>();

**public** **void** addObserver(Observer channel) {

channels.add(channel);

}

**public** **void** removeObserver(Observer channel) {

channels.remove(channel);

}

**public** **void** setNews(String news) {

**this**.news = news;

**for**(Observer channel:**this**.channels) {

channel.update(news);

}

}

}

**public** **class** ObservablePattern {

**public** **static** **void** main(String[] args) {

Subject observable = **new** Subject();

Client observer = **new** Client();

observable.addObserver(observer);

observable.setNews("News");

}

}

**Output:**

updated to NewsChannel -News

**Factory Method Pattern**

- Factory pattern is one that returns an instance of one of several possible classes depending on the data provided it.

- Factory method defines a separate factory method for creating an object.

- Factory pattern provides one of the best ways to create an object.

- Factory pattern revolves around the creation of object at run time.

- Factory Pattern deals with creation of objects delegated to a separate factory class.

- Factory pattern, we create objects without exposing the creation logic to the client and refers to newly created object using a common interface.

- Factory Pattern can be used where we need to create an object of any one of sub classes depending on the data provided.

- Factory pattern removes the instantiation of actual implementation of the classes from client code.

Encapsulated Object Creation

Object creation is defined(encapsulated) in a separate operation (factory method).

This make it easy to change the instantiation independently from the class( by adding new subclasses).

Deferred Object Creation

Creators defers instantiation to subclasses by calling an abstract factory method.

Factory method uses subclasses to specify which class to instantiate statically at compile time.

**Examples**

**Factory method design pattern is used by the Spring ApplicationContext to create spring beans in container i**n the correct order as per given configuration.

ApplicationContext context = new ClassPathXmlApplicationContext(“Beans.xml”);

BankAccountService bankAccountService = context.getBean(“bankAccountService”);

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

BankAccountService bankAccountService = context.getBean(BankAccountService.class);

**public** **interface** Logger {

**public** **void** error(String message);

**public** **void** debug(String message);

}

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.PrintStream;

**public** **class** FileLogger **implements** Logger {

PrintStream out;

**public** FileLogger() {}

**public** FileLogger(String file) **throws** IOException {

out = **new** PrintStream(**new** FileOutputStream(file),**true**);

}

**public** **void** error(String message) {

out.println("File ERROR:"+message);

}

**public** **void** debug(String message) {

out.println("File DEBUG:"+message);

}

}

**public** **class** ConsoleLogger **implements** Logger {

**public** **void** error(String message) {

System.***err***.println("ERROR:"+message);

}

**public** **void** debug(String message) {

System.***err***.println("DEBUG:"+message);

}

}

**import** java.io.IOException;

**public** **class** LoggerFactory {

**public** **static** Logger getLogger(String loggerType) **throws** IOException {

**if** (loggerType.equals("FILE")) {

Logger logger = **new** FileLogger("Logger.file");

**return** logger;

}

**if** (loggerType.equals("CONSOLE")) {

Logger logger = **new** ConsoleLogger();

**return** logger;

}

**return** **null**;

}

}

**import** java.io.IOException;

**public** **class** TestFactoryExapmple {

**public** **static** **void** main(String[] args) **throws** IOException {

//LoggerFactory lf = new LoggerFactory();

Logger logf = LoggerFactory.*getLogger*("FILE");

Logger logc = LoggerFactory.*getLogger*("CONSOLE");

logf.debug("File Debug");

logf.error("File Error");

logc.debug("Console Debug");

logc.error("Console Error");

}

}

**Output:**

DEBUG:Console Debug

ERROR:Console Error

**Abstract Factory Pattern**

- Abstract Factory provides an interface for creating families of a related or dependent object without specifying their concrete classes.

- Abstract Factory lets a class return a factory of classes, so this is the reason that AbstractFactory Pattern is one level higher than the factory pattern.

- Abstract Factory defines a separate factory object for creating object.

-

**Builder pattern**

- When the object contains lot of attributes, Builder pattern solves the issue by providing a way to build the object step-by-step and provide a method that will actually return the final object.

- Builder pattern provides clear separation between the construction and representation of an object.

- Builder pattern supports internal representation of objects.

- We want that a class can create different representation of a complex object in the same construction process.

- A class delegates object creation to a builder object instead of instantiating concrete classes directly.

- Builder pattern comes handy when a large complex object is to be built specially when it can be built in multiple combinations.

**public** **class** Computer {

//required parameters

**private** String HDD;

**private** String RAM;

//optional parameters

**private** **boolean** isGraphicsCardEnabled;

**private** **boolean** isBluetoothEnabled;

**public** String getHDD() {

**return** HDD;

}

**public** **void** setHDD(String hDD) {

HDD = hDD;

}

**public** String getRAM() {

**return** RAM;

}

**public** **void** setRAM(String rAM) {

RAM = rAM;

}

**public** **boolean** isGraphicsCardEnabled() {

**return** isGraphicsCardEnabled;

}

**public** **void** setGraphicsCardEnabled(**boolean** isGraphicsCardEnabled) {

**this**.isGraphicsCardEnabled = isGraphicsCardEnabled;

}

**public** **boolean** isBluetoothEnabled() {

**return** isBluetoothEnabled;

}

**public** **void** setBluetoothEnabled(**boolean** isBluetoothEnabled) {

**this**.isBluetoothEnabled = isBluetoothEnabled;

}

**private** Computer(ComputerBuilder builder) {

**this**.HDD=builder.HDD;

**this**.RAM=builder.RAM;

**this**.isGraphicsCardEnabled=builder.isGraphicsCardEnabled;

**this**.isBluetoothEnabled=builder.isBluetoothEnabled;

}

//Builder Class

**public** **static** **class** ComputerBuilder{

// required parameters

**private** String HDD;

**private** String RAM;

// optional parameters

**private** **boolean** isGraphicsCardEnabled;

**private** **boolean** isBluetoothEnabled;

**public** ComputerBuilder(String hDD, String rAM) {

HDD = hDD;

RAM = rAM;

}

**public** **void** setGraphicsCardEnabled(**boolean** isGraphicsCardEnabled) {

**this**.isGraphicsCardEnabled = isGraphicsCardEnabled;

}

**public** **void** setBluetoothEnabled(**boolean** isBluetoothEnabled) {

**this**.isBluetoothEnabled = isBluetoothEnabled;

}

**public** Computer build() {

**return** **new** Computer(**this**);

}

}//end static Computer Builder

}

**public** **class** TestBuilderPattern {

**public** **static** **void** main(String[] args) {

Computer comp = **new** Computer.ComputerBuilder("1TB", "8 GB RAM").build();

comp.setBluetoothEnabled(**true**);

comp.setGraphicsCardEnabled(**false**);

System.***out***.println("HDD"+comp.getHDD()+"\n RAM Size:"+comp.getRAM());

System.***out***.println("Blue Tooth Enabled"+comp.isBluetoothEnabled()+"\n Graphics Card Enabled:"+comp.isGraphicsCardEnabled());

StringBuilder sb= **new** StringBuilder("sb");

sb.append(4).append(**false**);

System.***out***.println("String Builder :"+sb);

String st = **new** String("s");

System.***out***.println("String :"+st);

}

}

**OUTPUT:**

HDD1TB

RAM Size:8 GB RAM

Blue Tooth Enabledtrue

Graphics Card Enabled:false

String Builder :sb4false

String :s

<https://thoughts-on-java.org/builder-pattern-hibernate/>

**Prototype Pattern**

- Prototype pattern is basically creation of new instances through cloning existing instances.

- Prototype pattern is used when creating instance of a class is very time consuming or complexity in some way. Then rather than creating more instances, you make copy of the original instance, modifying them as appropriate.

- Prototype pattern uses java cloning to copy the object.

- Many times, we want the copy of the object and its change should not affect the old object. The answer to this is prototype design pattern.

- Protype is creational pattern that lets you copy existing objects without making your code dependent on their classes.

**public** **class** Customer **implements** Cloneable {

String name ="guru";

String number="2112";

**public** Object clone() **throws** CloneNotSupportedException{

**return** **super**.clone();

}

**public** Object deepClone() **throws** CloneNotSupportedException{

Customer cust = (Customer)**super**.clone();

**return** cust;

}

**public** **boolean** equals(Object o)

{

Customer ce = (Customer)o;

**return** ce.name == name;

}

}

**public** **class** TestPrototype {

**public** **static** **void** main(String[] args) **throws** CloneNotSupportedException {

Customer c1 = **new** Customer();

Customer c2 = (Customer)c1.clone();

Customer c3 = (Customer)c1.deepClone();

System.***out***.println("c1.equals(c2)"+c1.equals(c2));

System.***out***.println("c1.equals(c3)"+c1.equals(c3));

System.***out***.println("(c1==c2)"+(c1==c2));

System.***out***.println("(c1==c3)"+(c1==c3));

String s1="guru";

String s2=s1;

System.***out***.println("(s1==s2)"+(s1==s2));

System.***out***.println("s1.equals(s2)"+s1.equals(s2));

}

}

**OUTPUT:**

c1.equals(c2)true

c1.equals(c3)true

(c1==c2)false

(c1==c3)false

(s1==s2)true

s1.equals(s2)true

**Singleton Pattern**

- It ensures that a class has only one instance provides the global point of access it.

**Example**

SessionFactory will be implemented as singleton design pattern for better performance. Singleton SessionFactory is nothing but one instance per the data source.

SessionFactory is also immutable and it will be created as singleton while the server initializes itself.  
Since SessionFactory implemented as singleton, it is thread safe too.

**Different ways to create singleton pattern**

- Eager Initialization in Singleton

- Static block Initialization in Singleton

- Lazy initialization in Singleton

- Thread safe in Singleton

- Implement singleton class with serialization

- Bill pugh Singleton implementation

**SingletonEnum**

//It supports only early loading or Eager initialization

**public** **enum** SingletonEnum {

***INSTANCE***;

**public** **void** doProcess() {

System.***out***.println("doProcess");

}

**public** **static** **void** main(String args[]) **throws** CloneNotSupportedException {

SingletonEnum instance1 = SingletonEnum.***INSTANCE***;

System.***out***.println("HashCode of instance1 "+instance1.hashCode());

SingletonEnum instance2 = SingletonEnum.***INSTANCE***;

System.***out***.println("HashCode of instance2 "+instance2.hashCode());

}

}

**Output:**

HashCode of instance1 366712642

HashCode of instance2 366712642

**SingletonExampleCloneNotSupported**

**public** **class** SingletonExampleCloneNotSupported **implements** Cloneable {

**private** SingletonExampleCloneNotSupported() {

}

**public** **static** SingletonExampleCloneNotSupported *singletonInstance* = **new** SingletonExampleCloneNotSupported();

**public** Object clone() **throws** CloneNotSupportedException{

**throw** **new** CloneNotSupportedException();

}

**public** **static** **void** main(String args[]) **throws** CloneNotSupportedException {

SingletonExampleCloneNotSupported instance1 = SingletonExampleCloneNotSupported.*singletonInstance*;

System.***out***.println("HashCode of instance1 "+instance1.hashCode());

SingletonExampleCloneNotSupported instance2 = (SingletonExampleCloneNotSupported)instance1.clone();

System.***out***.println("HashCode of instance1 "+instance1.hashCode());

System.***out***.println("HashCode of instance2 "+instance2.hashCode());

}

}

**Output:**

HashCode of instance1 366712642

Exception in thread "main" java.lang.CloneNotSupportedException

at com.example.pattern.creational.Singleton.SingletonExampleCloneNotSupported.clone(SingletonExampleCloneNotSupported.java:8)

at com.example.pattern.creational.Singleton.SingletonExampleCloneNotSupported.main(SingletonExampleCloneNotSupported.java:13)

**SingletonSerializable**

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**import** java.io.ObjectInput;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutput;

**import** java.io.ObjectOutputStream;

**import** java.io.Serializable;

**public** **class** SingletonSerializable **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = -9008590298141760838L;

**private** **static** SingletonSerializable *instance* = **new** SingletonSerializable();

**private** SingletonSerializable() {}

**public** **static** SingletonSerializable getInstance() {

**return** *instance*;

}

**protected** Object readResolve() {

**return** *getInstance*();

}

**public** **static** **void** main(String[] args) **throws** Exception {

SingletonSerializable instance1 = SingletonSerializable.*getInstance*();

ObjectOutput out = **null**;

out = **new** ObjectOutputStream(**new** FileOutputStream("Singleton.temp"));

out.writeObject(instance1);

out.close();

ObjectInput in = **new** ObjectInputStream(**new** FileInputStream("Singleton.temp"));

SingletonSerializable instance2 = (SingletonSerializable)in.readObject();

System.***out***.println("HashCode of instance1 "+instance1.hashCode());

System.***out***.println("HashCode of instance2 "+instance2.hashCode());

}

}

**OUTPUT:**

HashCode of instance1 1550089733

HashCode of instance2 1550089733

**SingletonSynchronizedBlock**

**public** **class** SingletonSynchronizedBlock {

**static** SingletonSynchronizedBlock *instance* =**null**;

**private** SingletonSynchronizedBlock() {

}

**public** **static** SingletonSynchronizedBlock getInstance() {

**synchronized**(SingletonSynchronizedBlock.**class**) {

**if** (*instance* == **null**) {

*instance* = **new** SingletonSynchronizedBlock();

}

}

**return** *instance*;

}

}

**SingletonBillPugh**

**public** **class** SingletonBillPugh {

**private** SingletonBillPugh() {

}

**private** **static** **class** StaticInstance{

**static** SingletonBillPugh *instance*= **new** SingletonBillPugh();

}

**public** **static** SingletonBillPugh getInstance() {

**return** StaticInstance.*instance*;

}

}