**DESIGN PATTERNS**

STRUCTURAL

* Adapter D P
* Bridge D P
* Filter D P
* Composite
* Decorator
* Façade
* Flyweight
* Proxy

CREATIONAL

* Factory D P
* Singleton D P
* Builder DP
* Prototype

J2EE

* MVC
* Business Delegate
* Composite Entity
* Data Access Object (DAO)
* Front Controller
* Intercepting Filter
* Service Locator
* Transfer Object

BEHAVIORAL

* Chain of responsibility
* Command
* Interpreter
* Iterator
* Mediator
* Momento
* Observer
* State D P
* Null object D P
* Strategy D P
* Template D P
* Visitor D P

**Creational** Patterns These design patterns provides way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case.

**Structural** Patterns These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities.

**Behavioral** Patterns These design patterns are specifically concerned with communication between objects.

**J2EE Patterns** These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center.

## Factory Pattern

This section describes factory pattern and its implementation.

Factory pattern is one of most used design pattern in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

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

**Implementation**

We're going to create a Shape interface and concrete classes implementing the Shape interface. A factory class ShapeFactory is defined as a next step.

FactoryPatternDemo, our demo class will use ShapeFactory to get a Shape object. It will pass information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object it needs.

## Singleton Design Pattern

This section describes singleton pattern and its implementation.

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under creational pattern as this pattern provides one of the best way to create an object.

This pattern involves a single class which is responsible to creates own object while making sure that only single object get created. This class provides a way to access its only object which can be accessed directly without need to instantiate the object of the class.

**Implementation**

We're going to create a SingleObject class. SingleObject class have its constructor as private and have a static instance of itself.

SingleObject class provides a static method to get its static instance to outside world.SingletonPatternDemo, our demo class will use SingleObject class to get a SingleObject object.

## Builder Design Pattern

This section describes builder pattern and its implementation.

Builder pattern builds a complex object using simple objects and using a step by step approach. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

A Builder class builds the final object step by step. This builder is independent of other objects.

**Implementation**

We've considered a business case of fast-food restaurant where a typical meal could be a burger and a cold drink. Burger could be either a Veg Burger or Chicken Burger and will be packed by a wrapper. Cold drink could be either a coke or pepsi and will be packed in a bottle.

We're going to create an Item interface representing food items such as burgers and cold drinks and concrete classes implementing the Item interface and a Packing interface representing packaging of food items and concrete classes implementing the Packing interface as burger would be packed in wrapper and cold drink would be packed as bottle.

We then create a Meal class having ArrayList of Item and a MealBuilder to build different types of Mealobject by combining Item. BuilderPatternDemo, our demo class will use MealBuilder to build a Meal.

## Prototype Design Pattern

This section describes prototype pattern and its implementation.

Prototype pattern refers to creating duplicate object while keeping performance in mind. This type of design pattern comes under creational pattern as this pattern provides one of the best way to create an object.

This pattern involves implementing a prototype interface which tells to create a clone of the current object. This pattern is used when creation of object directly is costly. For example, a object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as as and when needed thus reducing database calls.

**Implementation**

We're going to create an abstract class Shape and concrete classes extending the Shape class. A class ShapeCache is defined as a next step which stores shape objects in a Hashtable and returns their clone when requested.

PrototypPatternDemo, our demo class will use ShapeCache class to get a Shape object.

## Adapter Design Pattern

This section describes adapter pattern and its implementation.

Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.

This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugins the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

We are demonstrating use of Adapter pattern via following example in which an audio player device can play mp3 files only and wants to use an advanced audio player capable of playing vlc and mp4 files.

**Implementation**

We've an interface MediaPlayer interface and a concrete class AudioPlayer implementing theMediaPlayer interface. AudioPlayer can play mp3 format audio files by default.

We're having another interface AdvancedMediaPlayer and concrete classes implementing theAdvancedMediaPlayer interface.These classes can play vlc and mp4 format files.

We want to make AudioPlayer to play other formats as well. To attain this, we've created an adapter class MediaAdapter which implements the MediaPlayer interface and uses AdvancedMediaPlayerobjects to play the required format.

AudioPlayer uses the adapter class MediaAdapter passing it the desired audio type without knowing the actual class which can play the desired format. AdapterPatternDemo, our demo class will useAudioPlayer class to play various formats.

## Bridge Design Pattern

This section describes bridge pattern and its implementation.

Bridge is used where we need to decouple an abstraction from its implementation so that the two can vary independently. This type of design pattern comes under structural pattern as this pattern decouples implementation class and abstract class by providing a bridge structure between them.

This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other.

We are demonstrating use of Bridge pattern via following example in which a circle can be drawn in different colors using same abstract class method but different bridge implementer classes.

**Implementation**

We've an interface DrawAPI interface which is acting as a bridge implementer and concrete classes RedCircle, GreenCircle implementing the DrawAPI interface. Shape is an abstract class and will use object of DrawAPI. BridgePatternDemo, our demo class will use Shape class to draw different colored circle.

## Filter Design Pattern

This section describes filter pattern and its implementation.

Filter pattern or Criteria pattern is a design pattern that enables developers to filter a set of objects, using different criteria, chaining them in a decoupled way through logical operations. This type of design pattern comes under structural pattern as this pattern is combining multiple criteria to obtain single criteria.

**Implementation**

We're going to create a Person object, Criteria interface and concrete classes implementing this interface to filter list of Person objects. CriteriaPatternDemo, our demo class uses Criteria objects to filter List of Person objects based on various criteria and their combinations.

## Composite Design Pattern

This section describes composite pattern and its implementation.

Composite pattern is used where we need to treat a group of objects in similar way as a single object. Composite pattern composes objects in term of a tree structure to represent part as well as whole hierarchies. This type of design pattern comes under structural pattern as this pattern creates a tree structure of group of objects.

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

We are demonstrating use of Composite pattern via following example in which show employees hierarchy of an organization.

**Implementation**

We've a class Employee which acts as composite pattern actor class. CompositePatternDemo, our demo class will use Employee class to add department level hierarchy and print all employees.

## Decorator Design Pattern

This section describes decorator pattern and its implementation.

Decorator pattern allows adding new functionality an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

We are demonstrating use of Decorator pattern via following example in which we'll decorate a shape with some color without alter shape class.

**Implementation**

We're going to create a Shape interface and concrete classes implementing the Shape interface. We then create a abstract decorator class ShapeDecorator implementing the Shape interface and havingShape object as its instance variable.

RedShapeDecorator is concrete class implementing ShapeDecorator.

DecoratorPatternDemo, our demo class will use RedShapeDecorator to decorate Shape objects.

## Façade Design Pattern

This section describes facade pattern and its implementation.

Facade pattern hides the complexities of the system and provides an interface to the client using which the client can access the system. This type of design pattern comes under structural pattern as this pattern adds an interface to exiting system to hide its complexities.

This pattern involves a single class which provides simplified methods which are required by client and delegates calls to existing system classes methods.

**Implementation**

We're going to create a Shape interface and concrete classes implementing the Shape interface. A facade class ShapeMaker is defined as a next step.

ShapeMaker class uses the concrete classes to delegates user calls to these classes.FacadePatternDemo, our demo class will use ShapeMaker class to show the results.

## Flyweight Design Pattern

This section describes flyweight pattern and its implementation.

Flyweight pattern is primarily used to reduce the number of objects created, to decrease memory footprint and increase performance. This type of design pattern comes under structural pattern as this pattern provides ways to decrease objects count thus improving application required objects structure.

Flyweight pattern try to reuse already existing similar kind objects by storing them and creates new object when no matching object is found. We'll demonstrate this pattern by drawing 20 circle of different locations but we'll creating only 5 objects. Only 5 colors are available so color property is used to check already existing Circle objects.

**Implementation**

We're going to create a Shape interface and concrete class Circle implementing the Shape interface. A factory class ShapeFactory is defined as a next step.

ShapeFactory have a HashMap of Circle having key as color of the Circle object. Whenever a request comes to create a circle of particular color to ShapeFactory. ShapeFactory checks the circle object in itsHashMap, if object of Circle found, that object is returned otherwise a new object is created, stored in hashmap for future use and returned to client.

FlyWeightPatternDemo, our demo class will use ShapeFactory to get a Shape object. It will pass information (red / green / blue/ black / white) to ShapeFactory to get the circle of desired color it needs.

## Proxy Design Pattern

This section describes proxypattern and its implementation.

In Proxy pattern, a class represents functionality of another class. This type of design pattern comes under structural pattern.

In Proxy pattern, we create object having original object to interface its functionality to outer world.

**Implementation**

We're going to create a Image interface and concrete classes implementing the Image interface.ProxyImage is a a proxy class to reduce memory footprint of RealImage object loading.

ProxyPatternDemo, our demo class will use ProxyImage to get a Image object to load and display as it needs.

## Chain of Responsibility Design Pattern

This section describes chain of responsibility pattern and its implementation.

As the name suggest, the chain of responsibility pattern creates a chain of receiver objects for a request. This pattern decouples sender and receiver of a request based on type of request. This pattern comes under behavioral patterns.

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

**Implementation**

We've created an abstract class AbstractLogger with a level of logging. Then we've created three types of loggers extending the AbstractLogger.

Each logger checks the level of message to its level and print accordingly otherwise does not print and pass the message to its next logger.

## Command Design Pattern

This section describes command pattern and its implementation.

Command pattern is a data driven design pattern and falls under behavioral pattern category. A request is wrapped under a object as command and passed 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.

**Implementation**

We've created an interface Order which is acting as a command. We've created a Stock class which acts as a request.

We've concrete command classes BuyStock and SellStock implementing Orderinterface which will do actual command processing. A class Broker is created which acts as a invoker object. It can take order and place orders.

Broker object uses command pattern to identify which object will execute which command based on type of command. CommandPatternDemo, our demo class will use Broker class to demonstrate command pattern.

## Interpreter Design Pattern

This section describes interpreter pattern and its implementation.

Interpreter pattern provides way to evaluate language grammar or expression. This type of pattern comes under behavioral patterns. This pattern involves implementing an expression interface which tells to interpret a particular context. This pattern is used in SQL parsing, symbol processing engine etc.

**Implementation**

We're going to create an interface Expression and concrete classes implementing the Expressioninterface. A class TerminalExpression is defined which acts as a main interpreter of context in question. Other classes OrExpression, AndExpression are used to create combinational expressions.

InterpreterPatternDemo, our demo class will use Expression class to create rules and demonstrate parsing of expressions.

## Iterator Design Pattern

This section describes iterator pattern and its implementation.

Iterator pattern is very commonly used design pattern in Java and .Net programming environment. This pattern is used to get a way to access the elements of a collection object in sequential manner without any need to know its underlying representation.

Iterator pattern falls under behavioral pattern category.

**Implementation**

We're going to create a Iterator interface which narrates navigation method and a Container interface which returns the iterator. Concrete classes implementing the Container interface will be responsible to implement Iterator interface and use it

IteratorPatternDemo, our demo class will use NamesRepository, a concrete class implementation to print a Names stored as a collection in NamesRepository.

## Mediator Design Pattern

This section describes mediator pattern and its implementation.

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. Mediator pattern falls under behavioral pattern category.

**Implementation**

We're demonstrating mediator pattern by example of a Chat Room where multiple users can send message to Chat Room and it is the responsibility of Chat Room to show the messages to all users. We've created two classes ChatRoom and User. User objects will use ChatRoom method to share their messages.

MediatorPatternDemo, our demo class will use User objects to show communication between them.

## Memento Design Pattern

This section describes mementopattern and its implementation.

Memento pattern is used to reduce where we want to restore state of an object to a previous state. Memento pattern falls under behavioral pattern category.

**Implementation**

Memento pattern uses three actor classes. Memento contains state of an object to be restored. Originator creates and stores states in Memento objects and Caretaker object which is responsible to restore object state from Memento.

We've created classes Memento, Originator and CareTaker.

MementoPatternDemo, our demo class will use CareTaker and Originator objects to show restoration of object states.

## Observer Design Pattern

This section describes observer pattern and its implementation.

Observer pattern is used when there is one to many relationship between objects such as if one object is modified, its depenedent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

**Implementation**

Observer pattern uses three actor classes Subject, Observer and Client. Subject an object having methods to attach and de-attach observers to a client object. We've created classes Subject, Observerabstract class and concrete classes extending the abstract class the Observer.

ObserverPatternDemo, our demo class will use Subject and concrete class objects to show observer pattern in action. TUTORIALS POINT Simply Easy Learning Page 95

## State Design Pattern

This section describes state pattern andits implementation.

In State pattern a class behavior changes based on its state. This type of design pattern comes under behavior pattern.

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

**Implementation**

We're going to create a State interface defining a action and concrete state classes implementing theState interface. Context is a class which carries a State.

StaePatternDemo, our demo class will use Context and state objects to demonstrate change in Context behavior based on type of state it is in.

## Null Object Design Pattern

This section describes null object pattern and its implementation.

In Null Object pattern, a null object replaces check of NULL object instance. Instead of putting if check for a null value, Null Object reflects a do nothing relationship. Such Null object can also be used to provide default behaviour in case data is not available.

In Null Object pattern, we create a abstract class specifying the various operations to be done, concreate classes extending this class and a null object class providing do nothing implemention of this class and will be used seemlessly where we need to check null value.

**Implementation**

We're going to create a AbstractCustomer abstract class defining opearations, here the name of the customer and concrete classes extending the AbstractCustomer class. A factory classCustomerFactory is created to return either RealCustomer or NullCustomer objects based on the name of customer passed to it.

NullPatternDemo, our demo class will use CustomerFactory to demonstrate use of Null Object pattern.

## StrategyDesign Pattern

This section describes strategypattern and its implementation.

In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

**Implementation**

We're going to create a Strategy interface defining a action and concrete strategy classes implementing the Strategy interface. Context is a class which uses a Strategy.

StrategyPatternDemo, our demo class will use Context and strategy objects to demonstrate change in Context behaviour based on strategy it deploys or uses.

## TemplateDesign Pattern

This section describes templatepattern and its implementation.

In Template pattern, an abstract class exposes defined way(s)/template(s) to execute its methods. Its subclasses can overrides the method implementations as per need basis but the invocation is to be in the same way as defined by an abstract class. This pattern comes under behavior pattern category.

**Implementation**

We're going to create a Game abstract class defining operations with a template method set to be final so that it cannot be overridden. Cricket and Football are concrete classes extend Game and override its methods.

TemplatePatternDemo, our demo class will use Game to demonstrate use of template pattern.

## VisitorDesign Pattern

This section describes visitorpattern and its implementation.

In Visitor pattern, we use a visitor class which changes the executing algorithm of an element class. By this way, execution algorithm of element can varies as visitor varies. This pattern comes under behavior pattern category. As per the pattern, element object has to accept the visitor object so that visitor object handles the operation on the element object.

**Implementation**

We're going to create a ComputerPart interface defining accept opearation. Keyboard, Mouse, Monitor and Computer are concrete classes implementing ComputerPart interface.

We'll define another interface ComputerPartVisitor which will define visitor class operations. Computer uses concrete visitor to do corresponding action.

VisitorPatternDemo, our demo class will use Computer, ComputerPartVisitor classes to demonstrate use of visitor pattern.

## MVCDesign Pattern

This section describes MVCpattern and its implementation.

MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.

• Model - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.

• View - View represents the visualization of the data that model contains.

• Controller - Controller acts on both Model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps View and Model separate.

**Implementation**

We're going to create Student object acting as a model.StudentView will be a view class which can print student details on console and StudentController is the controller class responsible to store data in Student object and update view StudentView accordingly.

MVCPatternDemo, our demo class will use StudentController to demonstrate use of MVC pattern.

## Business DelegateDesign Pattern

This section describes business delegatepattern and its implementation.

Business Delegate Pattern is used to decouple presentation tier and business tier. It is basically use to reduce communication or remote lookup functionality to business tier code in presentation tier code. In business tier we've following entities.

• Client - Presentation tier code may be JSP, servlet or UI java code.

• Business Delegate - A single entry point class for client entities to provide access to Business Service methods.

• LookUp Service - Lookup service object is responsible to get relative business implementation and provide business object access to business delegate object.

• Business Service - Business Service interface. Concrete classes implement this business service to provide actual business implementation logic.

**Implementation**

We're going to create a Client, BusinessDelegate, BusinessService, LookUpService, JMSService and EJBService.

These classes will be representing various entities of Business Delegate pattern.

BusinessDelegatePatternDemo, our demo class will use BusinessDelegate and Client to demonstrate use of Business Delegate pattern.

## Composite Entity DesignPattern

This section describes composite entitypattern and its implementation.

Composite Entity pattern is used in EJB persistence mechanism. A Composite entity is an EJB entity bean which represents a graph of objects. When a composite entity is updated, internally dependent objects beans get updated automatically as being managed by EJB entity bean. Following are the participants in Composite Entity Bean.

• Composite Entity - It is primary entity bean.It can be coarse grained or can contain a coarse grained object to be used for persistence purpose.

• Coarse-Grained Object -This object contains dependent objects. It has its own life cycle and also manages life cycle of dependent objects.

• Dependent Object - Dependent objects is an object which depends on Coarse-Grained object for its persistence lifecycle.

• Strategies - Strategies represents how to implement a Composite Entity.

**Implementation**

We're going to create CompositeEntity object acting as CompositeEntity.

CoarseGrainedObject will be a class which contains dependent objects.

CompositeEntityPatternDemo, our demo class will use Client class to demonstrate use of Composite Entity pattern.

## Data Access ObjectDesign Pattern

This section describes data access objectpattern and its implementation.

Data Access Object Pattern or DAO pattern is used to separate low level data accessing API or operations from high level business services. Following are the participants in Data Access Object Pattern.

• Data Access Object Interface - This interface defines the standard operations to be performed on a model object(s).

• Data Access Object concrete class -This class implements above interface. This class is responsible to get data from a datasource which can be database / xml or any other storage mechanism.

• Model Object or Value Object - This object is simple POJO containing get/set methods to store data retrieved using DAO class.

**Implementation**

We're going to create a Student object acting as a Model or Value Object.StudentDao is Data Access Object Interface.StudentDaoImpl is concrete class implementing Data Access Object Interface.

DaoPatternDemo, our demo class will use StudentDao demonstrate use of Data Access Object pattern.

## Front ControllerDesign Pattern

This section describes front controllerpattern and its implementation.

The front controller design pattern is used to provide a centralized request handling mechanism so that all requests will be handled by a single handler. This handler can do the authentication/ authorization/ logging or tracking of request and then pass the requests to corresponding handlers. Following are the entities of this type of design pattern.

• Front Controller - Single handler for all kind of request coming to the application (either web based/ desktop based).

• Dispatcher - Front Controller may use a dispatcher object which can dispatch the request to corresponding specific handler.

• View - Views are the object for which the requests are made.

**Implementation**

We're going to create a FrontController, Dispatcher to act as Front Controller and Dispatcher correspondingly. HomeView and StudentView represent various views for which requests can come to front controller.

FrontControllerPatternDemo, our demo class will use FrontController ato demonstrate Front Controller Design Pattern.

## Intercepting FilterDesign Pattern

This section describes intercepting filterpattern and its implementation.

The intercepting filter design pattern is used when we want to do some pre-processing / post-processing with request or response of the application. Filters are defined and applied on the request before passing the request to actual target application. Filters can do the authentication/ authorization/ logging or tracking of request and then pass the requests to corresponding handlers. Following are the entities of this type of design pattern.

• Filter - Filter which will perform certain task prior or after execution of request by request handler.

• Filter Chain - Filter Chain carries multiple filters and help to execute them in defined order on target.

• Target - Target object is the request handler

• Filter Manager - Filter Manager manages the filters and Filter Chain.

• Client - Client is the object who sends request to the Target object.

**Implementation**

We're going to create a FilterChain, FilterManager, Target, and Client as various objects representing our entities.AuthenticationFilter and DebugFilter represents concrete filters.

InterceptingFilterDemo, our demo class will use Client to demonstrate Intercepting Filter Design Pattern.

## Service LocatorDesign Pattern

This section describes service locatorpattern and its implementation.

The service locator design pattern is used when we want to locate various services using JNDI lookup. Considering high cost of looking up JNDI for a service, Service Locator pattern makes use of caching technique.

For the first time a service is required, Service Locator looks up in JNDI and caches the service object. Further lookup or same service via Service Locator is done in its cache which improves the performance of application to great extent. Following are the entities of this type of design pattern.

• Service - Actual Service which will process the request. Reference of such service is to be looked upon in JNDI server.

• Context / Initial Context -JNDI Context, carries the reference to service used for lookup purpose.

• Service Locator - Service Locator is a single point of contact to get services by JNDI lookup, caching the services.

• Cache - Cache to store references of services to reuse them.

• Client - Client is the object who invokes the services via ServiceLocator.

**Implementation**

We're going to create a ServiceLocator, InitialContext, Cache, and Service as various objects representing our entities.Service1 and Service2 represents concrete services.

ServiceLocatorPatternDemo, our demo class is acting as a client here and will use ServiceLocator to demonstrate Service Locator Design Pattern.

## Transfer ObjectDesign Pattern

This section describes transfer objectpattern and its implementation.

The Transfer Object pattern is used when we want to pass data with multiple attributes in one shot from client to server. Transfer object is also known as Value Object. Transfer Object is a simple POJO class having getter/setter methods and is serializable so that it can be transferred over the network.

It does not have any behavior. Server Side business class normally fetches data from the database and fills the POJO and send it to the client or pass it by value. For client, transfer object is read-only. Client can create its own transfer object and pass it to server to update values in database in one go. Following are the entities of this type of design pattern.

• Business Object - Business Service which fills the Transfer Object with data.

• Transfer Object -Simple POJO, having methods to set/get attributes only.

• Client - Client either requests or sends the Transfer Object to Business Object.

**Implementation**

We're going to create a StudentBO as Business Object, Student as Transfer Object representing our entities.

TransferObjectPatternDemo, our demo class is acting as a client here and will use StudentBO andStudent to demonstrate Transfer Object Design Pattern.