**+DESIGN PATTERNS**

**Design Patterns** are very popular among software developers. A design pattern is a well described solution to a common software problem. I have written extensively on **java design patterns**.

## Java Design Patterns

Some of the benefits of using design patterns are:

Design Patterns are already defined and provides **industry standard approach** to solve a recurring problem, so it saves time if we sensibly use the design pattern. There are many java design patterns that we can use in our java based projects.

Using design patterns promotes **reusability** that leads to more **robust** and highly maintainable code. It helps in reducing total cost of ownership (TCO) of the software product.

Since design patterns are already defined, it makes our code easy to understand and debug. It leads to faster development and new members of team understand it easily.

**Design patterns:**

Pattern means set of guide lines.

Design patterns are solutions to commonly reoccurring problems in software development.

Design patterns are well proven solutions to common software problems.

Design patterns are best practices to use software technologies effectively in application development.

Design patterns used in analysis and requirement phase of  SDLC.

Design patterns can be implemented by using programming language.

**Advantages:**

**Reusable.**

These are already defined solutions to common re occurring problems so it reduces time.

They are already defined so Easy to understand and debug.

**Categorization:**

These are categorized into two parts.

Java SE Design patterns.

Java EE Design patterns.

**Java SE Design patterns:**

 In Java SE there are mainly three types.

**Creational Design patterns:**

In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design.

**Factory Pattern:**

Factory pattern is one of the most used design patterns 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.

Ex: Circle, Rectangle, Square implements Shape interface.

ShapeFactory and FactoryPatternDemo (Main method).

**Abstract Factory Pattern:**

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. This type of design pattern comes under creational pattern as this pattern provides one of the best ways to create an object.

In Abstract Factory pattern an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

Ex: Circle, Rectangle, Square implements Shape interface.

Red, Green, Blue implements Color interface.

ShapeFactory, ColorFactory implements Abstract class ‘AbstractFactory’.

AbstractFactoryPatternDemo creates factory classes through FactoryProducer.

ShapeFactory and FactoryPatternDemo(Main method).

**Singleton Pattern:**

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 ways to create an object.

This pattern involves a single class which is responsible to create an object while making sure that only single object gets 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.

SingleObject, SinglePatternDemo

**Prototype Pattern**

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 ways 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, an 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 and when needed thus reducing database calls.

[**Builder Pattern.**](http://www.instanceofjava.com/2016/08/builder-design-pattern-java-example.html)

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.

Interfaces - Item, Packing

Abstract Class – Burger implements Item

ColdDrink implements item.

Class: Bottle implements Packing, Wrapper implements Packing, Meal, MealBuilder,

Coke extends ColdDrink, Pepsi extends ColdDrink

Class: ChickenBurger extends Burger, VegBurger extends Burger.

**Structural Design patterns:**

In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities. Examples of Structural Patterns include: Adapter pattern: 'adapts' one interface for a class into one that a client expects.

**Adapter Pattern**

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 plugin the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

Interfaces: MediaPlayer, AdvancedMediaPlayer

Classes : VlcPlayer, Mp4Player implements AdvancedMediaPlayer

Class: MediaAdapter implements MediaPlayer

**Bridge Pattern**

Bridge is used when 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.

Abstract Class – Shape refers DrawAPI interface

GreenCircle, RedCircle implements DrawAPI interface

Circle Class extends Shape abstract class.

**Composite Pattern**

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 hierarchy. 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 that contains group of its own objects. This class provides ways to modify its group of same objects.

Employee, CompositePatternDemo

**Decorator Pattern**

Decorator pattern allows a user to add new functionality to 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.

**Circle** class implements **Shape** interface

**Rectangle** class implements **Shape** interface

**ShapeDecorator** abstract class implements **Shape** interface.

**RedShapeDecorator** extends **ShapeDecorator** class.

**Facade Pattern**

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 existing system to hide its complexities.

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

Circle, Square, Rectangle implements **Shape** interface

ShapeMaker uses all those classes.

**Flyweight Pattern**

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

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

**Circle** implements **Shape** interface

ShapeFactory is a class

FlyweightPatternDemo

**Proxy Pattern**

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.

**Filter Pattern**

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

Person is a separate class

AndCriteria, OrCriteria, CriteriaFemale, CriteriaMale, CriteriaFemale, CriteriaSingle classes are implementing Criteria interface.

**Behavioral Design patterns:**

Behavioral patterns provide solution for the better interaction between objects and how to provide lose coupling and flexibility to extend easily.

**Chain of Responsibility Pattern**

As the name suggests, 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.

**Command Pattern**

Command pattern is a data driven design pattern and falls under behavioral pattern category. 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 executes the command.

**Interpreter Pattern**

Interpreter pattern provides a way to evaluate language grammar or expression. This type of pattern comes under behavioral pattern. 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.

**Iterator Pattern**

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.

**Mediator Pattern**

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 maintenance of the code by loose coupling. Mediator pattern falls under behavioral pattern category.

**Memento Pattern**

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

**Observer Pattern**

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.

**State Pattern**

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.

**Strategy Pattern**

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.

**Template Pattern**

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

**Visitor Pattern**

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 vary as and when 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.

**Null Object Pattern**

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 an abstract class specifying various operations to be done, concrete 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.

**MVC pattern**

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.

**Java EE Design patterns**

**Business Delegate Pattern**

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 have 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.

**Composite Entry Pattern**

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 object is an object which depends on coarse grained object for its persistence lifecycle.

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

**Data Access object Pattern**

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 data source 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.

**Front Controller Pattern**

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 kinds of requests 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.

**Intercepting Filter Pattern**

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.

**Service Locator Pattern**

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 that invokes the services via ServiceLocator.

**Transfer Object Pattern**

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 shot. Following are the entities of this type of design pattern.

**Business Object** - Business Service 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.

**Factory Method Pattern:**

**Problem**: Using new keyword we cannot create object with flexibility and by applying restrictions.

**Solution**: Use Factory pattern (or) Factory method.

By defining a abstract class or an interface but let the subclass  decide which class object to instantiate.

A method of a class capable of constructing and returning its own class object or other class object is called "factory method".

There are two types of factory methods.

**Static factory method.**

**Instance factory method.**

**1.Static Factory method:**

A static method defined to construct and return object of same class or different is known as static factory method.

* + - * Some of the pre defined static factory methods are as follows.

**Thread th= Thread.currentThread();**

**Class c=Class.forName();**

**Runtime rt=Runtime.getRuntime();**

**Calendar c=Calendar.getInstance();**

* **Instance Factory method:**

A non static method defined to construct and return object of same class or different is known as instance factory method. Some of the pre defined instance factory methods are as follows.

**String s= new String("instance of");  
 String s1=s.concat("java");**

**StringBuffer sb=new StringBuffer("instance of");  
sb=sb.subString(0,2);**

**Date d= new Date();  
String s=d.toString();**

[**Singleton Design Pattern**](http://www.instanceofjava.com/2015/05/singleton-design-pattern-in-java.html)

**Problem:**

Instead of creating multiple objects of same class having same data and wasting memory, degrading performance it is recommended to create only one object and use it multiple times.

**Solution:**

 Use Singleton Java Class.

Java class that allows us to create one object per JVM is is called as singleton java class.

The logger class of Log 4J API is given as singleton java class.

**Rules:**

It must have only private constructors.

It must have private static reference variable of same class.

It must have public static factory method having the logic of singleton.

Static method should create and return only one object.

All these rules close all the doors of creating objects for java class and opens only one door to create object.

factory method where singleton logic is placed

The method of a class capable of creating and returning same class or other class object is known as factory method.

[**Builder design pattern**](http://www.instanceofjava.com/2016/08/builder-design-pattern-java-example.html)

Design patterns are solutions to software design problems.

Design patterns classified into three types.

Creational, Structural and behavioral design patterns.

Creational patterns helps us to create objects in a manner suitable to the given situation.

Builder design pattern is one of the creational  design pattern in java.

Builder design pattern helps us to create complex class object.

Builder design pattern helps us to separate the construction process of a complex object from its representation so that same object construction process can be created in different representations.

Means it will separate complex construction into two parts initialization of class instance and return class instance.

When a class having more number of fields and constructor of that class take care of assigning initial values.

And when we want to create object of the class we need to pass all parameters and should be in same order which constructor is accepting.

Builder design pattern helps us to create same class object by passing required number of fields by using separate builder class object.

Builder design pattern is useful when object creation is very complex.

Advantages of builder design pattern:

Builder design pattern simplifies complex object creation.

Builder design pattern provides separation between instance creation and representation

Reusability

## *Core J2EE patterns are broadly classified in the following three categories based on the layering of the multi tiered systems. A specific patterns applies to a specific layer of the system under development*

## *[Presentation Tier Patterns](http://www.javagyan.com/tutorials/corej2eepatterns/presentation-tier-patterns)*

* [**Intercepting Filter**](http://www.javagyan.com/tutorials/corej2eepatterns/presentation-tier-patterns/intercepting-filter-1)intercepts incoming requests and outgoing responses and applies a filter. These filters may be added and removed in a declarative manner, allowing them to be applied unobtrusively in a variety of combinations. After this preprocessing and/or post-processing is complete, the final filter in the group vectors control to the original target object. For an incoming request, this is often a Front Controller, but may be a View.
* [**Front Controller**](http://www.javagyan.com/tutorials/corej2eepatterns/presentation-tier-patterns/front-controller)is a container to hold the common processing logic that occurs within the presentation tier and that may otherwise be erroneously placed in a View. A controller handles requests and manages content retrieval, security, view management, and navigation, delegating to a Dispatcher component to dispatch to a View.
* [**Application Controller**](http://www.javagyan.com/tutorials/corej2eepatterns/presentation-tier-patterns/application-controller) centralizes control, retrieval, and invocation of view and command processing. While a Front Controller acts as a centralized access point and controller for incoming requests, the Application Controller is responsible for identifying and invoking commands, and for identifying and dispatching to views.
* [**Context Object**](http://www.javagyan.com/tutorials/corej2eepatterns/presentation-tier-patterns/context-object) encapsulates state in a protocol-independent way to be shared throughout your application. Using Context Object makes testing easier, facilitating a more generic test environment with reduced dependence upon a specific container.
* [**View Helper**](http://www.javagyan.com/tutorials/corej2eepatterns/presentation-tier-patterns/view-helper)encourages the separation of formatting-related code from other business logic. It suggests using Helper components to encapsulate logic relating to initiating content retrieval, validation, and adapting and formatting the model. The View component is then left to encapsulate the presentation formatting. Helper components typically delegate to the business services via a Business Delegate or an Application Service, while a View may be composed of multiple subcomponents to create its template.
* [**Composite View**](http://www.javagyan.com/tutorials/corej2eepatterns/presentation-tier-patterns/composite-view) suggests composing a View from numerous atomic pieces. Multiple smaller views, both static and dynamic, are pieced together to create a single template. The Service to Worker and Dispatcher View patterns represent a common combination of other patterns from the catalog. The two patterns share a common structure, consisting of a controller working with a Dispatcher, Views, and Helpers. Service to Worker and Dispatcher View have similar participant roles, but differ in the division of labor among those roles. Unlike Service to Worker, Dispatcher View defers business processing until view processing has been performed.
* [**Service to worker**](http://www.javagyan.com/tutorials/corej2eepatterns/presentation-tier-patterns/service-to-worker) performs core request handling and invoke business logic before control is passed to the view. It centralizes control and request handling to retrieve a presentation model before turning control over to the view. The view generates a dynamic response based on the presentation model.
* [**Dispatcher View**](http://www.javagyan.com/tutorials/corej2eepatterns/presentation-tier-patterns/dispatcher-view)combines a controller and dispatcher with views and helpers to handle client requests and prepare a dynamic presentation as the response. Controllers do not delegate content retrieval to helpers, because these activities are deferred to the time of view processing. A dispatcher is responsible for view management and navigation and can be encapsulated either within a controller, a view, or a separate component.

## *[Business Tier Patterns](http://www.javagyan.com/tutorials/corej2eepatterns/business-tier-patterns)*

* [**Business Delegate**](http://www.javagyan.com/tutorials/corej2eepatterns/business-tier-patterns/business-delegate) reduces coupling between remote tiers and provides an entry point for accessing remote services in the business tier. A Business Delegate might also cache data as necessary to improve performance. A Business Delegate encapsulates a Session Façade and maintains a one-to-one relationship with that Session Façade. An Application Service uses a Business Delegate to invoke a Session Façade.
* [**Service Locator**](http://www.javagyan.com/tutorials/corej2eepatterns/business-tier-patterns/service-locator) encapsulates the implementation mechanisms for looking up business service components. A Business Delegate uses a Service Locator to connect to a Session Façade. Other clients that need to locate and connect to Session Façade, other business-tier services, and web services can use a Service Locator.
* [**Session Façade**](http://www.javagyan.com/tutorials/corej2eepatterns/business-tier-patterns/session-facade)provides coarse-grained services to the clients by hiding the complexities of the business service interactions. A Session Façade might invoke several Application Service implementations or Business Objects. A Session Façade can also encapsulate a Value List Handler.
* [**Application Service**](http://www.javagyan.com/tutorials/corej2eepatterns/business-tier-patterns/application-service) centralizes and aggregates behavior to provide a uniform service layer to the business tier services. An Application Service might interact with other services or Business Objects. An Application Service can invoke other Application Services and thus create a layer of services in your application.
* [**Business Object**](http://www.javagyan.com/tutorials/corej2eepatterns/business-tier-patterns/business-object) implements your conceptual domain model using an object model. Business Objects separate business data and logic into a separate layer in your application. Business Objects typically represent persistent objects and can be transparently persisted using Domain Store.
* [**Composite Entity**](http://www.javagyan.com/tutorials/corej2eepatterns/business-tier-patterns/composite-entity) implements a Business Object using local entity beans and POJOs. When implemented with bean-managed persistence, a Composite Entity uses Data Access Objects to facilitate persistence.
* [**The Transfer Object**](http://www.javagyan.com/tutorials/corej2eepatterns/business-tier-patterns/transfer-object) pattern provides the best techniques and strategies to exchange data across tiers (that is, across system boundaries) to reduce the network overhead by minimizing the number of calls to get data from another tier.
* [**The Transfer Object Assembler**](http://www.javagyan.com/tutorials/corej2eepatterns/business-tier-patterns/transfer-object-assembler) constructs a composite Transfer Object from various sources. These sources could be EJB components, Data Access Objects, or other arbitrary Java objects. This pattern is most useful when the client needs to obtain data for the application model or part of the model.
* [**The Value List Handler**](http://www.javagyan.com/tutorials/corej2eepatterns/business-tier-patterns/value-list-handler) uses the GoF iterator pattern to provide query execution and processing services. The Value List Handler caches the results of the query execution and return subsets of the result to the clients as requested. By using this pattern, it is possible to avoid overheads associated with finding large numbers of entity beans. The Value List Handler uses a Data Access Object to execute a query and fetch the results from a persistent store.

## *[Integration Tier Patterns](http://www.javagyan.com/tutorials/corej2eepatterns/integration-tier-patterns)*

* [**Data Access Object**](http://www.javagyan.com/tutorials/corej2eepatterns/integration-tier-patterns/data-access-object) enables loose coupling between the business and resource tiers. Data Access Object encapsulates all the data access logic to create, retrieve, delete, and update data from a persistent store. Data Access Object uses Transfer Object to send and receive data.
* [**Service Activator**](http://www.javagyan.com/tutorials/corej2eepatterns/integration-tier-patterns/service-activator) enables asynchronous processing in your enterprise applications using JMS. A Service Activator can invoke Application Service, Session Façade or Business Objects. You can also use several Service Activators to provide parallel asynchronous processing for long running tasks.
* [**Domain Store**](http://www.javagyan.com/tutorials/corej2eepatterns/integration-tier-patterns/domain-store) provides a powerful mechanism to implement transparent persistence for your object model. It combines and links several other patterns including Data Access Objects.
* [**Web Service Broker**](http://www.javagyan.com/tutorials/corej2eepatterns/integration-tier-patterns/web-service-broker) exposes and brokers one or more services in your application to external clients as a web service using XML and standard web protocols. A Web Service Broker can interact with Application Service and Session Façade. A Web Service Broker uses one or more Service Activators to perform asynchronous processing of a request.