# Overview:

* There are some few patters:
  + Business Delegate Pattern.
  + Composite Entity Pattern.
  + Data Access Object.
  + Front Controller.
  + Intercepting Filter.
  + Service Locator.
  + Transfer Object.

# Business Delegate Pattern:

* Used to decouple the presentation tier and the business tier:
  + Used to reduce communication or remote lookup functionality to business tier code in presentation tier code.
* Acts as client-side business abstraction:
  + Provides an abstraction for, and thus hides, the implementation of the business services.
* Use the pattern when:
  + You want loose coupling between presentation and business tiers
  + You want to orchestrate calls to multiple business services.
  + You want to encapsulate service lookups and service calls.
* **Components** include:
  + **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:**
    - A responsible to get relative business implementation and provide business object access to business delegate object.
  + **Business Service:**
    - Concrete classes implement this business service to provide actual business implementation logic.

# Composite Entity Pattern:

* 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.
* The following are the participants:
  + **Context:**
    - Entity beans are not intended to represent every persistent object in the object model.
    - Entity beans are better suited for coarse-grained persistent business objects.
  + **Composite Entity:**
    - Primary entity bean.
    - Can be coarse grained or can contain a coarse-grained object to be used for persistence purpose.
  + **Coarse-Grained Object:**
    - Contains dependent objects
      * Has its own life cycle and manages life cycle of dependent objects.
  + **Dependent Object:**
    - An object, which depends on coarse-grained object for its persistence lifecycle.
  + **Strategies:**
    - Represents how to implement a Composite Entity.

# Data Access Object Pattern:

* Many real-world J2EE applications need to use persistent data at some point
* Access to data varies depending on the source of the data.
* Access to persistent storage varies greatly depending on the type of storage (relational databases, object-oriented databases, flat files, and so forth) and the vendor implementation.
* The data may reside in mainframe systems, LightweightDirectory Access Protocol (LDAP) repositories, etc.
* This pattern is used to separate low-level data accessing API or operations from high-level business services.
* Data Access Object Interface:
  + This interface defines the standard operations to be performed on a model object(s).
* Data Access Object concrete class:
  + Implements above interface and is responsible to get data from a data source, which can be database/xml or any other storage mechanism.
* Model Object or Value Object:
  + Simple POJO (Plain Old Java Object) containing get/set methods to store data retrieved using DAO classes.

# Front Controller Pattern:

* All requests that come from a resource in an application will be handled by a single handler and then dispatched to the appropriate handler for that type of request.
* Not used as widely since the MVC pattern was released.
* The presentation-tier request handling mechanism must control and coordinate processing of each user across multiple requests.
* When a user accesses the view directly without going through a centralized mechanism, two problems may occur:
  + Each view is required to provide its own system services, often resulting in duplicate code.
  + View navigation is left to the views, which may result in mixed view content and view navigation.
* In addition, distributed control is more difficult to maintain, since changes will often need to be made in numerous places.
* This pattern will ensure a system has a centralized access point for presentation-tier request handling to support the integration of system services, content retrieval, view management, and navigation.
* The following are participants:
  + Controller:
    - The initial contact point for handling all requests in the system.
    - May delegate to a helper to complete authentication and authorization of a user or to initiate contact retrieval.
  + View:
    - Represents and displays information to the client
    - Retrieves information from a model
    - Helpers support view by encapsulating and adapting the underlying data model for use in the display.
  + Dispatcher:
    - Responsible for view management and navigation, managing the choice of the next view to present the user, and providing the mechanism for vectoring control to this resource.
  + Helper:
    - Responsible for helping a view or controller complete its processing
    - Have numerous responsibilities
      * Gathering data required by the view and storing this intermediate model.

# Intercepting Filter:

* A J2EE pattern, which creates pluggable filters to process common services in a standard manner without requiring changes to core request processing code.
* 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.
* Participants include the following:
  + **Filter:**
    - Performs certain tasks prior or after execution of request by request handler.
  + **Filter Chain:**
    - Carries multiple filters and help to execute them in defined order target.
  + **Target:**
    - The request handler
  + **Filter Manager:**
    - Manages the filters and Filter Chain.
  + **Client:**
    - The object who sends request to the Target object.

# Service Locator:

* The service locator pattern is a relatively old pattern that was very popular with J2EE.
  + Goal of this pattern is to improve the modularity of your application by removing the dependency between the client and the implementation of an interface.
* Used whenever we want to locate/fetch various services using JNDI (Java Naming and Directory Interface) which, typically, is redundant and expensive lookup
* Encapsulates the process involved in obtaining a service with a strong abstraction layer.
  + Uses a central registry known as the “service locator” which on request returns the information necessary to perform a certain task.
* The ServiceLocator is responsible for returning instances of services when they are requested for by the service consumers or the service clients.
* This pattern addresses this expensive lookup by making use of caching techniques
  + The very first time a particular service is requested, the service Locator looks up in JNDI, fetches the relevant service and the finally caches this service object.
  + Further lookups of the same service via Service Locator is done in its cache, which improves the performance of the application largely.
* The following are participants:
  + **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:**
    - A single point of contact to get services by JNDI lookup caching the services
  + **Cache:**
    - Stores references of services to reuse them
  + **Client:**
    - The object that invokes the services via ServiceLocator.

# Transfer Object:

* A frequently used design pattern when we want to pass data with multiple attributes in one shot from client to server, to avoid multiple calls to a remote server
* A simple POJO class having getter/setters methods and is serializable so that it can be transferred over the network
  + Does not have any behavior
* A server side business class normally fetches data from the database and fills the POJO and then sends it to the client
* The following are participants of this pattern:
  + **Business Object:**
    - Fills the Transfer Object with data
  + **Transfer Object:**
    - Simple POJO having methods to set/get attributes only
  + **Client:**
    - Either requests or sends the Transfer Object to Business Object.