**Design Patterns**

Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Design patterns can speed up the development process by providing tested, proven development paradigms(pattern).

Reusing design patterns helps to prevent subtle issues that can cause major problems, and it also improves code readability for coders and architects who are familiar with the patterns.

**Gang of Four (GOF)** - According to them, design patterns are primarily based on the following principles of object orientated design.

* Program to an interface not an implementation
* Favor object composition over inheritance (interface methods to define the behavior rather than inheriting implementation from base class)

**Creational Patterns**

These design patterns provide a way to create objects while hiding the creation logic. The creational patterns aim to separate a system from how its objects are created, composed, and represented.

**Structural Patterns**

These design patterns concern class and object composition. A structural design pattern serves as a blueprint for how different classes and objects are combined to form larger structures.

**Structural class patterns** use inheritance to combine the interfaces or implementations of multiple classes. Structural class patterns are relatively rare.

**Structural object patterns** use object composition to combine the implementations of multiple objects.

**Behavioral Patterns**

These design patterns are specifically concerned with communication between objects.

**Architectural Patterns**

An architectural pattern is a general, reusable solution to a commonly occurring problem in software architecture within a given context. The architectural patterns address various issues in software engineering, such as computer hardware performance limitations, high availability and minimization of a business risk. Ex: ETL, NLP, Master data hub, Data mart etc.,

**J2EE Patterns**

These design patterns are concerned with specific J2EE challenges and how they can be addressed.

1. **Creational Patterns**
   1. **Factory Pattern**

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

Factory pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method.

Steps:

Create an interface.

Create concrete classes implementing the same interface.

Create a Factory to generate object of concrete class based on given information.

Use the Factory to get object of concrete class by passing an information such as type.

* 1. **Abstract Factory Pattern**

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called as factory of factories. 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.

Steps:

Create an interface for Shapes.

Create concrete classes implementing the same interface.

Create an interface for Colors.

Create concrete classes implementing the same interface.

Create an Abstract class to get factories for Color and Shape Objects.

Create Factory classes extending AbstractFactory to generate object of concrete class based on given information. (ShapeFactory, ColorFactory)

Create a Factory generator/producer class to get factories by passing an information such as Shape or Color

Use the FactoryProducer to get AbstractFactory in order to get factories of concrete classes by passing an information such as type. (FactoryProducer.getFactory("SHAPE");, FactoryProducer.getFactory("COLOR");)

* 1. **Singleton Pattern**

Ensure a class only has one instance, and provide a global point of access to it. This is useful when exactly one object is needed to coordinate actions across the system.

* 1. **Builder Pattern**

Allows you to create different flavors of an object while avoiding constructor pollution. Useful when there could be several flavors of an object. Or when there are a lot of steps involved in creation of an object. (telescoping constructor anti-pattern)

Builder pattern builds a complex object using simple objects and using a step by step approach. A Builder class builds the final object step by step. This builder is independent of other objects. Separate the construction of a complex object from its representation so that the same construction process can create different representations.

* 1. **Prototype Pattern**

Prototype pattern refers to creating duplicate object while keeping performance in mind i.e., create object based on an existing object through cloning. This pattern involves implementing a prototype interface which tells to create a clone of the current object. 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.

1. **Structural Patterns**
   1. **Adapter Pattern**

Adapter pattern works as a bridge between two incompatible interfaces. Interfaces may be incompatible but the inner functionality should suit the need. The Adapter design pattern allows otherwise incompatible classes to work together by converting the interface of one class into an interface expected by the clients. There are two variations of the Adapter pattern:

The class adapter implements the adaptee's interface whereas the object adapter uses composition to contain the adaptee in the adapter object.

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.

* 1. **Bridge Pattern**

Bridge pattern is about preferring composition over inheritance. The bridge pattern is a design pattern used in software engineering that is meant to "decouple an abstraction from its implementation so that the two can vary independently". 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.

An abstraction and its implementation should be defined and extended independently from each other.

A compile-time binding between an abstraction and its implementation should be avoided so that an implementation can be selected at run-time.

* 1. **Composite Pattern**

The composite pattern is a partitioning design pattern. The composite pattern describes that a group of objects is to be treated in the same way as a single instance of an object. The intent of a composite is to "compose" objects into tree structures to represent part as well as whole hierarchy. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.

* 1. **Decorator Pattern**

Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This pattern creates a decorator class which wraps the original class and provides additional functionality keeping class methods signature intact.

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

* 1. **Flyweight Pattern**

Flyweight pattern is primarily used to reduce the number of objects created and to decrease 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.

* 1. **Proxy Pattern**

In proxy pattern, a class represents functionality of another class. A proxy, in its most general form, is a class functioning as an interface to something else. A proxy is a wrapper or agent object that is being called by the client to access the real serving object behind the scenes. Use of the proxy can simply be forwarding to the real object, or can provide additional logic. In the proxy extra functionality can be provided, for example caching when operations on the real object are resource intensive, or checking preconditions before operations on the real object are invoked.

* 1. **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.

1. **Behavioral Patterns**
   1. **Chain of responsibility**

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. 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.

The chain-of-responsibility pattern is a design pattern consisting of a source of command objects and a series of processing objects. Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain.

* 1. **Command**

Command pattern is a data driven design pattern in 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.

Four terms always associated with the command pattern are command, receiver, invoker and client. A command object knows about receiver and invokes a method of the receiver. Values for parameters of the receiver method are stored in the command, the receiver object to execute these methods is also stored in the command object by aggregation. The receiver then does the work when the execute() method in command is called. An invoker object knows how to execute a command, and optionally does bookkeeping about the command execution. The invoker does not know anything about a concrete command, it knows only about the command interface. Invoker object(s), command objects and receiver objects are held by a client object, the client decides which receiver objects it assigns to the command objects, and which commands it assigns to the invoker. The client decides which commands to execute at which points. To execute a command, it passes the command object to the invoker object.

* 1. **Interpreter**

Interpreter pattern provides a way to evaluate language grammar or expression. This pattern involves implementing an expression interface which tells to interpret a particular context. This pattern is used in SQL parsing, symbol processing engine etc. Interpreter pattern is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar.

* 1. **Iterator**

Iterator 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.

* 1. **Mediator**

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.

* 1. **Memento**

Memento pattern is used to restore state of an object to a previous state. Intent - Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

The Memento pattern is implemented with three objects: the originator, a caretaker and a memento. The originator is some object that has an internal state. The caretaker is going to do something to the originator, but wants to be able to undo the change. The caretaker first asks the originator for a memento object. Then it does whatever operation (or sequence of operations) it was going to do. To roll back to the state before the operations, it returns the memento object to the originator. The memento object itself is an opaque object (one which the caretaker cannot, or should not, change).

* 1. **Observer**

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.

The Observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods. It is mainly used to implement distributed event handling systems.

* 1. **State**

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

* 1. **Strategy**

In Strategy pattern, a class behavior or its algorithm can be changed at run time. Intent is to define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

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.

* 1. **Template method**

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. Intent is to define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

* 1. **Visitor**

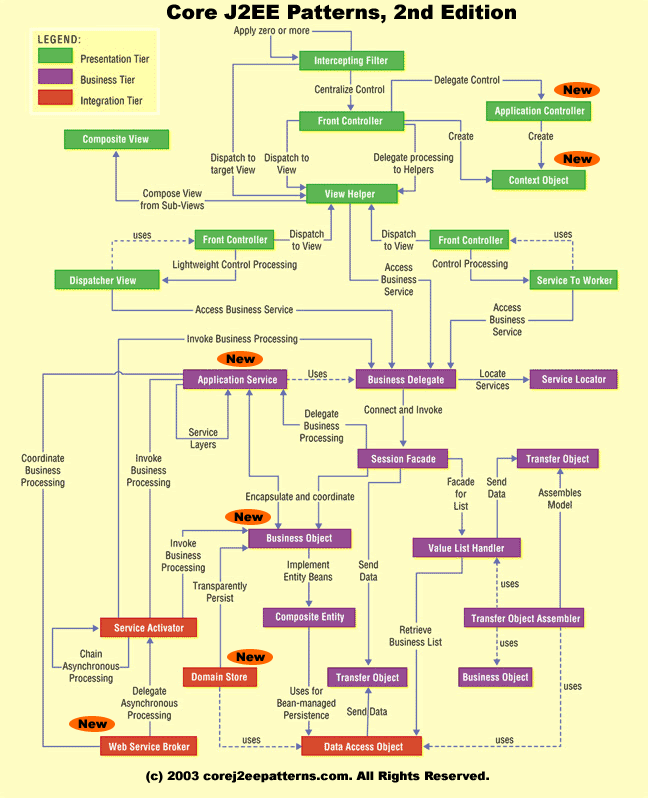
In Visitor pattern, we use a visitor class which changes the executing algorithm of an element class. By this way, execution algorithm of element can vary as and when visitor varies. As per the pattern, element object has to accept the visitor object so that visitor object handles the operation on the element object Intent is to represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

* 1. **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 behavior 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 implementation of this class and will be used seamlessly where we need to check null value.

1. **J2EE Pattern**



**Presentation Tier**

**Intercepting Filter** 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** 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** 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** 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** 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** 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.

**Business Tier**

**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** 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** 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** 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** 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** 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** 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** 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** 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.

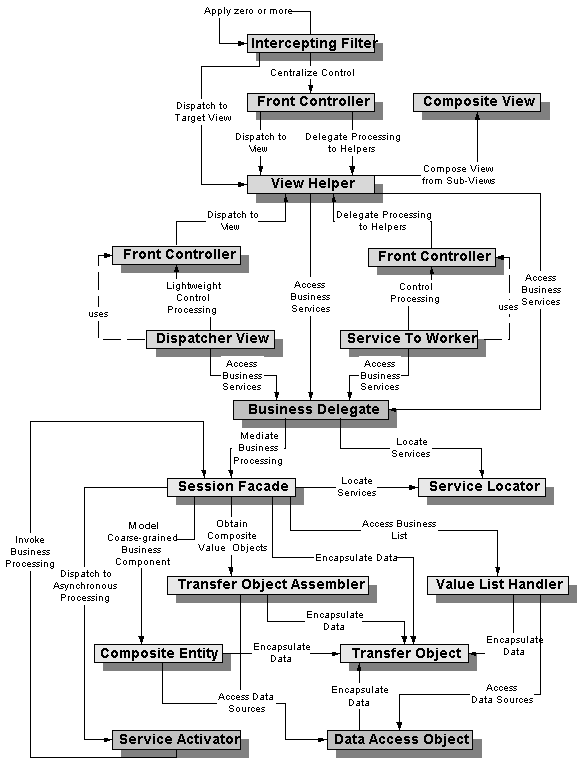
**Integration Tier**

**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** 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** 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** 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.



**Intercepting Filter**

EX: Logging and Authentication

Context : The presentation-tier request handling mechanism receives many different types of requests, preprocessing and post-processing of a client Web request and response are required.

**Create pluggable filters to process common services in a standard manner without requiring changes to core request processing code. The filters intercept incoming requests and outgoing responses, allowing preprocessing and post-processing. We are able to add and remove these filters unobtrusively, without requiring changes to our existing code.**

The responsibilities of the components participating in this patterns are :

FilterManager :The FilterManager manages filter processing. It creates the FilterChain with the appropriate filters, in the correct order, and initiates processing.

FilterChain : The FilterChain is an ordered collection of independent filters.

Filters : These are the individual filters that are mapped to a target. The FilterChain coordinates their processing.

Target : The Target is the resource requested by the client.

Strategies- available implementation strategies are

*Custom Filter Strategy*: Filter is implemented via a custom strategy defined by the developer. To implement the Custom Filter Strategy, the developer could use the Decorator pattern to wrap filters around the core request processing logic.

*Standard Filter Strategy* : Filters are controlled declaratively using a deployment descriptor, as described in the servlet specification version 2.3. The Servlet 2.3 specification includes a standard mechanism for building filter chains and unobtrusively adding and removing filters from those chains.

*Base Filter Strategy*: A base filter serves as a common superclass for all filters. Common features can be encapsulated in the base filter and shared among all filters.

*Template Filter Strategy*: In this case, the base filter is used to dictate the general steps that every filter must complete, while leaving the specifics of how to complete that step to each filter subclass. Typically, these would be coarsely defined, basic methods that simply impose a limited structure on each template. This strategy can be combined with any other filter strategy, as well.

**Front Controller Pattern**

Context :The presentation-tier request handling mechanism must control and coordinate processing of each user across multiple requests. Such control mechanisms may be managed in either a centralized or decentralized manner.

**Use a controller as the initial point of contact for handling a request. Typically, a controller coordinates with a dispatcher component. Dispatchers are responsible for view management and navigation. Thus, a dispatcher chooses the next view for the user**

The responsibilities of the components participating in this patterns are :

Controller

The controller is the initial contact point for handling all requests in the system. The controller may delegate to a helper to complete authentication and authorization of a user.

Dispatcher

A dispatcher is responsible for view management and navigation, managing the choice of the next view to present to the user. The dispatcher uses the RequestDispatcher object.

Helper

A helper is responsible for helping a view or controller complete its processing. Helpers can service requests for data from the view by simply providing access to the raw data or by formatting the data as Web content.

View

A view represents and displays information to the client. The view retrieves information from a model. Helpers support views by encapsulating and adapting the underlying data model for use in the display.

Strategies for implementation

Servlet Front Strategy

This strategy suggests implementing the controller as a servlet.

JSP Front Strategy

This strategy suggests implementing the controller as a JSP page.

Command and Controller Strategy

Based on the Command pattern [GoF], the Command and Controller Strategy suggests providing a generic interface to the helper components to which the controller may delegate responsibility

Physical Resource Mapping Strategy

All requests are made to specific physical resource names rather than logical names. An example is the following URL: http://some.server.com/resource1.jsp.

Logical Resource Mapping Strategy

Requests are made to logical resource names rather than to specific physical names. The physical resources to which these logical names refer may then be modified in a declarative manner.

For example, the URL http://some.server.com/process may be mapped as follows:

process=resource1.jsp

Multiplexed Resource Mapping Strategy

This is actually a substrategy of Logical Resource Naming Strategy. This strategy maps not just a single logical name, but an entire set of logical names, \*.ctrl = servletController

Base Front Strategy

Used in combination with the Servlet Front Strategy, this strategy suggests implementing a controller base class, whose implementation other controllers may extend.

Filter Controller Strategy

Filters provide similar support for centralizing request processing control (see Intercepting Filter pattern). Thus, some aspects of a controller can reasonably be implemented as a filter.

**View Helper Design Pattern**

Context: The system creates presentation content, which requires processing of dynamic business data.

Intermingling the business and systems logic with the view processing reduces modularity and also provides a poor separation

**A view contains formatting code, delegating its processing responsibilities to its helper classes, implemented as JavaBeans or custom tags. Helpers also store the view's intermediate data model and serve as business data adapters.**

Participants :

View

A view represents and displays information to the client. The information that is used in a dynamic display is retrieved from a model. Helpers support views by encapsulating and adapting a model for use in a display.

Helper

A helper is responsible for helping a view or controller complete its processing. Thus, helpers have numerous responsibilities, including gathering data required by the view and storing this intermediate model, in which case the helper is sometimes referred to as a value bean.

ValueBean

A value bean is another name for a helper that is responsible for holding intermediate model state for use by a view.

BusinessService

The business service is a role that is fulfilled by the service the client is seeking to access. Typically, the business service is accessed via aBusiness delegate.

Strategies

JSP View Strategy

The JSP View Strategy suggests using a JSP as the view component.

Servlet View Strategy

The Servlet View Strategy utilizes a servlet as the view.

JavaBean Helper Strategy

The helper is implemented as a JavaBean. Using helpers results in a cleaner separation of the view from the business processing in an application,

Custom Tag Helper Strategy

The helper is implemented as a custom tag (more upfront work than does the JavaBean Helper Strategy, since custom tag development is moderately complicated )

Business Delegate as Helper Strategy

Helper components often make distributed invocations to the business tier. We suggest using a business delegate in order to hide the underlying implementation details of this request, such that the helper simply invokes a business service without knowing details about its physical implementation and distribution

Transformer Helper Strategy

The helper is implemented as an eXtensible Stylesheet Language Transformer. This is particularly useful with models that exist as structured markup, such as eXtensible Markup Language (XML), either natively within legacy systems or via some form of conversion

**CompositeView Design Pattern**

Context : Sophisticated Web pages present content from numerous data sources, using multiple subviews that comprise a single display page. Additionally, a variety of individuals with different skill sets contribute to the development and maintenance of these Web pages.

**Use composite views that are composed of multiple atomic subviews. Each component of the template may be included dynamically into the whole and the layout of the page may be managed independently of the content.**

Participants

Composite View

A composite view is a view that is an aggregate of multiple subviews.

View Manager

The View Manager manages the inclusion of portions of template fragments into the composite view. The View Manager may be part of a standard JSP page runtime engine, in the form of the standard JavaServer Pages (JSP page) pages include tag (<jsp:include>).

Adv : reusable page layouts.

Included View

An included view is a subview that is one atomic piece of a larger whole view. This included view could also potentially be a composite, itself including multiple subviews.

Strategies :

JSP page View Strategy

Servlet View Strategy

JavaBean View Management Strategy

View management is implemented using JavaBeans components. The view delegates to the JavaBean, which implements the custom logic to control view layout and composition.

Standard Tag View Management Strategy

View management is implemented using standard JSP page page tags, such as the <jsp:include> tag

Custom Tag View Management Strategy

View management is implemented using custom tags , which is the preferred strategy.

Transformer View Management Strategy

View management is implemented using an XSL Transformer.

Early-Binding Resource Strategy

This is another name for translation-time content inclusion, as described in the Standard Tag View Management Strategy. recommended if a view includes headers and footers that change infrequently.

Late-Binding Resource Strategy

This is another name for runtime-content inclusion, as described in the Standard Tag View Management Strategy

**Dispatcher View -** The Dispatcher View pattern, like the Service to Worker pattern, describes a common combination of other patterns from the catalog. Both of these macro patterns describe the combination of a controller and dispatcher with views and helpers

Intermingling business logic with view processing also reduces modularity and provides a poor separation of roles among Web production and software development teams.

**Combine 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.**

**Service to Worker**

**Combine a controller and dispatcher with views and helpers (see "Front Controller" on page 172 and "View Helper" on page 186) to handle client requests and prepare a dynamic presentation as the response. Controllers delegate content retrieval to helpers, which manage the population of the intermediate model for the view. A dispatcher is responsible for view management and navigation and can be encapsulated either within a controller or a separate component.**

in the Service to Worker pattern, the dispatcher might be more sophisticated. The dispatcher may invoke a business service to determine the appropriate view to display.

**Business Delegate**

Problem : Presentation-tier components interact directly with business services. This direct interaction exposes the underlying implementation details of the business service application program interface (API) to the presentation tier.

**Use a Business Delegate to reduce coupling between presentation-tier clients and business services. The Business Delegate hides the underlying implementation details of the business service, such as lookup and access details of the EJB architecture.**

Component s :

BusinessDelegate

The BusinessDelegate's role is to provide control and protection for the business service.

LookupService

The BusinessDelegate uses the LookupService to locate the BusinessService. The LookupService encapsulates the implementation details of BusinessService lookup.

BusinessService

The BusinessService is a business-tier component, such as an enterprise bean or a JMS component, that provides the required service to the client.

**Session Facade**

Enterprise beans encapsulate business logic and business data and expose their interfaces, and thus the complexity of the distributed services, to the client tier.

**Use a session bean as a facade to encapsulate the complexity of interactions between the business objects participating in a workflow. The Session Facade manages the business objects, and provides a uniform coarse-grained service access layer to clients.**

**Service Locator**

Service lookup and creation involves complex interfaces and network operations.

**Use a Service Locator object to abstract all JNDI usage and to hide the complexities of initial context creation, EJB home object lookup, and EJB object re-creation. Multiple clients can reuse the Service Locator object to reduce code complexity, provide a single point of control, and improve performance by providing a caching facility.**

**Transfer Object Assembler**

server-side business components are implemented using session beans, entity beans, DAOs, and so forth. Application clients frequently need to access data that is composed from multiple objects.

**Use a Transfer Object Assembler to build the required model or submodel. The Transfer Object Assembler uses Transfer Objects to retrieve data from various business objects and other objects that define the model or part of the model.**

**Transfer Object**

Application clients need to exchange data with enterprise beans.

**Use a Transfer Object to encapsulate the business data. A single method call is used to send and retrieve the Transfer Object. When the client requests the enterprise bean for the business data, the enterprise bean can construct the Transfer Object, populate it with its attribute values, and pass it by value to the clien**

**Value List Handler**

The client requires a list of items from the service for presentation. The number of items in the list is unknown and can be quite large in many instances.

**Use a Value List Handler to control the search, cache the results, and provide the results to the client in a result set whose size and traversal meets the client's requirements.**

This pattern creates a ValueListHandler to control query execution functionality and results caching. The ValueListHandler directly accesses a DAO that can execute the required query. The ValueListHandler stores the results obtained from the DAO as a collection of Transfer Objects. The client requests the ValueListHandler to provide the query results as needed. The ValueListHandler implements an Iterator pattern [GoF] to provide the solution.

**Composite Entity**

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.

**Use Composite Entity to model, represent, and manage a set of interrelated persistent objects rather than representing them as individual fine-grained entity beans. A Composite Entity bean represents a graph of objects. - Drastically reduces the number of entities being used in application.**

**Service Activator**

Enterprise beans and other business services need a way to be activated asynchronously.

**Use a Service Activator to receive asynchronous client requests and messages. On receiving a message, the Service Activator locates and invokes the necessary business methods on the business service components to fulfill the request asynchronously.**

**Data Access Object**

**Use a Data Access Object (DAO) to abstract and encapsulate all access to the data source. The DAO manages the connection with the data source to obtain and store data.Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, this pattern allows the DAO to adapt to different storage schemes without affecting its clients or business components. Essentially, the DAO acts as an adapter between the component and the data source.**

**Correlate J2EE Pattern with Software development pattern.**

Session Facade (Singleton)

Value Object Assembler (Bridge)

Service Locator Pattern (Prototype)

Business Delegate (Abstract Factory)

Composite Entity (Flyweight)

Value List Handler (Mediator)

Service Locator (Strategy)

Composite Entity (Decorator)

Value Object (State)

Service to Worker (Iterator)

Data Access Object (Chain of Responsibility)

Intercepting Filter (Model View Controller II)

View Helper (Memento)

Composite View (Builder)

Dispatcher View (Factory Method)