J2EE Design Patterns

**Presentation Tier**

## **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.**

**Intercepting Filter**

**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. To summarize, the Intercepting Filter pattern provides the ability to manipulate a request prior to processing or to manipulate the response before sending the results of the request.

## Benefits

The following lists the benefits of using the Intercepting Filter pattern:

* Centralizes pre-processing of requests
* Centralizes post-processing of responses

## When to Use

You should use the Intercepting Filter pattern when:

* You need to pre-process a request or response.
* You need to post-process a request or response.

### Front Controller

**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. The Front Controller pattern creates central control logic for presentation request handling. The Front Controller is different from the Intercepting Filter in that the Front Controller is determining processing based on the request and an Intercepting Filter is modifying the request.

## Benefits

The following lists the benefits of using the Front Controller pattern:

* Centralizes control logic
* Improves reusability
* Improves separation of concerns

## When to Use

You should use the Front Controller pattern to:

* Apply common logic to multiple requests
* Separate processing logic from view

### Application Controller

**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. The Application Controller pattern is used to centralize retrieval and invocation of request-processing components, such as commands and views.

## Benefits

The following lists the benefits of using the Application Controller pattern:

* Improves extensibility
* Improves separation of concerns

## When to Use

You should use the Application Controller pattern to:

* Apply common control logic
* Have centralized view management

### Context Object

**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. The Context Object pattern is used to encapsulate the specifics of protocol implementation to be shared.

## Benefits

The following lists the benefits of using the Context Object pattern:

* Improves reusability and maintainability
* Allows code to be portable across operating systems

## When to Use

You should use the Context Object pattern when:

* Components need access to system information
* Decouple application from underlining protocols and system interfaces

### View Helper

**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. The View Helper pattern separates the processing logic from the view.

## Benefits

The following is a benefit of using the View Helper pattern:

* Separates logic from the view

## When to Use

You should use the View Helper pattern to:

* Encapsulate view-processing logic

### Composite View

**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. The Composite View pattern combines simple views into a more complex view without handling the content or layout.

## Benefits

The following lists the benefits of using the Composite View pattern:

* Code duplication is reduced because you can create common headers, footers, and other components.
* Views can be changed based on access authorization.

## When to Use

You should use the Composite View pattern when:

* You want common view components.
* You view component changes based on authorization.

### Service to Worker

**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. The Service to Worker pattern performs request handling and invokes business logic before control is passed to the view.

## Benefits

The following is a benefit of using the Service to Worker pattern:

* Improves separation of concerns

## When to Use

You should use the Service to Worker pattern to:

* Centralize business logic for requests

### Dispatcher View

**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. The Dispatcher View pattern handles the request and generates a response while managing limited business processing.

## Benefits

The following lists the benefits of using the Dispatcher View pattern:

* Separates processing logic from view
* Improves reusability

## When to Use

You should use the Dispatcher View pattern when:

* You have static views.
* You have limited business processing.

### Business Tier Patterns

## ***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.**

### Business Delegate

**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. The Business Delegate pattern hides the complexity of remote communication with business components from the client.

## Benefits

The following lists the benefits of using the Business Delegate pattern:

* Minimizes coupling of clients to business services
* Hides remoteness
* Improves performance

## When to Use

You should use the Business Delegate pattern when you:

* Want to encapsulate access to business services from multiple client types
* Translate exceptions into application exceptions
* Hide details of service creation

### Service Locator

**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. The Service Locator pattern uses a consistent approach to locating business components regardless of the type of components.

## Benefits

The following is a benefit of using the Service Locator pattern:

* Standardized approach to retrieving business components

## When to Use

You should use the Service Locator pattern when:

* You have many different business services that are located in different ways.

### Session Facade

**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. The Session Façade pattern provides a coarse-grained service of business components to remote clients. This is the same as a Façade pattern as described in [GOF Design Patterns](http://www.javagyan.com/system/errors/NodeNotFound?suri=wuid:gx:2edba72fc01ad874), but just provides an interface to a service instead of code.

## Benefits

The following lists the benefits of using the Session Facade pattern:

* Reduces the number of calls to the business component from the client
* Reduces coupling between the tiers
* Improves performance by reducing fine-grained calls from client
* Provides a cleaner API to the client

## When to Use

* You should use the Session Facade pattern when:
* You have a series of calls to make to business components from the client.

### Application Service

**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. The Application Service pattern centralizes and aggregates business components. An application service could be thought of as a helper to the Session Façade that takes care of all the business logic and work flow.

## Benefits

The following lists the benefits of using the Application Service pattern:

* Centralizes and improves reusability of business logic
* Simplifies the Session Façade by eliminating the business logic

## When to Use

You should use the Application Service pattern when you:

* Start to see duplicated business logic in the Session Façade

### Business Object

**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. The Business Object pattern separates business data from logic.

## Benefits

The following is a benefit of using the Business Object pattern:

* Separates persistence from business logic.

## When to Use

You should use the Business Object pattern when you:

* Want to increase reusability of business logic.

### Composite Entity

**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 Composite Entity pattern aggregates business entities into a coarse grained entity.

## Benefits

The following lists the benefits of using the Composite Entity pattern:

* Increases maintainability
* Improves network performance

## When to Use

You should use the Composite Entity pattern to:

* Avoid remote entity beans
* Leverage bean managed persistence (BMP) with custom persistence implementation
* Encapsulate POJO business objects

### Transfer Object

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 pattern uses an object to carry data across tiers.

## Benefits

The following lists the benefits of using the Transfer Object pattern:

* Reduces network traffic
* Reduces code duplication

## When to Use

You should use the Transfer Object pattern when:

* You need to send objects between tiers.

### Transfer Object Assembler

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 Transfer Object Assembler pattern builds a composite transfer object and returns to the client.

## Benefits

The following is a benefit of using the Transfer Object Assembler pattern:

* Improves network performance

## When to Use

You should use the Transfer Object Assembler pattern when:

* You have several transfer objects that are sent between tiers.

### Value List Handler

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. The Value List Handler pattern caches results and allows the client to traverse and select from the results.

## Benefits

The following lists the benefits of using the Value List Handler pattern:

* Caches search results
* Improves network performance
* Improves separation of concerns

## When to Use

You should use the Value List Handler pattern when you:

* Want to iterate through a set of objects.
* Implement read-only lists without transactions.

### Integration Tier Patterns

## **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.**

Data Access Object

**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. The Data Access Object pattern encapsulates access to a persistent store by managing the connection with the data store.

## Benefits

The following lists the benefits of using the Data Access Object pattern:

* Reduces code complexity in client
* Improves code reuse
* Provides easier migration to new data store

## When to Use

You should use the Data Access Object pattern to:

* Decouple data access from the business logic
* Provide all data access from in a separate layer

### Service Activator

**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. The Service Activator pattern handles asynchronous requests to business components.

## Benefits

The following lists the benefits of using the Service Activator pattern:

* Allows the client to continue processing
* Integrates JMS into application

## When to Use

* You should use the Service Activator pattern when:
* You need to invoke a business service in an asynchronous manner.

### Domain Store

**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. The Domain Store pattern separates the persistence of an object from the object model. This pattern really became relevant with the advent of object relational model frameworks and products. You would use the domain store and data access object at the same time.

## Benefits

The following is a benefit of using the Domain Store pattern:

* Decouples business logic from persistence logic

## When to Use

You should use the Domain Store pattern when:

* You do not want to use entity beans.
* Object model uses are complex.

### Web Service Broker

**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. The Web Service Broker pattern exposes and brokers services using XML and web protocols.

## Benefits

The following is a benefit of using the Web Service Broker pattern:

* Exposes existing services to web

## When to Use

You should use the Web Service Broker pattern when you:

* Need to expose services as web services