Study of J2EE Patterns

Anshuman Tiwari 04329004
Kaushal Mittal 04329024

Abstract: In this report we are giving introduction to common patterns of J2EE application with implementation of a web-centric application. The J2EE platform is designed to provide server-side and client-side support for developing distributed, multi tier application. Commonly we separate a web-based application into three tier architecture. View layer and Controller layer comes under the ***presentation tier***, used to interact with the client. The ***Business tier*** deals with business logic and the data accessing from database. The ***Integration tier*** is a legacy system contains database like thing.

1. Model-View-Controller (MVC)

Model-View-Controller architecture is used for interactive web-applications. This model minimizes the coupling between business logic and data presentation to web user. This model divides the web based application into three layers:

1. **Model:** Model domain contains the business logics and functions that manipulate the business data. It provides updated information to view domain and also gives response to query. And the controller can access the functionality which is encapsulated in the model.
2. **View:** View is responsible for presentation aspect of application according to the model data and also responsible to forward query response to the controller.
3. **Controller:** Controller accepts and intercepts user requests and controls the business objects to fulfill these requests. An application has one controller for related functionality. Controller can also be depends on the type of clients.

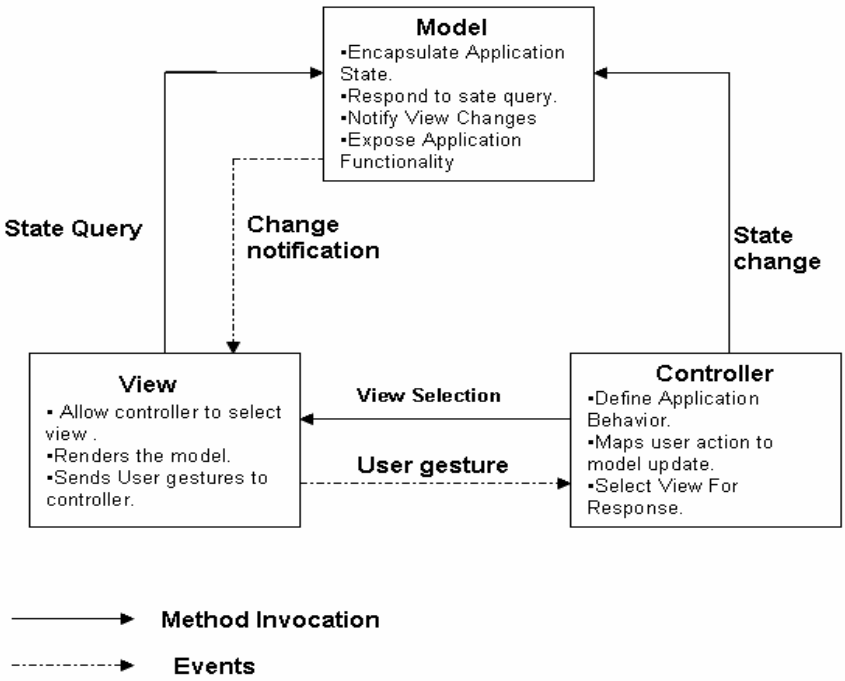

Figure 1: MVC Architecture

2. Intercepting Pattern

- **Pattern Context:** Presentation layer has different types of requests having different types of processing. Some of the requests require some preprocessing.

- **Problem:** The request should satisfy several constraints before get enter to the system like client must be authenticate, request came from a trusted network or not, Client has valid session or not.

- **Solution:** plugged a simple filter which will check all the constraints without any change in core request processing.

- **Class Diagram**

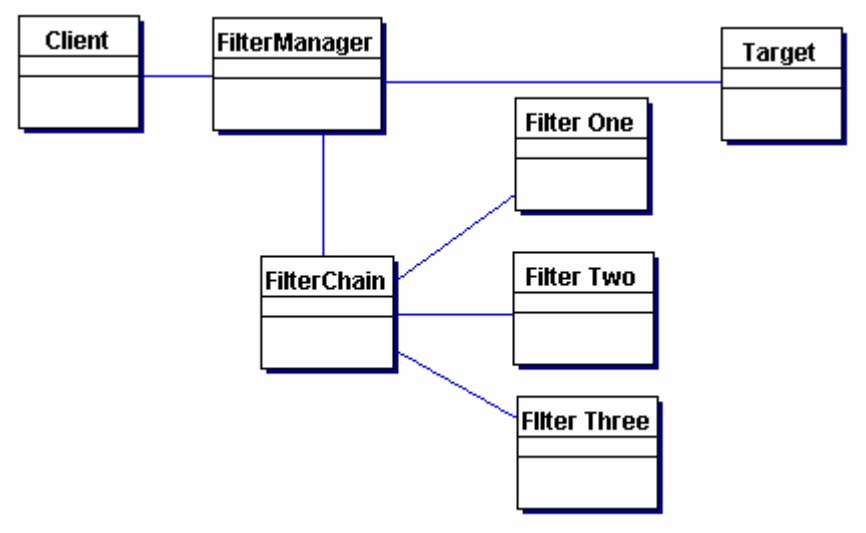

Figure 2: Intercepting pattern class diagram

- **Consequences:**
  - Filters provide a centralized approach for handling processing across multiple requests.
  - Filters create application partitioning and increase reuse due to their standard interface.
  - Numerous services are combined in varying permutations without a single recompile of the core code base.

3. Front Controller Pattern

- **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**.**

- **Problem:** In centralized approach each view is required to provide its own system services, often resulting in duplicate code.

- **Solution:** Use a controller as the initial point of contact for handling a request.

- **Class Diagram:**

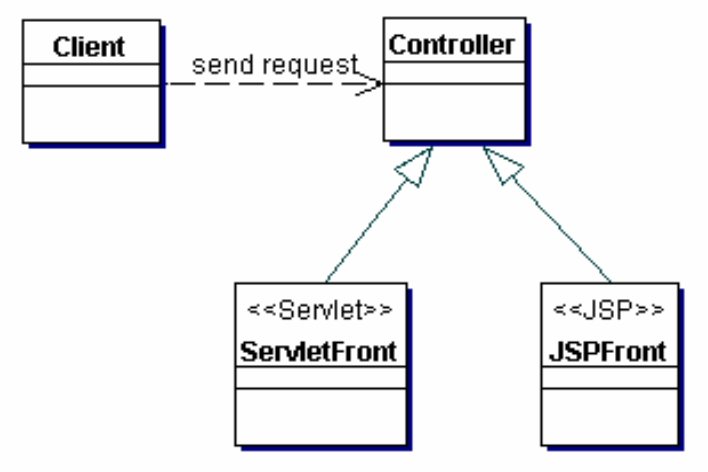

Figure 3: Front Controller pattern class diagram

- **Consequences:**

  - Centralized control: A controller provides a central place to handle system services and business logic across multiple requests
  - Improves Manageability: Easy to monitor control flow.
  - Improves Reusability: A controller promotes cleaner application partitioning and encourages reuse, as code that is common among components moves into a controller or is managed by a controller.

4. View Helper

- **Pattern Context** - The system creates presentation content, which requires processing of dynamic business data.

- **Problem** – Presentation tier keeps on changing and it needs to deal with the business data. So the logic for formatting and presenting the business data is separated from the business tier into the view helper.

- **Solution** – The view uses the view helper to convert the business data into the presentable format. .

- **Class Diagram**

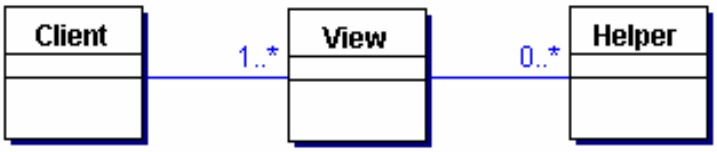

Figure 4: View Helper pattern class diagram

- **Consequences:**

  - Improves application partitioning, reuse and maintainability.
  - Improves role separation.
  - Easy testing

5. Composite view

- **Pattern Context**: A single view usually is composed of multiple views.
- **Problem:** Keeping multiple independent views increases problem of maintainability and reuse.
- **Solution:** The final view is composed of independent views that contain the response in presentable format.
- **Class Diagram**

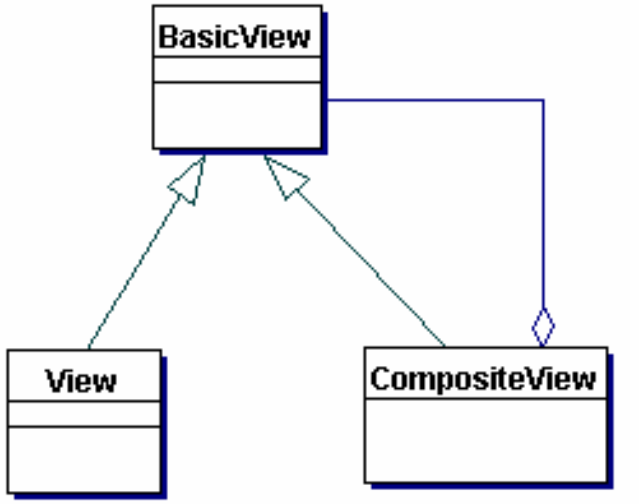

Figure 5: Composite pattern class diagram

- **Consequences:**

- Improves Modularity and Reuse
- Enhances Flexibility
- Enhances Maintainability and Manageability
- Reduces Manageability

6. Dispatcher view

- **Pattern Context**: System controls flow of execution and access to presentation processing, which is responsible for generating dynamic content.
- **Problem:** The problem is a combination of the problems solved by the Front Controller and View Helper patterns in the presentation tier. There is no centralized component for managing access control, content retrieval or view management

- **Solution:** Combine a controller and dispatcher with views and helpers to handle client requests and prepare a dynamic presentation as the response.

- **Class Diagram**

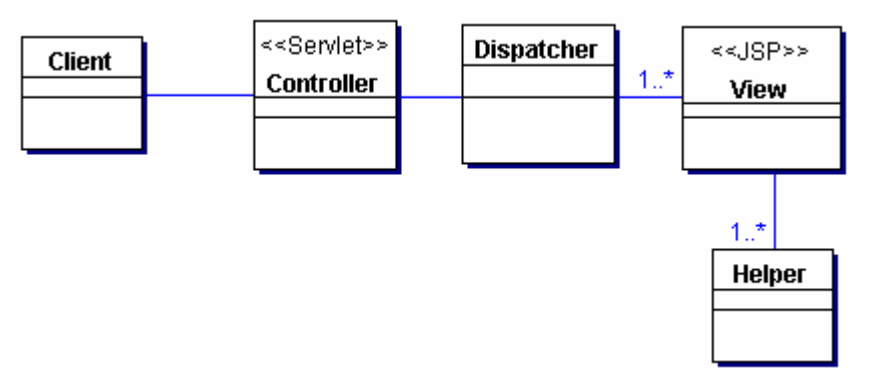

Figure 6: Dispatcher pattern class diagram

- **Consequences**:
  - Centralizes Control and Improves Reuse and Maintainability
  - Improves Application Partitioning
  - Improves Role Separation

7. Business delegate

- **Pattern Context:** A multi-tiered, distributed system requires remote method invocations to send and receive data across tiers.

- **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. As a result, the presentation-tier components are vulnerable to changes in the implementation of the business services: When the implementation of the business

services change, the exposed implementation code in the presentation tier must change too.

- **Solution:** 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.

- **Consequences:**
  - Reduces Coupling.
  - Improves Manageability.
  - Impacts Performance.
  - Hides Remoteness
- **Class Diagram**

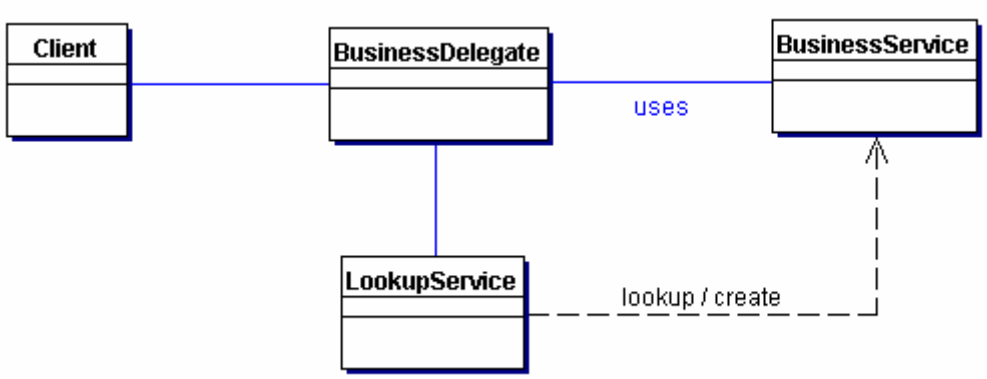

Figure 7: Business Delegate pattern class diagram

8. Service locator

- **Pattern Context:** Service lookup and creation involves complex interfaces and network operations.

- **Problem:** J2EE clients interact with service components, such as Enterprise JavaBeans (EJB) and Java Message Service (JMS) components, which provide business services and persistence capabilities. To interact with these components, clients must either locate the service component (referred to as a lookup operation) or create a new component.

- **Solution:** 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.

- **Class Diagram**

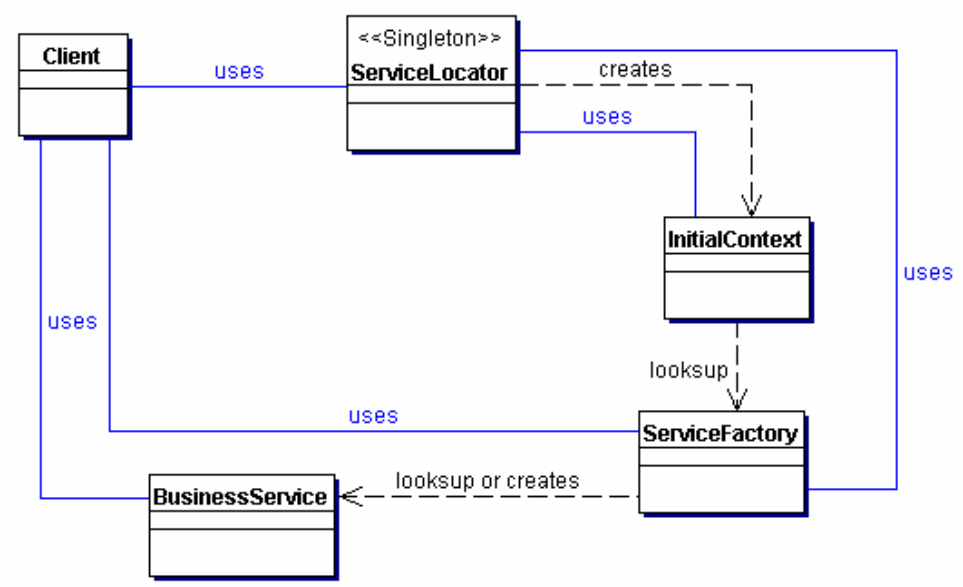

Figure 8: Service Locator pattern class diagram

- **Consequences:**
  - Abstracts Complexity.
  - Provides Uniform Service Access to Clients.
  - Improves Network Performance.

9. Session Façade

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

- **Problem:**
  - Tight coupling, which leads to direct dependence between clients and business objects;
  - Too many method invocations between client and server, leading to network performance problems;
  - Lack of a uniform client access strategy, exposing business objects to misuse.

- **Solution:** 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.

- **Class Diagram**

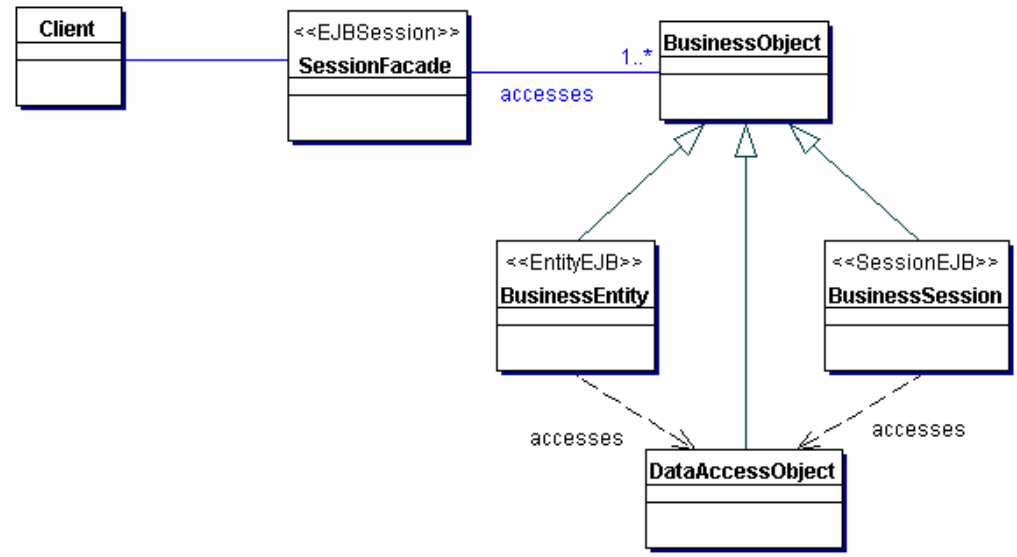

Figure 9: Session Facade pattern class diagram

- **Consequences:**
  - Introduces Business-Tier Controller Layer
  - Exposes Uniform Interface Reduces Coupling.
  - Increases Manageability

10. Composite Entity

- **Pattern 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.

- **Problem:** In a Java 2 Platform, Enterprise Edition (J2EE) application, clients -- such as applications, Java Server Pages (JSP) pages, servlets, and JavaBeans components -- access entity beans via their remote interfaces. Thus, every client invocation potentially routes through network stubs and skeletons, even if the client and the enterprise bean are in the same JVM, OS, or machine. When entity beans are fine-grained objects, clients tend to invoke more individual entity bean methods, resulting in high network overhead.

- **Solution:** 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.

- **Class Diagram**

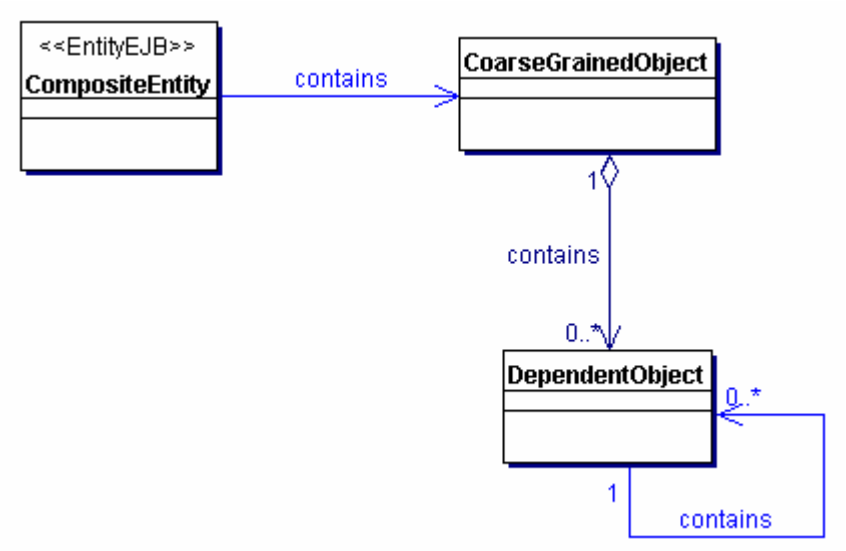

Figure 10: Composite Entity pattern class diagram

- **Consequences**:
  - Improves Network Performance
  - Overhead of Multi-level Dependent Object Graphs
  - Improves Manageability by Reducing Entity Beans

11. Value List Handler

- **Pattern Context:** 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.

- **Problem:** Most Java 2 Platform, Enterprise Edition (J2EE) applications have a search and query requirement to search and list certain data. In some cases, such a search and query operation could yield results that can be quite large. It is impractical to return the full result set when the client's requirements are to traverse the results, rather than process the complete set. Typically, a client uses the results of a query for read-only purposes, such as displaying the result list. Often, the client views only the first few matching records, and then may discard the remaining records and attempt a new query. The search activity often does not involve an immediate transaction on the matching objects. The practice of getting a list of values represented in entity beans by calling an `ejbFind()` method, which returns a collection of remote objects, and then calling each entity bean to get the value, is very network expensive and is considered a bad practice.

- **Solution:** 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.

- **Class Diagram:**

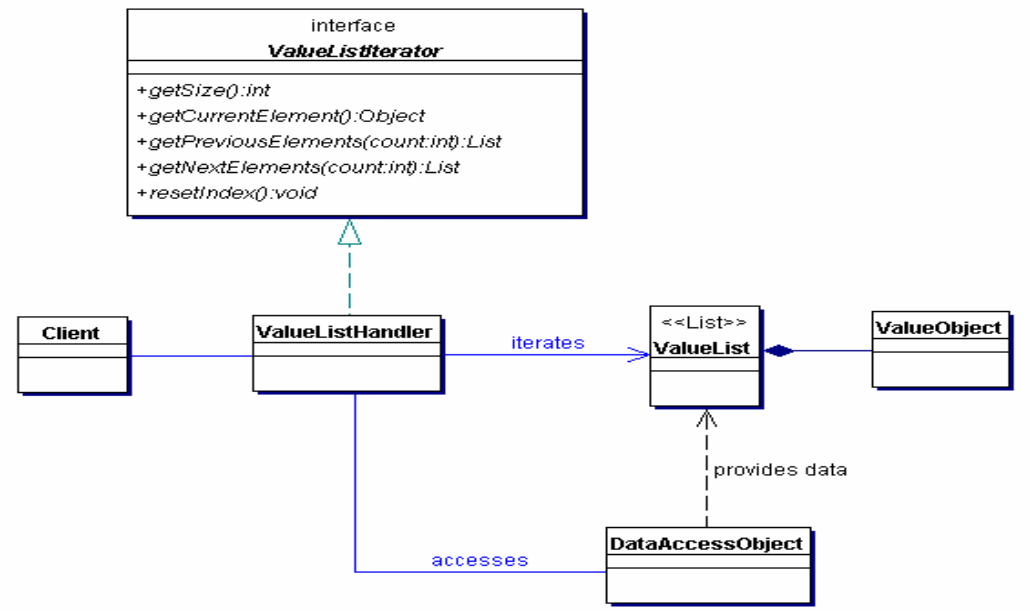

Figure 11: Value List Handler pattern class diagram

- **Consequences:**

  - Provides Alternative to EJB Finders for Large Queries
  - Caches Query Results on Server Side
  - Provides Better Querying Flexibility

12. Data Access Object

- **Pattern Context**: Access to data varies depending on the source of the data. Access to persistent storage, such as to a database, varies greatly depending on the type of storage (relational databases, object-oriented databases, flat files, and so forth) and the vendor implementation.

- **Problem:** Many real-worlds applications need to use persistent data at some point. For many applications, persistent storage is implemented with different mechanisms, and there are marked differences in the APIs used to access these different persistent storage mechanisms. Other applications may need to access data that resides on separate systems. For example, the data may reside in mainframe systems, Lightweight Directory Access Protocol (LDAP) repositories, and so forth. Another example is where data is provided by services through external systems such as business-to-business (B2B) integration systems, credit card bureau service, and so forth.

- **Solution:** 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.

- **Class Diagram:**

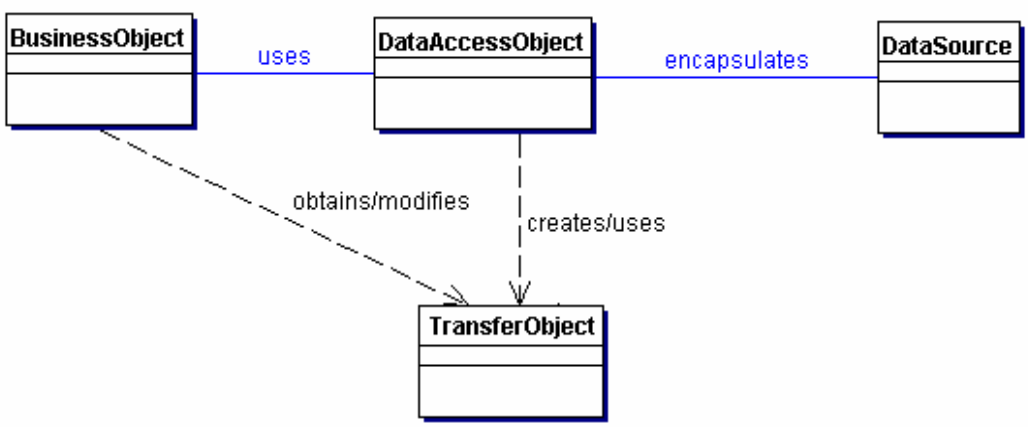

Figure 12: Data Access Object pattern class diagram

- **Consequences:**
  - Enables Transparency
  - Reduces Code Complexity in Business Objects
  - Adds Extra Layer
  - Centralizes All Data Access into a Separate Layer

Summary

Presentation Tier Patterns:

- *Intercepting Pattern:* Facilitates Pre processing and Post processing of a request.
- *View Helper:* provides a centralized controller for managing the handle of a request.
- *Composite View:* Creates an aggregate view from atomic components.
- *Service to Worker:* Combines a dispatcher component with the front controller and the view helper pattern.
- *Dispatch View:* Combines a dispatcher component with front controller pattern and the view helper pattern, deferring many activities to view processing.

Business logic Tier Patterns

- *Business Delegate:* Decouples presentation tier and service tier and provide a facade and proxy interface to the services.
- *Session Façade:* Hides business object complexities, Centralize workflow handling.

- *Value object:* Facilitates data exchanges between tiers by reducing network overhead.
- *Composite Entity:* Hides business object complexities, Centralize workflow handling.
- *Value object Assembler:* Assembles a composite value object from multiple data sources.
- *Value List Handler:* Manages queue execution, results caching, and results processing.
- *Service locator:* Encapsulate complexities of business service lookup and creation.

Integration Tier

- *Data Access objects:* Abstract data sources and provides transparent access to data.
- *Service Activator:* Activator facilitates asynchronous processing of EJB components.

Analysis

Pattern diagram for most of the web-centric applications are follows a general structure that is like this:

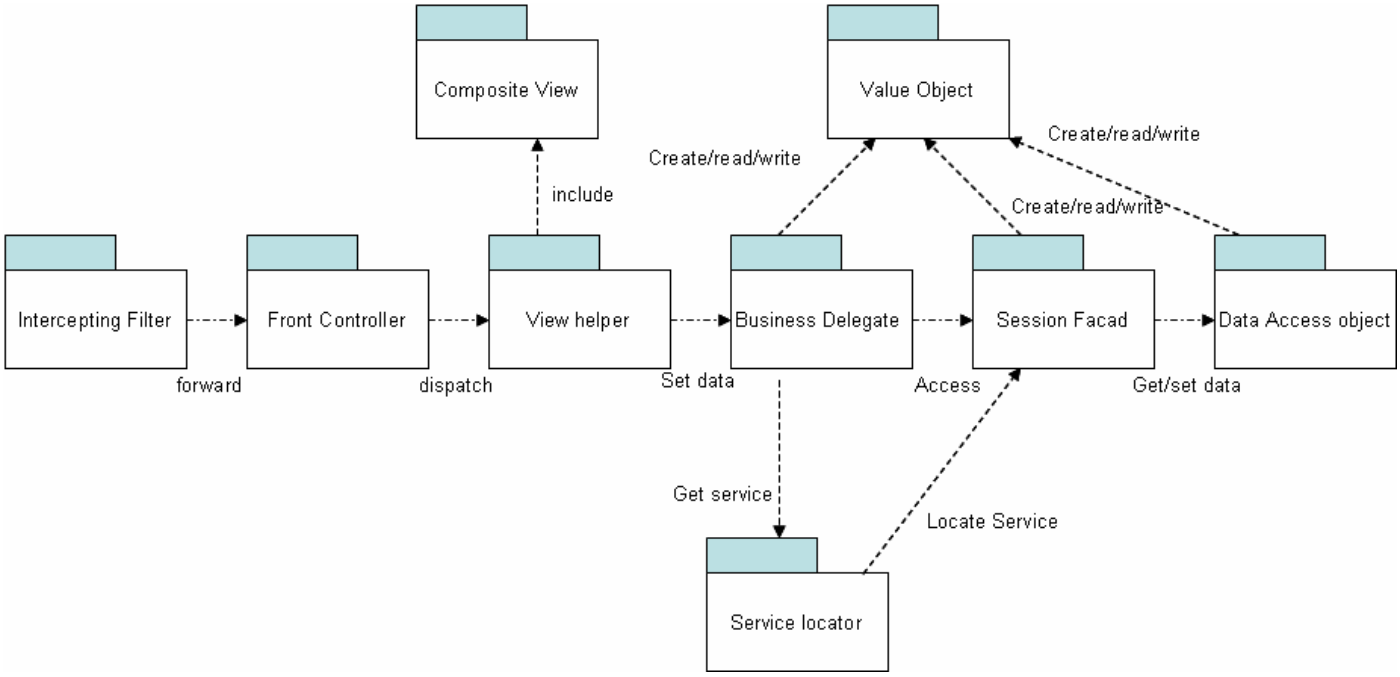

Figure 13: General Flow of pattern