Value Object Assembler

Context

In a J2EE application, the 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.

Problem

Application clients typically require the data for the model or parts of the model to present to the user or to use for an intermediate processing step before providing some service. The application model is an abstraction of the business data and business logic implemented on the server side as business components. A model may be expressed as a collection of objects put together in a structured manner (tree or graph). In a J2EE application, the model is a distributed collection of objects such as session beans, entity beans, or DAOs and other objects. For a client to obtain the data for the model, such as to display to the user or to perform some processing, it must access individually each distributed object that defines the model. This approach has several drawbacks:

- Because the client must access each distributed component individually, there is a tight coupling between the client and the distributed components of the model over the network
- The client accesses the distributed components via the network layer, and this can lead to performance degradation if the model is complex with numerous distributed components. Network and client performance degradation occur when a number of distributed business components implement the application model and the client directly interacts with these components to obtain model data from that component. Each such access results in a remote method call that introduces network overhead and increases the chattiness between the client and the business tier.
- The client must reconstruct the model after obtaining the model's parts from the distributed components. The client therefore needs to have the necessary business logic to construct the model. If the model construction is complex and numerous objects are involved in its definition, then there may be an additional performance overhead on the client due to the construction process. In addition, the client must contain the business logic to manage the relationships between the components, which results in a more complex, larger client. When the client constructs the application model, the construction happens on the client side. Complex model construction can result in a significant performance overhead on the client side for clients with limited resources.
- Because the client is tightly coupled to the model, changes to the model require changes to the client. Furthermore, if there are different types of clients, it is more difficult to manage the changes across all client types. When there is tight coupling between the client and model implementation, which occurs when the client has direct knowledge of the model and manages the business component relationships, then changes to the model necessitate changes to the client. There is the further problem of code duplication for model access, which occurs when an application has many types of clients. This duplication makes client (code) management difficult when the model changes.

Forces

- Separation of business logic is required between the client and the server-side components.
- Because the model consists of distributed components, access to each component is associated with a network overhead. It is desirable to minimize the number of remote method calls over the network.
- The client typically needs only to obtain the model to present it to the user. If the client must interact with multiple components to construct the model on the fly, the chattiness between the client and the application increases. Such chattiness may reduce the network performance.
- Even if the client wants to perform an update, it usually updates only certain parts of the model and not the entire model.
- Clients do not need to be aware of the intricacies and dependencies in the model implementation. It is desirable to have loose coupling between the clients and the business components that implement the application model.
- Clients do not otherwise need to have the additional business logic required to construct the model from various business components.

Solution

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

The Value Object Assember constructs a composite value object that represents data from different business components. The value object caries the data for the model to the client in a single method call. Since the model data can be complex, it is recommended that this value object be immutable. That is, the client obtains such value objects with the sole purpose of using them for presentation and processing in a read-only manner. Clients are not allowed to make changes to the value objects.

When the client needs the model data, and if the model is represented by a single coarse-grained component (such as a Composite Entity), then the process of obtaining the model data is simple. The client simply requests the coarse-grained component for its composite value object. However, most real-world applications have a model composed of a combination of many coarse-grained and fine-grained components. In this case, the client must interact with numerous such business components to obtain all the data necessary to represent the model. The immediate drawbacks of this approach can be seen in that the clients become tightly coupled to the model implementation (model elements) and that the clients tend to make numerous remote method invocations to obtain the data from each individual component.

In some cases, a single coarse-grained component provides the model or parts of the model as a single value object (simple or composite). However, when multiple components represent the model, a single value object (simple or composite) may not represent the entire model. To represent the model, it is necessary to obtain value objects from various components and assemble them into a new composite value object. The server, not the client, should perform such “on-the-fly” construction of the model.

Structure

Figure 1.1 shows the class diagram representing the relationships for the Value Object Assembler pattern.

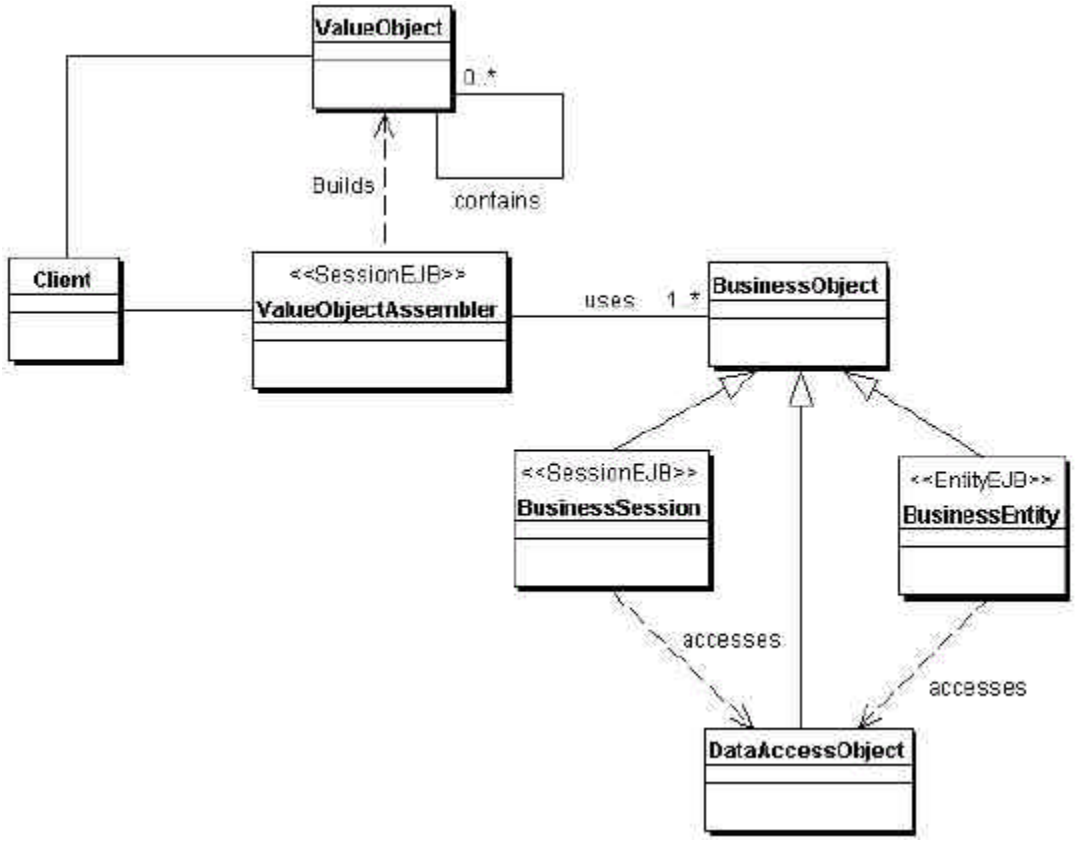

Figure 1.1 Value Object Assembler class diagram.

Participants and Responsibilities

The sequence diagram in Figure 1.2 shows the interaction between the various participants in the Value Object Assembler pattern.

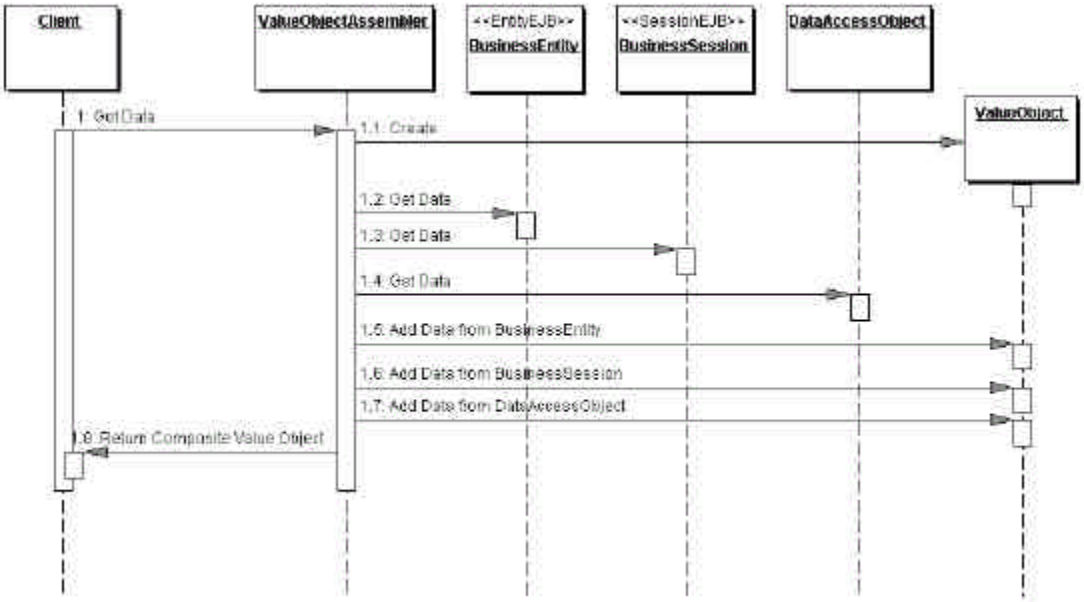

Figure 1.2 Value Object Assembler sequence diagram.

ValueObjectAssembler

The ValueObjectAssembler is the main class of this pattern. The ValueObjectAssembler constructs a new value object based on the requirements of the application when the client requests a composite value object. The ValueObjectAssembler then locates the required BusinessObject instances to retrieve data to build the composite value object. BusinessObjects are business-tier components such as entity beans and session beans, DAOs, and so forth.

Client

If the ValueObjectAssembler is implemented as an arbitrary Java object, then the client is typically a Session Facade that provides the controller layer to the business tier. If the ValueObjectAssembler is implemented as a session bean, then the client can be a Session Facade or a Business Delegate.

BusinessObject

The BusinessObject participates in the construction of the new value object by providing the required data to the ValueObjectAssembler. Therefore, the BusinessObject is a role that can be fulfilled by a session bean, an entity bean, a DAO, or a regular Java object.

ValueObject

The ValueObject is a composite value object that is constructed by the ValueObjectAssembler and returned to the client. This represents the complex data from various components that define the application model.

BusinessObject

BusinessObject is a role that can be fulfilled by a session bean, entity bean, or DAO. When the assembler needs to obtain data directly from the persistent storage to build the value object, it can use a DAO. This is shown as the DataAccessObject object in the diagrams.