Value Objects, VO Factory, and Generic Attribute Access

Part of the Upcoming book: EJB Design Patterns, by Floyd Marinescu
floyd@middleware-company.com
519-574-7480

Synopsis:
Included in this plop submission are the Value Object, Domain Value Object, Custom Value Object, Value Object Factory, and Generic Attribute Access patterns, which describe mechanisms for accessing and transporting data between tiers in an efficient, maintainable manner, in an EJB environment.

Part of the *EJB Design Patterns* book (a draft is freely downloadable off of www.TheServerSide.com, should contain about 20+ patterns by the time its complete), my intent in writing these patterns are to document common solutions to problems that EJB developers face (many of which are tied to EJB and not applicable in other technologies), in a form both concise enough for an expert to browse but at a level that would allow a novice developer with a basic understanding of EJB to grasp and apply these advanced concepts.

Notes for reviewers: If you are pressed for time, then feel free to focus on pages 1 – 11.

Value Object

The client tier in an EJB system needs a way to transfer bulk data with the server.

How can a client exchange bulk data with the server without making multiple fine-grained network calls?

*** * ***

In any distributed application there are two ways that a client can interact with a server. The first is to read some data from the server for display purposes; the second is to change some data on the server by creating, updating or removing data. In an EJB context, these types of operations typically involve the exchange of data between the client (servlet, applet, etc), and a Session Bean, Entity Bean, or Message Driven Bean.

When large amounts of data need to be exchanged, this can be achieved by loading many parameters into a method call (when updating data on the server), or by making multiple fine-grained calls to the server to retrieve data (when a client needs to read data from the server). The former option can quickly get out of hand when dealing with large amounts of parameters, and the latter option can be a performance killer.

Imagine the scenario where a client UI needs to display a set of attributes that live on the server, these attributes could live in an entity bean or be accessible through a session bean. One way that the client could get the data it needs is by executing multiple fine-grained calls to the server, as in figure X.1.

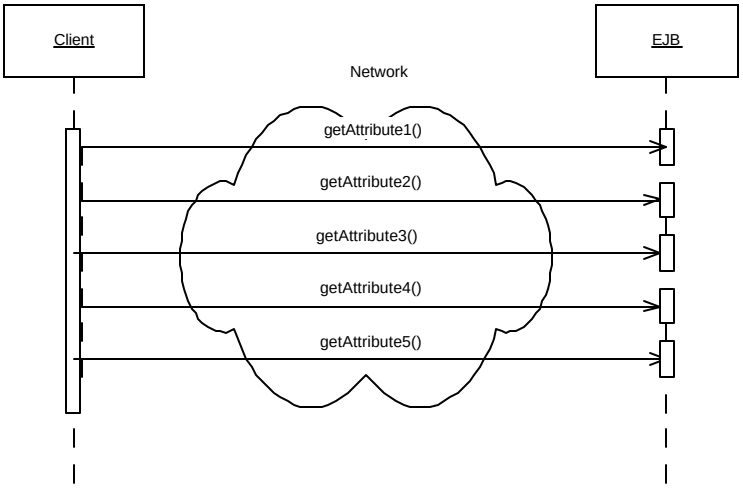

Figure X.1: The wrong way to get data from the server

The problem with this approach is that each call to the server is a network call, requiring serialization and deserialization of return values, blocking on the client while the EJB server intercepts the call to the server and performs transaction and security checks, and of course the retrieval of the attribute in question. Furthermore, each method call might actually execute in its own separate transaction if the client is not using Java Transaction API client demarcated transactions.

Executing multiple network calls in this fashion will contribute to significant degradation in performance. A better alternative is required, one that would allow the client to get all the data it requires in one bulk call.

Therefore,

Create plain java classes called Value Objects, which contain and encapsulate bulk data in one network transportable bundle.

A value object is a plain serializable Java class that represents a snapshot of some server side data, as in the following code example:

```
import java.io.Serializable;

public class SomeValueObject implements Serializable {

     private long   attribute1;
     private String attribute2;
     private String attribute3;

     …

     public long getAttribute1();
     public String getAttribute2();
     public String getAttribute3();
     …

}//SomeValueObject
```

Value objects can be used both for the reading operations and the update operations in a distributed system. When a client needs to update some data in the server, it can create a Value Object that wraps all the information the server needs to perform the updates, and send it to the server (usually to a Session Façade) for processing. Of course, it could also send data to the server using zillions of fine-grained parameters, but this is a very brittle approach. Whenever one parameter needs to be added or removed, the method signature needs to change. By wrapping with a value object, changes are isolated to the value object itself.

Where Value Objects are clearly needed is for reading operations. When a client needs to read some server side data (usually for the purpose of populating a client-side UI), the client can get all the data it needs in one bulk network call by wrapping the data in Value Object form.

From the previous example, the server side EJB would create a value object (as in figure X.2) and populate it with the attributes that the client required. This data would then be returned to the client in one bulk return value – the value object.

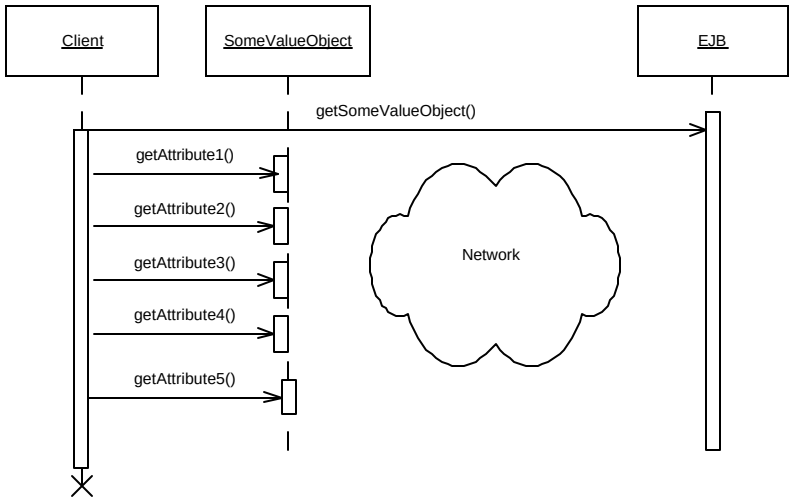

Figure X.1: The right way to read data from the server

Value objects are basically 'envelopes', used to transport any kind of data between the tiers in a J2EE system.

A common problem developers face when using value objects is choosing at what granularity to design them. That is, how do you choose how much data to wrap with a value object? At what point do you decide a value object is necessary? As the primary method of exchange between client and server, value objects form part of the interface that separates client developers and server developers. At the beginning of a project, client and server developers need to agree on the value object design at about the same time they need to decide upon what EJB interfaces. Despite this need, designing value objects at this stage can be difficult since developers often don't completely understand exactly what units of data should be transferred between client and server, at the beginning of a project.

A easy place to start in designing value objects is as copies of server side entity beans (or domain objects), as described in the ***Domain Value Object*** pattern. Designing Domain Value Objects is easy, as project teams usually have a good idea the domain model a project will utilize early on, since these requirements are established initial design phases. Thus, making Domain Value Objects the unit of exchange between client and server can help get a team up and running quicker.

Ultimately, data exchanged between the client and server should be designed to fit the client's needs. Thus, as the project progresses and the needs of the clients become finalized, Domain Value Objects often become cumbersome as units of exchange, too coarse grained to be useful to the fine-grained needs of the client.

A client may need access to data that simply isn't encapsulated in any Domain Value Objects. At this point, developers can design ***Custom Value Objects,*** that is, value objects that wrap arbitrary sets of data, completely driven on the particular needs of the client.

The differences between these two design paradigms can have significant impact on the design of the application as a whole. Although they are mutually exclusive, they can and usually do co-exist in any J2EE application.

When deciding where to put the logic to create and consume value objects, the ***Value Object Factory*** pattern illustrates a proven and maintainable solution.

Domain Value Object

A client wants to access and manipulate data from the server side domain object model.

How can a client access and manipulate the server side domain (entity bean) object model without the performance overhead of remote calls?

A domain model, or domain object model refers to the layer of objects in your system that map to real world concepts, such as a Person, a Bank Account, or a Car. In an EJB setting, the most common example of a domain model is your application specific layer of entity beans. A Domain Object in this sense is a particular entity bean, such as a Car entity bean. Other technologies for creating domain models can also be used in an EJB setting, such as Java Data Objects, Data Access Objects, or the proprietary domain object layer provided by an object-relational mapping tool.

In a distributed scenario, domain object models live completely on the server. However, depending on the implementation technology chosen (entity beans, JDO, etc), a domain object model can manifest itself into two types: one that can be accessed by clients remotely across a network (entity beans), and one that cannot be accessed by clients remotely (JDO, proprietary frameworks, etc). For the latter, only other business components on living on the server can access the domain model.

Entity beans are a remotely accessible. A client can access an EJBObject stub that maintains a direct connection to a particular entity bean living on the EJB server across a network. However, as explained in the ***Session Façade*** pattern, accessing entity beans remotely is a very poor practice. The most powerful way to optimize on entity bean access is not to access entity beans from the client at all. Instead, a client would execute a method on a Session Façade, which would directly interact with entity beans through their local interfaces, allowing complex operations to be completed in one network call. Using the Session Façade pattern, entity beans are thus no longer accessible by a client across a network.

This optimizes on the performance problems associated with using entity beans directly from clients, but puts the client in a difficult situation: If the client cannot access entity beans (or any type of domain object), how can it work with the same data object abstractions (the same domain model) on the client side as are being used on the server side? How is a client to read and display the attributes of a particular domain object that lives on the server side, and work with this data using the same object oriented semantics on the client side?

For example, consider a Car dealership application, in which the business requirements of the system define the notion of a Car and a Manufacturer, implemented as entity beans on the server side. For the client-side portion of this application, the most intuitive thing to do would be to also display and update similar Car and Manufacturer abstractions on the client side. A client could read data from the Car and Manufacturer, update values on the Car and Manufacturer and naturally traverse relationships to other related entity beans if it needed data from them.

Performance concerns require that the client not have direct access to the entity bean domain object model, and other types of object domain models cannot even be accessed remotely. However, to be able to work with an applications domain model on the client side is desirable. Thus, a mechanism is needed to allow clients to traverse and manipulate the same data object abstractions as exist on the server.

Therefore,

Design Value Object copies of the server side domain objects (entity beans). Clients may operate on local value object copies of domain object data, performing bulk reads and updates.

Domain Value Objects are a special application of the Value Object pattern. Whereas the Value Object pattern simple says to use Value Objects to exchange *data* between client and server, the Domain Value Object pattern says that Value Objects can be used to provide clients with optimized local access to value object copies of server side domain objects. Thus, Domain Value Objects have one-to-one correspondence with domain objects on the server side. If you have an Account entity bean, then you will have an Account value object, as in figure X.1.

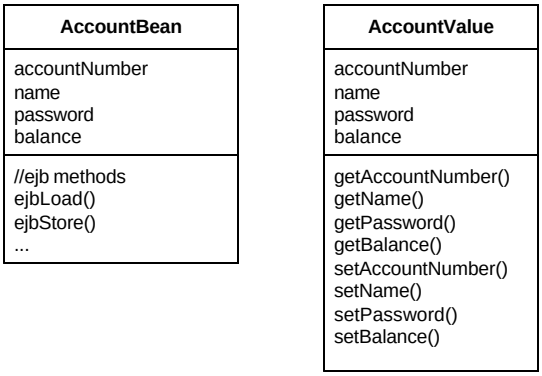

Figure X.1: Account EJB and Account Value Object

Using Domain Value Objects provides a simple way for clients to interact with the entity bean object model in a distributed application:

Displaying entity bean data. When a client wants to display the data from an Account entity bean, it could call a *getValueObject* method on the Account, which would retrieve an AccountValueObject, containing a copy of all the attributes of the Account Entity Bean. The client could then perform multiple *get* calls on this local value object, without the burden of network calls.

Modifying entity bean data. If a client wants to modify the contents of the Account entity bean, it would perform its update by calling *set* methods on its local copy of the AccountValueObject, then pass this updated value object to the server for merging with the Account Entity Bean.

Displaying data from that spans related entity beans. Entity beans can contain references to other entity beans in a one to one, one to many, or many to many fashion. Often a client will be interested in getting data from multiple related entity beans, but manually traversing to get separate value object copies would involve significant network call overhead. A better solution would be to assemble a value object copy of these related entity beans on the server side, and pass it to the client in one network call. This can be achieved using special value objects called *Aggregate Value Objects* (value objects that contain references to other value objects). Aggregate Value Objects can be created that contain value object copies of hierarchies of related entity beans. The client would navigate this **local** hierarchy of value objects just as if it would navigate the remote entity beans themselves.

Creating entity beans. To create a new Account, a client could locally create and populate an AccountValueObject, and pass this object to the Session Façade, which would use it in the *ejbCreate* method on the AccountHome. The less maintainable alternative would be to pass all the attributes of the Account as method parameters to the session façade and then to ejbCreate(). For example, which looks more maintainable: *ejbCreate(attrib1, attrib2, attrib3, attrib4, attrib5, …),* or *ejbCreate(aValueObject)*?

Much debate has arisen as to whether Value Objects should be **mutable** or **immutable**. That is, should a client be allowed to modify a value object (by calling *set* methods on it), or should value objects be read-only, without *set* methods. When using Domain Value Objects, it makes sense to make them mutable. The client knows

it is interacting with a client side copy of an entity bean, it can read and update the value object as though it were doing so on the entity bean itself. Since the client knows the value object came from the server, it is reasonable to give it the responsibility of *knowing to send* the value object back to the server once modifications have been made. Where immutable value objects make sense is when value objects don't represent server side entity beans, such as in the *Custom Value Object* pattern. Here, value objects simply represent arbitrary collections of read-only data, and updates should not be performed through the value objects.

Designing Value Objects as copies of server side entity beans has the following benefits:

Domain model data structures replicated to client in one network call. Copies of entity beans and even multiple entity beans can be assembled on the server and passed to the client in one network call. The client can then traverse the local value objects, reading and updating without incurring network call overhead. A client can then update the server by passing the value object back.

Easy to quickly build a functional site. Early in the development process, the specific data access needs of the client is unclear and always changing. Whereas the needs of the client UI's are unclear, the applications entity beans object model is has usually already been built. A functional application can quickly be built using the entity bean value objects as the medium of exchange between client and server.

Client-side attribute validation. Syntactic validation of entity bean attributes can be performed on the client side by embedding this validation logic in value object *set* methods. This allows errors with entity bean editing and creation to be caught on the client side instead of using up a network call only to have exceptions be thrown from the server.

As well as the following tradeoffs:

Couples client to server side domain object model. With the domain value object pattern, a client is working with a direct copy of a server side domain object (entity bean). Thus, Session Façade or not, the client is effectively coupled to object model that lives on the server. If an entity bean is changed, its corresponding value object must be changed, thus any clients using that value object must be recompiled.

Does not always map well to needs of clients. The entity bean object model used on the server side often does not map well to the clients needs. Different UI's may require different sets of data that simply don't map to 'bundles of data' that entity bean value objects provide. A client may want one or two attributes of an entity bean that has 20 attributes. To use a Domain Value Object to transfer 20 attributes to the client when only two are needed is a waste of network resources.

Results in a parallel hierarchy. Domain Value Objects duplicate objects in the domain model, resulting in duplicate attributes and methods.

Combersome for updating domain objects. Merging changes from an aggregate value object (a domain object that contains other domain objects), is difficult and cumbersome. What if only one domain value object deep in the tree was changed? Ugly code needs to be written to detect this.

The reader may have noted that the above examples have implied that domain value objects could be created and consumed by the entity beans themselves. Back in the EJB 1.X days (before entity beans had local interfaces), it was common to see entity beans expose a *getValueObject* and a *setValueObject* method, instead of fine-grained *getAttribute/setAttribute* methods. Every entity bean in an application housed logic that created a value object copy of itself (*getValueObject)* and logic that updated itself based on changed values in a value object (*setValueObject*). The reason was that all calls to an entity bean were potentially remote calls, even if they came from session beans or other entity beans co-located in the same server. The Domain Value Object pattern arose out of this need to optimize on calls to entity beans, be they from non-ejb clients or from other session and entity beans. With the introduction of EJB 2.0 local interfaces, session beans and other entity beans no longer need to use value objects to optimize on entity bean data access. Instead, they can simply use fine-grained *getAttribute/setAttribute* methods on the entity bean, now Domain Value Objects be used properly: to exchange domain object copies between client and server.

Since domain value objects should not be created and consumed on the domain objects themselves, this begs the question: Where should value objects be created and consumed? The ***Value Object Factory*** pattern provides a best practice for this type of code. Another related pattern is the **Custom Value Object** pattern, which takes the opposite perspective to the entity bean value object pattern: Value Objects should be immutable, and map to the specific needs of the client, not the domain model.

Custom Value Objects

A client finds that the domain object model and associated domain value objects don't map well to its needs.

How can value objects be designed when domain value objects don't fit?

The Value Object pattern introduced the notion of using a value object to pass bulk data between client and server. The Domain Value Object pattern described a common method of designing value objects – by mapping directly to the object model used on the server side. Although this method of designing value objects works well early on in a project, EJB clients often have much more fine-grained data access needs.

For example, consider a Car entity bean. A Car could potentially be described by hundreds of attributes (colour, weight, model, length, width, height, year, etc). In most typical scenario's, a client is only interested in a small subset of those attributes. For example, consider a web page that lists a cars model, year and type. To populate this page, it would be extremely wasteful to transfer a CarValueObject *(with all its attributes)* to the client, when it only wants to list 3 simple attributes from the car.

A client may have even more complicated needs. Imagine a client that required just one or two attributes from 5 different related entity beans. In order to optimize on network calls, a value object representation could be constructed on the server side that wraps all the required data into one network transportable bundle. One solution would be to create a Domain Value Object that contains links to other domain value objects. Thus the hierarchy of entity beans on the server side would be copied into a symmetric hierarchy of domain value objects. This approach is terrible for performance and cumbersome in practice. If a client only needs one or two attributes from each server side entity bean, transferring the complete domain object model as value objects to the client would waste time and network bandwidth.

Another problem is that often a client may be require data that comes from a variety of data sources other than the domain objects on the server. Datasources such as straight JDBC calls, Java Connector Architecture (JCA) adapters, also need to be wrapped in value objects and returned to the client.

Therefore,

Design custom value objects that wrap arbitrary sets of data as needed by the client, completely decoupled from the layout of the domain model on the server.

Custom Value Objects are just like normal value objects, except they are typically immutable and don't map to any specific data structures on the server (in contrast to mutable domain value objects). Custom Value Objects advocate a use case driven approach, in which value objects are designed around the needs of the client.

From the Car example, imagine a client only wanted to display the attributes of a car related to its engine. For this case, a value object that wraps those particular attributes should be created and passed to the client. This custom value object would contain a subset of the Car's attributes, as in figure X.1.

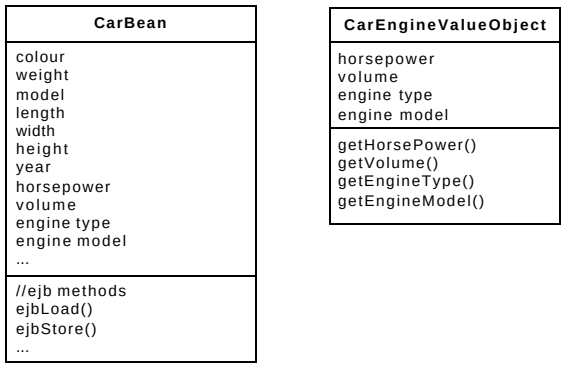

Figure X.1: A Custom Value Object wrapping a subset of data

In general, if an EJB client requires attributes X,Y and Z, then a value object that wraps X,Y and Z, and *only* those attributes would be created. With this approach, a value object acts as contract that provides the client with the data it needs, while encapsulating the data that the server has. Custom value object design works perfectly with the Session Façade pattern, in which the details of the entity bean object model are hidden behind a set of session beans. The correct way to think of value objects in this respect is merely as data and not as representing any server side business abstraction such as an entity bean. If all of the data happens to come from one entity bean, fine; but if not, that's the server's problem of how to populate the value object and doesn't concern the client.

Custom Value Objects are typically used for UI-specific read-only operations, and are made **immutable.** That is, Custom Value Objects cannot be changed; they are only for display purposes. Since a Custom value object is merely a grouping of data, and not really related to any server side business object, it doesn't make sense to update it. Typically, updates are done via entity bean value objects (since they represent a real business object and can encapsulate validation logic) or through use-case specific methods on Session Facades.

Custom Value Objects are almost always created via a Value Object Factory (see the **Value Object Factory** pattern), and are tied to the specific needs of the client.

ValueObjectFactory

A J2EE system using value objects finds that its value object layer tends to change very often.

How should value object creation and consumption logic be implemented, in order to minimize the impact of frequent changes in the value object layer on the rest of the system?

*** * ***

Value Objects have a tendency to change often. Domain Value Objects change whenever the domain objects change (adding a new attribute to an entity bean, etc). Custom Value Objects are just use case specific data holders for transporting data across a network; they can change as frequently as your application's presentation view. A medium to large application could potentially have tens to even hundreds of different Value Objects, each of which would require custom logic to create it. A critical question then becomes: how and where should this logic be implemented, in order to decouple and protect the rest of this system from value object changes?

A common solution employed in EJB 1.X applications is to place *getXXXValueObject/setXXXValueObject* methods directly on entity beans. In this scenario, the entity bean would be responsible for populating this value object, and updating itself based on the attributes of the *set* value object. The problem with this approach is that it tightly couples the value object layer to the entity bean layer. That is, placing use case specific value object creation code on an entity bean could cause serious dependencies between

your entity beans and your clients in medium to large sized applications. Every time a web page changed and a different view of the data model was required, you would have to add a new method to an entity bean, recompile your entity bean, and redistribute your remote interfaces to any client using them.

Entity beans are supposed to be reusable business components, which can be separately assembled to create an application. In order to build truly reusable business components, it is important to maintain strict separation between your application logic and your business logic, allowing the two to evolve separately. Some other solution is required for creating and consuming entity beans, one that can decouple value object related logic from other components in the system.

Therefore,

Place the responsibility for creating and consuming Value Objects in a *ValueObjectFactory*.

A Value Object Factory separates the logic related to Value Objects (part of the application domain) from other components in your system such as entity beans (part of the business domain). When new views or different subsets of server side data become necessary, new value object creation methods can be added to the *ValueObjectFactory*. These new methods will interact with the entity bean layer (or any other source of data such as JCA connectors, straight JDBC, etc), calling getters and traversing relationships as required to generate domain or custom value objects. The advantage to this approach is that the entity beans themselves do not need to know about these different views of their data, in-fact, no code on an entity bean needs to be changed at all.

A ValueObjectFactory (using our Car example) would like:

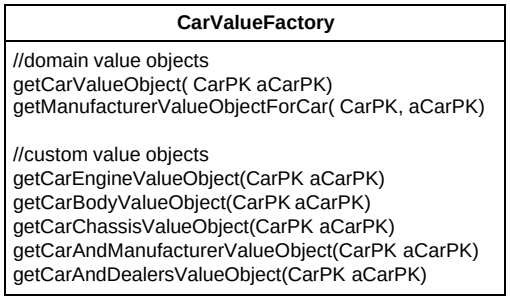

Figure x.x: CarValueFactory

There are two fundamental ways to implement the ValueObjectFactory pattern, depending on whether the client of the factory is a session bean or a non-ejb client such as a servlet. When used behind a session bean façade, the value object factory can be implemented as a plain java class that simply stores creation/consumption logic for different value objects in its methods. This type of Factory lends itself well to reuse as the value objects it generates can be reused across different session beans and/or in different projects.

When used from a non-ejb client, the ValueObjectFactory should be implemented as a stateless session bean. A typical interaction between this client and the value object is outlined in figure x.2. Here, a servlet client wants to get a Custom Value Object called CarAndManufacturerValueObject , so it queries a CarValueFactory for this object. The CarValueFactory then creates and populates this view value object by calling *get* methods on the Car and its related Manufacturer entity bean through their **local interfaces.**

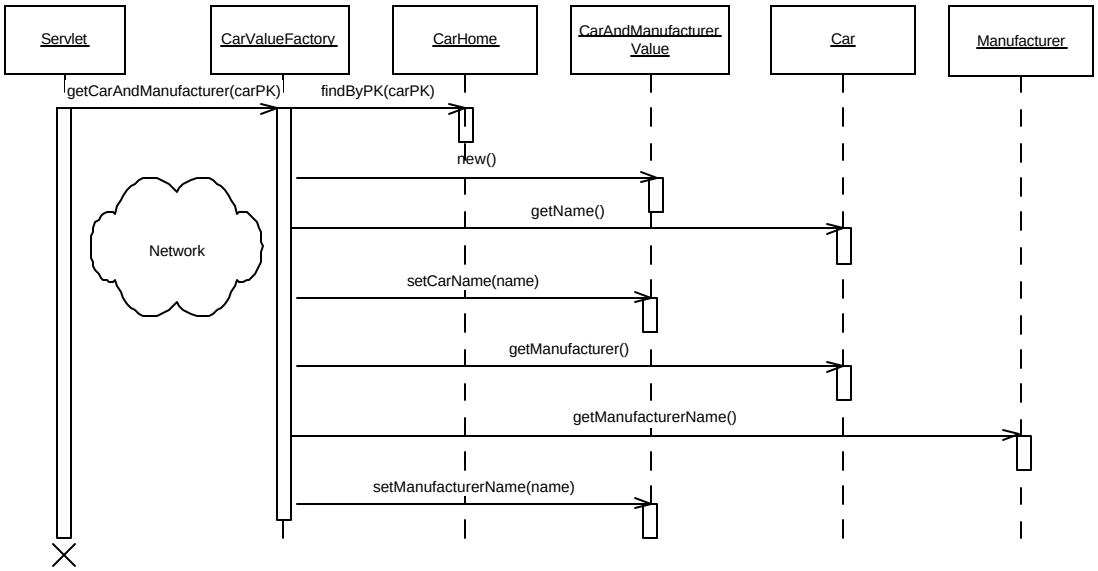

Figure X.2: Car Value Factory as Session Bean Interactions

Value Object Factories can be used to easily create any type of value objects. Even complex hierarchies of Aggregate Value Objects (domain value objects that contain other domain value objects) can be created that map to different *slices* of the server side entity bean object model. Complex value object hierarchies can be created by explicitly writing logic that knows how to navigate (and copy) a use-case specific *slice* of a hierarchy of entity beans. These value object hierarchies can all be created upfront on the server, and passed to the client in one network call.

One important benefit that results from this practice is that the entity beans in our application are now fully reusable. For example, imagine two separate development teams in a corporation working on separate applications. These two teams can reuse the same entity bean business components (a beautiful example of EJB reuse in action by the way) by using separate Value Object Factories. The teams could achieve complete reuse by each maintaining its own separate value object factory that passed out use-case specific value object 'slices' of entity bean state independently of the other team. By maintaining their own Value Object Factory, they could also develop and deploy their own applications completely independently from each other. This concept is illustrated in figure X.3.

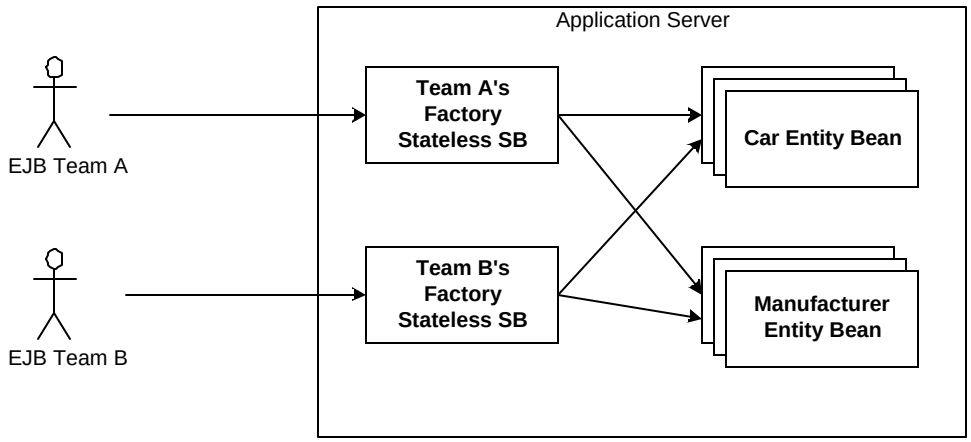

Figure X.3: Achieving Entity Bean Reuse with Value Object Factories

Note that the Value Object factory pattern does **not** imply creating one Value Object Factory for each entity bean class. For example you don't necessarily need to create a CarValueFactory for a Car entity bean. This would result in explosion of VO factories. Where requirements permit, it can be more straightforward to create a Value Object Factory for a whole set of entity beans and/or other sources of server side data.

Value Object Factories provide a way to read data from the server, but what of updating? Similar techniques used for reading server side data can also be used for updating. That is, clients can either pass a Domain Value Object or a Custom Value Object to the server, where it can in term perform CRUD operations on entity beans or other datastores on the server side.

For Domain Value Objects (which are typically made mutable), a client will perform its updates onto a value object locally, and then update the server by passing a the DVO to an *updateXXXEntity* method on a ValueObjectFactory, which would copy the attributes of the value object into the appropriate entity bean, using fine-grained *set* methods on the entity beans local interface. Clients can create entity beans similarly by populating a DVO locally, and passing it to a *createXXXEntity* method on the factory.

Using the previous example, if the application administrator wanted to update a particular Car or a Manufacturer, these updates would be done with separate UI displays (one for the Car, and one for the Manufacturer). Updates would be performed and either a Car or a Manufacturer Domain Value Object would be sent back to the server for updating in a single transaction, as in figure X.4.

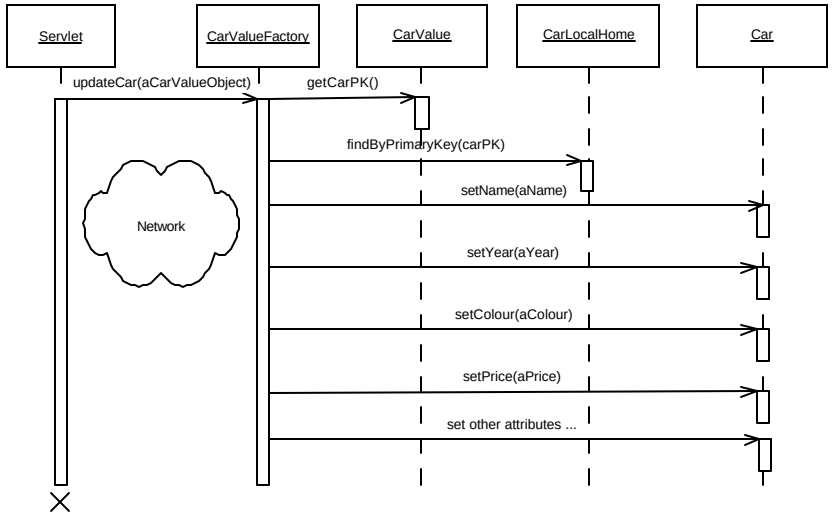

Figure X.4: Updating using a Value Object Factory

For performing any sort of update above and beyond updating domain objects, or the attributes of other fixed sets of data on the server should be done by passing Custom Value Objects the Session/Message Façade. Remember that the Façade is supposed to contain all the business logic required to execute use cases in an application, such as placing an order on Amazon, or transferring funds at a bank. For these types of operations, a client will typically create a Custom Value Object that contains all the data required to perform the update, pass this value object to the façade, which will in turn create, update or delete any number of server side resources.

The advantages to the Value Object Factory approach are numerous:

Better maintainability. By separating between your application logic (use-cases) and your data object model (entity beans), the two can evolve separately. Entity beans no longer need to be changed and recompiled when the needs of the client change.

Encourages entity bean reuse. Entity beans can be reused across projects, since different Value Object Factories can be written to suit the needs of different applications.

Allow for creating complex value object graphs. By writing value object creation logic upfront, developers can create complex graphs that mimic one-to-one, one-to-many, many-to-many, cyclic, and combinations of such relationships, providing clients with fine grained control over what parts of entity bean data they need to display. For non-web clients such as java apps and applets, the ability to get non-tabular data is particularly important.

Increases performance. When the Value Object Factory is uses as a Session Façade, attributes from multiple entity beans can be passed to the client with just one network call.

The Value Object Factory pattern can build maintainable and flexible systems, providing a simple and consistent method for creating arbitrarily complex value objects and passing them to the client in one bulk network call, without causing dependencies between value objects and other components in a J2EE system.

Generic Attribute Access

An entity bean client needs to access the attributes of an entity bean.

How can an entity bean client efficiently access and manipulate the attributes of an entity bean in a bulk, generic fashion?

*** * ***

Directly invoking methods on entity beans from non-EJB clients incurred significant performance overhead when using the entity bean remote interface. Even if using the local interface of an entity bean, this approach is discouraged due to tight coupling, poor transaction integrity and maintainability concerns. The Value Object pattern, alone or combined with the Session Façade pattern served to correct these problems.

Value Objects patterns provide methods for efficient bulk access to the attributes of entity beans and also for marshalling data between session facades and client tiers. These approaches are effective, but also suffer several drawbacks:

High cost of change over time. The use cases in an application change over time. Different clients may need to access different views or subsets of server side data than were initially programmed. Using the value object approach (even with Value Object Factories), server side code (such as new value objects and associated creation logic) must be written to satisfy the changing data access needs of the client. Once an EJB project has been launched, access to server side programmers tends to be expensive, as is the EJB re-deployment process.

Does not scale well from small to large entity beans. Imagine a stock/bonds entity bean for a financial application. Such an entity bean could have well over 200 attributes. To write and expose getters/setters for all of those attributes could be a nightmare of tedious coding and an explosion in interface size.

Poor maintainability. Value Objects create a new layer, which can explode to thousands of objects in a large application. Imagine a distributed system with 30 entity beans. Each of those thirty entity beans would likely have a Domain Value Object to marshal their state to and from the client tier. The applications use cases may also require that data from those entity beans be used in several Custom Value Objects. Thus, a medium sized system could require hundreds of value objects, each with a particular value object factory method to create it. Since the value object layer is tightly coupled to the entity bean layer, changes in entity bean attributes will cause ripples that could require changes in multiple value objects as well. In large applications, the value object layer can prove to very difficult to maintain.

Although simple and effective for small to medium sized systems, for large systems, the Value Object approach can slow down the development process, contribute to bloated data access code and reduce maintainability. The development/maintainability problems associated with value objects boil down to the fact that they are a ***domain specific*** interface to your data model. Dictionary.com defines a domain as a *sphere of activity, concern, or function*. For example, consider an *Account* entity bean as a domain. An *AccountValue* value object (as in figure X.X in the Value Object Chapter X) is thus an object that is specific to the domain of an account entity bean, and cannot be re-used across other entity beans in different domains; it would not make sense to return an Account Value Object from a Car Entity Bean. Value Objects are thus domain specific interfaces to entity beans. Since Value Objects are not reusable across domains, this results in EJB applications with potentially thousands of lines of code that essentially does the same thing – copies domain attributes data from an entity bean into a domain specific value object, and returns this to the client. For each use case in a system, this essentially identical logic is duplicated all over the place.

If the domain of an application is relatively simple, value objects are a great way to get the job done. If your domain requirements are more complex, an alternative to Value Objects may be needed; one that decouples attribute access from your domain model, but still allows bulk access and transport of data across tiers.

Therefore,

Abstract entity bean attribute access logic into a generic *Attribute Access* interface, using HashMapto pass key-value attributes in and out of entity beans, and across tiers in your system.

The *Attribute Access* interface is implemented by entity beans, and looks like this:

```
public interface AttributeAccess {
        public Map getAttributes(Collection keysOfAttributes);
        public Map getAllAttributes();
        public void setAttributes(Map keyAndValuePairs);
}
```

Source x.x: Attribute Access Interface

Attribute Access provides a generic interface that completely replaces domain specific value object factory methods. The standard Java HashMap, available in any JDK, can replace an entire layer of Value Objects. The only dependency been client and entity bean code is the naming conventions placed on the *keys* used to identify attributes, described later in this pattern.

Clients access an entity beans attributes through the attribute access. Figure X.X in the Value Object chapter illustrates how a client used to interact with an entity bean through its domain specific getter/setter methods. Instead of explicitly calling a getter method for each attribute in an entity bean, a client can make one call to the *getAllAttributes()* method on the Attribute Access interface, and execute getters locally on the returned HashMap. Updates can be made on the same HashMap, and the entity bean can be updated by passing that same HashMap back to it in the *setAttributes(aHashMap)* method.

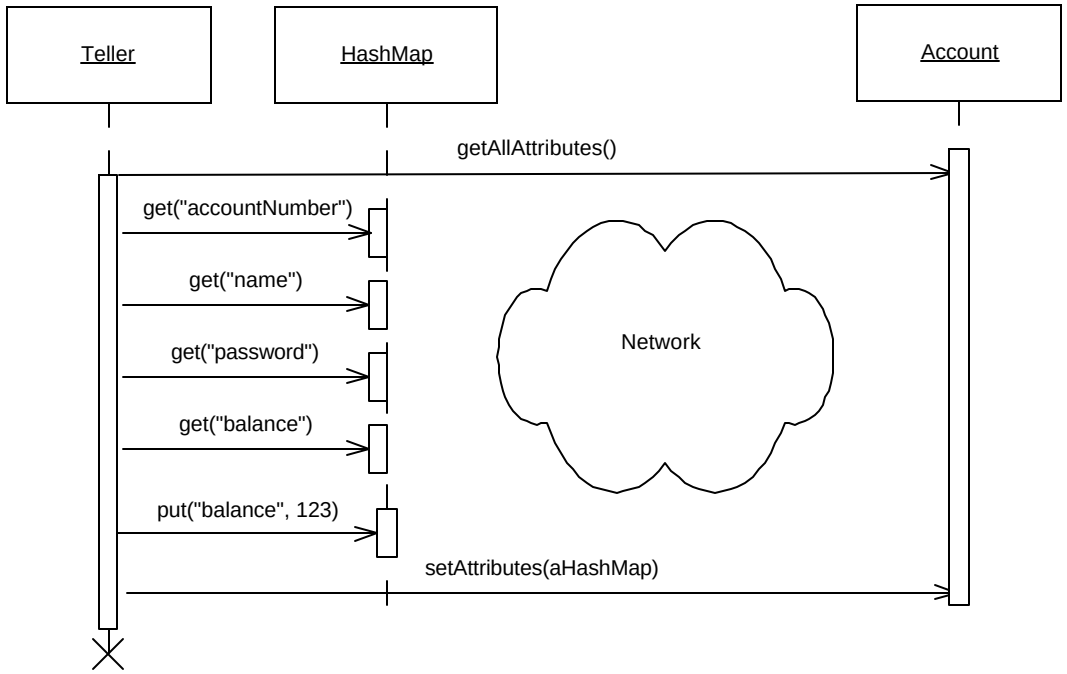

Figure X.1: Client Accessing Entity Bean Attributes In Bulk

Instead of implementing Custom Value Objects when a client needs a smaller subset of an entity beans data, a client can just call the *getAttributes(Collection keysOfAttributes)* method, passing in a collection containing strings which identify the attributes a client is interested in. A client can thus dynamically decide which subsets of entity bean data they want to see at run time, eliminating the need for manual, design time programming of Value Objects.

Custom Value Objects that contain data that spans multiple entity beans can also be replaced by HashMaps and the AttributeAccess interface. From behind a Session Bean Façade, session beans can grab attributes from different entity beans and easily combine them before passing to clients. One drawback with using the Generic Attribute Access pattern is that the HashMap class was designed for storing a flat map of attributes. Thus, copying the attributes from graphs of related entity beans to a client in one bulk network call is not supported explicitly by pattern. Depending upon the complexity of the application, an application specific policy for flattening object graphs into a flat HashMap will be needed.

Like the interface itself, the implementation of the AttributeAccess interface is generic. Under BMP, an entity bean can be further simplified by storing all of its attributes in a private, internal HashMap, rather than the obvious hard-coding of attributes that usually takes place. On large entity beans, this optimization can simplify an entity beans code greatly. Using this internal HashMap, the implementation of the methods on AttributeAccess thus become completely generic, reusable across BMP entity beans:

```
private java.util.HashMap attributes;

/**
 * Returns key/value pairs of entity bean attributes
 * @return java.util.Map
 */
```

```
public Map getAllAttributes()
{
      return(HashMap)attributes.clone();
}

/**
 * Used by clients to specify exactly the attributes they are interested in
 * @return java.util.Map
 * @param keysofAttributes the name of the attributes the client is
 * interested in
 */
public Map getAttributes(Collection keysofAttributes)
{
      Iterator keys = keysofAttributes.iterator();
      Object aKey = null;
      HashMap aMap = new HashMap();

      while ( keys.hasNext() )
      {
            aKey = keys.next();
            aMap.put( aKey, this.attributes.get(aKey));
      }

      //map now has all requested data
      return aMap;
}


/**
 * Used by clients to update particular attributes in the entity bean
 * @param keyValuePairs java.util.Map
 */
public void setAttributes(Map keyValuePairs)
{
      Iterator entries = keyValuePairs.entrySet().iterator();
      Map.Entry anEntry = null;

      while ( entries.hasNext() )
      {
            anEntry = (Map.Entry)entries.next();
            this.attributes.put(anEntry.getKey(), anEntry.getValue());
      }
}
```

Source X.2: Attribute Access BMP Implementation

In CMP, using an internal map of attributes is not possible, since the implementation of an entity beans classes is abstracted behind container generated *get* and *set* methods. When an internal map is not possible, the AttributeAccess interface can be implemented generically using the Java Reflection API. That is, in *setAttributes*, reflection can be performed on the key-value of the attribute a client wants to set. That is, if the key-value is *XXX* the *setAttribute* implementation will attempt to call *setXXX(…)* on the entity bean. Similarly, in the *getAttributes* method, reflection can be used to find all *get* methods on an entity bean, invoke them and populate a map for the client. If a developer would prefer not to use the Reflection API, then implementing the AttributeAccess method

cannot be done generically for CMP. The developer will need to interlace his AttributeAccess implementation with IF statements that call hard coded *get/set* methods on the entity bean, depending on the key-value string.

In order to make the AttributeAccess implementation code reusable across all entity beans, a super class can be used to implement the interface methods, which are completely generic whether using the reflection method or the internal map of attributes style of implementation. All entity beans wanting to make use of the AttributeAccess services need only subclass the super class implementation, thus automatically exposing this unified interface to their attributes with no extra coding involved.

The final piece of the puzzle is how to name the keys, which identify the attributes of an entity bean. Not only do the attributes need to be identified by key, but client and server side programmers must need to agree on the naming conventions. Some sort of 'contract' is required between client and server. Several possibilities are discussed:

Establish a consistent naming convention. The client and the server can agree upon a documented, consistent naming convention for attributes. For an Account entity bean, *"com.bank.Account.accountName"* would be an example of consistent convention. The drawback with this approach is that the contract exists in the minds of developers only, when developing, it is easy to misspell the attribute name, resulting in costly development errors.

Define static final member variables in the entity bean remote or local interface. An entity bean client can make calls to the entity bean using references to static final variables containing the correct key-string required to get an attribute. For example, in-order to retrieve the attributes from an *Employee* entity bean*;* an entity bean client could use the following code:

```
      //ask employee for his personal attributes
      Collection aCollection = new Vector();
      aCollection.add(Employee.NAME);
      aCollection.add(Employee.EMAIL);
      aCollection.add(Employee.SEX);
      aCollection.add(Employee.SSN);

      Map aMap = employee.getAttributes(aCollection);
```

Source X.3: Getting attributes using final static variables

Where the entity beans remote interface contains the following definitions:

```
public interface Employee extends EJBObject, AttributeAccess
{

      //since attributes are stored in a hashmap in the entity bean,
      //we need a central place to store the 'keys' used to reference
      //attributes, so that the clients and the entity bean won't need
      //need to be hardcoded with knowledge of the attribute key strings
      public final static String ID = "EMPLOYEEID";
      public final static String NAME = "NAME";
      public final static String EMAIL = "EMAIL";
      public final static String AGE = "AGE";
      public final static String SSN = "SSN";
      public final static String SEX = "SEX";
...
}
```

Source X.4: A Remote Interface Attribute Contract

This approach works fine when the client of the entity bean calls the entity bean directly (not through an intermediary). There are many instances where this is not the case, such as a 3 tier architecture using the *Session*

Façade pattern. Here, it is the presentation logic in the servlets that chooses which an attribute to display to the browser, but it is a Session Bean that interacts directly with an entity bean. The problem is that the code that decides which attributes to display (needs to know the key-value names) needs to be in the servlet layer. This would require the servlet to have access to the entity bean remote interface, in order to use the key name contract. This would couple the presentation logic to the entity bean layer, reducing the benefits of the *Session Façade* pattern. A couple of solutions are discussed:

Shared final static member variables class. With this approach both the clients and the entity beans share access to a separate class consisting of final static strings (the keys of the attributes). This approach does not tie a client to a specific entity bean interface, reducing coupling between client and server. One disadvantage to this method is that should the key mappings need to be updated or added to, the new class would need to be redistributed to the client (and their JVM would thus need to be restarted).

Place the Attribute contract in a JNDI tree. In this approach, a singleton of sorts is maintained by placing a class containing the keys in a JNDI tree, accessible by client and server. Client and server code would not need to be recompiled or rebooted, when keys are changed/updated, since a central object in a JNDI tree would always contain the freshest copy of keys. The tradeoff with this solution is the overhead incurred in grabbing the contract from the JNDI tree whenever key values are required.

The ***Generic Attribute Access*** pattern has many advantages:

Excellent maintainability - eliminates the Value Object layer. The extra layer of domain specific value objects and all that repetitive value object creation logic is now eliminated in favour of the generic reusable Map and Attribute Access interfaces. This represents a potential reduction of thousands of lines of value object and factory logic to the minimal amount required for the AttributeAccess implementation.

Allows for dynamic addition of attributes at runtime. When using BMP, this pattern can easily be extended to allow for the ability to add and remove attributes from an entity bean dynamically. This can be achieved by adding an *addAttribute* and *removeAttribute* method to the interface, which simply performs operations on the attribute HashMap.

One interface across all entity beans. Entity bean clients can manipulate entity beans consistently via the *AttributeAccess* interface, simplifying client code. Entity beans are also simplified, as the *AttributeAccess* can be encapsulated in a super-class.

One data object (Map) across all clients. A Map of attributes is reusable from the *Session Façade* down to the JSPs. In particular, using a Map as a data container significantly reduces the complexity of the JSP code, as pages don't need to be written with use case specific Value Objects that are tightly coupled to the entity bean layer.

Scales well to large entity beans. Whether an entity bean has 20 or 2000 attributes, attribute access logic is simplified to just a few lines.

Low cost of maintenance over time. New views of server side data can be created that does not require any server side programming. Clients can dynamically decide which attributes to display.

Like all patterns, using ***Generic Attribute Access*** has its tradeoffs:

Additional Overhead per method call. For each attribute call, clients must use an attribute key to identify attributes. Finally, attributes need to be *cast* to their appropriate type after being extracted from a Map object.

Need to maintain a contract for attribute keys. Since attributes are requested by string, clients need to remember the key strings used to identify attributes. Defining a key-attribute contract (discussed earlier in this pattern), can alleviate these dependencies.

Loss of strong typing/compile time checking. Using value objects, values passed by *gets* or *sets* were always of the correct type, any errors would be passed at compile time. Using ***Generic Attribute Access,*** attribute

access must be managed by the client at runtime by casting objects to their correct type and associating the correct attribute type with the correct key.

Lack of support for entity bean relationships. AttributeAccess is just that, an interface for accessing the individual attributes of an entity bean. Support for marshalling state of multiple entity beans in one network call is out of scope for this pattern.

Many developers are uncomfortable with adding the dangers of weak typing and complexity of maintaining a key-value contract across the client and server. These developers should keep in mind that the Generic Attribute Access pattern basically has two components: Using the AttributeAccess interface to an entity bean's attributes, and using a HashMap as an alternative to Value Objects for marshalling data across the network. Most of the benefits and complexities of this pattern arise due to the use of a HashMap as a value object alternative (need for a key-value contract between client and server, weak typing at the client, etc). Luckily, this pattern can be applied without using a HashMap between client and server. A Session Façade could be used that only has value objects as method parameters and return values, but internally operates on entity beans using the AttributeAccess interface. This Session Façade would convert HashMap data into strong-types value objects for the clients, but still benefit from the reuse and maintainability benefits of having a generic interface into the entity bean layer.

Overall, the ***Generic Attribute Access*** provides a domain-generic method of managing the state of entity bean objects, eliminating the bulky repetitive code associated with domain specific entity bean data access. Maintaining this level of abstraction requires extra care on part of the developers, but for large scale applications, the indirection may be necessary. Where simple domain model is involved the *Value Object* through a *Session Façade* is simple way to get the job done. Where the requirements are more complicated, developers should consider the ***Generic Attribute Access*** pattern.