The EJB Session Facade pattern wraps entity beans with session beans so that the entity beans aren't directly accessed by clients. The pattern goes beyond simple wrapping to help you structure the objects called by the session beans. This article delves into the details of this useful EJB pattern.

In the past couple of years Enterprise JavaBeans™ (EJBs) have really started to make an impact in the Java™ object design space. During this time, one of the most common EJB patterns that we have seen employed is the notion of a Session Facade. This is a very powerful and simple notion, and one that has been beneficial to a large number of developers. However, I have also seen that there are a number of misconceptions about exactly what this pattern means, and a great deal of misunderstanding about how it should be applied in practice.

In an attempt to bring more light to the subject, in this article I will reiterate some of the basic notions of what a Facade is and how the Session Facade pattern operates, and explore some of the ramifications of this pattern. Hopefully this will dispel some of the misconceptions and help developers in the proper application of the pattern.

**What are Session Facades and why do you need them?**

The Session Facade pattern has been clearly described in a number of places, namely [Sun 2001] and [Brown 2000]. Rather than recover all of that ground, I will attempt to summarize the reasoning for it here: The basic problem is that in an EJB design, EJB clients (for example, Servlets, Java applications, and so on) should not directly access Entity beans.

There are a number of reasons for this:

- A significant runtime performance penalty can be incurred when crossing the network on an RMI-IIOP call. If a client asks an Entity bean representing, say a Bank Account for two pieces of data (say the account balance and the customer name who owns the account), then that would require 2 network calls. When this is multiplied by a large number of attributes, this overhead quickly becomes significant. A potential solution to this involves what [Monson-Haefel] describes as "bulk accessors," which are methods on an Entity bean that create and return value objects to represent the data in the Entity bean. This is in fact the solution employed by the CopyHelper Access Beans in VisualAge® for Java. However, this has an
unfortunate downside in that it assumes that all requests will need ALL of the data in the EJB, resulting in returning unnecessary data to the user, and incurring additional overhead for marshalling and unmarshalling the larger value objects.

• More importantly, if you allow EJB clients to directly access an Entity bean, then the client requires knowledge of the Entity implementation that goes beyond what clients should have. For instance, manipulating an Entity bean requires knowledge of the Entity relationships (associations, inheritance) that are involved, inappropriately exposing the client to all of the details of the business model. Also, manipulating more than one Entity bean requires the use of client-side transactions -- another complication that EJBs are meant to remove from a client design, not add to it.

The solution that most architects have found in order to avoid directly accessing the Entity beans in an EJB design can be found in the Facade described in [Gamma]. [Gamma] describes the Facade pattern in the following way, "Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use." The application of this idea to EJBs has been generally construed to mean that you should create a Session EJB that acts as a Facade and then "wraps" a set of Entity beans that make up a subsystem. In this way the clients are isolated from the details of the Entity bean implementation, and do not have to manage the details of transaction management themselves.

However, the problem is that most people stop there. They then blithely go on and begin wrapping Entities inside Sessions, without considering the rest of what the Facade pattern describes, and what its ramifications are in an EJB design. This is probably due to coming by the information on Facade "second-hand" as it were, and not studying the original pattern. If we do take the time to understand the ramifications of Facade, then we can see a number of other beneficial design possibilities inherent in the pattern.

**Key points of the Facade pattern**

There are a number of key points made about the Facade pattern in [Gamma] that we need to understand. The first few are found in the Applicability section of the Facade pattern, which describes when you will want to apply the Facade pattern. They are: "Use the Facade pattern when...you want to provide a simple interface to a complex subsystem" and "Use the Facade pattern when... you want to layer your subsystems. Use a Facade to define an entry point to each subsystem level." There are a couple of ideas we can extract out of that discussion of the Facade pattern. The first is that a Facade should provide an abstract view of the subsystem, rather than simply directly wrapping the API of the whole subsystem itself. One of the most heinous abuses of this pattern that I unfortunately have seen in practice a number of times is where a developer creates a Session bean that directly wraps all of the methods of the Entity bean homes and Entity bean objects without providing any additional abstraction. Remember that the idea here is to reduce the complexity of the overall system, not just shift the complexity to another object.
The second, subtler point involves layering. The idea here is that you may employ multiple Facades to hide the details of successive subsystems. So here you can imagine that a Session Facade might be layered on top of other Facades that further abstract away the details of the underlying business logic. This is a crucial point. It becomes clearer when you look at the following two statements from the Collaborations Section and the "Related Patterns" section of the Facade pattern in [Gamma] respectively:

- "Clients communicate with the subsystem by sending requests to the Facade, which forwards them to the appropriate subsystem object(s)."³
- "A facade merely abstracts the interface to subsystem objects to make them easier to use; it doesn't define new functionality."⁴

I can summarize these points in the following statement: Facades don't do the real work of a system; they instead delegate to other objects that in turn do the work. The ramification of that is that you must have these objects in place in order to make the pattern work as intended.

This point is the key difference between two popular representations of this pattern, [Sun 2000] and [Sun 2001]. The first description [Sun 2000] which is part of the J2EE blueprints, calls the pattern "Session Entity Facade." Its intent states that it "provides a single interface to a collection of enterprise beans." It describes a pattern by which all data access is through Entity beans, and a Session bean provides an interface to those beans. Now, the problem is that [Sun 2000] is unnecessarily EJB-centric. It does not mention other object types at all, and assumes that the only objects in your system are EJBs. In my experience, I have seen that this can result in bloated Session objects that are not reusable at all across projects, and can pose problems when there are slightly different requirements within the same project.

Now, [Sun 2001] is more general and does not suffer from this problem. It calls this pattern simply "Session Facade," and its solution instead states that you should "Use a session bean as a facade to encapsulate the complexity of interactions between the business objects participating in a workflow." It does not restrict you to having the business objects be EJBs at all, and is thus a more flexible approach.

**Rules to live by for Session Facades**

So how do we apply these rules about Facades to Sessions and what does this mean for our EJB designs? There are three basic principles that I apply when designing my Session Facades:

- They don't do work themselves; they delegate to other objects to do the real work. This means that each method in a session facade should be small (five lines of code or less not counting exception handling logic).
- They provide a simple interface. This means that the number of facade methods should be relatively small (no more than two dozen or so in each Session bean).
- They are the client interface to the underlying system. They should encapsulate the subsystem-specific knowledge and not unnecessarily expose it.
So how does this work? What other types of objects can you delegate work to, and what advantage does that confer in your designs? I've covered some of this in my book [Brown 2001], so some of the details can be found there. However, in summary, here are the four kinds of objects that I generally find in most of my EJB designs:

- **Value objects** are Serializable Java beans that contain data requested by a client. They contain a subset of the data contained the Entity beans and other data sources. They are the return types for Session EJB methods. Value Objects and their uses are described in both [Sun 2001] and [EJB 2.0]. Note that [Fowler 2001] refers to these as "Data Transfer Objects," which is also the name used in [Brown 1999]. Personally, I feel that Data Transfer Object is a more descriptive term, but unfortunately, the Sun terminology seems to be becoming commonplace.

- **Object Factories** [Brown 1999] [Brown 2000] are responsible for building value objects. They know about the different data sources, create instances of the value objects, fill in the instances of the value objects, and so on. Each factory can retrieve and update data to and from multiple data sources. There should be a factory for every "root" object in your object model. (Root objects are those that "contain" other objects.) In a way, an Object Factory is acting as a Facade onto the JDBC or Entity bean persistence subsystem, implementing the layering principle from [Gamma].

- **Entity EJBs** should be standard "data sources" that can be globally useful across the enterprise. Entity beans should not contain application-specific domain logic, nor should they be constrained to only work within a single application. Note that Entity beans are optional and are not a required part of this architecture; a Factory could just as simply obtain data directly from a data source like a JMS queue or a JDBC Connection.

- **An Action object** represents a unique business process that a session bean may invoke. Action objects are required to handle business processes that are not related to simply creating, reading, updating, or deleting data. Like Object Factories, Action objects also act as inner-layer Facades.

**An example of EJB objects**

A problem with describing patterns like these is that the examples that are large enough to warrant the use of the pattern are too large to include in the description of the pattern itself. However, let me try to illustrate what some of these objects would look like by the following, admittedly simple example.

Let's say that we're building an ATM system for a Bank. This is one of the hoariest of OO design clichés, and it certainly has been done in many other books and articles, but it has enough interesting twists to it to serve our purposes. In this analysis, we've found two kinds of EJBs.

- The ATM's connection to the bank is represented as a Session bean. There are methods on this bean to do the kind of things that you can do with an ATM—make deposits, make withdrawals, and transfer money between accounts.

- Accounts are represented as Entity beans (in our case CMP, but it doesn't really matter to the example). They have methods to return the balance of the account, and debit and credit the account.
The Remote Interface for our ATM Session bean is as follows:

```java
package com.ibm.bankexample.ejbs;
import com.ibm.bankexample.domain.*;

/**
 * This is the Enterprise Java Bean Remote Interface
 * for the ATM example.
 */
public interface ATM extends javax.ejb.EJBObject {

    void deposit(java.lang.String accountNumber, double amount)
        throws java.rmi.RemoteException,
        com.ibm.bankexample.domain.FactoryException;

    java.util.Vector getAccounts(java.lang.String userid)
        throws java.rmi.RemoteException,
        com.ibm.bankexample.domain.FactoryException;

    void transfer(java.lang.String fromAccount, java.lang.String toAccount,
        double amount)
        throws java.rmi.RemoteException,
        com.ibm.bankexample.domain.InsufficientFundsException,
        com.ibm.bankexample.domain.FactoryException;

    void withdraw(java.lang.String accountNumber, double amount)
        throws java.rmi.RemoteException,
        com.ibm.bankexample.domain.InsufficientFundsException,
        com.ibm.bankexample.domain.FactoryException;
}
```

Likewise, the Remote Interface for our Account EJB is as follows:

```java
package com.ibm.bankexample.ejbs;

/**
 * This is the Enterprise Java Bean Remote Interface
 * for the Account Entity EJB.
 */
public interface Account extends javax.ejb.EJBObject {

    void deposit(double amount) throws java.rmi.RemoteException;

    java.lang.String getAccountNumber() throws java.rmi.RemoteException;

    double getBalance() throws java.rmi.RemoteException;

    java.lang.String getUserid() throws java.rmi.RemoteException;

    void setBalance(double newValue) throws java.rmi.RemoteException;

    void setUserid(java.lang.String newUserid) throws java.rmi.RemoteException;

    void withdraw(double amount) throws java.rmi.RemoteException;
}
```

Now, there are two other object types that we have identified that are useful in our system. The first is a value object that that represents the account information for display on the ATM. This class looks like the following:
public class AccountValue implements java.io.Serializable {
    private java.lang.String accountNumber;
    private double balance;
}

Of course, the AccountValue class also has getter and setter methods for the attributes, but these are left out for brevity.

Now, we almost have enough information to understand the implementation of the getAccounts() method of the ATM EJB. The implementation of this method is as follows:

```java
public java.util.Vector getAccounts(String userid) throws FactoryException {
    AccountFactory fact = new AccountFactory();
    Vector result = fact.getAccounts(userid);
    return result;
}
```

This method shows the standard pattern of methods in a Session Facade EJB. It finds the appropriate helper object (Action or, as in this case, Factory), invokes a business method on the helper, and then returns the result.

As this method indicates, we need to have an AccountFactory class to build AccountValues from Accounts. The class definition for this class is below:

```java
public class AccountFactory {
    private static AccountHome accountHome = null;
}
```

The implementation of the getAccounts(userid) method in AccountFactory is shown below:

```java
public java.util.Vector getAccounts(String userid) throws FactoryException {
    try {
        Vector vect = new Vector();
        AccountHome home = getAccountHome();
        Enumeration accountRefs = home.findByUserid(userid);
        while (accountRefs.hasMoreElements()) {
            Account acc = (Account) accountRefs.nextElement();
            AccountValue valueObject = new AccountValue();
            valueObject.setAccountNumber(acc.getAccountNumber());
            valueObject.setBalance(acc.getBalance());
            vect.addElement(valueObject);
        }
        return vect;
    } catch (Exception e) {
        throw new FactoryException("Cannot generate accounts due to wrapped exception "+ e);
    }
}
```

This method uses a cached AccountHome instance, which is obtained from the following method:
private AccountHome getAccountHome() {
    if (accountHome == null) {
        try {
            java.lang.Object homeObject = getInitialContext().lookup("com/ibm/bankexample/ejbs/Account");
            accountHome = (AccountHome) javax.rmi.PortableRemoteObject.narrow((org.omg.CORBA.Object) homeObject, AccountHome.class);
        } catch (Exception e) {
            // Error getting the home interface
            System.out.println("Exception " + e + " in createTimeSheetHome()");
        }
    }
    return accountHome;
}

As [Brown 2001] and [Gunther 2000] describe, caching EJB homes is a best practice in WebSphere® because of the time it takes to obtain a JNDI InitialContext and to obtain the EJB Home from the InitialContext.

So now that you've seen how Sessions, Entities, and Factories tie together, let's look at an example of an Action class. In this case, we have a Transfer object that handles the job of transferring money from one account to another. The Transfer is created in the following implementation of the transfer() method in the ATM EJB.

```java
public void transfer(String fromAccount, String toAccount, double amount) throws InsufficientFundsException, FactoryException {
    Transfer trans = new Transfer();
    trans.transfer(fromAccount, toAccount, amount);
}
```

Again, notice the same flow. However, this method does not need to return the value from the Action object. The Transfer class definition is as follows:

```java
public class Transfer {
    private static AccountHome accountHome;
}
```

The transfer() method's implementation is shown below:

```java
public void transfer(String fromAccount, String toAccount, double amount) throws InsufficientFundsException, FactoryException {
    try {
        Account from = getAccountHome().findByPrimaryKey(new AccountKey(fromAccount));
        Account to = getAccountHome().findByPrimaryKey(new AccountKey(toAccount));
        if (from.getBalance() < amount)
            throw new InsufficientFundsException();
        to.deposit(amount);
        from.withdraw(amount);
    } catch (Exception e) {
        throw new FactoryException("cannot perform transfer. Nested exception is " + e);
    }
}
```
As you can see, this method in the Transfer object handles the details of locating the two Account Entities, ensuring that the "From" account has a sufficient balance, depositing the transfer amount in the "To" account, and withdrawing the transfer amount from the "From" account. In the same way, you can see that other methods of Action objects could implement the other business rules in your system.

**Reasons for EJB objects**

So why do we need this second layer of objects? After all, didn't we move to Enterprise JavaBeans from CORBA and RMI to make things simpler? Why not just put all of the logic in your EJBs? There are several reasons for this. First and foremost, this is simply an application of layering. It is never a good idea to place too much behavior in a single object. If you layer the objects called by your EJBs in this way, you can gain the following benefits:

- Placing the behavior in a set of objects one level in from the session makes it easier to test them in isolation, perhaps even outside of the context of a J2EE application server.
- Multiple different Session Facade objects can use the same inner-layer objects without fear of inappropriate transaction semantics and without the potential network and marshalling/unmarshalling overhead of cross-Session bean calls.
- A second layer of objects allows you to vary the implementation of those objects (by using the Strategy pattern from [Gamma]) to take advantage of particular features of an application server, while still allowing the entire design to remain portable across Application Servers. For instance, [Brown 2000a] describes some particular caching strategies for speeding up EJB performance that work under the WebSphere Application Server, Advanced Edition, but that would not work under the IBM CICS EJB support. By providing two implementations of the same Factory class or Action class, you can keep the overall design portable, while taking maximum advantage of the peculiarities of each server.
- In cases where you do not need a JTA transaction context (for example, you are only working against a single data source), this pattern allows you to choose to deploy and build your applications either with or without EJBs. For instance, in some simple query cases it may be significantly more efficient to call a Factory directly from a Servlet in order to avoid the overhead of the EJB calls.

Also, we have found through review of several projects that reuse only rarely occurs at the Session level. This is because each Session will have a specific combination of transactional settings and method signatures for a specific application. Having a second layer of objects can instead result in reuse at the inner-layer level, where we have seen reuse in many projects, both within projects (across different Session beans) and across projects in the enterprise.

We have seen that if you employ this design strategy then your designs can often use Stateless session beans as your Facade objects. Since each Stateless Session bean is not unique to a single user, this allows you to gain the additional scalability that stateless beans provide.

Now that we've seen the kinds of objects that sit behind the facade, we can start to look at what kind of methods the facade will present to the outside world. We have seen that Facade methods generally fall into the following types:
• A **Collector** method often begins with "get" and returns a single object or a collection of objects (represented as an Enumeration in EJB 1.0 or a Java Collection in EJB 1.1 and 2.0). The collector method will defer its implementation to a Factory object as shown in [Brown 2000].

• An **Updater** method locates and updates an Entity bean or a set of Entity beans based on information held in Value objects passed in as the arguments. The method name often begins with "update" or "set." An updater method's implementation can be either deferred to a factory as in [Brown 2000] or it may be in a separate class.

• An **Action** method (like `transfer(String fromAcctNum, String toAcctNum, BigDecimal amount)`) defers its implementation to an Action object.

### Rules for creating Session Facades

Now that you've seen what the Session Facade interfaces look like, and what the objects are behind the Session Facades, the next question is "How many of these things do I have?" You don't want to have too many Session Facades, otherwise you lose the benefits of the Facade pattern. However, a single Session Facade for an entire application might become a "God Object" and cause problems of its own. Here are some rules for designing Session Facades to achieve the right level of granularity.

- Look for functional subsystems in your application. For instance subsystems named Order Management, Billing, and Shipping might be the source of three potential Facade objects in an application.
- Go back to your use cases and look for related groups of use cases. A group of related use cases (like buy a stock, sell a stock, and get a price quote) might suggest a cohesive subsystem like "Stock Trading." This single cohesive subsystem will probably share many inner-layer objects and be a good candidate for a Session Facade. [Sun 2001] discusses this approach in more depth.
- Do NOT make each individual use case into a Session Facade. This results in a system with far too large of a distribution cross-section. The clients will have to manage too many EJB references and Homes in this case.
- After an initial pass, look at the relationships between the second-layer objects in your design. If you see that there are disjoint groups of Value objects, Factories, and Actions then separate the Facade into two or more facades based around the actual groupings.

### Another Session pattern

So, now that you've been convinced that you should employ stateless Session Facades to wrap your data sources, and that you should employ strict layering in the implementation of these facades, you may be wondering where the user-specific application state (if any) will be held. Also, you may be wondering where use cases will actually be implemented, based on my insistence that session facades be designed to handle the requirements of multiple use cases.

To understand how this is applied, let's take the following definition of a Control object from [Jacobson 1999]: “Control classes represent coordination, sequencing transactions, and control of other objects and are often used to encapsulate control related to a specific use case.”
The solution here is to implement special "use-case controller" objects that correspond one-to-one with a single use case. Each use-case controller will maintain some user-specific state that corresponds to the information necessary to determine where in the use case a particular user is (for example, how much they have executed) and the information that the user has entered that is necessary to carry out the next steps of the use case. In this way, we have found a place where use cases can be realized and written in a truly GUI-independent way.

Now, what are the options for implementing these Use-case controller objects? Since they must by nature be stateful, we have two options that present themselves in a J2EE architecture:

- One option is to make the Use-case controllers serializable objects (JavaBeans) and then store and retrieve them in the HttpSession. This will, of course, only work in an application that is using the Servlet API. However, this will apply to a good many applications, even those that use an Applet or Application client GUI, since communication to server-side objects via HTTP (possibly using SOAP) is a common design strategy even in these cases. Using HttpSessions as the storage mechanism makes it easy on the programmer, as in most application servers (like WebSphere) the details of session persistence and failover are handled automatically. However, in WebSphere there is a limit to the size of the HttpSession that can be reasonably persisted (see [Gunther]), so this option will only work in the case where the Use-case controller objects are relatively small.
- Another option is to use Stateful Session beans as the Use-case controllers. Stateful session beans are a good solution in that the programmer does not have to worry about how the session information is maintained. In application servers where stateful session beans are automatically stored in a shared persistent store, this solution will even handle failover. However, in application servers like WebSphere where session beans are not stored in a shared persistent store, this solution can result in failure of the user session in the case of a server crash. Also, using a Stateful session bean in this way brings us back to the problem of increasing the total distribution cross-section of the application. Although in this case, since the Stateful session bean is itself acting as a facade, the problem is mitigated in that each client will only need to know about a small number of use-case controllers.

This pattern brings us full-circle back to our initial discussion of the Facade pattern. The successive layering in this pattern gives us the ability to focus only on the task at hand, while at the same time allowing us to compose our solutions out of reusable bits and pieces. We aren’t forced to design the entire system at once, but can apply these patterns iteratively and watch a useful set of reusable objects emerge as a result.

**Summary**

In this article I’ve briefly examined some of the rules that go into building Session Facades. You have seen how you can structure the objects that the Session Facades call, and how you can go about designing Session Facades. You have also seen a few approaches to designing Use-case controllers that mesh well with your session facades. Hopefully, applying these rules should improve the flexibility and performance of your EJB designs.
Acknowledgments

Thanks to Martin Fowler for reviewing an early draft of this paper and providing a number of insightful comments. Also thanks to Craig Larman for his very probing comments which helped me understand that there are really two complementary patterns for Session facades.

Footnotes

1 Erich Gamma, Richard Helms, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Design, Addison-Wesley, 1995, p. 185.

2 Erich Gamma, Richard Helms, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Design, Addison-Wesley, 1995, p. 186.

3 Erich Gamma, Richard Helms, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Design, Addison-Wesley, 1995, p. 187.

4 Erich Gamma, Richard Helms, Ralph Johnson, and John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Design, Addison-Wesley, 1995, p. 193.

Related topics

- Erich Gamma, Richard Helms, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Design*, Addison-Wesley, 1995. [Gamma]
- Martin Fowler, *Analysis Patterns: Reusable Object Oriented Models*, Addison-Wesley, 1997. [Fowler]
- "Session Facade," Sun Java Center J2EE Patterns, Java Developer Connection. [Sun 2001]
- Sun Microsystems, "EJB 2.0 Specification," Public Draft 2. [EJB 2.0]