This chapter contains a set of diverse patterns that solves problems involving transaction control, persistence, and performance. The chapter includes:

**Version Number.** Used to program your entity beans with optimistic concurrency checks that can protect the consistency of your database, when dealing with use cases that span transactions and user think time.

**JDBC for Reading.** The section on this performance-enhancing pattern discusses when to disregard the entity bean layer and opt for straight JDBC access to the database, for performance reasons, and discusses all the semantics involved with doing so.

**Data Access Command Bean.** Provides a standard way to decouple an enterprise bean from the persistence logic and details of the persistence store. Makes it really easy to write persistence logic.

**Dual Persistent Entity Bean.** A pattern for component developers, the Dual Persistent Entity Bean pattern shows how to write entity beans that can be compiled once and then deployed in either a CMP or a BMP engine-simply by editing the deployment descriptors.
Version Number

When a client initiates an update on the server side, based on data that it has read in a previous transaction, the update may be based on stale data.

How can you determine if the data used to update the server is stale?

Transactions allow developers to make certain assumptions about the data they handle. One of these assumptions is that transactions will operate in isolation from other transactions, allowing developers to simplify their code by assuming that the data being read and written in a transaction is fresh and consistent.

In an EJB context, this means that when a use case is executed (usually as a method on the session façade running under a declarative transaction), the code can update a set of entity beans with the assumption that no other transactions can modify the same entity beans it is currently modifying.

While transaction isolation works well when a use case can be executed in just one transaction, it breaks down for use cases that span multiple transactions. Such use cases typically occur when a user needs to manually process a piece of data before performing an update on the server. Such a use case requires an interval of user think time (that is, a user entering updates into a form). The problem with user think time is that it is too long, which makes it infeasible (and impossible in EJB) to wrap the entire process of reading from the server, thinking by the user, and updating of the server in one transaction. Instead, data is usually read from the server in one transaction, processed by the user, and then updated on the server in a second transaction.

The problem with this approach is that we no longer have guarantees of isolation from changes by other transactions, since the entire use case is not wrapped in a single transaction. For example, consider a message board administrative system, in which multiple individuals have moderator access on a forum of messages. A common use case is to edit the contents of a user-posted message for broken links or improper content. At the code level, this involves getting a message’s data in one transaction, modifying it during user think time, and then updating it in a second transaction. Now consider what can happen when two moderators A and B try to edit the same message at the same time:

1. Moderator A reads Message X in a transaction.
3. Moderator A performs local updates on his copy of the Message.
4. Moderator B performs local updates on her copy of the Message.
5. Moderator A updates Message X in one transaction.
6. Moderator B updates Message X in one transaction.

Once Step 6 occurs, all updates executed by Moderator A will be overwritten by those changes made by Moderator B. In Step 5, Moderator A successfully updated Message X. At this point, any copies of the message held by other clients is said to be stale, since it no longer reflects the current state of the Message entity bean. Thus, Moderator B updated the message on the basis on stale data.

In a message board system, such issues may not be much cause for concern, but imagine the ramifications of similar events happening in a medical or a banking system—they could be disastrous. The crux of the problem here is that the Moderator A’s and Moderator B’s actions were not isolated from each other. Because separate transactions were used for the read and update steps, there was no way to automatically check when the data used to update the server was based on a read that had become stale.

Therefore:

**Use version numbers to implement your own staleness checks in entity beans.**

A version number is simply an integer that is added to an entity bean (and its underlying table) as a member attribute. The purpose of this integer is to identify the state of an entity bean at any point in time. This can be achieved by incrementing the bean’s version number whenever an entity bean is updated. This incrementing of versions allows the detection of updates based on stale data, using the following procedure:

1. **Carry the version number along with any other data read from an entity bean during read transactions.** This is usually done by adding an entity bean’s version number to any data transfer objects used to copy its data to the client.

2. **Send the version number back to the entity bean along with any updated data.** When it comes time to perform the update, carry the original version number back with the newly updated data, and compare it with the entity bean’s current version before performing any updates.

3. **Increment the entity bean’s version number when performing an update.** If the current version of the entity bean is equal to that of the updated data from the client, then update the entity bean and increment its version.

4. **Reject the update if the version numbers do not match.** An update carrying an older version number than currently in the entity bean means that the update is based on stale data, so throw an exception.
Using version numbers in this manner will protect against the isolation problems that can occur when a use case spans multiple transactions. Consider the forum moderator example. If, before Step 1, the version number of message X was 4, then both Moderator A and Moderator B will retrieve this version number in their local copy of the message. At Step 5, Moderator A’s update will succeed, since the version he is carrying (4) matches that in Message X. At this point, Message X’s version number will be incremented from 4 to 5. At Step 6, Moderator B’s update will fail, since the version number this moderator is carrying (4) does not match the current version of Message entity bean X, which is currently 5.

When a stale update is detected, the usual recovery procedure is to notify the end user that someone has beat them to the update, and ask them to reapply their changes on the latest copy of server-side data.

The implementation of the Version Number pattern differs slightly, depending on the mechanisms used to access the entity beans. If we use data transfer objects to get and set data in bulk on the entity beans directly (as done with EJB 1.X applications), then the version number is added to the DTO in the entity bean’s `getXXXDTO` method, and the version number is checked with the current version in the entity bean’s `setXXXDTO` method, as in the following code block:

```java
public void setMessageDTO(MessageDTO aMessageDTO) throws NoSuchMessageException {
    if (aMessageDTO.getVersion() != this.getVersion())
        throw new NoSuchMessageException();

    this.setSubject(aMessageDTO.getSubject());
    this.setBody(aMessageDTO.getBody());
}
```

However, as discussed in the DTOFactory pattern, using DTOs as a mechanism for accessing entity beans directly is a deprecated practice as of EJB 2.0. Instead, the DTOFactory/session façade is responsible for getting data from an entity bean and updating the entity bean by directly calling `get/set` methods via the entity bean’s local interface.

Using this paradigm, a session bean is responsible for updating an entity bean directly via its `set` methods; thus the entity bean can no longer automatically check the version of a set of data before it updates itself. Instead, developers must adopt a programming convention and always remember to pass the version of a set of data they are about to update before beginning the update procedure, as in the following session bean method:

```java
public void updateMessage(MessageDTO aMessageDTO) {
    Message aMessage;
```
try //to update the desired message
{
    aMessage = this.messageHome.findByPrimaryKey(
        aMessageDTO.getMessageID() );

    aMessage.checkAndUpdateVersion(aMessageDTO.getVersion());

    //update the message
    aMessage.setBody(    aMessageDTO.getBody()    );
    aMessage.setSubject( aMessageDTO.getSubject() );
}
catch(IncorrectVersionException e)
{
    this.ctx.setRollbackOnly();
    throw new StaleUpdateException();
}
catch (...)
{
    ...
}

Upon the call to checkAndUpdateVersion, the Message entity bean will check
the version with its own internal version number and throw an IncorrectVer-
sionException if the versions do not match. If the versions do match, then the
entity bean will increment its own internal counter, as in the following code
block:

    public void checkAndUpdateVersion(long version)
    throws IncorrectVersionException
    {
        int currentVersion = this.getVersion();
        if( version != currentVersion)
            throw new IncorrectVersionException();
        else
            this.setVersion( ++currentVersion );
    }

The version numbering scheme described here can also be thought of as
implementing your own optimistic concurrency. Instead of having entity
beans being used by a long-running use case be locked from concurrent access,
we allow multiple users to access the data, and only reject an update when we
detect that stale data was used as a basis for the update. Databases that imple-
ment optimistic concurrency use a similar scheme to allow multiple clients to
read data, only rejecting writes when collisions are detected.

Similar implementations can be found that use timestamps instead of ver-
sion numbers. These two implementations are basically identical, although
using version numbers is simpler and protects against possible problems that
can occur in the unlikely event that the server’s clock is rolled back, or if the
database date and time come down to a small enough interval to eliminate the
possibility of invalid staleness checks.
The Version Number pattern guarantees that use cases executed across transactions will be properly isolated from each other’s changes, in the same way that use cases that execute within a single transaction are guaranteed to be isolated from the operations of other transactions. However, what happens in the infrequent event that both moderators attempt to update the server (Steps 5 and 6) at the exact same time? In this example, two instances of the Message entity bean could be loaded into memory with both containing the same version number. The call to checkAndUpdateVersion will thus succeed in both instances. Once the first transaction commits, the question then becomes: what happens when the second transaction attempts to commit?

The answer is that the second transaction will be correctly rolled back. Since both transactions are happening at the same time, the same transaction isolation level semantics that protect use cases that execute within one transaction will protect this particular operation from conflicts. The way it achieves this depends on how your database/application server handles concurrency:

- **Isolation of READ_COMMITTED with application server CMP verified updates.** Here the application server will compare the changed attributes in the Message entity bean (including the version number) with that in the database before committing. If the contents do not match (because a previous transaction incremented the version number and other attributes), then the application server will roll back the transaction. This is an optimistic concurrency check implemented at the application server level, allowing you to use a transaction isolation level of just READ_COMMITTED, since the application server guarantees consistency.

- **Isolation of READ_COMMITTED with verified updates implemented in BMP.** BMP developers can manually implement verified updates by comparing the version number in the current bean to that in the database in ejbStore. This can be achieved by modifying the SQL UPDATE statement to include a where version=X clause. Even if Moderator A’s transaction updated the database milliseconds before, this where clause will fail and the developer can manually roll back the exception.

- **Isolation of SERIALIZABLE with Database (DB) that supports optimistic concurrency.** If optimistic concurrency is not implemented at the application server level, then a transaction isolation level of SERIALIZABLE must be used to ensure consistency. If the database itself implements optimistic concurrency checks, then it will automatically roll back the transaction of Moderator B’s when it detects that ejbStore is trying to overwrite the data inserted by the first transaction.

- **Isolation of SERIALIZABLE with a DB that uses pessimistic concurrency.** Again, SERIALIZABLE must be used since the application server
won’t enforce consistency. However, since the database is using a pes-
simistic concurrency strategy, it will lock Message X’s row in the data-
base, forcing the MessageEntity.ejbLoad() of the second transaction to
wait until the MessageEntity.ejbStore() from the first transaction com-
pletes and commits. This means that when Moderators B’s transaction
calls checkAndUpdateVersion this check will correctly fail, since the mes-
sage X was not ejbLoad()ed until after Moderator A’s transaction had
committed.

■ Isolation of SERIALIZABLE with a SELECT FOR UPDATE. Some
application servers allow the CMP engine to be configured to issue
a SELECT FOR UPDATE during ejbLoad, by editing a deployment
descriptor setting. The purpose of this is to force a database that uses
optimistic concurrency to actually lock the underlying row. This will
cause the transactions to execute as in the previous option.

The takeaway point here is that, in the rare instance where the updates are
happening at the same time, consistency is maintained, and either the second
transaction will be detected at checkAndUpdateVersion time or the application
server or database will detect the collision and roll back the transaction—
either way, consistency is maintained.

Another important point to consider when using the Version Number pat-
tern is that it can cause problems when you have legacy or non-Java applica-
tions updating the same data as your EJB application. Legacy applications will
probably be using version numbers, resulting in consistency problems
between the EJB application and the legacy application. If it is under your con-
trol, ensure that other non-Java or legacy applications also properly update the
version number when performing updates. If changing the legacy applications
is completely beyond your control, then another solution is to implement trig-
gers in the database that will update the version numbers in the database auto-
matically. If you take this approach, don’t forget to remove the version number
incrementing code from your entity bean.

The Version Number pattern is most often used as a way to protect against
stale updates that occur when using data transfer objects. Once a DTO is used
to copy some data off of the server, this data could potentially be stale. Version
numbers help us detect the stale data at update time.
JDBC for Reading

In an EJB system that uses a relational database in the back end, an EJB client needs to populate a tabular user interface with server-side data, for display purposes.

When should a session façade perform direct database access instead of going through the entity bean layer?

* * *

Perhaps the most common use case encountered in distributed applications is the need to present static server-side data to a client in tabular form. Examples of tabular UIs constitute the majority of Web pages, where data is listed in tables or rows, such as a list of items in a catalog (as opposed to nontabular UIs such as the rare treelike or circular UI). Furthermore, this tabular data is usually read-only; clients tend to do a lot more browsing than updating of the pages they surfs.

One common scenario is an application that requires the presentation of a large amount of read-only data to the user, perhaps in the form of an HTML table. The table may represent line items in a large order, information on all employees in a company, or the characteristics of all products a company produces.

In Figure 3.1, each row in the table corresponds to one employee in the system and his/her department. On the server side, we would model this with an Employee and a Department entity bean. One way to populate the table would be to call a getEmployees() method on a session façade/data transfer object factory, which would call a finder method on an EmployeeHome object, return all employee’s, find each employee’s related Department entity bean, and create a custom data transfer object with the combined data from these two entity beans. The session bean would then return a collection of EmployeeDepartmentDTOs to the client.

<table>
<thead>
<tr>
<th>Employee</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam Berman</td>
<td>Development</td>
</tr>
<tr>
<td>Eileen Sauer</td>
<td>Training</td>
</tr>
<tr>
<td>Ed Roman</td>
<td>Management</td>
</tr>
<tr>
<td>Clay Roach</td>
<td>Architecture</td>
</tr>
</tbody>
</table>

Figure 3.1 HTML table of employees.
Depending on the EJB Server and applications, there are numerous problems with this approach:

- **The $n + 1$ entity bean database calls problem.** With BMP and certain implementations of CMP, retrieving data from $N$ entity beans will require $N + 1$ database calls. Although a good CMP implementation will allow bulk loading, developers should be aware of this dire problem. The $N + 1$ calls problem is as follows: In order to read data from $N$ entity beans, one must first call a finder method (one database call). The container will then execute ejbLoad() individually on each entity bean returned by the finder method, either directly after the finder invocation or just before a business method invocation. This means that ejbLoad() (which will execute a database call) will need to be called for each entity bean. Thus, a simple database query operation requires $N + 1$ database calls when going through the entity bean layer! Each such database call will temporarily lock a database connection from the pool, open and close connections, open and close result sets, and so on. Since most distributed systems have a separate box for the database, each of these database round trips would require a network call, slowing down the speed of each round trip and locking valuable database resources from the rest of the system. For our Employee and Departments example, running this use case will actually require $2N + 1$ database calls (one finder, $N$ Employee ejbLoads(), and $N$ Department ejbLoads()).

- **Remote call overhead.** If it goes through the entity bean remote interface (as opposed to the local interface), this method would also require $3N$ remote calls for $N$ rows of employee and department data. The remote calls break down as follows:

  - N calls to getValueObject() for each Employee.
  - N calls to getDepartment() on each Employee.
  - N calls to getValueObject() on each Department.

After grabbing each set of value objects, the session bean would then combine the value objects into the EmployeeProjectViewObjects.

- **Cumbersome for simple join operations.** Whether we use BMP or CMP, this typical use case requires the instantiation of multiple entity beans and traversal of their relationships. Imagine a slightly more complex scenario in which the table needed to list data from an Employee and a related Department, Project, and Company. This would not only require tens of lines of spaghetti code, but would significantly slow down a system because of the database calls, remote calls, and all the application server overhead incurred when traversing multiple entity bean relationships.
When the client side mainly requires tabular data for read-only listing purposes, the benefits of querying through the entity bean layer are less clear. Using local interfaces and a good CMP implementation will definitely reduce the performance problems with listing data via entity beans, but BMP developers are not so lucky. In BMP, these problems can only be alleviated by turning on entity bean caching, a luxury usually only available for single EJB server (or nonclustered) deployments in which the database is never modified outside of the EJB application. The remaining BMP developers are faced with a serious performance problem. Querying through the entity bean layer simply to list read-only data causes unacceptable performance problems.

Therefore:

**In BMP, perform listing operations on relational databases using JDBC. Use entity beans for update operations.**

If the data that the client UI requires is mainly used for listing purposes, then using JDBC to directly read the rows and columns required by the client can be far faster and more efficient than going through the entity bean layer. Using the previous example, the entire table of employees and departments could be read in bulk from the database in just one JDBC call, as opposed to the potentially required $3N$ remote calls and $N + 1$ database calls required if it is read through the entity bean layer.

After reading in the ResultSet, the data could then be added to Employee-DepartmentDTOs just as in the previous example, or it could be marshaled to the client by using HashMaps (as in the Data Transfer HashMap pattern) or in tabular form using RowSets, as in the Data Transfer Rowset pattern.

The decision to use straight JDBC instead of entity beans for reading data is a tough one for most developers, and has been the subject of raging debates ever since the advent of entity beans. After all, entity beans provide a nice encapsulation of data and data logic, they hide the persistence details such as the type of database being used, they model the business concepts in your system, and they make use of many container features such as pooling, concurrency, transactions, and so on. To go to a non-OO method of data access seems like a step back. Like all design patterns, there are trade-offs.

Using JDBC for reading purposes has the following advantages:

- **No transactional overhead for simple query operations.** Read-only operations do not need to use transactions. Querying the database from a stateless session bean with transactions turned off is more lightweight than querying entity beans. Often it is impossible to query an entity bean without a transaction.

- **Takes advantage of DB built-in caching.** Databases have sophisticated and powerful caches. By using JDBC for these operations we can make better use of the DB’s built-in cache. This becomes important when executing queries that span tables, because the database can cache the
results of this one bulk query, rather than cache individual table queries generated by entity bean \textit{ejbLoad}s calls. The next time a query is run, the one bulk JDBC query will come directly from the database cache.

- **Retrieve the exact data your use case requires.** Using JDBC, you can select the exact columns required across any number of tables. This stands in contrast to using an entity bean layer, in which the client may only need a couple of attributes from a variety of related entity beans. Those entity beans will need to load all of their attributes from the database even if a client only needs one attribute.

- **Perform queries in ONE BULK READ.** All the data a client requires is grabbed in one bulk database call. This is in direct contrast to the \(N+1\) database calls problem associated with entity beans.

Here are the trade-offs:

- **Tight coupling between business and persistence logic.** When working with an entity bean, a developer doesn’t know what the underlying persistence mechanism is. With this pattern, session bean data querying logic is now coupled to the JDBC APIs and is thus coupled to a relational database. However, other design patterns such as the Data Access Object pattern (not covered in this book) can be used to alleviate this problem.

- **Bug prone and less maintainable.** Bug-prone JDBC code is now mixed around the session bean layer, instead of nicely encapsulated behind entity beans. Changes to the database schema will require changes to multiple code fragments across the session façade. Again, the Data Access Object pattern can help here.

Finally, this pattern does not imply that entity beans should not be used at all, only that there are more efficient alternatives when the client needs to temporarily list data. In this pattern, JDBC is used for listing behavior, and the entity bean layer is used for updating behavior in an application.

Whereas the integrity of business/data objects and their relationships with other business objects are not that important when listing tables of data on a client, these concepts are critical when performing updates. Entity beans (or any other data object framework) encapsulate both data and \textit{rules} for changing that data. When updating an attribute on an entity bean, the entity bean may need to perform validation logic on its changes and institute updates on other entity beans in an application.

For example, consider an application with Book and Chapter entity beans. When modifying the title of a Chapter entity bean, the Chapter will need to perform validation on the new title, and internally call and modify its Book bean to notify it to change its table of contents. The Book entity bean may then need to modify other entity beans, and so on.
Performing updates via JDBC from the session façade forces a developer to write spaghetti code that mixes business logic with the complexities of data logic. All the rules, relationships, and validations required by particular business concepts would have to be hacked in the form of updates on rows and tables. The system would become very brittle to changes in the business requirements of the application.

Thus, where the client UI requires read-only tabular data and entity bean caching is not possible, use JDBC to read in data from the database, instead of going through the entity bean layer. All updates should still go through the domain object (entity bean) layer.

The JDBC for Reading pattern occurs behind a session façade or a data transfer object factory. Depending on what type of object is used to transfer the ResultSets contents to the client. (DTOFactory implies that DTOs will be returned to the client, whereas HashMaps or RowSets can be returned from the session façade).

**Related Patterns**

Fast Lane Reader (J2EE Blueprints)
Data Access Command Beans

An enterprise bean needs to access a persistent data store.

How can persistence logic and persistent store details be decoupled and encapsulated away from enterprise bean business logic?

* * *

When programming with a session bean layer that directly accesses the database (no entity beans), or when writing bean-managed persistent entity beans, a common practice is to mix the persistence logic in with the session bean or entity bean. For session beans, this usually entails writing data-store-specific access code (such as JDBC) mixed in with business logic. For entity beans, the standard practice is to write JDBC in the `ejbCreate()`, `ejbLoad()`, `ejbStore()` and `ejbRemove()` methods.

Although this gets the job done, this approach suffers from several drawbacks:

- **Data logic mixed in with business logic.** Mixing persistence logic in with business logic has terrible consequences for maintainability. Business logic becomes harder to distinguish among spaghetti persistence code, and persistence code becomes spread across the business layer instead of localized to one layer.

- **Tight coupling to a particular persistent data store (database) type.** By coding a particular persistence API into your business logic (such as JDBC), you tightly couple your application to one particular data store type (OODBMS, RDBMS, LEGACY). This makes it difficult to switch between data store types. Furthermore, on projects that include a legacy integration as well as a more modern data store, two different persistence APIs would be mixed in with business logic, making the code even more convoluted.

- **Vulnerable to data schema changes.** Minor schema changes in the database will require modification and recompilation of the persistence logic and also the business logic, since the two are tightly coupled.

- **Replication of logic.** JDBC programming requires repetitive coding (finding data sources, getting the connection, declaring prepared statements, parsing the results of a ResultSet, closing statements and connections, and so on) that needs to be replicated across all the EJBs that access the database.

The problems with coupling and maintainability described above make it difficult to write truly reusable business components. In many cases, reusability across data store types is not that important. Only projects whose requirements dictate the use of multiple types of data stores (RDBMS, ODBMS, LDAP, and so on), now or in the future, need to be concerned with the coupling of the
persistence logic to the details of the database implementation. However, coupling or not, the mixing of persistence logic with enterprise beans still poses significant maintainability problems.

Therefore:

Encapsulate persistence logic into data access command beans, which decouple enterprise beans from all persistence logic.

A data access command bean (DACB) is a plain Java bean style object that exposes a simple get/set/execute interface to the enterprise bean clients. Data access command beans encapsulate persistence logic and all details about the persistent data store, forming a separate, decoupled persistence layer that exists beneath the EJB layers.

A Data Access Command Bean pattern is similar to the original Command pattern (Gamma, et al., 1994), in that it exposes a very simple interface to its clients (see Figure 3.2). All a client needs to do is create a data access command bean, set any information it needs to perform its task, call execute, and then call getters to retrieve the data from the command, as well as a next method if the command returns multiple rows of values.

For example, consider an EmployeeServices session bean that handles all the management of employees for a company. EmployeeServices exposes (among others) methods to create and to search for employees within an organization. An example of the session façade, this bean doesn’t use a domain model, instead it interacts directly with the database.

To decouple the EmployeeServices bean from persistence logic, two data access command beans would be created, one that handles the creation of an employee, and one that handles finding all employees with the same name. The class diagrams for these DACBs are listed in Figure 3.3.
By using these data access command beans, the code in EmployeeServices is greatly simplified. The following code shows how the EmployeeServices session bean interacts with the InsertEmployeeCommand:

```java
InsertEmployeeCommand insEmp = null;
try {
    insEmp = new InsertEmployeeCommand();
    insEmp.setEmail("me@home.com");
    insEmp.setId(id);
    insEmp.setName("Ed");
} catch (DataCommandException e) {
    this.ctx.setRollbackOnly();
    throw new EJBException(e.getMessage());
}
```

Using the QueryEmployeeByName command is slightly different, since the command could potentially return multiple employees by the same name:

```java
try {
    QueryEmployeeByNameCommand query = 
        new QueryEmployeeByNameCommand();
    query.setName(name);
    query.execute();
} 
```
84 Chapter Three

Vector employees;
EmployeeDTO anEmployee;
while (query.next())
{
    anEmployee = new EmployeeDTO(query.getId(),
                                 query.getName(),
                                 query.getEmail());
}

return employees;

} catch (DataCommandException e)
{
    this.ctx.setRollbackOnly();
    throw new EJBEException(e.getMessage());
}

Note that the data access commands throw a DataCommandException, a
generic exception that serves to completely decouple the session bean client
from the fact the details of the database type.

Data access command beans are implemented in an inheritance hierarchy as
illustrated in Figure 3.4. Every data access command inherits from one of two
abstract classes: the BaseReadCommand and the BaseUpdateCommand.
These two reusable classes centralize all the setup, database execution and
cleanup code common in persistence logic.

Figure 3.4 Command inheritance implementation.
Implementing data access command beans is simple. If you are implementing an insert, update, or delete, then the class must extend from BaseUpdateCommand. If you are implementing a query, then extend from BaseReadCommand. The abstract superclasses remove most of the details of persistence logic even from the data access command bean developer, who only needs to code in the JNDI name of the data source used, the actual use cases specific SQL string to be executed, and all the use-case-specific gets/sets:

```java
public class InsertEmployeeCommand extends BaseUpdateCommand
{
    static String statement =
        "insert into Employees (EMPLOYEEID,NAME,EMAIL) values (?,?,?)";
    static final String dataSourceJNDI = "bookPool";

    protected InsertEmployeeCommand() throws DataCommandException
    {
        super(dataSourceJNDI, statement);
    }

    public void setEmail(String anEmail) throws DataCommandException
    {
        try{
            pstmt.setString(3, anEmail);
        } catch (SQLException e) {
            throw new DataCommandException(e.getMessage());
        }
    }

    //more sets
}
```

The advantages to the Data Access Command Bean pattern are:

- **Decouples business logic from persistence logic.** All the tedious and repetitive persistence logic is encapsulated behind a simple Java bean/command style interface. Business logic no longer needs to worry about ResultSet parsing, driver/statement tracking, and so on.

- **Creates a persistence layer.** Extracting all persistence logic to a layer of data access command beans (beneath the EJB layers) helps both layers to change independently of each other, helping to minimize the effects of changes to one layer on the other.

- **Data source independent.** DACBs can access relational database management systems (RDBMSs), object-oriented database management systems (OODBMSs), legacy adaptors, or any persistence store, all
transparently to the client. In fact, migrations between databases are
easier since persistence logic is localized to one layer.

- **Useable on any tier.** Although this chapter illustrates the pattern in an
  EJB context, data access command beans can provide a clean, robust
  persistence mechanism in any scenario (EJB, Servlets, JSPs, Taglibs, and
  so on).

- **Consistent interface.** The command style interface remains consistent
  for all DACBs. Even multiple types of data stores can be supported, all
  transparent to the user.

The cons of this pattern are:

- **Adds an extra layer of objects.** An extra layer of command beans must
  be written to separate the persistence logic from the other layers.

- ** Doesn’t support advanced JDBC features.** Features such as batch
  updating are not explicitly supported by the data access command bean
  (as illustrated here).

The Data Access Command Bean pattern provides a simple extensible way
to decouple persistence logic from enterprise beans, making it an attractive
mechanism for handling persistence behind the session façade and BMP entity
beans. DACBs should be used in parallel with the JDBC for Reading pattern.
The interface of the command beans can also be slightly modified to support
returning RowSets directly, for the Data Transfer RowSet pattern.

**Related Patterns**

- Data Access Command Bean (Matena and Stearns, 2001)
- Data Access Object (J2EE Blueprints; Alur, et al., 2001)
Dual Persistent Entity Bean

An EJB developer needs to write entity bean components that support both CMP and BMP.

**How can an entity bean be designed to support either CMP or BMP at deployment time?**

***

The environment in which an entity bean component will be deployed can vary widely from project to project. In the best case, a team will have access to an application server with a good CMP implementation, which they can use to gain significant performance enhancements that are not possible when using BMP. Often a team will be using an application server with poor CMP support or lack of support for their database. In this case, BMP is a requirement. This puts an entity bean component developer in a tough situation. How can they provide a component that can fit both situations?

One way to achieve this is to ship two separate versions of the same entity bean component. One packaged for CMP, the other for BMP. Unfortunately, this approach would require that the component developer maintain two separate code bases/components, making testing, debugging, and maintenance more difficult.

A truly portable EJB component should be deployable in any J2EE-compliant server, in a wide variety of environments and configurations. By portable, this means that the component should be customizable without any reprogramming or compiling. The only source of modification should be the deployment descriptors.

Therefore:

**To build more portable components, write entity beans that support both CMP and BMP, by separating business logic into a CMP-compliant superclass and BMP persistence logic into a subclass. Deployment descriptor settings can be used to select between the two at deployment time.**

Entity beans can be made to support both CMP and BMP by splitting entity bean logic into two classes: a CMP-compliant superclass, and a subclass that extends the superclass implementations of `ejbStore`, `ejbLoad`, and other methods. This new component can be used to choose its persistence mode at deployment time, by making minor changes to the standard `ejb-jar.xml` file.

For example, consider an Account entity bean. The Account entity bean contains two attributes: an `account id` and a `balance`. It also has three business methods: `deposit`, `withdraw`, and `balance`, and one special finder method: `findByBalance(int)`. As a dual persistent entity bean, the Account entity bean would look like Figure 3.5.
The CMP superclass contains the business methods and abstract get/set methods (abstract attribute accessors are required by EJB 2.X CMP), and simple implementations of required EJB methods such as set/unSetEntityContext and ejbCreate(). Note that the implementations of ejbLoad, ejbStore, and ejbRemove are empty implementations. Finder methods do not need to be implemented in the CMP class, since these are declared separately in the deployment descriptor.

The BMP subclass provides concrete implementations of the accountID and balance attributes, and their get/set accessors. Other than that the only extra logic this class requires is real implementations of persistence-related methods: ejbCreate, ejbLoad, ejbStore, and ejbRemove. Finder methods also need to be implemented, whereas the CMP superclass relied on query definitions in the ejb-jar.xml file. Note that the BMP does not need to reimplement the business logic methods, set/unSetEntityContext, or ejbActivate/Passivate, since these are inherited from the superclass.

At deployment time, the CMP or BMP classes can be chosen simply by changing the ejb-jar.xml file. Specifically, the <ejb-class> tag will need to refer to either the CMP superclass or the BMP subclass. Obviously, the <persistence-type> tag will need to select “container” or “bean managed” as well. If you

---

<table>
<thead>
<tr>
<th>abstract AccountCMPBean</th>
</tr>
</thead>
<tbody>
<tr>
<td>entityContext ctx</td>
</tr>
<tr>
<td>//EJB 2.0 accessors</td>
</tr>
<tr>
<td>abstract getBalance()</td>
</tr>
<tr>
<td>abstract setBalance(int)</td>
</tr>
<tr>
<td>abstract getAccountID()</td>
</tr>
<tr>
<td>abstract setAccountID(int)</td>
</tr>
<tr>
<td>//business methods</td>
</tr>
<tr>
<td>withdraw(int)</td>
</tr>
<tr>
<td>deposit(int)</td>
</tr>
<tr>
<td>balance()</td>
</tr>
<tr>
<td>//ejb required methods</td>
</tr>
<tr>
<td>setEntityContext(ctx)</td>
</tr>
<tr>
<td>unSetEntityContext()</td>
</tr>
<tr>
<td>ejbCreate(id, balance)</td>
</tr>
<tr>
<td>ejbPostCreate(id, balance)</td>
</tr>
<tr>
<td>ejbStore()</td>
</tr>
<tr>
<td>ejbLoad()</td>
</tr>
<tr>
<td>ejbRemove()</td>
</tr>
<tr>
<td>ejbActivate()</td>
</tr>
<tr>
<td>ejbPassivate()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AccountBMPBean</th>
</tr>
</thead>
<tbody>
<tr>
<td>entityContext</td>
</tr>
<tr>
<td>accountID</td>
</tr>
<tr>
<td>balance</td>
</tr>
<tr>
<td>//overridden accessors</td>
</tr>
<tr>
<td>getBalance()</td>
</tr>
<tr>
<td>setBalance(int)</td>
</tr>
<tr>
<td>getAccountID()</td>
</tr>
<tr>
<td>setAccountID(int)</td>
</tr>
<tr>
<td>//overridden ejb methods</td>
</tr>
<tr>
<td>ejbCreate(id, balance)</td>
</tr>
<tr>
<td>ejbStore()</td>
</tr>
<tr>
<td>ejbLoad()</td>
</tr>
<tr>
<td>ejbRemove()</td>
</tr>
<tr>
<td>//hard coded finders</td>
</tr>
<tr>
<td>ejbFindByPrimaryKey()</td>
</tr>
<tr>
<td>ejbFindByBalance(int)</td>
</tr>
</tbody>
</table>

---

**Figure 3.5** A dual persistent entity bean.
choose CMP, the *ejb-jar.xml* will need to be configured with CMP specific tags to add a schema, attributes, finders, and so on. The schema will also need to be mapped to an underlying data store using the proprietary mechanism provided by the application server it is being deployed on. On the other hand, if you deploy with BMP, the *ejb-jar.xml* will likely need to add a SQL *DataSource* via the `<resource-ref>` tags, and that’s it.

Besides creating more portable entity beans, another use of this pattern is migrating BMP entity beans to CMP. Many pre-EJB 2.0 applications were written in BMP. The CMP support provided by the EJB 1.X specifications were often insufficient for the needs of nontrivial applications, furthermore, many CMP implementations available on the market at the time suffered from poor performance. All of these legacy EJB applications could benefit by moving from EJB 1.X BMP to newer and more sophisticated CMP. Unfortunately, the migration process from BMP to CMP can be very tricky. One solution would be to completely rewrite the component using CMP. This option would require a lot more up-front work, and would essentially require cutting and pasting business logic from one entity bean to the other. This is hardly an efficient way to convert BMP beans to CMP. Using the Dual Persistent Entity Bean pattern, an existing BMP entity bean can be refactored into CMP by creating a superclass and moving code to it, leaving only the attributes, attribute accessors, and persistence-related methods in the subclass. The new superclass can be tested and deployed, and the subclass can be removed later if necessary.