

---

# 8

## *Distributed Component Models*

*Beings are, so to speak, interrogated with regard to their being. But if they are to exhibit the characteristics of their being without falsification, they must for their part have become accessible in advance as they are in themselves. The question of being demands that the right access to being be gained and secured in advance with regard to what it interrogates.*

—Martin Heidegger  
*Being and Time*

As a phenomenologist, Heidegger believed that there was no purpose in seeking an underlying truth to a being beyond that truth it exposed to the world outside—in other words, one is what one does. The phenomenological approach actually applies very well to software engineering, in which one of your central tasks, especially in a business system, is to identify key concepts in the business, capture them in software, and expose them to the outside world. When I introduced Enterprise JavaBeans as the centerpiece of the Java 2 Enterprise Edition in Chapter 6, *Other Enterprise APIs*, I noted that EJB was a distributed component model. A component model specifies how objects should be written in order to make themselves accessible to the outside world. A good component model is backed by an open standard so that a system can be built from components developed by many independent sources.

EJB takes care of almost everything covered in this book for you. In other words, if you deploy a container-managed EJB application, you can be successful without knowing JDBC or component model issues. Java development, however, has not reached the point at which everyone is using Enterprise JavaBeans. Furthermore, EJB does not address all distributed-computing concerns perfectly. There are

three common situations in which you will want to know more of the details of enterprise database application development than EJB requires:

- Container-managed persistence does not work for most EJB deployment environments. For example, systems that save some of their data to specialized data stores, such as a digital asset management repository, require bean-managed persistence.
- Not all applications require the power of an EJB solution.
- Even when the features supported by EJB are required, there may be other issues—for example, cost—that make the choice of EJB impractical.

You have already covered what you need to know for the first scenario, the JDBC API. Because EJB worries about the details of managing your JDBC-based transaction, a bean-managed EJB developer does not really need to understand all of the issues covered in this chapter. Anyone faced with the other two scenarios, however, will need to tackle some or all of the issues that distributed component models generally handle for you. This chapter shows you how to address those issues. It also introduces concepts, such as *façades*, that can help support EJB environments.

As I mentioned in Chapter 6, some of the issues EJB handles for you include security, searching, transactions, and persistence. In this chapter, you will look at three of those issues independent of EJB: security, searching, and transactions. Because the focus of this book is database programming, an entire chapter—Chapter 9, *Persistence*—is dedicated to persistence. The last part of the equation is building a client to use your component model. Chapter 10, *The User Interface*, addresses user interface issues.

## *Kinds of Distributed Components*

Distributed components can be broken into two categories: *persistent components* and *process-oriented components*. Persistent components represent the fundamental, unchanging business concepts that make up your model. Process-oriented components, however, manage the business processes that tie your persistent components together. EJB refers to process-oriented components as session beans; it refers to persistent components as entity beans.

If you think about the problem of a banking application, an `Account` is a persistent component, whereas a `BankTransaction`—not to be confused with a database transaction—is process-oriented. The `Account` is persistent because it is something the bank wants to track across an extended period of time. The `BankTransaction`, however, does not really represent a thing that needs to be tracked on its own across time. It simply exists during a client session to support the withdrawal of money from a checking account or the deposit of money into a savings account.

If you write your own distributed application without the assistance of EJB, nothing demands you make this distinction. For the sake of being able to best leverage the tools in this book for a migration to EJB, however, you will continue with this distinction. You need persistent business objects that represent our key concepts and non-persistent process objects that represent operations on those persistent business objects.

## *Process-Oriented Components*

Process-oriented components manage your business transactions. Perhaps you have noticed by this point that the term *transaction* is heavily overloaded. On the one hand, I have used it for business transactions such as deposits, withdrawals, account creation, etc. On the other hand, I have used it in terms of database and component transactions. Process-oriented components support the former: business transactions.

In the banking application, the class `BankTransaction` represents common business processes that support account management. This “session” component has a one-to-one relationship with the client it serves. If you and I both use the application on our respective computers, we will each have our own `BankTransaction` instance. These session components do not require a lot of infrastructure to support them. They will use the architecture’s underlying transaction support to manage your component transactions—to begin them and either to commit them or roll them back—and you will need to be able to get references to them from clients, but they have no issues unique to themselves that you need to support.

## *Persistent Components*

Persistent components are more complex than session components; EJB did not even require support for persistent components until Version 1.1. Today, support for persistent components comes in the form of entity beans. These components represent something in your system that needs to last beyond the current session. One key feature of an entity component is that it has a clear identity. By identity, I refer to the same thing we mean when we talk about the identity of things in the real world. What is it about the persistent component that makes you consider it to be the same component across time?

My bank account, for example, is not interchangeable with your bank account. Furthermore, if you and I both need to see my account information, something needs to guarantee that we are both looking at the same thing. In the database, this is generally accomplished via primary keys. Anything about an object can change over time and we know it is still the same object because it shares the same primary key with its past state.

Depending on the nature of your application, what constitutes a unique identifier can be highly dependent on the kind of component you support. An ideal unique identifier has no meaning built into it; it is just a number or a string that uniquely identifies the component. Unfortunately, not all systems have been built with good unique identifiers. If you are tasked with building business components that persist against a legacy database, you may find that some use social security numbers, email addresses, or other meaningful values as their primary keys.

In a well-designed distributed system, you will generate unique identifiers to support the identification of your business objects. In the banking application, you use a Java `long` field, `objectID`, in each persistent component. No two components in the system will share the same `objectID`. You therefore need a mechanism for generating unique `long` values.

You could rely on the ID generation mechanism supported by your database of choice. Unfortunately, there is no standard for ID generation. Even if there were a standard supported by JDBC, you would leave out the ability to support such non-JDBC data stores as object databases and digital media repositories. You are therefore going to develop a custom, data-store-independent ID generation tool.

From a performance perspective, you do not want to have to go to your data store each time you need a new unique ID. Such a scheme could end up doubling the time it takes to create new objects in your system. It could also be very costly if you engage in clustering.\* A solution to the problem of ID generation is a *node identifier*.

A node identifier is a unique key that enables a server to generate numbers within a process that are guaranteed to be unique even if other processes also generate unique IDs. On start up, a node in a cluster finds a sequence generator somewhere on the network and requests a unique node identifier. The sequence generator will go to some sort of persistent store to grab the next available node ID, increment it, and save the incremented value back to the persistent store. That value will then be returned to the requesting node to use as a seed in `objectID` generation.

Consider a banking application with two server nodes enabling clients to create new accounts. The first server gets a node identifier 1 from the sequence generator and the second server gets a 2. Using that 1, the first server can begin generating unique IDs without consulting the sequence generator again. It generates its unique `objectID` values by multiplying the node identifier by one million and then adding an internally incremented number. Thus, the first `objectID` it generates will be 1000000, the second, 1000001, and so on. It will continue until the process ends or

---

\* *Clustering* is having multiple processes serving up objects to clients. In such an environment, you would end up with a bottleneck at the central ID generation point if that point were responsible for generating each and every identifier.

it reaches 1999999—whichever comes first. When it reaches one of those points, it goes to the sequence generator for a new node identifier and generates new objectID values all over again. It might, for example, get 3 as the next node identifier and thus generate 3000000 as the next objectID after it generates 1999999.

You may have noted that you have simply traded one central bottleneck for another. This bottleneck, however, is negligible. After all, a node only grabs a new node identifier at every million new objects or at every start up. Excepting for system startup, the chances of two nodes asking for a node identifier at the same time are quite slim. As a result, the bottleneck is more of a potential than an actual bottleneck.

Another potential downside of the node-identifier approach is that it is capable of rapidly burning unique identifiers. You therefore should tailor the number of object IDs generated before a new node identifier is retrieved to be representative of the uptime of a server. If, for example, a node is likely to be up only five minutes at a time, you want to make sure it is only going to multiply the node identifier by a small number such as 1,000 or 10,000. If the node is up for weeks, months, or years at a time, you can push that number up to 1,000,000 or 10,000,000. With the multiplier set correctly and a 64-bit object ID, it should be virtually impossible to run out of object IDs for most business systems.

You can probably come up with any number of alternate algorithms that accomplish the same task. Example 8-1 provides the code for an abstract `SequenceGenerator` class that implements this algorithm without relying on any specific data store technology. It provides a factory method for accessing a sequence generator specific to whatever data store you might be using.

*Example 8-1. A Generic API for Generating Unique Sequences Across Multiple Processes*

```
package com.imaginary.lwp;

import com.imaginary.lwp.jdbc.JDBCGenerator;

public abstract class SequenceGenerator {
    static private long         currentNode = -1L;
    static private SequenceGenerator generator = null;
    static private long         nextID     = -1L;

    /**
     * Generates the next unique value in the sequence
     * having the specified name. By creating a generic
     * sequence generation scheme, we can re-use this
     * scheme for other kinds of sequences.
     * @param seq the name of the desired sequence
     * @return the next value in the specified sequence
     * @throws com.imaginary.lwp.SequenceException
     * the desired sequence could not be generated
     */
}
```

*Example 8-1. A Generic API for Generating Unique Sequences Across Multiple Processes (continued)*

```

static public synchronized long generateSequence(String seq)
    throws SequenceException {
    if( generator == null ) {
        // the class name for a concrete sequence generator
        // for example, com.imaginary.lwp.jdbc.JDBCGenerator
        String cname = System.getProperty(LWPPProperties.SEQ_GEN);

        if( cname == null ) {
            // use the JDBC generator if none specified
            generator = new JDBCGenerator();
        }
        else {
            try {
                // instantiates a concrete sequence generator
                generator =
                    (SequenceGenerator)Class.forName(cname).newInstance();
            }
            catch( Exception e ) {
                throw new SequenceException(e);
            }
        }
    }
    return generator.generate(seq);
}

/**
 * A convenience method that uses the unique
 * object ID generation algorithm to create unique
 * object IDs without having to go to the data store
 * for every single ID.
 * @return a unique objectID
 * @throws com.imaginary.lwp.SequenceException
 * the next ID could not be determined
 */
static public synchronized long nextObjectID() throws SequenceException {
    // if this is the first object ID after process start,
    // (currentNode == -1)
    // or if we have used up all of our ID's
    // (nextID >= 999999L)
    // we need to go to the data store and get a new one
    if( currentNode == -1L || nextID >= 999999L ) {
        currentNode = generateSequence("node");
        // if currentNode < 1, this is the first ever
        if( currentNode < 1 ) {
            // start at 1, keep all objectID > 0
            nextID = 1;
        }
    }
}

```

*Example 8-1. A Generic API for Generating Unique Sequences Across Multiple Processes (continued)*

```

        else {
            nextID = 0;
        }
    }
    else {
        nextID++;
    }
    return ((currentNode*1000000L) + nextID);
}

public SequenceGenerator() {
    super();
}

/**
 * Generators for specific data storage technologies
 * will implement this method to generate
 * the next value in the sequence.
 * @param seq the sequence to be generated
 * @return the next value in the specified sequence
 * @throws com.imaginary.lwp.SequenceException
 * the sequence could not be generated
 */
public abstract long generate(String seq) throws SequenceException;
}

```

This class supports any kind of sequence generation, not simply node or objectID generation. Anything wanting a unique identifier asks for the next number in the sequence by calling the `generateSequence()` method with the name of the sequence to be generated. This method relies on a concrete implementation of the class to support the actual generation of unique identifiers. It specifically looks to a system property for the name of a `SequenceGenerator` subclass. It instantiates an instance of that class and then calls the `generate()` method to generate the next value in the sequence. By taking this indirect approach, your system can hide any number of sequence generation algorithms behind the same API. Your application is not at all dependent on any particular algorithm.

The `SequenceGenerator` class uses the `generateSequence()` method within a custom method for generating unique `objectID` values, `nextObjectID()`. Whatever your mechanism for generating sequences, the `nextObjectID()` method will use a sequence named `node` to generate node identifiers and the `objectID` values that follow.

The abstract `generate()` method must be implemented by concrete subclasses that provide the actual algorithm for generating sequences. Example 8-2 is a JDBC implementation of the abstract sequence generator from Example 8-1.

*Example 8-2. A JDBC Implementation of the SequenceGenerator Abstract Class*

```

package com.imaginary.lwp.jdbc;

import com.imaginary.lwp.SequenceException;
import com.imaginary.lwp.SequenceGenerator;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

/**
 * A JDBC-based sequence generator that implements LWP's
 * <CODE>SequenceGenerator</CODE> interface. To use this sequence
 * generator, your database must have the following data model:
 * <PRE>
 * CREATE TABLE SEQGEN(
 *     NAME          VARCHAR(25)      NOT NULL PRIMARY KEY,
 *     NEXT_SEQ      BIGINT           NOT NULL DEFAULT 1,
 *     LASTUPDATETIME BIGINT          NOT NULL);
 *
 * CREATE UNIQUE INDEX SEQGEN_IDX ON SEQGEN(NAME, LASTUPDATETIME);
 * </PRE>
 * <BR>
 * Last modified $Date: 2000/08/08 17:17:20 $
 * @version $Revision: 1.12 $
 * @author George Reese (borg@imaginary.com)
 */
public class JDBCGenerator extends SequenceGenerator {
    /**
     * The SQL to insert a new sequence number in the table.
     */
    static public final String INSERT =
        "INSERT INTO SEQGEN(NAME, NEXT_SEQ, LASTUPDATETIME) " +
        "VALUES(?, ?, ?)";

    /**
     * Selects the next sequence number from the database.
     */
    static public final String SELECT =
        "SELECT NEXT_SEQ, LASTUPDATETIME " +
        "FROM SEQGEN " +
        "WHERE NAME = ?";

```

*Example 8-2. A JDBC Implementation of the SequenceGenerator Abstract Class (continued)*

```

/**
 * The SQL to one-up the current sequence number.
 */
static public final String UPDATE =
    "UPDATE SEQGEN " +
    "SET NEXT_SEQ = ?, " +
    "LASTUPDATETIME = ? " +
    "WHERE NAME = ? " +
    "AND LASTUPDATETIME = ?";

/**
 * Creates a new sequence.
 * @param conn the JDBC connection to use
 * @param seq the sequence name
 * @throws java.sql.SQLException a database error occurred
 */
private void createSequence(Connection conn, String seq)
    throws SQLException {
    PreparedStatement stmt = conn.prepareStatement(INSERT);

    stmt.setString(1, seq);
    stmt.setLong(2, 1L);
    stmt.setLong(3, (new java.util.Date()).getTime());
    stmt.executeUpdate();
}

/**
 * Generates a sequence for the specified sequence in accordance with
 * the <CODE>SequenceGenerator</CODE> interface.
 * @param seq the name of the sequence to generate
 * @return the next value in the sequence
 * @throws com.imaginary.lwp.SequenceException an error occurred
 * generating the sequence
 */
public synchronized long generate(String seq) throws SequenceException {
    Connection conn = null;

    try {
        PreparedStatement stmt;
        ResultSet rs;
        String dsn = System.getProperty(LWPProperties.DSN);
        Context ctx = new InitialContext();
        DataSource ds = (DataSource)ctx.lookup(dsn);
        long nid, lut, tut;

        conn = ds.getConnection();
        conn.setAutoCommit(false);
        stmt = conn.prepareStatement(SELECT);

```

*Example 8-2. A JDBC Implementation of the SequenceGenerator Abstract Class (continued)*

```
stmt.setString(1, seq);
rs = stmt.executeQuery();
if( !rs.next() ) {
    try {
        // if the sequence does not exist, create it
        createSequence(conn, seq);
    }
    catch( SQLException e ) {
        String state = e.getSQLState();

        // if a duplicate was found, retry sequence generation
        // 23505 == duplicate unique index field
        if( state.equals("23505") ) {
            return generate(seq);
        }
        throw new SequenceException(e);
    }
    return 0L;
}
// the next identifier
nid = rs.getLong(1);
// a last update ID to verify concurrency
lut = rs.getLong(2);
tut = (new java.util.Date()).getTime();
if( tut == lut ) {
    tut++;
}
stmt = conn.prepareStatement(UPDATE);
stmt.setLong(1, nid+1);
stmt.setLong(2, tut);
stmt.setString(3, seq);
stmt.setLong(4, lut);
try {
    int rc = stmt.executeUpdate();

    if( rc != 1 ) {
        conn.rollback();
        return generate(seq);
    }
    else {
        conn.commit();
    }
}
catch( SQLException e ) {
    throw new SequenceException(e);
}
return nid;
}
```

*Example 8-2. A JDBC Implementation of the SequenceGenerator Abstract Class (continued)*

```
    catch( SQLException e ) {
        throw new SequenceException(e);
    }
    catch( NamingException e ) {
        throw new SequenceException(e);
    }
    finally {
        if( conn != null ) {
            try { conn.close(); }
            catch( SQLException e ) { }
        }
    }
}
```

## Security

Security for distributed systems falls into two problem domains: *business component security* and *network security*. Proper business component security prevents users from doing things they should not be able to do to the shared business objects. Proper network security, on the other hand, prevents snoopers from intercepting your network traffic and seeing things they should not.

### Component Security

Component security involves authenticating the people accessing your system and validating their access requests against a set of established privileges. Distributed computing, especially in a web environment, introduces a few quirks that make component security especially troublesome. Fortunately for EJB users, security is handled for you by your EJB container at every level. You just specify security policies at deployment time.\*

#### Authentication

Today applications tend to rely on a weak authentication method based on a user ID/password combination. This authentication method requires you to store a list of users and their passwords in the system. When a client presents a user ID and password, and the specified password matches the password you have in your database for that user ID, your system is considered to have authenticated that user. In other words, your system views the fact that the correct password was provided as proof that the user is who that user claims to be.

---

\* If you are intent on building your own security system, you should probably use the Java security API as specified in the `java.security` package. A discussion of this package is well beyond the scope of this book.

Upon initial authentication of a user, your system needs a way to keep track of that user, i.e., to keep track of the user’s identity across time. One of the greater challenges of security in an Internet environment is that you cannot rely on a constant network connection between the client and server. While a client Java application maintains a constant connection, a servlet-based application uses HTTP and therefore does not maintain a connection. The challenge of identity is thus the challenge of knowing that two HTTP requests are in fact from the same user.

Once you know that the users are who they say they are, you need to validate at each step of the way the type of access they are requesting. To avoid the hassle of reauthentication, you should, upon authentication, provide a client with some sort of token that identifies the user to the system. The client then takes that token and sends it to the server every time it wishes to perform an operation. By placing this burden on the client, you make it responsible for solving the identity problem. Figure 8-1 is an activity diagram that illustrates what needs to happen at a conceptual level.

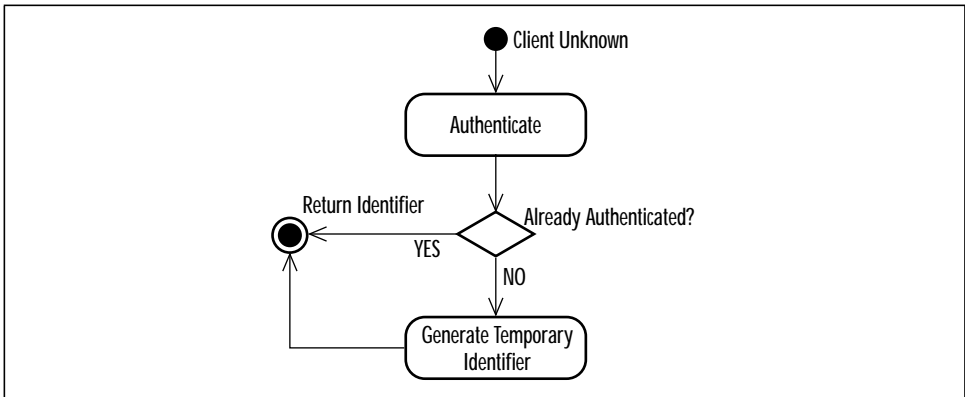


Figure 8-1. An activity diagram illustrating authentication

To avoid tying your infrastructure to a particular technology for storing users and passwords, you need to use a design similar to the one you used for sequence generation. Specifically, the banking application will rely on a generic Authenticator interface that can be implemented by any number of technology-specific classes, and these classes can be assigned to the application at deployment time. Example 8-3 shows the Authenticator interface.

*Example 8-3. The Data Store Independent Authenticator Interface*

```
package com.imaginary.lwp;
```

```
/**
```

```
 * Authenticates a user ID/password pair. Different applications may provide
```

*Example 8-3. The Data Store Independent Authenticator Interface (continued)*

```

* their own authenticator and specify it in their LWP configuration
* file using the &quot;imaginary.lwp.authenticator&quot; property.
* <BR>
* Last modified $Date: 2000/08/08 17:17:20 $
* @version $Revision: 1.12 $
* @author George Reese (borg@imaginary.com)
*/
public interface Authenticator {
    /**
     * Authenticates the specified user ID against the specified
     * password.
     * @param uid the user ID to authenticate
     * @param pw the password to use for authentication
     * @throws com.imaginary.lwp.AuthenticationException the
     * user ID failed to authenticate against the specified password
     */
    void authenticate(String uid, String pw) throws AuthenticationException;
}

```

Just as the `SequenceGenerator` class requires subclasses to implement a `generate()` method to perform the specific sequence generation tasks, the `Authenticator` interface requires subclasses to implement an `authenticate()` method to perform authentication specific to a given technology. The focus of this book is JDBC, so the banking application uses a JDBC implementation of the `Authenticator` interface in Example 8-4.

*Example 8-4. A JDBC Implementation of the Authenticator Interface*

```

package com.imaginary.lwp.JDBCAuthenticator;

import com.imaginary.lwp.Authenticator;
import com.imaginary.lwp.AuthenticationException;
import com.imaginary.lwp.AuthenticationRole;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

/**
 * Implements the <CODE>Authenticator</CODE> interface to authenticate
 * a user ID/password against values stored in a database. This class
 * expects the following table structure:
 * <TABLE>
 * <TR>

```

*Example 8-4. A JDBC Implementation of the Authenticator Interface (continued)*

```

* <TH><CODE>LWP_USER</CODE></TH>
* </TR>
* <TR>
* <TD><CODE>USER_ID (VARCHAR(25))</CODE></TD>
* </TR>
* <TR>
* <TD><CODE>PASSWORD (VARCHAR(25))</CODE></TD>
* </TR>
* </TABLE>
* If you want a more complex authentication scheme, you should
* write your own <CODE>Authenticator</CODE> implementation.
* <BR>
* Last modified $Date: 2000/08/08 17:17:20 $
* @version $Revision: 1.12 $
* @author George Reese (borg@imaginary.com)
*/
public class JDBCAuthenticator implements Authenticator {
    /**
     * The SQL SELECT statement.
     */
    static public final String SELECT =
        "SELECT USER_ID, PASSWORD FROM LWP_USER WHERE USER_ID = ?";

    /**
     * Authenticates the specified user ID against the specified
     * password.
     * @param uid the user ID to authenticate
     * @param pw the password to use for authentication
     * @throws com.imaginary.lwp.AuthenticationException the
     * user ID failed to authenticate against the specified password
     */
    public void authenticate(String uid, String pw)
        throws AuthenticationException {
        Connection conn = null;

        try {
            String dsn = System.getProperty(LWPProperties.DSN);
            Context ctx = new InitialContext();
            DataSource ds = (DataSource)ctx.lookup(dsn);
            PreparedStatement stmt;
            String actual;
            ResultSet rs;

            conn = ds.getConnection();
            stmt = conn.prepareStatement(SELECT);
            stmt.setString(1, uid);
            rs = stmt.executeQuery();
            if( !rs.next() ) {

```

*Example 8-4. A JDBC Implementation of the Authenticator Interface (continued)*

```

        throw new AuthenticationException("Invalid user ID or " +
                                         "password.");
    }
    actual = rs.getString(1);
    if( rs.isNull() ) {
        throw new AuthenticationException("No password "+
                                         "specified for " +
                                         uid);
    }
    if( !actual.equals(pw) ) {
        throw new AuthenticationException("Invalid user ID or " +
                                         "password.");
    }
}
catch( SQLException e ) {
    e.printStackTrace();
    throw new AuthenticationException(e);
}
catch( NamingException e ) {
    e.printStackTrace();
    throw new AuthenticationException(e);
}
finally {
    if( conn != null ) {
        try { conn.close(); }
        catch( SQLException e ) { }
    }
}
}
}

```

You should be concerned that this example relies on a data model with unencrypted passwords. Of course, it does not have to. Such databases as MySQL support the concept of password fields so that you do not have to store unencrypted passwords. Relying on such features, however, ties you to specific database engines. A better solution is to encrypt passwords before you store them in the database. The book *Java Cryptography* by Jonathan Knudsen (O'Reilly & Associates) provides an in-depth discussion of encrypting all kinds of data in Java.

Once a client is authenticated, it needs an identity token that it can use for future calls to the server. A primary goal of an identity token is to avoid continuous reauthentication of a client. If you are familiar with web development, an identity token is kind of like a cookie—without the privacy concerns. A good identity token has two requirements:

- It must live for only the lifetime of the client session.
- It must not be forgeable.

To support the banking application, you will use an identifier token class called `Identifier`. Once a person is successfully authenticated, the server generates and stores a unique `Identifier` instance. It also hands the client a copy of this instance. Whenever a client makes a call to the server, it passes its copy to the server. If the two copies match, the client is considered authenticated for that method call. After a certain amount of inactivity, the `Identifier` is invalidated, and the client must login again to generate a new `Identifier`.

The important security feature here is that the server associates a temporary unique number with a user ID. If an `Identifier` comes in when the unique number does not match the currently valid unique number, the operation is rejected. The only way for someone to forge access is to snoop, grab the entire serialized `Identifier`, and use it before the session is invalidated. Forgery is impossible using such encryption as SSL.

Because the `Identifier` instances expire, they should not live much longer than the expected lifetime of the client session they support. The hard part is making the identifier unforgeable. In order to do that, the `Identifier` class has a key field that gets randomly generated upon login. To make it truly secure, you use Java's `java.security.SecureRandom` class. The actual code for this class is included in the examples on this book's web site at <http://www.oreilly.com/catalog/jdbc2>.

## ***Validation***

The picture you should have now is of a client that logs in to a server, gets an `Identifier` instance, and then passes it to every method it calls on the server. Under this component model, RMI remote interfaces must be written with every method accepting an `Identifier` as one of its arguments. As you will see when you get to the actual banking classes, I make the `Identifier` the first argument to every method. In a shared environment like the banking application, you probably need to move beyond authentication to validation. In other words, you need to not only make sure that the client represents who it claims to represent, but also that that person can actually perform the operation in question.

The `Identifier` class provides static methods that enable a business object to check if a particular `Identifier` supports a specific operation. These methods come in the form of `validateXXX()` for which `XXX` is the name of an operation. These `validateXXX()` methods can then be coded to perform some sort of lookup in an access control list. The implementation class for the account entity, for example, might have a method `getBalance()` that will look like this:

```
public double getBalance(Identifier id) {
    if( !Identifier.validateRead(id, this) ) {
        throw new IllegalReadException();
    }
    return balance;
}
```

To simplify things, however, you will support a `prepareRead()` method in a `BaseEntity` that performs the check for any read operation:

```
protected void prepareRead(Identifier id) {
    if( !Identifier.validateRead(id, this) ) {
        throw new IllegalReadException();
    }
}
```

As a result, `getBalance()` can be simplified:

```
public double getBalance(Identifier id) {
    // this will throw a runtime exception if access
    // is disallowed
    prepareRead(id);
    return balance;
}
```

While this simplification may appear trivial, it gains importance when the `prepareXXX()` methods become larger vehicles for managing transactions. Throughout the rest of this book, I will use a class called `BaseEntity` that provides convenience methods to be called from higher level methods.

## *Network Security*

The default network communication in RMI is provided by the standard `java.net.Socket` class. This is the same class you might use for non-RMI TCP/IP network communication. It will send data across the network unchanged from its original form. As long as you are not concerned with what prying eyes might see, this state of affairs will work just fine for you.

Encrypting your network communications actually involves very few changes to the way you write application code. It is largely a matter of installing a custom socket factory. I will briefly outline the steps here required to install a custom socket factory for RMI. A more detailed discussion of these issues can be found in *Java Network Programming* by Elliotte Rusty Harold (O'Reilly & Associates).

Your first task is to decide what sort of socket will handle your network communications. In fact, this discussion is not limited to helping you encrypt your RMI communications. It will also help you perform such things as compression of large amounts of binary data. JDK 1.2 lets you support different sockets for different objects, so the choice of which socket to use depends very much on the type of data coming in and out of an object.

For encryption, you will likely want to use a secure socket layer (SSL) socket. Unfortunately, Java does not ship with any SSL socket implementations. You have to buy these from third-party vendors.

Next, you need to write an implementation of `java.rmi.server.RMIClientSocketFactory*` to hand out the client sockets you wish to use. This pattern is an excellent example of the factory design pattern mentioned in the previous chapter. By relying on a class that constructs sockets rather than relying on direct instantiation of the sockets themselves, you'll find that the sky is the limit for the type of sockets you can use for your RMI communications. Example 8-5 shows a custom socket factory for creating a fictional `SSLClientSocket`.

*Example 8-5. A Custom Client Socket Factory for RMI*

```
import java.io.IOException;
import java.io.Serializable;
import java.net.Socket;
import java.rmi.server.RMIClientSocketFactory;

public class SSLClientSocketFactory
implements RMIClientSocketFactory, Serializable {
    public Socket createSocket(String h, int p)
        throws IOException {
        return new SSLSocket(h, p);
    }
}
```

Naturally, you also have to write a server socket factory that implements `java.rmi.server.RMIServerSocketFactory`. Example 8-6 is an example of an RMI server socket factory.

*Example 8-6. A Factory for Generating Server SSL Sockets*

```
import java.io.IOException;
import java.io.Serializable;
import java.net.Socket;
import java.rmi.server.RMIServerSocketFactory;

public class SSLServerSocketFactory
implements RMIServerSocketFactory, Serializable {
    public ServerSocket createServerSocket(int p)
        throws IOException {
        return new SSLServerSocket(p);
    }
}
```

---

\* If you are lucky, the custom socket package you use will ship with RMI client and server socket factories, so that you do not need to write them yourself.

After all this work, you still have not touched any application code. The final step is when you integrate your custom sockets into your application. Back in Chapter 6, you saw how RMI classes can export themselves either by extending `UnicastRemoteObject` or calling `UnicastRemoteObject.exportObject()`. The constructor for `UnicastRemoteObject` and the static method `exportObject()` both have alternate signatures that enable you to provide a custom socket factory for a specific object.

In Chapter 6, there was a `BallImpl` object that extended `UnicastRemoteObject`. If you wanted to install your SSL socket factories from this chapter, you would simply change the constructor to look like this:

```
public Ball() throws RemoteException {
    super(0,
        new SSLClientSocketFactory(),
        new SSLServerSocketFactory());
}
```

When RMI needs to create a sockets to handle its network communications for the ball component, it will now use these factory classes to generate those sockets instead of the normal socket factories.

## *Database Security*

In a multitier environment, you generally put together multiple technologies with their own authentication and validation mechanisms. You might, for example, have EJB component authentication and validation in the middle tier, but also have database authentication and validation in the data storage tier. A servlet environment might even compound that with web server security.

Multitier applications generally grant a single user ID/password to the middle tier application server. The database trusts that the application server properly manages security. Other tools accessing that same database use different user ID and password combinations to distinguish them and their access rights. Individual users, in turn, are managed either at the web server or application server layer.

A web-based application, for example, could use the web server to manage which users can see which web pages. Those web pages present only the screens containing data a specific user is allowed to see or edit. The web server, in turn, will use a single user ID and password to authenticate itself with the application server. It does not matter who the actual end user is. Similarly, the application server will support access by a handful of different client applications authenticated by user ID/password combinations on a per-client application basis. Finally, the application server will use a single user ID/password pair for all database access.

## Transactions

One of the most important features of EJB is transaction management. Whether you have a simple two-tier application or a complex three-tier application, if you have a database in the mix, you have to worry about transactions. Transactions were discussed in the context of JDBC in both Chapter 3, *Introduction to JDBC*, and Chapter 4, *Advanced JDBC*. Transactions in a three-tier environment are much more complex, but they face many of the same issues. For example, if you perform a transfer from a savings account object to a checking account object, you want to make sure that a failure at any point in that transaction results in a return to the original state of affairs. For example, if the savings account successfully debits itself but the crediting of the checking account fails, the savings account needs to get back the amount debited.

Transaction management at the distributed component level means worrying about a lot of details in the code for every single transaction; a mistake in any one of those details can place the system permanently in an inconsistent state. At the very least, a transaction in a distributed component environment needs to do the following:\*

- Recognize when a transaction begins.
- Track changes that occur during a transaction.
- Lock down the modified objects against modification by other transactions for the duration of the transaction.
- Recognize when the transaction ends.
- Notify the persistence library that changes need to be saved and save them within a single data store transaction.
- Commit or roll back the data store transaction.
- Commit or roll back changes to the business objects.
- Unlock the locked objects so that they may be accessed again for subsequent transactions.

On top of this, components need to recognize when the transaction management has failed. In other words, once a change has been made to an object, it needs to expect a commit or rollback. If it does not receive one in a reasonable amount of time, it needs to roll itself back.

The best solution to the problem of transaction management is to use an infrastructure that manages it all for you, such as Enterprise JavaBeans.† If EJB is not an option, then you should attempt to write a shared library that captures as many of

---

\* Even more is required to support transactions across multiple data sources, also called *two-phase commits*.

† I want to emphasize that I am not recommending writing your own transaction management system. I am covering these issues because they are important to understanding distributed database application programming. EJB is much simpler and more robust than what I present here in this book.

the details of transaction management as is possible. The solution is a Transaction object that monitors a given transaction. You will leave the burden of determining when a transaction begins and ends to the application developer, but some tools in the Transaction class will be provided to make that task easier. The Transaction class should handle everything else.\*

## Transaction Boundaries

The first attack on transaction-boundary recognition might be to have code that looks like this:

```
public void debit(Identifier id, double amt) {
    // get a transaction
    Transaction trans = Transaction.getCurrent(id);

    trans.begin();
    // perform the application logic
    amount -= amt;
    trans.end();
}
```

One problem is that the `debit()` method cannot be reused in the context of another transaction. You cannot, for example, call the `debit()` method from a `transfer()` method because the `debit()` method attempts to end the transaction. The transaction will no longer be valid when you attempt to call `credit()` in the other account!

A more flexible approach to transaction management would enable the developer to write the same code in every single transactional method without worrying about the context the method is being called in. Consider this modified version of the `debit()` method:

```
public void debit(Identifier id, double amt)
throws TransactionException {
    Transaction trans = Transaction.getCurrent(id);
    boolean ip = trans.isInProgress();

    if( !ip ) {
        trans.begin();
    }
    amount -= amt;
    if( !ip ) {
        trans.end();
    }
}
```

---

\* EJB does not require you to worry about transaction boundaries. It recognizes transaction boundaries through a very complex transaction management mechanism. For a detailed discussion of EJB transaction management, look at *Enterprise JavaBeans* by Richard Monson-Haefel (O'Reilly & Associates).

If the application developer follows this paradigm for all method calls, it will not matter whether the method were called as part of a greater transaction or as its own transaction. The application developer has one more problem to worry about: exceptions. The debit method can only encounter `Error` conditions, so it may not seem like much of a concern. Consider the `transfer()` method, however:

```
public void transfer(Identifier id, Account a, Account b, double amt)
throws TransactionException {
    Transaction trans = Transaction.getCurrent(id);
    boolean ip = trans.isInProgress();
    boolean success = false;

    if( !ip ) {
        trans.begin();
    }
    try {
        a.debit(id, amt);
        b.credit(id, amt);
        success = true;
    }
    finally {
        // some exception may have occurred, rollback if it did
        if( success ) {
            if( !ip ) {
                trans.end();
            }
        }
        else if( !ip ) {
            try { trans.rollback(); }
            catch( Exception e ) { }
        }
    }
}
```

The success flag is made true only when the entire set of business logic has completed. Because the business logic is captured in a try block, any exception is certain to trigger a rollback if the business logic does not complete for any reason whatsoever, *and* this method is where the transaction began. The code could include a `catch()` block if you wanted special handling to occur for specific exceptions, but in this case there is no reason to catch particular exceptions.

## ***Tracking Changes***

The code for the `debit()` and `transfer()` methods is missing some of the security discussed earlier in the chapter; namely, it does not check with the `Identifier` class to see if the update is valid. I intentionally left this part out since you have another issue needs addressing: tracking changes.

For the `Transaction` class to intelligently manage transactional operations, it needs to know which objects are being created, modified, or deleted. You can take advantage of the `prepareXXX()` methods mentioned earlier to mark an object as modified in a particular way for a given transaction.

The security code alone might look something like this:

```
if( !Identifier.validateUpdate(id, this) ) {
    throw new ValidationException("Illegal access!");
}
```

By combining the change tracking code in a single method in the `BaseEntity` class, you would end up with code like this:

```
protected synchronized void prepareUpdate(Identifier id)
throws TransactionException {
    Transaction trans = Transaction.getCurrent(id);

    if( !Identifier.validateUpdate(id, this) ) {
        throw new ValidationException("Illegal access!");
    }
    // associate this change with the current
    // transaction
    trans.prepareUpdate(this);
}
```

The first part of this method performs the security check. The second part notifies the current transaction that the object has been modified by calling the `prepareUpdate()` method. The `debit()` method can now look like this:

```
public void debit(Identifier id, double amt)
throws TransactionException {
    Transaction trans = Transaction.getCurrent(id);
    boolean ip = trans.isInProgress();
    boolean success = false;

    if( !ip ) {
        trans.begin();
    }
    try {
        // security check and change notification
        prepareUpdate(id);
        amount -= amt;
        success = true;
    }
    finally {
        if( success ) {
            if( !ip ) {
                trans.end();
            }
        }
    }
}
```

```
        else {
            try { trans.rollback(); }
            catch( Exception e ) { }
        }
    }
}
```

Certainly there is a lot more code in that method than the core business logic of `amount -= amt`, but it is code that can exist in each transactional method without the coder having to make a lot of transaction-based coding decisions. Because the `prepareUpdate()` tells the transaction that the account has been modified, the transaction can add it to a list of modified objects associated with the transaction. When you end the transaction, the `Transaction` class then goes through all of the modified objects and makes sure that their new state is saved to the persistent data store. Among the complexities the `Transaction` class needs to handle here is the complexity of a transaction that makes a change to an object and deletes it. The `Transaction` class can make sure that only the delete operation is sent to the persistent data store.

## *Other Transaction Management Issues*

Once a transaction has been told it is over, it needs to save the current state of the objects that took part in the transaction to the persistent data store. You will cover the actual saving of the state to a relational database in Chapter 9. What you need to think about at this point is that the transaction object be able to track which objects have changed and what sort of changes occurred. Once the transaction ends, it needs to tell the persistence mechanism to insert, update, or delete the object in the data store. The `prepareUpdate()` method in the `BaseEntity` class took care of that for us.

The only unaddressed piece other than persistence is making sure that two transactions do not interfere with each other. An example of such a situation might be a transaction for which I am in the bank withdrawing cash and my wife is at the ATM doing a transfer. We have \$100 in our checking account, and we are both attempting to withdraw \$75. Obviously, we should not both be able to succeed. The transaction management infrastructure should make sure that does not happen.

We can make sure only one transaction is touching an object at a time by adding the following code to the `prepareUpdate()` method just after the security check:

```
if( transaction != null ) {
    if( !trans.equals(transaction) ) {
        throw new TransactionException("Illegal " +
            "concurrent transactions!");
    }
}
```

```
else {  
    transaction = trans;  
}
```

By throwing an exception, this prevents the system from ending up in a situation called a *deadlock*, in which one transaction waits on a lock held by another, while that other transaction waits on a lock held by the first. Of course the `BaseEntity` also needs to set the transaction to null in its `commit()` and `rollback()` implementations.

## Lookups and Searches

Before a client can make any changes to an object, it needs to find that object. There are three scenarios for getting a reference to a distributed component:

- Looking it up by its unique identifier
- Searching for it based on a set of criteria
- Asking another related component for a reference to it

The last scenario is the simplest and begs the question of the other two. Specifically, given a reference to a `Customer` component, you should be able to get all of that `Customer`'s `Account` objects. How did you get a reference to that `Customer` in the first place?

EJB uses a kind of meta-component called a *home* to manage operations on a component as a class, including lookups, searches, and creates. Whether you call it a home or something else, a distributed database application needs some way to get access to the business objects stored on the server. You will take advantage of the home metaphor, but you'll put a new spin on it.

Getting a reference to an object using its unique identifier is fairly simple. A `find()` method in the home accepts an `Identifier` and an `objectID` as parameters and returns a reference to the component identified by that `objectID`:

```
public abstract Account find(Identifier id,  
                             long oid)  
    throws FindException;
```

Using the persistence library that will be discussed in Chapter 9, you can then search the persistence store for an object that has the specified `objectID`.

Performing searches on criteria other than a unique identifier gets more complex. First, because the criteria may not identify an object instance uniquely, you need to handle a collection of those objects. Second, the number of search criteria combinations can become overwhelming. Whereas one client screen may want to search on balance and customer last names, other client screens may wish to

search on social security numbers, gender, or marital status. A solid business component needs to be able to support all these permutations.

EJB actually encounters serious problems with these issues. Under the EJB component model, any bean has a home interface responsible for the creation of new instances of that component, destroying instances of it, and performing searches. When a client performs a search that returns a collection, most EJB implementations return a `Java Enumeration` or `Collection` that contains the full set of beans matching the specified criteria. Unfortunately, searches that return thousands or millions of records cause serious performance problems for EJB. In addition, EJB requires you to specify a distinct `find()` method for each combination of search criteria you wish to offer. To address these issues, you will use a generic search mechanism in the banking application that returns a specialized collection.\*

The core of your searching library is a class called `SearchCriteria`. It can represent any arbitrary set of search criteria—independent of persistence technology—so the home can pass those search criteria to a persistence engine for interpretation. The `AccountHome` class thus has a single method for searches on arbitrary criteria:

```
public abstract Collection find(Identifier id,
                               SearchCriteria sc)
    throws FindException;
```

The task of the `SearchCriteria` class is to associate attributes with values via some sort of operator. For example, if you want to search for all accounts with a balance of less than \$100, the `SearchCriteria` class would have “balance” as the attribute, “<” as the operator, and 100.00 as the value. Such a triplet is encapsulated in a class called `SearchBinding`. The `SearchCriteria` ties multiple bindings together with an `AND` or `OR`. You can thus perform complex queries joining many bindings. Finally, a `SearchCriteria` is itself a `SearchBinding`. Using this feature, a person can perform a search that groups bindings together. Figure 8-2 shows how the following SQL search matches with a `SearchCriteria` instance:

```
“SELECT objectID FROM Account WHERE balance > 100 OR (openDate > '23-MAR-2000' AND
openDate < '31-MAR-2000’)”
```

In Chapter 9, I will show how the persistence library actually implements searching by taking a `SearchCriteria` instance and turning it into a SQL query.

I am still begging the question as to how you get a reference to a home object in the first place. JNDI enters the picture here. When an application is deployed, the system administrator enters home objects into a JNDI-supported directory so that clients can perform a JNDI lookup for the home.

---

\* The next release of the EJB specification will introduce a specialized query language, which should address some failings in its searching API.

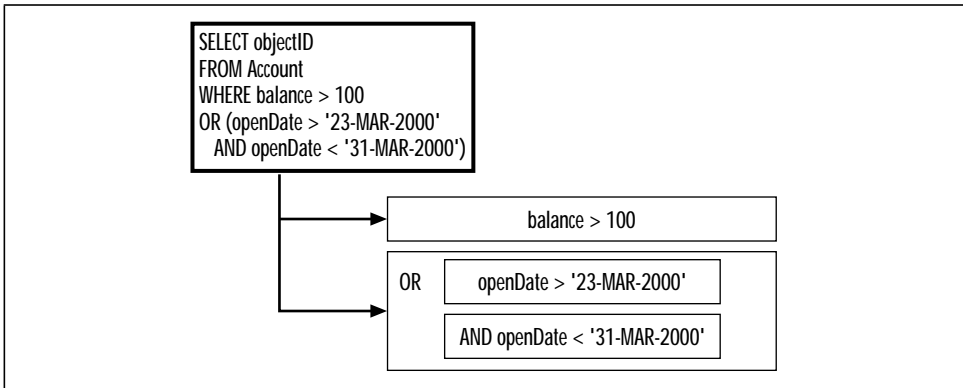


Figure 8-2. A graphic illustration of how bindings form a search criteria instance

## Entity Relationships

Relationships among entities is one of the most complex problems to handle in the object-oriented world. Imagine a huge film server such as the Internet Movie Database (IMDB). A film can have many directors and actors. Each director and actor can, in turn, be related to many films. If your application restored a film from its data store with all of its relationships intact, the act of loading a single film would cause a huge amount of data—most of which you are definitely not interested in—to load into memory. An enterprise system therefore needs to be much smarter about managing entity relationships than the use of simple entity attributes.

Enterprise JavaBeans does not directly address the problem of entity relationships. The problems you face for the banking application are therefore the same problems you would face if you were using EJB. As a result, the solutions discussed in this section are directly applicable to an Enterprise JavaBean system.

A crude solution to the problem is to store the unique identifiers for related entities instead of the entities themselves; after all, this is what you do in the database. In other words, a bank account would have a `customerID` instead of a `Customer` attribute. The `Account` component would load the `Customer` into memory using the `customerID` only when it needed the reference.

Unfortunately, this solution is both cumbersome and not very object-oriented. It is cumbersome because the `Account` coder is forced to deal with the `Customer` relationship at two levels: as a unique identifier and as an entity component. This treatment of the same concept using two different representations opens the code up to error. The more serious architectural problem, however, is the move away from the object metaphor. The relationship being modeled in the code is no longer the relationship between a `Customer` and an `Account`, but between a `Customer` and a number.

A better solution is an object placeholder that looks to clients like the actual object it represents but does not contain all data associated with the actual object it represents. I call this approach a *façade*.<sup>\*</sup> When a façade first comes into being, it knows only the unique identifier of the component it serves. It loads that component only if a method call is made on the façade. An entity can thus store its relationships with other entities as façades and treat them as if they were the real components.

## *Façades*

A good façade can do much more than save you the effort of loading components. It can help performance by caching component attributes. The first benefit of caching attributes is the minimization of network calls to the component. If, for example, a client had a `JTable` with a list of `Account` components, it would make a network call to each account for each value stored in the table any time a redraw is requested. This chatty behavior results in a very slow GUI. By using façades and caching data in the façades, calls to get data from an account beyond the initial calls become local.

A façade can also poll its associated entity component to make sure nothing has changed. It can then integrate with the Swing event model on a client by supporting `PropertyChangeEvent` occurrences. When something does change, the façade can throw a `PropertyChangeEvent`. If you are not familiar with property change events, remember that they are a key part of the JavaBeans model. The idea is that objects can register themselves as being interested in changes that occur to the bound properties of other objects, a.k.a. beans. When a change occurs in a target object, it fires a `PropertyChangeEvent` that notifies all listening objects of the change. I will cover the Swing event model and how façades support it in detail in Chapter 10.

Example 8-7 shows the base class for a façade from which component-specific façades can be built. At its heart is the method `reconnect()`. This method performs a lookup for the entity behind this façade when circumstances demand it.

### *Example 8-7. A Generic Façade for Entity Components*

```
package com.imaginary.lwp;

import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import java.io.Serializable;
```

---

<sup>\*</sup> Façades are actually an implementation of the classic “proxy” pattern, not the classic “façade” pattern. More specifically, they help implement the distributed listener pattern from earlier in the book. I call them façades because they are much more than simple proxies and are, in fact, façades in the more colloquial dictionary sense.

*Example 8-7. A Generic Façade for Entity Components (continued)*

```

import java.net.MalformedURLException;
import java.rmi.Naming;
import java.rmi.NotBoundException;
import java.rmi.RemoteException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;

/**
 * The base class for all façade objects. This class
 * captures all functionality common to façades. Subclasses
 * should be written for each entity class.
 */
public abstract class BaseFacade implements Serializable {
    private      HashMap      cache      = new HashMap();
    private      Entity       entity     = null;
    private      Home        home       = null;
    private transient ArrayList listeners =
        new ArrayList();

    private      String      lastUpdateID = null;
    private      long       lastUpdateTime = -1L;
    private      long       objectID     = -1L;

    public BaseFacade() {
        super();
    }

    /**
     * Constructs a new façade that represents the entity
     * identified by the specified object identifier.
     * @param oid the unique identifier of the associated entity
     */
    public BaseFacade(long oid) {
        super();
        objectID = oid;
    }

    /**
     * Constructs a new façade that represents the specified
     * entity object.
     * @param ent the entity being represented
     * @throws java.rmi.RemoteException the entity is inaccessible
     */
    public BaseFacade(Entity ent) throws RemoteException {
        super();
        entity = ent;
    }

```

*Example 8-7. A Generic Façade for Entity Components (continued)*

```
        objectID = entity.getObjectID();
    }

    /**
     * Supports the JavaBeans event model by allowing other
     * objects to know when a change has occurred in this object.
     * @param l the object listening to this façade
     */
    public void addPropertyChangeListener(PropertyChangeListener l) {
        if( listeners == null ) {
            listeners = new ArrayList();
        }
        listeners.add(l);
    }

    /**
     * Enables an object to listen for changes in a particular
     * property in this façade. This implementation does
     * not currently care what property the listeners are listening for.
     * @param p the property, this is ignored
     * @param l the object listening to this façade
     */
    public void addPropertyChangeListener(String p, PropertyChangeListener l) {
        if( listeners == null ) {
            listeners = new ArrayList();
        }
        listeners.add(l);
    }

    /**
     * Assigns this façade to support the entity identified
     * by the specific object identifier.
     * @param oid the unique object identifier
     */
    public void assign(long oid) {
        if( objectID != -1L ) {
            throw new ReferenceReuseException("Reference already assigned.");
        }
        else {
            objectID = oid;
        }
    }

    public void assign(long oid, Entity ent) {
        assign(oid);
        entity = ent;
    }
}
```

*Example 8-7. A Generic Façade for Entity Components (continued)*

```

public void assign(long oid, HashMap vals) {
    assign(oid);
}

/**
 * Determines whether or not the specified attribute has
 * been cached.
 * @param attr the name of the attribute to test
 * @return true if the attribute has been cached
 */
protected boolean contains(String attr) {
    return cache.containsKey(attr);
}

// two are equal if they have the same objectID
public boolean equals(Object ob) {
    if( ob instanceof BaseReference ) {
        BaseReference ref = (BaseReference)ob;

        return (ref.getObjectID() == getObjectID());
    }
    else {
        return false;
    }
}

// fires a property change event
protected void firePropertyChange() {
    firePropertyChange(new PropertyChangeEvent(this, null, null, null));
}

protected void firePropertyChange(PropertyChangeEvent evt) {
    Iterator it;

    if( listeners == null ) {
        return;
    }
    it = listeners.iterator();
    while( it.hasNext() ) {
        PropertyChangeListener l = (PropertyChangeListener)it.next();

        l.propertyChange(evt);
    }
}

/**
 * Provides the cached value for the specified attribute.
 * This method will return null if the value has not been

```

*Example 8-7. A Generic Façade for Entity Components (continued)*

```
* cached, so a check to contains() should be made first.
* @param attr the name of the attribute to get
* @return the value of the cached attribute
*/
protected Object get(String attr) {
    return cache.get(attr);
}

// Provides the entity behind this reference.
public Entity getEntity() throws RemoteException {
    if( entity == null ) {
        reconnect();
    }
    return entity;
}

// Provides the last update ID.
public String getLastUpdateID() throws RemoteException {
    if( lastUpdateID == null ) {
        try {
            lastUpdateID = getEntity().getLastUpdateID();
        }
        catch( RemoteException e ) {
            reconnect();
            lastUpdateID = getEntity().getLastUpdateID();
        }
    }
    return lastUpdateID;
}

// Provides the timestamp from the last update.
public long getLastUpdateTime() throws RemoteException {
    if( lastUpdateTime == -1L ) {
        try {
            lastUpdateTime = getEntity().getLastUpdateTime();
        }
        catch( RemoteException e ) {
            reconnect();
            lastUpdateTime = getEntity().getLastUpdateTime();
        }
    }
    return lastUpdateTime;
}

// Provides the unique object identifier.
public long getObjectID() {
    return objectID;
}
```

*Example 8-7. A Generic Façade for Entity Components (continued)*

```

public int hashCode() {
    Long l = new Long(getObjectID());

    return l.hashCode();
}

public boolean hasListeners(String prop) {
    if( listeners == null ) {
        return false;
    }
    if( listeners.size() > 0 ) {
        return true;
    }
    else {
        return false;
    }
}

/**
 * Inserts the specified attribute and value into the
 * object cache.
 * @param attr the name of the attribute
 * @param val the value to be associated with the attribute
 */
protected void put(String attr, Object val) {
    cache.put(attr, val);
}

/**
 * This method provides a level of failover support and
 * initializes entity polling. This method is called
 * only when the façade has determined it has to load
 * the associated entity.
 * @throws java.rmi.RemoteException the server is inaccessible
 */
protected void reconnect() throws RemoteException {
    final BaseFacade ref = this;
    Thread t;

    // the home object is used to find entities; load it if null
    if( home == null ) {
        String url = System.getProperty(LWPPProperties.RMI_URL);
        ObjectServer svr;

        try {
            // the ObjectServer is an RMI object that provides homes
            svr = (ObjectServer)Naming.lookup(url);
        }
    }
}

```

*Example 8-7. A Generic Façade for Entity Components (continued)*

```
catch( MalformedURLException e ) {
    throw new RemoteException(e.getMessage());
}
catch( NotBoundException e ) {
    throw new RemoteException(e.getMessage());
}
try {
    // use the client identifier for doing a lookup
    Identifier id = Identifier.currentIdentifier();

    // if null, this is happening on the server
    // in that case, use a special server ID
    if( id == null ) {
        id = Identifier.getServerID();
    }
    // ask the server for the home for this class
    home = (Home)svr.lookup(id, getClass().getName());
}
catch( LWPEException e ) {
    throw new RemoteException(e.getMessage());
}
}
try {
    Identifier id = Identifier.currentIdentifier();

    // look up the entity
    entity = home.findByObjectID(id, objectID);
    lastUpdateID = entity.getLastUpdateID();
    lastUpdateTime = entity.getLastUpdateTime();
}
catch( PersistenceException e ) {
    throw new RemoteException(e.getMessage());
}
catch( FindException e ) {
    throw new RemoteException(e.getMessage());
}
}
catch( RemoteException e ) {
    // give it a second chance
    e.printStackTrace();
    String url = System.getProperty(LWPPProperties.RMI_URL);
    ObjectServer svr;

    try {
        svr = (ObjectServer)Naming.lookup(url);
    }
    catch( MalformedURLException salt ) {
        throw new RemoteException(salt.getMessage());
    }
}
```

*Example 8-7. A Generic Façade for Entity Components (continued)*

```

catch( NotBoundException salt ) {
    throw new RemoteException(salt.getMessage());
}
try {
    Identifier id = Identifier.currentIdentifier();

    if( id == null ) {
        id = Identifier.getServerID();
    }
    home = (Home)svr.lookup(id, getClass().getName());
    entity = home.findByObjectID(id, objectID);
    lastUpdateID = entity.getLastUpdateID();
    lastUpdateTime = entity.getLastUpdateTime();
}
catch( LookupException salt ) {
    throw new RemoteException(salt.getMessage());
}
catch( PersistenceException salt ) {
    throw new RemoteException(salt.getMessage());
}
catch( FindException salt ) {
    throw new RemoteException(salt.getMessage());
}
}
// With the entity loaded, begin polling for changes.
t = new Thread() {
    public void run() {
        while( true ) {
            synchronized( ref ) {
                if( entity == null ) {
                    return;
                }
                try {
                    if( lastUpdateTime == -1L ) {
                        lastUpdateTime = entity.getLastUpdateTime();
                    }
                    if( entity.isChanged(lastUpdateTime) ) {
                        lastUpdateTime = entity.getLastUpdateTime();
                        lastUpdateID = entity.getLastUpdateID();
                        firePropertyChange();
                    }
                }
            }
            catch( RemoteException e ) {
                // this will force a reload
                entity = null;
                return;
            }
        }
    }
}

```

*Example 8-7. A Generic Façade for Entity Components (continued)*

```
        try { Thread.sleep(30000); }
        catch( InterruptedException e ) { }
    }
}
};
t.setPriority(Thread.MIN_PRIORITY);
t.start();
}

public void removePropertyChangeListener(PropertyChangeListener l) {
    if( listeners == null ) {
        return;
    }
    listeners.remove(l);
}

public void removePropertyChangeListener(String p,
                                         PropertyChangeListener l) {
    if( listeners == null ) {
        return;
    }
    listeners.remove(l);
}

public synchronized void reset() {
    cache.clear();
    lastUpdateTime = -1L;
    lastUpdateID = null;
}
}
```

## ***Collections***

One-to-one relationships are clearly simplified with the use of façades. One-to-many and many-to-many relationships complicate things. How do you handle the modeling of a machine with one million parts?

You almost never want to send all one million parts across the network. When you do, users on the other end would certainly prefer not to wait for all one million parts to traverse the network before they work with the first part or even see any progress on the method call. This problem is more common when performing finds using home objects, since it is rare that an entity will have a one-to-one million relationship. It is all too common that people issue queries that will return one million rows. The standard Java Collections API does not address this problem.

The answer is a custom collection that hands the client elements in the collection just before the client goes to work with that row. It could grab the first 100 hits

before it returns and then send the client back elements in groups of 100 as the client is approaching the point when it needs them. If the client uses only the first 100 results, at most 200 elements will get sent across the network. The other 999800 elements will sit on the server. The code supporting this book (<ftp://ftp.oreilly.com/pub/examples/java/jdbc>) comes with just such a collection.