5.7 Oracle AQ Examples

This section offers numerous examples of using AQ, including packages you can install and reuse in your environment. In all these examples, unless otherwise noted, assume that I have (a) defined an Oracle account named AQADMIN to perform administrative tasks and (b) assigned the AQ_USER_ROLE to SCOTT to perform operational tasks. I then connect to AQADMIN.

After setting up the queue tables and queues, I connect to SCOTT and create this object type:

```
CREATE TYPE message_type AS OBJECT
   (title VARCHAR2(30),
    text VARCHAR2(2000));
```

I also grant EXECUTE privilege on this object to my AQ administrator:

```
GRANT EXECUTE ON message_type TO AQADMIN;
```

My AQ administrator then can create a queue table and a message queue as follows:

```
BEGIN
   DBMS_AQADM.CREATE_QUEUE_TABLE
      (queue_table => 'scott.msg',
       queue_payload_type => 'message_type');

   DBMS_AQADM.CREATE_QUEUE
      (queue_name => 'msgqueue',
       queue_table => 'scott.msg');

   DBMS_AQADM.START_QUEUE (queue_name => 'msgqueue');
END;
/
```

Notice that I do not need to specify the schema for the payload type. AQ assumes the same schema as specified for the queue table.

I will make use of these objects throughout the following examples; I will also at times supplement these queue objects with other, more specialized queue table and queues.

Oracle also provides a set of examples scripts for AQ. In Oracle 8.0.3, the following files were located in *$ORACLE_HOME/rdbms80/admin/aq/demo* :

aqdemo00.sql

The driver program for the demonstration

aqdemo01.sql

Create queue tables and queues

aqdemo02.sql

Load the demo package

aqdemo03.sql

Submit the event handler as a job to the job queue

aqdemo04.sql

Enqueue messages

5.7.1 Improving AQ Ease of Use

Let's start by constructing a package to make it easier to work with AQ objects. I am always looking for ways to shortcut steps I must perform to get things done. The complexity of AQ, along with all of the different records and structures, begs for a wrapper of code to perform common steps more easily.

I describe the program elements defined in the aq package later. To save a few trees, I will leave the reader to examine the package body to see how I constructed these programs. In most cases they are very straightforward.

First off, I define two subtypes so that you can declare variables using names instead of hard-coded declarations like RAW(16). These subtypes are as follows:

```
v_msgid RAW(16);
SUBTYPE msgid_type IS v_msgid%TYPE;

v_name VARCHAR2(49);
SUBTYPE name_type IS v_name%TYPE;
```


I also predefined two common exceptions so that you can trap these by name through a WHEN OTHERS clause and a hard-coding of the error number (see the *aqbrowse.sp* file for an example of using this named exception):

```
dequeue_timeout EXCEPTION
PRAGMA EXCEPTION_INIT (dequeue_timeout, -25228);

dequeue_disabled EXCEPTION;
PRAGMA EXCEPTION_INIT (dequeue_disabled, -25226);
```

Now let's run through the different procedures and functions in the packages. The aq.create_queue procedure combines the create table queue, create queue, and start queue steps into a single procedure call:

```
PROCEDURE aq.create_queue
   (qtable IN VARCHAR2,
    payload_type IN VARCHAR2,
    qname IN VARCHAR2,
    prioritize IN VARCHAR2 := NULL);
```

If the queue table already exists, it is not created. You can also provide a prioritization

string if you want to override the default.

The aq. create_priority_queue procedure has the same interface as aq.create_queue, but the default value for the prioritize parameter is the most common nonstandard string: order by the priority number, and within the same priority number, by the enqueue time.

```
PROCEDURE create_priority_queue
   (qtable IN VARCHAR2,
    payload_type IN VARCHAR2,
    qname IN VARCHAR2,
    prioritize IN VARCHAR2 := 'PRIORITY,ENQ_TIME');
```

The aq. stop_and_drop procedure is a neat little program. It combines the following operations: stop queue, drop queue, and drop queue table. But it also figures out when it is appropriate to execute each of those steps.

```
PROCEDURE aq.stop_and_drop (
   qtable IN VARCHAR2,
   qname IN VARCHAR2 := '%',
   enqueue IN BOOLEAN := TRUE,
   dequeue IN BOOLEAN := TRUE,
   wait IN BOOLEAN := TRUE);
```

Here are the rules followed by aq.stop_and_drop:

- Stop all queues within the specified queue table that match the queue name you provide. Notice that the default is `%', so if you do not provide a queue name, then all queues in the queue table are stopped.
- If you specify that you want to stop both enqueue and dequeue operations on queues, then those queues will also be dropped.
- If you stop and drop all queues in the queue table, then the queue table itself will be dropped.

The default values for this program specify that all queues in the specified queue table should be stopped and dropped, but only after any outstanding transactions on those queues are completed.

The rest of the aq programs retrieve information about queues and queue tables from the data dictionary views. You could write many more programs along these lines to make it easier to view the contents of the AQ views. In fact, the aq package will contain more programs by the time this book is printed, so check out the *aq.spp* file to see the latest set of functionality.

The aq. queue_exists function returns TRUE if a queue of the specified name exists:

```
FUNCTION aq.queue_exists (qname IN VARCHAR2) RETURN BOOLEAN;
```

The aq. qtable_exists function returns TRUE if a queue table of the specified name exists:

```
FUNCTION aq.qtable_exists (qtable IN VARCHAR2) RETURN
BOOLEAN;
```

The aq. msgcount function returns the number of messages in the specified queue:

```
FUNCTION aq.msgcount (qtable IN VARCHAR2, qname IN VARCHAR2)
```

```
      RETURN INTEGER
```

You have to specify both the queue table and the queue name so that the function can construct the name of the database table holding the queue messages. You could enhance this function so that you provide only the queue name and the function looks up the queue table name for you.

The aq. msgdata function returns the specified piece of information (the data_in argument) for a specific message ID in the queue table:

```
FUNCTION aq.msgdata (qtable_in IN VARCHAR2,
   msgid_in IN RAW,
   data_in IN VARCHAR2) RETURN VARCHAR2;
```

The data_in argument must be one of the columns in the aq$<qtable_in> database table, which contains all the messages for queues in that queue table.

For example, to obtain the correlation ID for a message in the "msg" queue table, you could call aq.msgdata as follows:

```
CREATE OR REPLACE FUNCTION corr_id (msg_id IN
aq.msgid_type)
   RETURN VARCHAR2
IS
   v_corr_id := aq.msgdata ('msg', msgid_in, 'corr_id');
END;
/
```

Call the aq. showmsgs procedure to show some of the message information for the specified queue:

```
PROCEDURE showmsgs (qtable IN VARCHAR2, qname IN VARCHAR2);
```

This procedure currently shows the priority, message state, number of retries, and correlation ID of messages in the queue. You can easily modify the procedure to show different pieces of information about the message. Remember that it is probably impossible to create a generic program like this that will display the *contents* of the message, since that is either a RAW or an instance of an object type. For this same reason, there is no generic enqueue or dequeue procedure.

I hope these programs will get you started on encapsulating commonly needed tasks at your site for performing queueing operations. There is much more to be done, particularly in the area of building queries (which can then be placed behind functions and in mini-report generator procedures) against the various data dictionary views.

5.7.2 Working with Prioritized Queues

The normal priority order for dequeuing is by enqueue time: in other words, "first in, first out" or FIFO. You can modify this priority order when you create a different value for the sort_list argument when you create a queue table. Since this value is specified for a queue table, you will be setting the default sorting for any queue defined in this queue table.

The only other option for the default sorting of queue messages is by the priority number. In the world of AQ, the lower the priority number, the higher the priority.

Suppose that I want to create a queue that manages messages of three different

priorities: low, medium, and high. The rule is very simple: dequeue high-priority messages before medium-priority messages, and medium-priority messages before low-priority messages.

As you might expect, I would strongly urge that when faced with a task like this one, you immediately think in terms of building a package to encapsulate your different actions and make your code easier to use. In this scenario, for example, I don't really want users of my prioritized queue to have to know about specific priority numbers. Instead, I want to provide them with programs that hide the details and let them concentrate on their tasks.

The following specification for a package offers an interface to a three-level prioritization queue. The payload type for this queue is the same message_type described at the beginning of the example section.

```
/* Filename on companion disk:

priority.spp */*
CREATE OR REPLACE PACKAGE priority
IS
   PROCEDURE enqueue_low (item IN VARCHAR2);

   PROCEDURE enqueue_medium (item IN VARCHAR2);

   PROCEDURE enqueue_high (item IN VARCHAR2);

   PROCEDURE dequeue (item OUT VARCHAR2);
END;
/
```

This is a very simple package specification. You can enqueue messages with one of three priorities, and you can dequeue messages. Here is a script that tests this package by helping me prioritize my chores for the evening:

```
/* Filename on companion disk:

priority.tst */*
DECLARE
  str varchar2(100);
BEGIN
   priority.enqueue_low ('Cleaning the basement');
   priority.enqueue_high ('Cleaning the bathroom');
   priority.enqueue_high ('Helping Eli with his non-French
homework');
   priority.enqueue_medium ('Washing the dishes');

   LOOP
      priority.dequeue (str);
      EXIT WHEN str IS NULL;
      DBMS_OUTPUT.PUT_LINE (str);
   END LOOP;
END;
/
```

I place four messages with different priorities in my queue. Notice that the order in which I enqueue does not correspond to the priorities. Let's run this script and see what I get:

```
SQL> @priority.tst
HIGH: Cleaning the bathroom
HIGH: Helping Eli with his non-French homework
MEDIUM: Washing the dishes
LOW: Cleaning the basement
```

As you can see, my messages have been dequeued in priority order.

You can view the entire package body in the *priority.spp* file. Let's take a look at the individual components I used to build this package. First, I define a set of constants as follows:

```
CREATE OR REPLACE PACKAGE BODY priority
IS
   c_qtable CONSTANT aq.name_type := 'hi_med_lo_q_table';
   c_queue CONSTANT aq.name_type := 'hi_med_lo_q';

   c_high CONSTANT   PLS_INTEGER := 1;
   c_medium CONSTANT PLS_INTEGER := 500000;
   c_low CONSTANT    PLS_INTEGER := 1000000;
```

I don't want to hard-code the names of my queue table and queue throughout my body, so I use constants instead. I also define constants for my three different priority levels. (Notice the space between these values; I will come back to that later.)

I have three different enqueue procedures to implement. Each of them performs the same basic steps. Here, for example, is the way I first implemented enqueue_low:

```
   PROCEDURE enqueue_low (item IN VARCHAR2)
   IS
      queueopts DBMS_AQ.ENQUEUE_OPTIONS_T;
      msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;
      item_obj message_type;
   BEGIN
      item_obj := message_type (priority, item);
      queueopts.visibility := DBMS_AQ.IMMEDIATE;
      msgprops.priority := c_low;
      DBMS_AQ.ENQUEUE (c_queue, queueopts, msgprops,
item_obj, g_msgid);
   END;
```

I declare my records to hold the queue options and message properties. I construct the object to be placed in the queue. I request that the operation be immediately visible (no commit required) and set the priority. Once these steps are complete, I enqueue the message.

I finished this procedure and then embarked on enqueue_medium. I quickly discovered that the only difference between the two was the assignment to the msgprops.priority field. I just as quickly put the kibosh on this approach. It made no sense at all to me to write (or cut-and-paste) three different procedures with all that code when there was virtually no difference between them. Instead I wrote a single, generic enqueue as follows:

```
   PROCEDURE

enqueue (item IN VARCHAR2, priority IN PLS_INTEGER)
   IS
      queueopts DBMS_AQ.ENQUEUE_OPTIONS_T;
      msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;
      item_obj message_type;
   BEGIN
      item_obj := message_type (priority, item);
      queueopts.visibility := DBMS_AQ.IMMEDIATE;
      msgprops.priority := priority;
      DBMS_AQ.ENQUEUE (c_queue, queueopts, msgprops,
item_obj, g_msgid);
   END;
```

And then I implemented the priority-specific enqueue procedures on top of this one:

```
   PROCEDURE

enqueue_low (item IN VARCHAR2)
   IS
   BEGIN
      enqueue (item, c_low);
   END;

   PROCEDURE

enqueue_medium (item IN VARCHAR2)
   IS
   BEGIN
      enqueue (item, c_medium);
   END;

   PROCEDURE

enqueue_high (item IN VARCHAR2)
   IS
   BEGIN
      enqueue (item, c_high);
   END;
```

It is extremely important that you always consolidate your code and modularize within package bodies as much as possible. You will then find it much easier to maintain and enhance your programs.

My enqueue procedures are now done. I have only a single dequeue and it is fairly straightforward:

```
   PROCEDURE dequeue (item OUT VARCHAR2)
   IS
      queueopts DBMS_AQ.DEQUEUE_OPTIONS_T;
      msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;
      item_obj message_type;
   BEGIN
      queueopts.wait := DBMS_AQ.NO_WAIT;
      queueopts.visibility := DBMS_AQ.IMMEDIATE;
      DBMS_AQ.DEQUEUE (c_queue, queueopts, msgprops,
item_obj, g_msgid);
      item := priority_name (item_obj.title) || ': ' ||
item_obj.text;
   EXCEPTION
      WHEN OTHERS
      THEN
         IF SQLCODE = -25228 /* Timeout; queue is likely
empty... */
         THEN
            item := NULL;
         ELSE
            RAISE;
         END IF;
   END;
```

Most of the code is taken up with the basic steps necessary for any dequeue operation: create the records, specify that I want the action to be immediately visible, and specify that AQ should not wait for messages to be queued. I then dequeue, and, if successful, construct the string to be passed back.

If the dequeue fails, I trap for a specific error that indicates that the queue was empty (ORA-025228). In this case, I set the item to NULL and return. Otherwise, I reraise the same error.

Notice that I call a function called priority_name as a part of my message passed back in dequeue. This function converts a priority number to a string or name as follows:

```
   FUNCTION priority_name (priority IN PLS_INTEGER) RETURN
VARCHAR2
   IS
      retval VARCHAR2(30);
   BEGIN
      IF    priority = c_high THEN retval := 'HIGH';
      ELSIF priority = c_low THEN retval := 'LOW';
      ELSIF priority = c_medium THEN retval := 'MEDIUM';
      ELSE
         retval := 'Priority ' || TO_CHAR (priority);
      END IF;
      RETURN retval;
   END;
```

This function offers some consistency in how the priorities are named.

This package contains the following initialization section:

```
BEGIN
   /* Create the queue table and queue as necessary. */
   aq.create_priority_queue (c_qtable, 'message_type',
c_queue);
END priority;
```

This line of code is run the first time any of your code references a program in this package. This aq procedure (see the earlier section, Section 5.7.1, "Improving AQ Ease of Use" ") makes sure that all elements of the priority queue infrastructure are ready to go. If the queue table and queue are already in place, it will not do anything, including raise any errors.

Remember the comment I made about the big gaps between the priority numbers? Take a look at the body for the priority package. If you add the header of the generic enqueue procedure to the specification, this package will support not only high, low, and medium priorities, but also any priority number you want to pass to the enqueue procedure.

5.7.2.1 More complex prioritization approaches

In many situations, the priority may be established by relatively fluid database information. In this case, you should create a function that returns the priority for a record in a table. Suppose, for example, that you are building a student registration system and that priority is given to students according to their seniority: if a senior wants to get into a class, she gets priority over a freshman.

If I have a student object type as follows (much simplified from a real student registration system),

```
CREATE TYPE student_t IS OBJECT
   (name VARCHAR2(100),
    enrolled_on DATE);
```

and a table built upon that object type as follows:

```
CREATE TABLE student OF student_t;
```

I might create a function as follows to return the priority for a student:

```
CREATE OR REPLACE FUNCTION reg_priority (student_in IN
student_t)
   RETURN PLS_INTEGER
IS
BEGIN
   RETURN -1 * TRUNC (SYSDATE - student_in.enrolled_on);
END;
/
```

Why did I multiply by -1 the difference between today's date and the enrolled date? Because the lower the number, the higher the priority.

Of course, this function could also be defined as a member of the object type itself.

5.7.3 Building a Stack with AQ Using Sequence Deviation

A queue is just one example of a "controlled access list." The usual definition of a queue is a FIFO list (first-in, first-out). Another type of list is a stack, which follows the LIFO rule: last-in, first-out. You can use AQ to build and manage persistent stacks with ease (its contents persist between connections to the database).

The files *aqstk.spp* and *aqstk2.spp* offer two different implementations of a stack using Oracle AQ. The package specifications in both cases are exactly the same and should be familiar to anyone who has worked with a stack:

```
CREATE OR REPLACE PACKAGE aqstk
IS
   PROCEDURE push (item IN VARCHAR2);
   PROCEDURE pop (item OUT VARCHAR2);
END;
/
```

You push an item onto the stack and pop an item off the stack. The differences between *aqstk.spp* and *aqstk2.spp* lie in the package body. A comparison of the two approaches will help you see how to take advantage of the many different flavors of queuing available.

The *aqstk.spp* file represents my first try at a stack implementation. I decided to create a prioritized queue. I then needed to come up with a way to make sure that the last item added to the queue always had the lowest priority. This is done by maintaining a global variable inside the package body (g_priority) to keep track of the priority of the most-recently enqueued message.

Every time I enqueue a new message, that global counter is decremented (lower number = higher priority) as shown in the following push procedure (bold lines show priority-related code):

```
   PROCEDURE

push (item IN VARCHAR2)
   IS
      queueopts DBMS_AQ.ENQUEUE_OPTIONS_T;
      msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;
      msgid aq.msgid_type;
      item_obj aqstk_objtype;
   BEGIN
      item_obj := aqstk_objtype (item);

 msgprops.priority := g_priority;
```

```
      queueopts.visibility := DBMS_AQ.IMMEDIATE;

  g_priority := g_priority - 1;

      DBMS_AQ.ENQUEUE (c_queue, queueopts, msgprops,
item_obj, msgid);
   END;
```

The problem with this approach is that each time you started anew using the stack package in your session, the global counter would be set at its initial value: 2^{30}. (I wanted to make sure that you didn't exhaust your priority values in a single session.) Why is that a problem? Because AQ queues are based in database tables and are persistent between connections. So if my stack still held a few items, it would be possible to end up with multiple items with the same priority.

To avoid this problem, I set up the initialization section of my stack package in *aqstk.spp* as follows:

```
BEGIN
   /* Drop the existing queue if present. */
   aq.stop_and_drop (c_queue_table);

   /* Create the queue table and queue as necessary. */
   aq.create_priority_queue (c_queue_table,
'aqstk_objtype', c_queue);
END aqstk;
```

In other words: wipe out the existing stack queue table, and queue and recreate it. Any leftover items in the stack will be discarded.

That approach makes sense if I don't want my stack to stick around. But why build a stack on Oracle AQ if you don't want to take advantage of the persistence? I decided to go back to the drawing board and see if there was a way to always dequeue from the top without relying on some external (to AQ) counter.

I soon discovered the sequence deviation field of the enqueue options record. This field allows you to request a deviation from the normal sequencing for a message in a queue. The following values can be assigned to this field:

DBMS_AQ.BEFORE

> The message is enqueued ahead of the message specified by relative_msgid.

DBMS_AQ.TOP

> The message is enqueued ahead of any other messages.

NULL (the default)

> There is no deviation from the normal sequence.

So it seemed that if I wanted my queue to act like a stack, I simply had to specify "top" for each new message coming into the queue. With this in mind, I created my push procedure as follows:

```
   PROCEDURE push (item IN VARCHAR2)
   IS
      queueopts DBMS_AQ.ENQUEUE_OPTIONS_T;
      msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;
      item_obj aqstk_objtype;
```

```
   BEGIN
      item_obj := aqstk_objtype (item);
      queueopts.sequence_deviation := DBMS_AQ.TOP;
      queueopts.visibility := DBMS_AQ.IMMEDIATE;
      DBMS_AQ.ENQUEUE (c_queue, queueopts, msgprops,
item_obj, g_msgid);
   END;
```

My pop procedure could now perform an almost-normal dequeue as follows:

```
   PROCEDURE pop (item OUT VARCHAR2)
   IS
      queueopts DBMS_AQ.DEQUEUE_OPTIONS_T;
      msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;
      msgid aq.msgid_type;
      item_obj aqstk_objtype;
   BEGIN
      /* Workaround for 8.0.3 bug; insist on dequeuing of
first message. */
      queueopts.navigation := DBMS_AQ.FIRST_MESSAGE;

      queueopts.wait := DBMS_AQ.NO_WAIT;
      queueopts.visibility := DBMS_AQ.IMMEDIATE;
      DBMS_AQ.DEQUEUE (c_queue, queueopts, msgprops,
item_obj, g_msgid);
      item := item_obj.item;
   END;
```

Notice that the first line of this procedure contains a workaround. A bug in Oracle 8.0.3 requires that I insist on dequeuing of the first message (which is always the last-enqueued message, since I used sequence deviation) to avoid raising an error. I was not able to confirm by the time of this book's printing whether this was necessary in Oracle 8.0.4.

With this second implementation of a stack, I no longer need or want to destroy the queue table and queue for each new connection. As a consequence, my package initialization section simply makes sure that all the necessary AQ objects are in place:

```
BEGIN
   /* Create the queue table and queue as necessary. */
   aq.create_queue ('aqstk_table', 'aqstk_objtype',
c_queue);
END aqstk;
```

So you have two choices with a stack implementation (and probably more, but this is all I am going to offer):

aqstk.spp

A stack that is refreshed each time you reconnnect to Oracle.

aqstk2.spp

A stack whose contents persist between connections to the Oracle database.

5.7.4 Browsing a Queue's Contents

One very handy feature of Oracle AQ is the ability to retrieve messages from a queue in a nondestructive fashion. In other words, you can read a message from the queue without *removing* it from the queue at the same time. Removing on dequeue is the default mode of AQ (at least for messages that are not part of a message group).

However, you can override that default by requesting BROWSE mode when dequeuing.

Suppose that I am building a student registration system. Students make requests for one or more classes and their requests go into a queue. Another program dequeues these requests (destructively) and fills the classes. But what if a student drops out? All of their requests must be removed from the queue. To perform this task, I must scan through the queue contents, but remove destructively only when I find a match on the student social security number. To illustrate these steps, I create an object type for a student's request to enroll in a class:

```
/* Filename on companion disk:

aqbrowse.sp */*
CREATE TYPE student_reg_t IS OBJECT
   (ssn VARCHAR2(11),
    class_requested VARCHAR2(100));
/
```

I then build a drop_student procedure, which scans the contents of a queue of previous objects of the type. The algorithm used is quite simple: within a simple LOOP, dequeue messages in BROWSE mode. If a match is found, dequeue in REMOVE mode for that specific message by its unique message identifier. Then continue in BROWSE mode until there aren't any more messages to be checked.

```
/* Filename on companion disk:

aqbrowse.sp */*
CREATE OR REPLACE PROCEDURE

drop_student (queue IN VARCHAR2, ssn_in IN VARCHAR2)
IS
   student student_reg_t;
   v_msgid aq.msgid_type;
   queue_changed BOOLEAN := FALSE;

   queueopts DBMS_AQ.DEQUEUE_OPTIONS_T;
   msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;

   /* Translate mode number to a name. */
   FUNCTION

mode_name (mode_in IN INTEGER) RETURN VARCHAR2
   IS
   BEGIN
      IF    mode_in = DBMS_AQ.REMOVE THEN RETURN 'REMOVE';
      ELSIF mode_in = DBMS_AQ.BROWSE THEN RETURN 'BROWSE';
      END IF;
   END;

   /* Avoid any redundancy; doing two dequeues, only
difference is the
      dequeue mode and possibly the message ID to be
dequeued. */
   PROCEDURE

dequeue (mode_in IN INTEGER)
   IS
   BEGIN
      queueopts.dequeue_mode := mode_in;
      queueopts.wait := DBMS_AQ.NO_WAIT;
      queueopts.visibility := DBMS_AQ.IMMEDIATE;

      IF mode_in = DBMS_AQ.REMOVE
      THEN
         queueopts.msgid := v_msgid;
```

```
         queue_changed := TRUE;
      ELSE
         queueopts.msgid := NULL;
      END IF;

      DBMS_AQ.DEQUEUE (queue_name => queue,
         dequeue_options => queueopts,
         message_properties => msgprops,
         payload => student,
         msgid => v_msgid);

      DBMS_OUTPUT.PUT_LINE
          ('Dequeued-' || mode_name (mode_in) || ' ' ||
student.ssn ||
           ' class ' || student.class_requested);
   END;
BEGIN
   LOOP
      /* Non-destructive dequeue */
      dequeue (DBMS_AQ.BROWSE);

      /* Is this the student I am dropping? */
      IF student.ssn = ssn_in
      THEN
         /* Shift to destructive mode and remove from queue.
            In this case I request the dequeue by msg ID.
            This approach completely bypasses the normal
order
            for dequeuing. */
         dequeue (DBMS_AQ.REMOVE);
      END IF;
   END LOOP;

EXCEPTION
   WHEN aq.dequeue_timeout
   THEN
      IF queue_changed
      THEN
         COMMIT;
      END IF;
END;
/
```

Notice that even in this relatively small program, I still create a local or nested procedure to avoid writing all of the dequeue code twice. It also makes the main body of the program more readable. I also keep track of whether any messages have been removed, in which case queue_changed is TRUE. I also perform a commit to save those changes as a single transaction.

Here is a script I wrote to test the functionality of the drop_student procedure:

```
/* Filename on companion disk:

aqbrowse.tst */*
DECLARE
   queueopts DBMS_AQ.ENQUEUE_OPTIONS_T;
   msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;
   student student_reg_t;
   v_msgid aq.msgid_type;
BEGIN
   aq.stop_and_drop ('reg_queue_table');

   aq.create_queue ('reg_queue_table', 'student_reg_t',
'reg_queue');

   queueopts.visibility := DBMS_AQ.IMMEDIATE;

   student := student_reg_t ('123-46-8888', 'Politics 101');
```

```
   DBMS_AQ.ENQUEUE ('reg_queue', queueopts, msgprops,
student, v_msgid);

   student := student_reg_t ('555-09-1798', 'Politics 101');
   DBMS_AQ.ENQUEUE ('reg_queue', queueopts, msgprops,
student, v_msgid);

   student := student_reg_t ('987-65-4321', 'Politics 101');
   DBMS_AQ.ENQUEUE ('reg_queue', queueopts, msgprops,
student, v_msgid);

   student := student_reg_t ('123-46-8888', 'Philosophy
101');
   DBMS_AQ.ENQUEUE ('reg_queue', queueopts, msgprops,
student, v_msgid);

   DBMS_OUTPUT.PUT_LINE ('Messages in queue: ' ||
      aq.msgcount ('reg_queue_table', 'reg_queue'));

   drop_student ('reg_queue', '123-46-8888');

   DBMS_OUTPUT.PUT_LINE ('Messages in queue: ' ||
      aq.msgcount ('reg_queue_table', 'reg_queue'));
END;
/
```

Here is an explanation of the different elements of the test script:

- Since this is a test, I first get rid of any existing queue elements and recreate them. This guarantees that my queue is empty when I start the test.
- I then perform four enqueues to the registration queue. In each case, I use the constructor method for the object type to construct an object. I then place that object on the queue. Notice that there are two requests for class enrollments for 123-46-8888 (the first and fourth enqueues).
- Next, I call my handy aq.msgcount function to verify that there are four messages in the queue.
- Time to scan and remove! I request that all class requests for the student with the 123-46-8888 social security number be dropped.
- Finally, I check the number of messages remaining in the queue (should be just two).

Here is the output from execution of the test script:

```
SQL> @aqbrowse.tst
stopping AQ$_REG_QUEUE_TABLE_E
dropping AQ$_REG_QUEUE_TABLE_E
stopping REG_QUEUE
dropping REG_QUEUE
dropping reg_queue_table
Messages in queue: 4
Dequeued-BROWSE 123-46-8888  class Politics 101
Dequeued-REMOVE 123-46-8888  class Politics 101
Dequeued-BROWSE 555-09-1798  class Politics 101
Dequeued-BROWSE 987-65-4321  class Politics 101
Dequeued-BROWSE 123-46-8888  class Philosophy 101
Dequeued-REMOVE 123-46-8888  class Philosophy 101
Messages in queue: 2
```

The first five lines of output show the drop-and-create phase of the script. It then verifies four messages in the queue. Next, you can see the loop processing. It browses

the first entry, finds a match, and then dequeues in REMOVE mode. Three browsing dequeues later, it finds another match, does the remove dequeue, and is then done.

5.7.4.1 A template for a show_queue procedure

You can also take advantage of BROWSE mode to display the current contents of a queue. The following code offers a template for the kind of procedure you would write. (It is only a template because you will need to modify it for each different object type (or RAW data) you are queueing.)

```
/* Filename on companion disk:




aqshowq.sp */*
CREATE OR REPLACE PROCEDURE show_queue (queue IN VARCHAR2)
IS
/* A generic program to dequeue in browse mode from a queue
to
   display its current contents.

   YOU MUST MODIFY THIS FOR YOUR SPECIFIC OBJECT TYPE.
*/
   obj <YOUR OBJECT TYPE>;
   v_msgid aq.msgid_type;

   queueopts DBMS_AQ.DEQUEUE_OPTIONS_T;
   msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;

   first_dequeue BOOLEAN := TRUE;
BEGIN
   LOOP
      /* Non-destructive dequeue */
      queueopts.dequeue_mode := DBMS_AQ.BROWSE;
      queueopts.wait := DBMS_AQ.NO_WAIT;
      queueopts.visibility := DBMS_AQ.IMMEDIATE;

      DBMS_AQ.DEQUEUE (queue_name => queue,
         dequeue_options => queueopts,
         message_properties => msgprops,
         payload => obj,
         msgid => v_msgid);

      /* Now display whatever you want here. */
      IF first_dequeue
      THEN
         DBMS_OUTPUT.PUT_LINE ('YOUR HEADER HERE');
         first_dequeue := FALSE;
      END IF;

      DBMS_OUTPUT.PUT_LINE ('YOUR DATA HERE');
   END LOOP;

EXCEPTION
   WHEN aq.dequeue_timeout
   THEN
      NULL;
END;
/
```

Check out the *aqcorrid.spp* file (and the layaway.display procedure), described in the next section, for an example of the way I took this template file and modified it for a specific queue.

5.7.5 Searching by Correlation Identifier

You don't have to rely on message identifiers in order to dequeue a specific message from a queue. You can also use application-specific data by setting the correlation identifier.

Suppose that I maintain a queue for holiday shopping layaways. All year long, shoppers have been giving me money towards the purchase of their favorite bean-bag stuffed animal. I keep track of the requested animal and the balance remaining in a queue of the following object type (found in *aqcorrid.spp*):

```
CREATE TYPE layaway_t IS OBJECT
   (animal VARCHAR2(30),
    held_for VARCHAR2(100),
    balance NUMBER
   );
/
```

When a person has fully paid for his or her animal, I will remove the message from the queue and store it in a separate database table. Therefore, I need to be able to identify that message by the customer and the animal for which they have paid. I can use the correlation identifier to accomplish this task.

Here is the package specification I have built to manage my layaway queue:

```
/* Filename on companion disk:

aqcorrid.spp */*
CREATE OR REPLACE PACKAGE BODY



 layaway
IS
   FUNCTION one_animal (customer_in IN VARCHAR2, animal_in
IN VARCHAR2)
      RETURN layaway_t;

   PROCEDURE make_payment
      (customer_in IN VARCHAR2,
       animal_in IN VARCHAR2,
       payment_in IN NUMBER);

   PROCEDURE display
      (customer_in IN VARCHAR2 := '%', animal_in IN
VARCHAR2 := '%');
 END layaway;
/
```

The layaway. one_animal function retrieves the specified animal from the queue utilizing the correlation identifier. The layaway. make_payment procedure records a payment for that stuffed animal (and decrements the remaining balance). The layaway.display procedure displays the contents of the queue by dequeuing in BROWSE mode.

I built a script to test this package as follows:

```
/* Filename on companion disk:

 aqcorrid.tst */*
DECLARE
   obj layaway_t;
```

```
BEGIN
   layaway.make_payment ('Eli', 'Unicorn', 10);

   layaway.make_payment ('Steven', 'Dragon', 5);

   layaway.make_payment ('Veva', 'Sun Conure', 12);

   layaway.make_payment ('Chris', 'Big Fat Cat', 8);

   layaway.display;

   obj := layaway.one_animal ('Veva', 'Sun Conure');

   DBMS_OUTPUT.PUT_LINE ('** Retrieved ' || obj.animal);
END;
/
```

Notice that I do not have to deal with the layaway object type unless I am retrieving an animal for final processing (i.e., the customer has paid the full amount and it is time to hand that adorable little pretend animal over the counter).

Here is the output from my test script:

```
SQL> @aqcorrid.tst
Customer             Animal               Balance
Eli                  Unicorn               39.95
Steven               Dragon                44.95
Veva                 Sun Conure            37.95
Chris                Big Fat Cat           41.95
** Retrieved Sun Conure
```

And if I run the same script twice more, I see the following:

```
SQL> @aqcorrid.tst
Input truncated to 1 characters
Customer             Animal               Balance
Steven               Dragon                39.95
Veva                 Sun Conure            37.95
Eli                  Unicorn               29.95
Chris                Big Fat Cat           33.95
** Retrieved Sun Conure

PL/SQL procedure successfully completed.

SQL> @aqcorrid.tst
Input truncated to 1 characters
Customer             Animal               Balance
Veva                 Sun Conure            37.95
Eli                  Unicorn               19.95
Steven               Dragon                34.95
Chris                Big Fat Cat           25.95
** Retrieved Sun Conure
```

Notice that the order of messages in the queue changes each time (as well as the balance remaining). That happens because I am dequeuing and enqueuing back into the queue. Since I have not specified any priority for the queue table, it always dequeues (for purposes of display) those messages most recently enqueued.

Let's now take a look at the implementation of this package (also found in *aqcorrid.spp*). Let's start with the one_animal function:

```
FUNCTION

one_animal (customer_in IN VARCHAR2, animal_in IN VARCHAR2)
```

```
   RETURN layaway_t
IS
   queueopts DBMS_AQ.DEQUEUE_OPTIONS_T;
   msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;

   retval layaway_t;
BEGIN
   /* Take immediate effect; no commit required. */
   queueopts.wait := DBMS_AQ.NO_WAIT;
   queueopts.visibility := DBMS_AQ.IMMEDIATE;

   /* Retrieve only the message for this correlation
identifier. */

 queueopts.correlation := corr_id (customer_in, animal_in);


   /* Reset the navigation location to the first message. */

 queueopts.navigation := DBMS_AQ.FIRST_MESSAGE;


   /* Locate the entry by correlation identifier and return
the object. */
   DBMS_AQ.DEQUEUE (c_queue, queueopts, msgprops, retval,
g_msgid);

   RETURN retval;

EXCEPTION
   WHEN aq.dequeue_timeout
   THEN
      /* Return a NULL object. */
      RETURN layaway_t (NULL, NULL, 0);
END;
```

Most of this is standard enqueue processing. The lines that are pertinent to using the correlation ID are in boldface. I set the correlation field of the dequeue options to the string returned by the corr_id function (shown next). I also set the navigation for the dequeue operation to the *first message* in the queue. I do this to make sure that Oracle AQ starts from the beginning of the queue to search for a match. If I do not take this step, then I raise the following exception when running my *aqcorrid.tst* script more than once in a session:

```
ORA-25237: navigation option used out of sequence
```

This behavior may be related to bugs in the Oracle 8.0.3 release, but the inclusion of the navigation field setting definitely takes care of the problem.

So when I dequeue from the layaway queue, I always specify that I want the first message with a matching correlation string. I have hidden away the construction of that string behind the following function:

```
FUNCTION corr_id  (customer_in IN VARCHAR2, animal_in IN
VARCHAR2)
   RETURN VARCHAR2
IS
BEGIN
   RETURN UPPER (customer_in || '.' || animal_in);
END;
```

I have taken this step because I also need to create a correlation string when I enqueue (shown in the following make_payment procedure). In order to minimize maintenance and reduce the chance of introducing bugs into my code, I do not want this

concatenation logic to appear more than once in my package.

Here is the make_payment procedure:

```
PROCEDURE make_payment
   (customer_in IN VARCHAR2, animal_in IN VARCHAR2,
payment_in IN NUMBER)
IS
   queueopts DBMS_AQ.ENQUEUE_OPTIONS_T;
   msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;
   layaway_obj layaway_t;
BEGIN
   /* Locate this entry in the queue by calling the
function.
      If found, decrement the balance and reinsert into the
queue.
      If not found, enqueue a new message to the queue. For
example
      purposes, the price of all my bean-bag animals is
$49.95. */

   layaway_obj := one_animal (customer_in, animal_in);

   /* Adjust the balance. We SHOULD check for 0 or negative
values,
      and not requeue if finished. I will leave that to the
reader. */

   IF layaway_obj.animal IS NOT NULL
   THEN
      layaway_obj.balance := layaway_obj.balance -
payment_in;
   ELSE
      /* Construct new object for enqueue, setting up
initial balance. */
      layaway_obj := layaway_t (animal_in, customer_in,
49.95 - payment_in);
   END IF;

   /* Don't wait for a commit. */
   queueopts.visibility := DBMS_AQ.IMMEDIATE;

   /* Set the correlation identifier for this message. */
   msgprops.correlation := corr_id (customer_in, animal_in);

   DBMS_AQ.ENQUEUE (c_queue, queueopts, msgprops,
layaway_obj, g_msgid);
END;
```

The first thing that make_payment does is attempt to retrieve an existing queue entry for this customer-animal combination by calling one_animal. Again, notice that I do *not* repeat the dequeue logic in make_payment. I am always careful to reuse existing code elements whenever possible. If I find a match (the animal field is not NULL; see the exception section in one_animal to understand how I set the "message not found" values in the returned object), then I update the remaining balance. If no match is found, then I construct an object to be placed in the queue.

Once my layaway object has been updated or created, I set the correlation identifier by calling the same corr_id function. Notice that when I enqueue, I set the correlation field of the message properties record. When I dequeue, on the other hand, I set the correlation field of the dequeue options record. Finally, I enqueue my object.

5.7.5.1 Wildcarded correlation identifiers

You can also specify wildcard comparisons with the correlation identifier, using the

standard SQL wildcarding characters _ (single-character substitution) and % (multiple-character substitution).

For example, if I set the value of "S%" for my queue options correlation field, as follows, then AQ will find a correlation for any message whose correlation identifier starts with an upper-case "S."

```
queueopts.correlation := "S%";
```

5.7.5.1.1 Tips for using the correlation identifier

When you are using the correlation identifier, remember these tips:

- When you enqueue, set the correlation field of the message properties record.
- When you dequeue, set the correlation field of the dequeue options record.
- Before you dequeue, set the navigation field of the dequeue options record to DBMS_AQ.FIRST_MESSAGE to avoid out-of-sequence errors.

5.7.6 Using Time Delay and Expiration

If you have started the Queue Monitor process, you can set up queues so that messages cannot be dequeued for a period of time. You can also specify that a message will expire after a certain amount of time has passed. These features come in handy when messages in your queue have a "window of applicability" (in other words, when there is a specific period of time in which a message should or should not be available for dequeuing).

If a message is not dequeued before it expires, that message is automatically moved to the exception queue (either the default exception queue associated with the underlying queue table or the exception queue specified at enqueue time). Remember that the time of expiration is calculated from the earlier dequeuing time. So if you specify a delay of one week (and you do this in seconds, as in 7 Ч 24 Ч 60 Ч 60) and an expiration of two weeks, the message would expire (if not dequeued) in three weeks.

To delay the time when a message can be dequeued, modify the delay field of the message properties record. Modify the expiration time by setting a value for the expiration field of the message properties record.

Now let's see how to use the expiration feature on messages to manage sales for products in my store. I created the following object type:

```
CREATE TYPE sale_t IS OBJECT
   (product VARCHAR2(30),
    sales_price NUMBER
   );
/
```

Here are the rules I want to follow:

- A product goes on sale for a specific price in a given period (between start and end dates).
- Every product that is on sale goes into my sales queue. When a message is enqueued, I compute the delay and expiration values based on the start and end dates.

- I can then check to see if a product is on sale by checking my sales queue: if I can dequeue it (nondestructively: that is, in BROWSE mode) successfully, then it is on sale.
- I never dequeue in REMOVE mode from the sales queue. Instead, I simply let the Queue Monitor automatically move the product from the sales queue to the exception queue when that message expires.

To hide all of these details from my application developers (who in turn will hide all programmatic details from *their* users, the people pushing buttons on a screen), I will construct a package. Here's the specification for this sale package:

```
/* Filename on companion disk:

aqtiming.spp */*
CREATE OR REPLACE PACKAGE sale
IS
   FUNCTION onsale (product_in IN VARCHAR2) RETURN BOOLEAN;

   PROCEDURE mark_for_sale
      (product_in IN VARCHAR2,
       price_in IN NUMBER,
       starts_on IN DATE,
       ends_on IN DATE);

   PROCEDURE show_expired_sales;

END sale;
/
```

So I can check to see if the Captain Planet game is on sale as follows:

```
IF sale.onsale ('captain planet')
THEN
   ...
END IF;
```

I can mark Captain Planet for a special sales price of $15.95 during the month of December as follows:

```
sale.mark_for_sale (
   'captain planet',
   15.95,
   TO_DATE ('01-DEC-97'),
   TO_DATE ('31-DEC-97'));
```

Finally, I can at any time display those products whose sales windows have expired as follows:

```
SQL> exec sale.show_expired_sales;
```

To test this code, I put together the following scripts. First, I create the queue table, queue for sales, and exception queue for sale items that expire in the original sales queue.

```
/* Filename on companion disk:

 aqtiming.ins */*
DECLARE
   c_qtable CONSTANT aq.name_type := 'sale_qtable';
   c_queue CONSTANT aq.name_type := 'sale_queue';
```

```
   c_exc_queue CONSTANT aq.name_type := 'sale_exc_queue';

BEGIN
   /* Create the queue table and queue as necessary. */
   aq.create_queue (c_qtable, 'sale_t', c_queue);

   /* Create a special exception queue for expired sales
listings. */
   aq.create_queue (c_qtable, 'sale_t', c_exc_queue,
      qtype => DBMS_AQADM.EXCEPTION_QUEUE);
END sale;
/
```

I then combine a number of sales-related operations into a single script:

NOTE: To run this script, you must have EXECUTE privilege on DBMS_LOCK. If you do not will see this error:

```
PLS-00201: identifier 'SYS.DBMS_LOCK' must be
declared
```

You or your DBA must connect to SYS and issue this command:

```
GRANT EXECUTE ON DBMS_LOCK TO PUBLIC;
```

```
/* Filename on companion disk:

aqtiming.tst */*
DECLARE
   FUNCTION seconds_from_now (num IN INTEGER) RETURN DATE
   IS
   BEGIN
      RETURN SYSDATE + num / (24 * 60 * 60);
   END;

   PROCEDURE show_sales_status (product_in IN VARCHAR2)
   IS
      v_onsale BOOLEAN := sale.onsale (product_in);
      v_qualifier VARCHAR2(5) := NULL;
   BEGIN
      IF NOT v_onsale
      THEN
         v_qualifier := ' not';
      END IF;

      DBMS_OUTPUT.PUT_LINE (product_in ||
         ' is' || v_qualifier || ' on sale at ' ||
         TO_CHAR (SYSDATE, 'HH:MI:SS'));
   END;
BEGIN
   DBMS_OUTPUT.PUT_LINE ('Start test at ' || TO_CHAR
(SYSDATE, 'HH:MI:SS'));

   sale.mark_for_sale ('Captain Planet', 15.95,
      seconds_from_now (30), seconds_from_now (50));

   sale.mark_for_sale ('Mr. Math', 12.95,
      seconds_from_now (120), seconds_from_now (180));

   show_sales_status ('Captain Planet');
   show_sales_status ('Mr. Math');
```

```
   DBMS_LOCK.SLEEP (30);
   DBMS_OUTPUT.PUT_LINE ('Slept for 30 seconds.');
   show_sales_status ('Captain Planet');
   show_sales_status ('Mr. Math');

   sale.show_expired_sales;

   DBMS_LOCK.SLEEP (100);
   DBMS_OUTPUT.PUT_LINE ('Slept for 100 seconds.');
   show_sales_status ('Captain Planet');
   show_sales_status ('Mr. Math');

   sale.show_expired_sales;

   DBMS_LOCK.SLEEP (70);
   DBMS_OUTPUT.PUT_LINE ('Slept for 70 seconds.');
   show_sales_status ('Captain Planet');
   show_sales_status ('Mr. Math');
END;
/
```

Here is the output from this test script (I insert my comments between chunks of output to explain their significance):

```
Start test at 12:42:57
```

The next four lines come from sale.mark_for_sale and show how the start and end dates were converted to seconds for delay and expiration. As you start using this technology, I strongly suggest that you encapsulate your date-time computations inside a wrapper like seconds_from_now so that you can keep it all straight.

```
Delayed for 30 seconds.
Expires after 20 seconds.
Delayed for 120 seconds.
Expires after 60 seconds.
```

I check the status of my sale items immediately, and neither is yet available at their sale price. The delay time is still in effect.

```
Captain Planet is not on sale at 12:42:58
Mr. Math is not on sale at 12:42:58
```

I put the program to sleep for 30 seconds and then check again. Now Captain Planet is on sale (the delay was only 30 seconds), but smart shoppers cannot pick up Mr. Math for that special deal.

```
Slept for 30 seconds.
Captain Planet is on sale at 12:43:28
Mr. Math is not on sale at 12:43:28
```

After another 100 seconds, Captain Planet is no longer on sale, but look at those copies of Mr. Math fly out the door!

```
Slept for 100 seconds.
Captain Planet is not on sale at 12:45:08
Mr. Math is on sale at 12:45:08
```

Why isn't Captain Planet on sale? The output from a call to sale.show_expired_sales makes it clear: the window of opportunity for that sale has closed, and the message has been "expired" into the exception queue.

```
Product              Price              Expired on
Captain Planet       $15.95             11/14/1997 12:42:57
```

After another 70 seconds, neither Captain Planet nor Mr. Math are on sale, and as you might expect, both appear on the exception queue:

```
Slept for 70 seconds.
Captain Planet is not on sale at 12:46:18
Mr. Math is not on sale at 12:46:18

Product              Price              Expired on
Captain Planet       $15.95             11/14/1997 12:42:57
Mr. Math             $12.95             11/14/1997 12:42:58
```

Yes, dear readers, these nifty Oracle AQ features do seem to work as documented!

Now let's examine the implementation of the programs in the sale package. Rather than reproduce the entire body in these pages, I will focus on the individual programs. You can find the full set of code in the *aqtiming.spp* file.

First, we have the sale. onsale function. This program returns TRUE if the specified product is currently available for dequeuing. Here is the code:

```
/* Filename on companion disk:

aqtiming.spp */*
FUNCTION onsale (product_in IN VARCHAR2) RETURN BOOLEAN
IS
   queueopts DBMS_AQ.DEQUEUE_OPTIONS_T;
   msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;
   obj sale_t;
BEGIN
   /* Take immediate effect; no commit required. */
   queueopts.wait := DBMS_AQ.NO_WAIT;
   queueopts.visibility := DBMS_AQ.IMMEDIATE;

   /* Reset the navigation location to the first message. */
   queueopts.navigation := DBMS_AQ.FIRST_MESSAGE;

   /* Dequeue in BROWSE. You never dequeue destructively.
You let
      the Queue Monitor automatically expire messages and
move them
      to the exception queue. */
   queueopts.dequeue_mode := DBMS_AQ.BROWSE;

   /* Retrieve only the message for this product. */
   queueopts.correlation := UPPER (product_in);

   /* Locate the entry by correlation identifier and return
the object. */
   DBMS_AQ.DEQUEUE (c_queue, queueopts, msgprops, obj,
g_msgid);

   RETURN obj.product IS NOT NULL;

EXCEPTION
   WHEN aq.dequeue_timeout
   THEN
      RETURN FALSE;
END;
```

This is a standard nondestructive dequeue operation. Notice that I use the correlation identifier to make sure that I attempt to dequeue a message only for this particular product. I also set navigation to "first message" to make sure I get the first message (in

enqueue time) for the product. In this case, I do not return any of the sale information. Instead, I return TRUE if I found a non-NULL product in the dequeued object. If I timeout trying to retrieve a message, I return FALSE.

Of course, I need to be able to put a product on sale. I do that with the sale.mark_for_sale procedure.

```
/* Filename on companion disk:

aqtiming.spp */*
PROCEDURE mark_for_sale
   (product_in IN VARCHAR2,
    price_in IN NUMBER,
    starts_on IN DATE,
    ends_on IN DATE)
IS
   queueopts DBMS_AQ.ENQUEUE_OPTIONS_T;
   msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;
   sale_obj sale_t;

BEGIN
   /* Calculate the delay number of seconds and the
expiration in same terms */
   msgprops.delay := GREATEST (0, starts_on - SYSDATE) * 24
* 60 * 60;
   msgprops.expiration := GREATEST (0, ends_on - starts_on)
* 24 * 60 * 60;

   DBMS_OUTPUT.PUT_LINE
      ('Delayed for ' || msgprops.delay || ' seconds.');
   DBMS_OUTPUT.PUT_LINE
      ('Expires after ' ||  msgprops.expiration || '
seconds.');

   /* Don't wait for a commit. */
   queueopts.visibility := DBMS_AQ.IMMEDIATE;

   /* Set the correlation identifier for this message to
the product. */
   msgprops.correlation := UPPER (product_in);

   /* Specify a non-default exception queue. */
   msgprops.exception_queue := c_exc_queue;

   /* Set up the object. */
   sale_obj := sale_t (product_in, price_in);

   DBMS_AQ.ENQUEUE (c_queue, queueopts, msgprops, sale_obj,
g_msgid);
END;
```

This procedure is a wrapper around the enqueue operation. First I convert the start and end dates to numbers of seconds for the delay and expiration field values, and I display those values for debugging purposes.

Then I set the other characteristics of the enqueue. Most importantly, I set the correlation ID so that I can look just for this product later in my dequeue operation (shown in the sale.onsale function), and I specify an alternate exception queue just for expired sales messages.

Finally, I include a program to show me the contents of my exception queue. The sale.show_expired_sales is interesting because it combines two different elements of Oracle AQ: use of the operational interface (the dequeue program) and direct access against the data dictionary view. I execute a cursor FOR loop against AQ$sale_qtable, which is the underlying database table created by Oracle AQ to hold messages for all queues in this

queue table. Notice that I request only those rows in the exception queue I specified in sale.mark_for_sale. I retrieve the message ID and then I dequeue explicitly for that message ID. Why do I do this? When a message is moved to the exception queue, its message state is set to EXPIRED. I cannot dequeue a message in this state using normal navigation.

```
/* Filename on companion disk:

aqtiming.spp */*
PROCEDURE show_expired_sales
IS
   obj sale_t;

   v_msgid aq.msgid_type;

   CURSOR exp_cur
   IS
      SELECT msg_id
        FROM AQ$sale_qtable
       WHERE queue = c_exc_queue
       ORDER BY enq_time;

   queueopts DBMS_AQ.DEQUEUE_OPTIONS_T;
   msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;
BEGIN
   FOR exp_rec IN exp_cur
   LOOP
      /* Non-destructive dequeue by explicit message ID. */
      queueopts.dequeue_mode := DBMS_AQ.BROWSE;
      queueopts.wait := DBMS_AQ.NO_WAIT;
      queueopts.visibility := DBMS_AQ.IMMEDIATE;
      queueopts.msgid := exp_rec.msg_id;

      DBMS_AQ.DEQUEUE (c_exc_queue, queueopts, msgprops,
obj, v_msgid);

      IF exp_cur%ROWCOUNT = 1
      THEN
         DBMS_OUTPUT.PUT_LINE (
            RPAD ('Product', 21) || RPAD ('Price', 21) ||
'Expired on');
      END IF;

      DBMS_OUTPUT.PUT_LINE (
         RPAD (obj.product, 21) ||
         RPAD (TO_CHAR (obj.sales_price, '$999.99'), 21) ||
         TO_CHAR (msgprops.enqueue_time, 'MM/DD/YYYY
HH:MI:SS'));
   END LOOP;

EXCEPTION
   WHEN aq.dequeue_timeout
   THEN
      NULL;
END;
```

The range of possible behaviors for enqueue and dequeue operations is truly remarkable. However, this flexibility has its dark side: it can take a lot of experimentation and playing around to get your code to work just the way you want it to. It took me several hours, for example, to put together, debug, test, and think about the sale package in *aqtiming.spp* before it all came together. Try not to get too frustrated, and take things a step at a time, so you are always working from sure footing in terms of your understanding of AQ and your program's behavior.

5.7.7 Working with Message Groups

In some cases, you may wish to combine multiple messages into a single "logical" message. For example, suppose that you were using AQ to manage workflow on invoices. An invoice is a complex data structure with a header row (or object), multiple line item rows (or objects), and so forth. If you were up and running on a fully object-oriented implementation in Oracle8, you might easily have a single object type to encapsulate that information. On the other hand, what if your invoice information is spread over numerous objects and you just don't want to have to restructure them or create a single object type to hold that information for purposes of queueing? And on yet another hand, what if you want to make sure that when a consumer process dequeues the header invoice information, it also must dequeue all of the related information?

Simply set up your queue to treat all messages queued in your own logical transaction as a single message group. Once you have done this, a message is not considered by AQ to be dequeued until all the messages contained in the same group have been dequeued.

5.7.7.1 Enqueuing messages as a group

Let's walk through the different steps necessary to group messages logically, and then we'll explore the consequences in the way that the dequeue operation works. This section will cover these steps:

- Creating a queue table that will support message grouping
- Enqueuing messages within the same transaction boundary

5.7.7.1.1 Step 1. Create a queue table that will support message grouping

To do this, you simply override the default value for the message_grouping argument in the call to DBMS_AQADM.CREATE_QUEUE_TABLE with the appropriate packaged constant as follows:

```
BEGIN
   DBMS_AQADM.CREATE_QUEUE_TABLE
      (queue_table => 'msg_with_grouping',
       queue_payload_type => 'message_type',
       message_grouping => DBMS_AQADM.TRANSACTIONAL);

   /* Now I will create and start a queue to use in the
next example. */
   DBMS_AQ.CREATE_QUEUE ('classes_queue',
'msg_with_grouping');
   DBMS_AQ.START_QUEUE ('classes_queue');
END;
/
```

One thing to note immediately is that all of the different messages in the queue must be of the same type, even though they may potentially hold different kinds, or levels, of information.

5.7.7.1.2 Step 2. Enqueue messages within the same transaction boundary

However, your queue table is enabled to store and treat messages as a group. You must still make sure that when you perform the enqueue operation, all messages you want in a group are committed at the same time.

This means that you should *never* specify DBMS_AQ.IMMEDIATE for the visibility of your enqueue operation in a message group-enabled queue. Instead, you should rely on

the DBMS_AQ.ON_COMMIT visibility mode. This mode ensures that all messages will be processed as a single transaction, giving AQ the opportunity to assign the same transaction ID to all the messages in that group.

Here is an example of an enqueue operation loading up all the classes for a student as a single message group:

```
PROCEDURE semester_to_queue (student_in IN VARCHAR2)
IS
   CURSOR classes_cur
   IS
      SELECT classyear, semester, class
        FROM semester_class
       WHERE student = student_in;

   queueopts DBMS_AQ.ENQUEUE_OPTIONS_T;
   msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;

   v_msgid aq.msgid_type;
   class_obj semester_class_t;

BEGIN
   /* This is the default, but let's make sure! */
   queueopts.visibility := DBMS_AQ.ON_COMMIT;

   FOR rec IN classes_cur
   LOOP
      class_obj := semester_class_t (student_in, rec.class,
rec.semester);
      DBMS_AQ.ENQUEUE
         ('classes_queue', queueopts, msgprops, class_obj,
v_msgid);
   END LOOP;

   /* Now commit as a batch to get the message grouping. */
   COMMIT;
END;
```

And that's all it takes to make sure that messages are treated as a group: enable the queue when you create the queue table, and make sure that all messages are committed together.

When you work with message groups, you'll find that you will almost always be using PL/SQL loops to either enqueue the set of messages as a group or dequeue all related messages.

Now let's take a look at the dequeuing side of message group operations.

5.7.7.2 Dequeuing messages when part of a group

To give you a full sense of the code involved, I will shift from individual programs to a package. Suppose that I want to place in a queue (as a single group) all of the classes for which a student is enrolled (that is, the semester_to_queue procedure shown in the previous section). But I also want to display (and simultaneously dequeue) the contents of that queue for each student. I can take advantage of the message grouping feature to do this.

Here is the specification of the package:

```
/* Filename on companion disk:

aqgroup.spp */*
```

```
CREATE OR REPLACE PACKAGE



schedule_pkg
IS
   PROCEDURE semester_to_queue (student_in IN VARCHAR2);

   PROCEDURE show_by_group;
END;
/
```

The *aqgroup.ins* file creates the data needed to demonstrate the functionality of the schedule_pkg package. I will not repeat the implementation of semester_to_queue; instead, let's focus on the code you have to write to *dequeue* grouped messages in the show_by_group procedure.

```
PROCEDURE show_by_group
IS
   obj semester_class_t;
   v_msgid aq.msgid_type;
   first_in_group BOOLEAN := TRUE;

   queueopts DBMS_AQ.DEQUEUE_OPTIONS_T;
   msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;
BEGIN
   /* Just dumping out whatever is in the queue, so no
waiting. */
   queueopts.wait := DBMS_AQ.NO_WAIT;

   /* Start at the beginning of the queue, incorporating
all enqueues. */
   queueopts.navigation := DBMS_AQ.FIRST_MESSAGE;

   LOOP
      /* Local block to trap exception: end of group. */
      BEGIN
         DBMS_AQ.DEQUEUE (c_queue, queueopts, msgprops,
obj, v_msgid);

         IF first_in_group
         THEN
            first_in_group := FALSE;

            DBMS_OUTPUT.PUT_LINE
               ('Schedule for ' || obj.student ||
                ' in semester ' || obj.semester);
         END IF;

         DBMS_OUTPUT.PUT_LINE ('*   ' || obj.class);

         /* Navigate to the next message in the group. */
         queueopts.navigation := DBMS_AQ.NEXT_MESSAGE;

      EXCEPTION
         WHEN aq.end_of_message_group
         THEN
            /* Throw out a break line. */
            DBMS_OUTPUT.PUT_LINE ('*****');

            /* Move to the next student. */
            queueopts.navigation :=
DBMS_AQ.NEXT_TRANSACTION;

            /* Set header flag for new student. */
            first_in_group := FALSE;
      END;
   END LOOP;
EXCEPTION
```

```
   WHEN aq.dequeue_timeout
   THEN
      /* No more students, no more message groups. */

NULL;



END;
```

5.7.8 Working with Multiple Consumers

In the simpler schemes of queueing, one producer puts a message on a queue and another agent, a consumer, retrieves that single message from a queue. A common variant of this process follows a broadcasting model, where a producer enqueues a message with the intention of distributing that message to many consumers. Oracle AQ allows you to perform this kind of broadcast in two different ways:[1]

> [1] Prior to Oracle AQ, the DBMS_ALERT package already supported this broadcast mechanism; I would not be surprised to find that DBMS_ALERT is redesigned to use AQ in a future release.

- Define a default subscriber list for a queue. Then any message that is placed on that queue is available for dequeuing by any of the agents in that subscriber list.
- Specify an "override" recipient list when you enqueue a specific message to the queue, by assigning a list (index-by table) of recipients to the recipient_list field of the message properties record.

In both of these cases, you must have defined the queue table in which your queue is defined to support multiple consumers. Here is an example of the creation of a queue table that supports multiple consumers:

```
BEGIN
   DBMS_AQADM.CREATE_QUEUE_TABLE
      (queue_table => 'msg',
       queue_payload_type => 'message_type',
       multiple_consumers => TRUE);
END;
/
```

Let's take a look at the different steps involved in using both the default subscriber list and the override recipient list. Suppose that in my student registration and management system, I want to define a default set of subscribers who are to receive notification of a student's change in major_pkg. When the student changes his or her major from mathematics or philosophy to business, however, notification is sent to the school psychologist and the professor of ethics.

I will demonstrate these techniques by constructing incrementally a package that supports the change-major operation.

5.7.8.1 Using the subscriber list

The default application behavior is to send out the major change notification to the president of the school and the single guidance counselor (it's a small place). I could just hard-code this logic into my programs, but instead, I will build a more flexible, encapsulated interface for this action and then deploy it for those two people.

First, I must create an object type to use in my queue. (All the elements of these initialization steps, including the creation of the queue table, can be found in the file *aqmult.ins* .)

```
CREATE TYPE student_major_t IS OBJECT
   (student VARCHAR2(30),
    major VARCHAR2(100));
/
```

I then create a queue table and queue based on this object type. Notice the specification of a multiple consumers queue:

```
/* Filename on companion disk:

 aqmult.ins */*
BEGIN
   /* Create the queue table and queue for multiple
consumers. */
   DBMS_AQADM.CREATE_QUEUE_TABLE
      (queue_table => 'major_qtable',
       queue_payload_type => 'student_major_t',
       multiple_consumers => TRUE);

 DBMS_AQADM.CREATE_QUEUE ('major_queue', 'major_qtable');

   DBMS_AQADM.START_QUEUE ('major_queue');
END;
/
```

Now I can construct my package. Here is the specification:

```
/* Filename on companion disk:

aqmult1.spp */*
CREATE OR REPLACE PACKAGE major_pkg
IS
   PROCEDURE add_reviewer (name_in IN VARCHAR2);

   PROCEDURE change_it_again
      (student_in IN VARCHAR2, new_major_in IN VARCHAR2);
END major_pkg;
/
```

So at this point, I can add a reviewer to the queue; this is a person who is to be notified *by default* of any major changes. I can also change the major of a student. Let's look at how I would use these programs. First of all, I need to specify the default reviewers:

```
/*Filename on companion disk:

aqmult2.ins */*
BEGIN
   major_pkg.add_reviewer ('President Runtheshow');
   major_pkg.add_reviewer ('Counselor Twocents');
END;
/
```

Now that my main subscribers are in place, I can change the major of a student and rest assured that entries will be made in the queue for all the people who need to know.

```
SQL> exec major_pkg.change_it_again ('Steven Feuerstein',
'Biology');
```

Wait a minute! That's not what I want -- I want to study the English language!

```
SQL> exec major_pkg.change_it_again ('Steven Feuerstein',
'English');
```

And so on. We're about to get into more detailed scenarios for both construction and testing, so I added the following steps to my installation script, *aqmult.ins* :

```
CREATE TABLE student_intention
   (name VARCHAR2(30),
    ssn CHAR(11),
    major_study VARCHAR2(100));

BEGIN
   INSERT INTO student_intention VALUES
      ('Steven Feuerstein', '123-45-6789', 'Mathematics');
   INSERT INTO student_intention VALUES
      ('Eli Feuerstein', '123-45-6780', 'Philosophy');
   INSERT INTO student_intention VALUES
      ('Veva Feuerstein', '123-45-6781', 'Pottery');
   INSERT INTO student_intention VALUES
      ('Chris Feuerstein', '123-45-6782', 'Art');
   COMMIT;
END;
/
```

You should run this script before playing around with *aqmult2.spp* or *aqmult3.spp* (the last two iterations in this exercise), described in the following code examples.

Now, each time I change my major (or someone else's), a message is written to the queue. By default, each message is read by two subscribers, the president and the guidance counselor. The way Oracle AQ works is that a message is not considered dequeued (and therefore *removed* , assuming that you are dequeuing in the default destructive mode) until all consumers specified by the subscriber list or the override recipients list have dequeued that message. You request messages for which you are a subscriber or a recipient by setting the appropriate value in the dequeue options consumer name field.

Here is how the process might work for our ever-changing student majors: each morning, the executive assistant of the president connects to the system and pulls out a report of any students who changed their major yesterday. Here is a procedure that might do this:

```
/* Filename on companion disk:

aqmult2.spp */*
PROCEDURE

show_changers_to (curious_in IN VARCHAR2)
IS
   obj student_major_t;
   v_msgid aq.msgid_type;

   queueopts DBMS_AQ.DEQUEUE_OPTIONS_T;
   msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;

   first_dequeue BOOLEAN := TRUE;
BEGIN
   queueopts.consumer_name := curious_in;

   /* Loop through the contents of the queue looking for
      matches on the specified recipient name. */
   LOOP
```

```
      /* Non-destructive dequeue */
      queueopts.wait := DBMS_AQ.NO_WAIT;
      queueopts.navigation := DBMS_AQ.FIRST_MESSAGE;
      queueopts.visibility := DBMS_AQ.IMMEDIATE;

      DBMS_AQ.DEQUEUE (queue_name => c_queue,
         dequeue_options => queueopts,
         message_properties => msgprops,
         payload => obj,
         msgid => v_msgid);

      IF first_dequeue
      THEN
         DBMS_OUTPUT.PUT_LINE
            ('Changed Majors on ' || TO_CHAR (SYSDATE-1));
         first_dequeue := FALSE;
      END IF;

      DBMS_OUTPUT.PUT_LINE (
         obj.student || ' changed major to ' || obj.major);
   END LOOP;

EXCEPTION
   WHEN aq.dequeue_timeout
   THEN
      NULL;
END;
```

This is a typical destructive dequeue operation, except that it will dequeue the message only if the specified curious person is in the default subscription list or is specified in a recipient list an enqueue time.

The following script demonstrates how this technology all works together:

```
/* Filename on companion disk:

aqmult2.tst */*
BEGIN
   major_pkg.change_it_again ('Steven Feuerstein',
'Philosophy');
   major_pkg.change_it_again ('Veva Feuerstein', 'English');
   major_pkg.change_it_again ('Eli Feuerstein', 'Strategic
Analysis');

   COMMIT;

   major_pkg.show_changers_to ('President Runtheshow');
END;
/
```

And here is the output from that script:

```
SQL> @aqmult2.tst
Changed Majors on 23-NOV-97
Steven Feuerstein changed major to Philosophy
Veva Feuerstein changed major to English
Eli Feuerstein changed major to Strategic Analysis
```

5.7.8.2 Overriding with a recipient list

Now let's add some code to the package to support the special logic for changing from math or philosophy to a business degree. (Surely we have enough MBAs in the world already!) I need to make changes to the change_it_again procedure. You will find this third iteration in the *aqmult3.spp* file.

In this final version, I need to find out what the current major is for my student, so that I can compare it to the new choice and see if it triggers my rule to notify two different nosey-bodies at the school. I could simply drop that query into the change_it_again procedure, but that practice leads to redundant coding of SQL statements in my application -- a serious no-no. I will surely want to fetch the major of a student in more than one place, so I should put that specific action inside a standard lookup function, which is shown here as a fragment of the major package:

```
/* Filename on companion disk:

aqmult3.spp */*
CREATE OR REPLACE package body major_pkg
IS
   /* Just showing this new part of the package. */
   FUNCTION

current_choice (student_in IN VARCHAR2) RETURN VARCHAR2
   IS
      CURSOR maj_cur
      IS
         SELECT major_study
           FROM student_intention
          WHERE name = student_in;
      maj_rec maj_cur%ROWTYPE;
   BEGIN
      OPEN maj_cur;
      FETCH maj_cur INTO maj_rec;
      RETURN maj_rec.major_study;
   END;

END major_pkg;
/
```

The sharp reader will no doubt also point out that I have embedded an UPDATE statement inside the change_it_again procedure as well. True. That should be converted into a procedure with a name like major_pkg.upd_major_study. I will leave that exercise for the reader.

Now I can use this current_choice function inside my upgraded change_it_again, as shown in the next code listing. First, an explanation: I declare a recipient list (which is actually an index-by table) to hold the school psychologist and the professor of ethics -- if needed. Then before I update the major, I retrieve the current choice using that function. After the update, I see if the condition is met. If so, I define two rows in the recipient list and assign that list to the recipient_list field of the message properties record. I then perform the same enqueue operation as before.

```
   PROCEDURE

change_it_again
      (student_in IN VARCHAR2, new_major_in IN VARCHAR2)
   IS
      queueopts DBMS_AQ.ENQUEUE_OPTIONS_T;
      msgprops DBMS_AQ.MESSAGE_PROPERTIES_T;
      major_obj major_t;
      those_who_need_to_know DBMS_AQ.AQ$_RECIPIENT_LIST_T;
   BEGIN
      /* What is the current major? */
      v_major := current_choice (student_in);

      /* Update the database table. */
      UPDATE student_intention
         SET major_study = new_major_in
       WHERE name = student_in;
```

```
      /* IF changing from math or philosophy to business,
         build a special recipient list and add that to
         the enqueue operation. */
      IF v_major IN (c_philosophy, c_mathematics) AND
         new_major_in = c_business
      THEN
         /* Notify the school psychologist and professor of
ethics. */
         those_who_need_to_know (1) := SYS.AQ$_AGENT
('Doctor Baddreams');
         those_who_need_to_know (1) := SYS.AQ$_AGENT
('Doctor Whatswrong);

         msgprops.recipient_list := those_who_need_to_know;
      END IF;

      /* Put a message on the queue so that everyone is
         properly notified. Notice I will coordinate
visibility
         of this message with a COMMIT of the entire
transaction.*/
      queueopts.visibility := DBMS_AQ.ON_COMMIT;

      /* Populate the object. */
      major_obj := student_major_t (student_in,
new_major_in);

      DBMS_AQ.ENQUEUE (c_queue, queueopts, msgprops,
major_obj, g_msgid);
   END;
```

The following script shows how the package now will automatically notify all of the right parties:

```
/* Filename on companion disk:

aqmult3.tst */*
DECLARE
   prez VARCHAR2(100) := 'President Runtheshow';
   counselor VARCHAR2(100) := 'Counselor Twocents';
   psych_dr VARCHAR2(100) := 'Doctor Baddreams';
   ethics_prof VARCHAR2(100) := 'Professor Whatswrong';
BEGIN
   major_pkg.change_it_again ('Steven Feuerstein',
'Philosophy');
   major_pkg.change_it_again ('Veva Feuerstein', 'English');

   major_pkg.show_changes_to (prez);
   major_pkg.show_changes_to (psych_dr);

   major_pkg.change_it_again ('Steven Feuerstein',
major_pkg.c_business);
   major_pkg.change_it_again ('Veva Feuerstein',
major_pkg.c_philosphy);

   major_pkg.show_changes_to (counselor);
   major_pkg.show_changes_to (ethics_prof);
END;
/
```

Here is the output from the execution of this script:

```
SQL> @aqmult3.tst
Showing to President Runtheshow Majors Changed on 23-NOV-97
Steven Feuerstein changed major to Philosophy
Veva Feuerstein changed major to English

Showing to Counselor Twocents Majors Changed on 23-NOV-97
```

```
Steven Feuerstein changed major to Philosophy
Veva Feuerstein changed major to English
Veva Feuerstein changed major to Philosophy

Showing to Professor Whatswrong Majors Changed on 23-NOV-97
Steven Feuerstein changed major to Business
```

As you *should* be able to tell from this section's examples, it's not terribly difficult to set up a queue table for multiple consumers. Nor is it hard to define lists of subscribers and recipients. You must remember, however, to set the message properties consumer name field to retrieve a message for a given agent. And you should remember that the message will stay queued until all agents who have access to that message have performed their dequeue (or, for some reason, the message is moved to the exception queue).

PREVIOUS 5.6 Oracle AQ Database Objects | HOME BOOK INDEX | NEXT 6. Generating Output from PL/SQL Programs

Library Home | Oracle PL/SQL Programming, Second Edition | Oracle PL/SQL Programming: Guide to Oracle8i Features | Oracle Built-in Packages | Advanced PL/SQL Programming with Packages | Oracle Web Applications: PL/SQL Developer's Introduction | Oracle PL/SQL Language Pocket Reference | Oracle PL/SQL Built-ins Pocket Reference