Test-First Development with Mock J2EE, JMS, and JNDI

Shaun Smith
ClearStream Consulting
205 5th Ave SW
Suite 3710
Calgary, Alberta Canada
1 403 809-3085
shaun@clrstream.com

Jennitta Andrea
ClearStream Consulting
205 5th Ave SW
Suite 3710
Calgary, Alberta Canada
1 403 264-5840
jennitta@clrstream.com

Gerard Meszaros
ClearStream Consulting
205 5th Ave SW
Suite 3710
Calgary, Alberta Canada
1 403 560-2408
gerard@clrstream.com

INTRODUCTION

Having recently begun developing applications that depend upon the Java Message Service (JMS) [5], we searched the Internet for experiences on testing JMS applications. Finding nothing, we set out to determine how applications built on JMS could be unit tested.

In a message-based architecture, processing is performed when a message is received on a queue. Message processing may involve arbitrary computations and database interactions and usually results in yet another message being placed on another queue. The challenge was to determine out how to develop in a test-first approach when the JMS API does not readily permit the usual MockObject [3] approach of passing the appropriately configured MockObjects into objects to be tested.

BACKGROUND

The JMS API defines "a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations" [6]. JMS is an implementation independent API for messaging products in the same way that JDBC is an implementation independent API for relational databases. With the introduction of Message Driven Beans in version 2.0 of the Enterprise Java Beans (EJB) [4] specification, JMS is poised to become an important part of J2EE development. The evolution of test strategies for JMS applications is necessary if we are to practice test-first development when using these technologies.

MockObjects were devised as a way of verifying that a piece of software that is expected to invoke specific methods on other components is indeed invoking those methods. The usual procedure when using MockObjects is to create a mock implementation of a class, create an instance of it, and configure it to expect certain method calls in the course of a test.

MockObjects examples typically show the MockObject being passed in to the object under test as one of the arguments of the particular method under test. This is certainly possible when you are defining the API of the object under test but presents an interesting dilemma when the object under test is expected to acquire the (possibly mocked) object by other means.

RECEIVING A MESSAGE

Applications use JMS to receive and send messages. Messages can be received in one of two ways: calling `receive()` on a `QueueReceiver` (which will block until a message is received),

```
Message m = aQueueReceiver.receive();
```

or by implementing the MessageListener interface and registering for notification when a message is received.

```
public class MyListener
  implements MessageListener {
    public void onMessage(
      Message message) {
        // process message
    }
}
...
aQueueReceiver.
  setMessageListener(myListener);
```

Implementations using either of these approaches appear readily testable. The latter though the direct invocation of the `onMessage()` of a `MessageListener` with an appropriately configured Message object and the former through the use of a mock `QueueReceiver` configured to return a Message object when `receive()` is called. But using the `receive()` technique offers a particularly problematic challenge—a `QueueReceiver` must be obtained via the JMS API and cannot be passed in. We'll see the details when trying to send a message.

SENDING A MESSAGE

As stated above, it is typical for the processing of a message to involve the generation and sending of new messages. The following code sample (devoid of error checking) illustrates what has to be done to send a message using JMS.

```
jndiContext = new InitialContext();
queueConnectionFactory =
  (QueueConnectionFactory)
  jndiContext.lookup(
    QUEUE_CONNECTION_FACTORY_NAME);
queue = (Queue)
  jndiContext.lookup(QUEUE_NAME);
```

```
queueConnection =
  queueConnectionFactory
    .createQueueConnection();
queueSession =
  queueConnection.createQueueSession(
      IS_TRANSACTED,
      ACKNOWLEDGEMENT_MODE);
queueSender =
  queueSession.createSender(queue);
message =
  queueSession.createTextMessage();
message.setText(anXmlDocument);
queueSender.send(message);
```

USING MOCKOBJECTS WITH JMS

The last line in our *Sending a Message* example is the line of primary interest. This is the line that actually sends a message—the line we want to confirm occurs or, depending on the test scenario, does not occur. Unfortunately, as the sample illustrates, replacing the `queueSender` with a MockObject is not easy. The `queueSender` object is obtained after a series of method calls which lead back to a Java Naming and Directory Service (JNDI) lookup on an `InitialContext` object that is instantiated. The `queueReceiver` of the message receipt example is obtained through a nearly identical set of message sends. However, even though it looks impossible, it turns out that a mock `QueueSender` *can* be substituted for the real one.

The key is JNDI. The "`new InitialContext()`" creates a new object that provides access to the JNDI directory—a directory that can be configured with MockObjects. The setup for a test can configure JNDI with mock objects that will be returned when performing a lookup.[1]

Once a mock `QueueConnectionFactory` and a mock `Queue` have been published in JNDI, all that is left to do is to configure these mocks and the mocks they return to correctly execute the series of method send required to produce the mock `QueueSender`. Having achieved this, it is standard procedure to determine if the `QueueSender` was asked to send a message to a queue or not.

CONCLUSIONS

This exercise with JMS has produced two interesting results. First, frameworks defined in terms of interfaces are easy to "mock out". EasyMock [2] was employed to create all the required mock objects, which was only possible because JMS is defined entirely by interfaces. This use of interfaces was intended to allow for the use of various underlying messaging products—but it also allows for the use of a completely mock messaging implementation. This is a result that we can apply to the development of our own testable frameworks and applications.

Second, JNDI provides an alternative way of getting MockObjects into an object under test. Papers on the use of MockObjects thus far have relied upon the ability to pass MockObject into the method under test. Publishing MockObjects using JNDI is another way and, in some cases, the only way to introduce MockObjects into a test scenario. The J2EE technologies tend to rely on JNDI to obtain references to services. This reliance provides an opportunity, not a challenge, for unit testing J2EE applications.

More generally, this result illustrates that the use of a directory to locate services, no matter what the language or framework, provides a means of introducing MockObjects to support testing. In technologies lacking a directory service, we have used a component factory instead. Like a directory service, a component factory may be configured by a test to return suitably initialized MockObjects.

ACKNOWLEDGEMENTS

The authors would like to thank Doug Berscht of Cognicase for providing the opportunity for this work and Denis Clelland for his enthusiastic support of XP at ClearStream Consulting.

REFERENCES

1. Freeman, S. "Developing JDBC applications test-first", Online at www.mockobjects.com/papers/jdbc_testfirst.html
2. Freese, T., EasyMock 0.8, Online at www.easymock.org
3. Mackinnon, T., Freeman, S., Craig, P. "Endo-Testing: Unit Testing with Mock Objects", *eXtreme Programming and Flexible Processes in Software Engineering - XP2000*, May 2000.
4. Sun Microsystems Inc., *Enterprise JavaBeans™ Specification, Version 2.0*, Online at java.sun.com/products/ejb/docs.html
5. Sun Microsystems Inc., *Java Message Service*, Online at java.sun.com/products/jms/docs.html
6. Sun Microsystems Inc., *Java Message Service Tutorial*, Online at java.sun.com/products/jms/tutorial/index.html

[1] "Real" JNDI can be configured with mock JMS objects, but during testing is easier to install a mock JNDI implementation too.