# Maintaining Transactional Context: A Model Problem

Daniel Plakosh
Santiago Comella-Dorda
Grace Alexandra Lewis
Patrick R.H. Place
Robert C. Seacord

*August 2001*

TECHNICAL REPORT
CMU/SEI-2001-TR-012
ESC-TR-2001-012

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

# Maintaining Transactional Context: A Model Problem

CMU/SEI-2001-TR-012
ESC-TR-2001-012

Daniel Plakosh
Santiago Comella-Dorda
Grace Alexandra Lewis
Patrick R.H. Place
Robert C. Seacord

*August 2001*

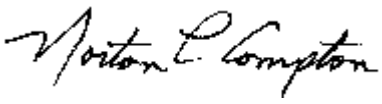
**COTS-Based Systems**

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER



Norton L. Compton, Lt Col., USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.



This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

# Table of Contents

# List of Figures

# List of Tables

# Abstract

Due to their size and complexity, modernizing enterprise systems often requires that new functionality be developed and deployed incrementally. As modernized functionality is deployed incrementally, transactions that were processed entirely in the legacy system may now be distributed across both legacy and modernized components.

In this report, we investigate the construction of adapters for a modernization effort that can maintain a transactional context between legacy and modernized components. One technique that is particularly useful in technology and product evaluations is the use of model problems—focused experimental prototypes that reveal technology/product capabilities, benefits, and limitations in well-bounded ways.

This report describes a model problem used to verify that a mechanism for maintaining a transactional context between legacy and modernized components exists and could be used to support the modernization of a legacy system. In this report, we describe a model problem constructed to verify the feasibility of building this mechanism. We also discuss the results of our investigation including the problems we encountered during the construction of the model problem and workarounds that were discovered.

# 1 Introduction

Due to their size and complexity, modernizing enterprise systems often requires that new functionality be developed and deployed incrementally. As modernized functionality is deployed incrementally, transactions that were processed entirely in the legacy system may now be distributed across both legacy and modernized components. Identifying and validating a design solution to this problem is a prerequisite to the overall modernization effort.

Our approach to identifying and validating design solutions involves the use of model problems—focused experimental prototypes that reveal technology/product capabilities, benefits, and limitations in well-bounded ways. Model problems are used to reduce design risk. However, no amount of modeling short of building the actual system can completely eliminate design risk. Therefore, the designer must be satisfied with achieving a degree of certainty in the design approach.

## 1.1 Model Problems

The use of model problems is a component-based software engineering technique described by Wallnau et al. [Wallnau 01].

A *model proble*m is actually a description of the design context. The overall process consists of the following steps, which are executed in sequence. There are two roles defined by the process: the *architect* and the *engineer*. The architect is the overall technical lead on the project who makes overall design decisions. The engineer is a designer who is tasked by the architect to execute the model problem.

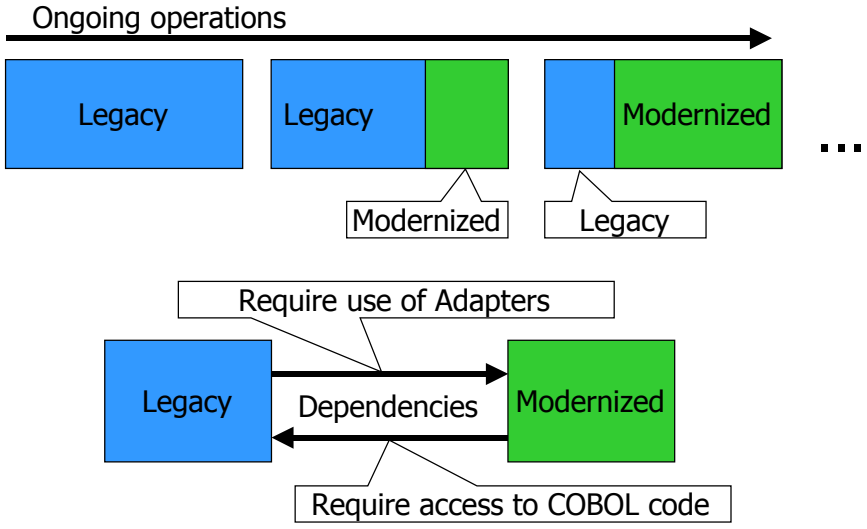
1. The architect and engineer identify a *design question*. This question initiates the model problem and refers to an unknown that is expressed as a hypothesis.
2. The architect and engineer define the *a priori evaluation criteria*. These criteria describe how the model solution will be shown to support or contradict the hypothesis.
3. The architect and engineer define the *implementation constraints*. These constraints specify the fixed (i.e., inflexible) part of the design context that governs the implementation of the model solution. These constraints might include things such as platform requirements, component versions, and business rules.
4. The engineer produces a *model solution* situated in the design context. The model solution is a minimal spanning application that uses only those features of a component (or components) that are necessary to support or contradict the hypothesis.

5. The engineer identifies *a posteriori evaluation criteria*. These evaluation criteria include the *a priori* criteria plus criteria that are discovered as a by-product of implementing the model solution.
6. Finally, the architect evaluates the model solution against the *a posteriori* criteria. The evaluation may result in the design solution being rejected or adopted, but often leads to the generation of new design questions that must be resolved in a similar fashion.

## 1.2 Case Study

This report focuses on a particular case study, which provides the context for the model problem. This case study involves the modernization of a large Retail Supply System (RSS) for a major U.S. retailer. The RSS consists of approximately 2 million lines of MicroFocus COBOL code running on a Solaris workstation. Data is stored in an Oracle 8i database. However, the overall architecture of the system has remained largely unchanged over 30 years, resulting in a system that is extremely brittle and difficult to maintain. (Comella-Dorda et al. provide a relevant description of an information system life cycle [Dorda 00]). As a result, the decision was made to modernize the RSS to a Java 2 Enterprise Edition (J2EE) platform. In particular, the modernized system will consist of Enterprise JavaBeans™ (EJBs) written in the Java programming language and deployed on an EJB application server.

Figure 1 shows an overview of the RSS modernization process. Initially the system consists completely of legacy COBOL code. At the completion of each increment, the percentage of legacy code decreases while the percentage of modernized code increases. Eventually, the system is completely modernized. Dependencies between legacy and modernized code require adapters to map between legacy and modernized system components.




™ Enterprise JavaBeans is a trademark of Sun Microsystems, Inc.

*Figure 1: RSS Modernization*

The modernization process consists of replacing legacy program elements with functionally equivalent EJBs. These beans are then deployed on a J2EE platform, in this case, the WebSphere application server developed by IBM.

As modernized functionality is deployed incrementally, transactions that were processed entirely in COBOL may now be distributed across both legacy and modernized components. Figure 2 shows the system after the incremental deployment of some modernized components. Apparent in this illustration is the fact that both the legacy COBOL code and modernized EJBs may update or access the database.

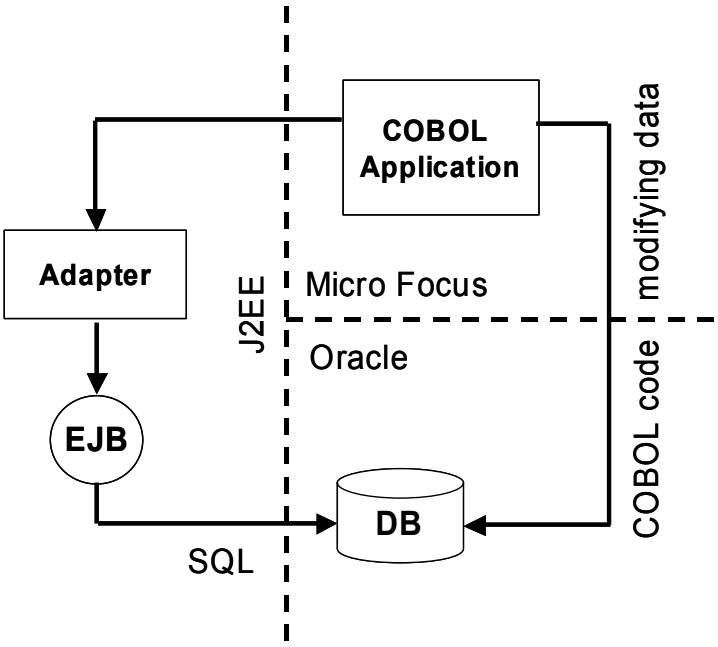



*Figure 2: The Operational System During Modernization*

An updated operation involving both legacy code and modern components is shown in the sequence diagram illustrated in Figure 3. In this diagram, the legacy COBOL module updates Table 2 by means of a SQL UPDATE. The COBOL module then invokes a method in a modernized component via an adapter that results in a SQL UPDATE to Table 1. Because the RSS modernization strategy is to maintain records in a single Oracle 8i database, there is no need to support two-phase commit in this scenario.[1]

To perform these updates within a transactional context, it is necessary to start and commit the transaction. Some scenarios might suggest that we alternately start or commit transactions in either the legacy or modernized components, but for simplicity we will assume that transactions are always started and committed from the legacy COBOL system.

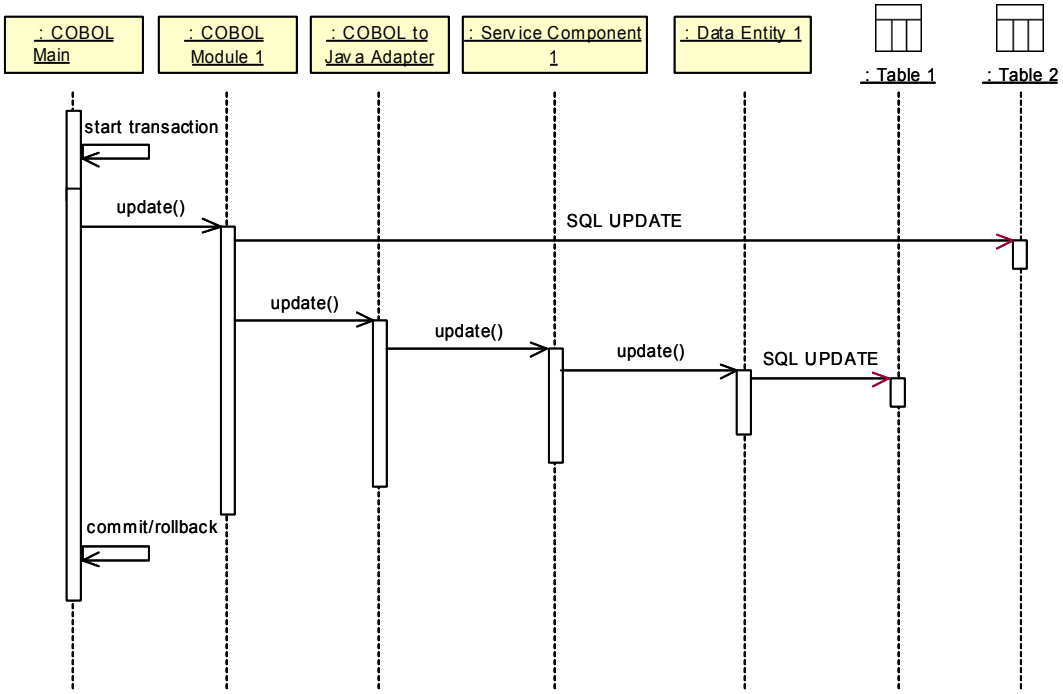



*Figure 3: Sequence Diagram Showing Transaction Update of Database Records*

The problem we now face is how to maintain transactional integrity across the COBOL-to-EJB interface.

[1] Transaction managers and resource managers use the two-phase commit (2PC) protocol with presumed roll back. That is, if something goes wrong, the transaction and resource managers involved in the transaction will always attempt to roll back their portions of the transaction. In Phase One, the prepare phase, the transaction manager asks all resource managers if they are ready and able to commit a transaction. If a resource manager responds negatively, it will automatically roll back any work it performed on behalf of the transaction and discard any knowledge it had of the transaction. In Phase Two, the commit phase, the transaction manager determines if there are any negative replies, and if so, instructs all resource managers to roll back. If all replies are positive, it will instruct the resource managers to commit.

There are several possible solutions, which we present in Section 2.

# 2 Contingency Planning

A fundamental tenement of component-based software engineering is *contingency-based design*. Simply put, contingency-based design allows for multiple design options to be pursued in parallel. This is critical when dealing with commercial components, as the implementations of these components are typically opaque to the architect, and their evolution is driven by the marketplace.

There are a number of possible design options that can be used to maintain the transactional context between the legacy COBOL system and the modernized system. Each of these contingencies is considered in this section of the report, along with our initial evaluation of the feasibility of each solution.

## 2.1 MQSeries

Existing modernization plans for the RSS assumed the use of MQSeries as a communication mechanism between the legacy COBOL and modernized EJB systems. MQSeries is an IBM product that provides asynchronous communications and uses independent queues to relay messages between communicating processes as shown in Figure 4.

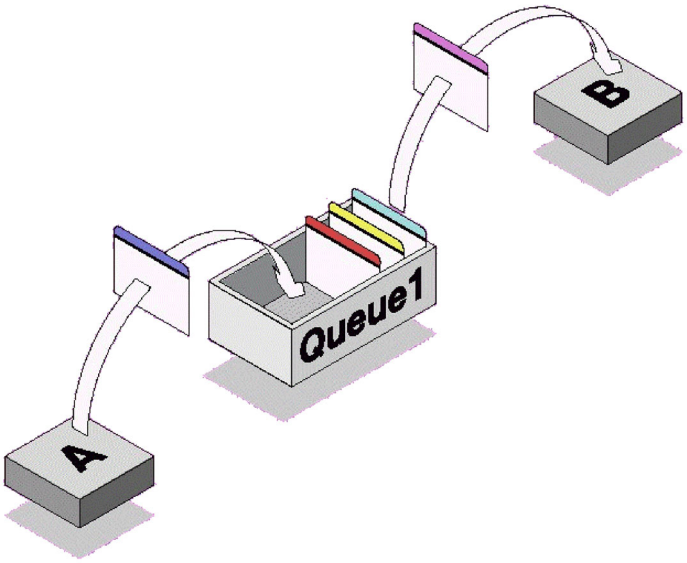



*Figure 4: Queue-Based Communication Using MQSeries*

MQSeries has some potential problems with respect to supporting transactions; in its current release, integration between MQSeries and WebSphere is limited. However, IBM claims that these products will be further integrated in the next release of WebSphere (Version 4.0) due out in the fall of 2001. However, even this future target state has severe limitations that can best be illustrated in an example. We assume that process A in Figure 4 represents the legacy system code written in MicroFocus COBOL and that B represents a modernized component developed as an EJB and deployed in the WebSphere application server. Queue1 is an input queue for the modernized component. The legacy code is passing a message via MQSeries to the modernized component to perform a function. We also assume that this function needs to be accomplished as part of a transaction. To do this, the MicroFocus COBOL program element will need to start a transaction and pass a message. In the planned Version 4 release of WebSphere, MQSeries will be able to maintain a transactional context through delivery of the message to the remote queue. However, once the EJB component removes the message from the queue, the transaction context is no longer maintained. This means that any database operations performed by the EJB component will take place outside of the transaction context.

We did not develop MQSeries further as a model problem solution due to this limitation in transaction propagation, although we maintained this option as a possible design contingency in case an asynchronous, message-oriented approach became a requirement.

## 2.2 Object Transaction Service

Object Transaction Service (OTS) is a distributed transaction-processing service specified by the Object Management Group (OMG). This specification extends the CORBA model and defines a set of interfaces to perform transaction processing across multiple CORBA objects. CORBA uses the Internet Inter-ORB Protocol (IIOP) as an interoperable protocol for communication between distributed objects.

As of the EJB Version 1.1 specification, the Remote Method Invocation (RMI) over IIOP has become the standard mechanism for supporting communication between a client and EJBs, and between EJB containers. IIOP is well suited for this purpose as it supports the propagation of both a transaction and security context. The WebSphere product, in particular, has been built around ComponentBroker ORB developed by IBM, even prior to the release of the EJB Version 1.1 specification.

To use OTS as a solution, we would need to find COBOL language bindings to a CORBA and OTS implementation. Optimally, if ComponentBroker had a MicroFocus COBOL interface, we could be fairly confident that this product would work in our target environment.

Although this approach appears to have potential, we had difficulty identifying a MicroFocus COBOL CORBA binding. A possible workaround was to use a Java CORBA binding, accessed through a MicroFocus COBOL-to-Java language interface. However, we decided not

to develop an OTS model problem at this time, but maintained it as a potentially viable design contingency.

## 2.3 Oracle Pro*COBOL

The Pro*COBOL precompiler is a programming tool that supports embedded SQL statements in high-level programming languages. This precompiler accepts the program as input, translates the embedded SQL statements into standard Oracle runtime library calls, and generates a source program that can be compiled, linked, and executed.

Although Pro*COBOL claims to be compatible with the MicroFocus Object COBOL Version 4.0 for 32-bit Windows® NT/95 compilers, it does not provide a solution for transaction management. Pro*COBOL is used primarily to preserve business logic in legacy COBOL programs when data is migrated to an Oracle database. Pro*COBOL supports transactions in embedded SQL statements, but does not solve the problem of maintaining a transactional context between legacy and modernized components. As a result, we eliminated this contingency as a possible design solution.

## 2.4 Net Express

MicroFocus Net Express® is an integrated development environment (IDE) for developing procedural COBOL/Object COBOL-based applications. Net Express supports mixed-language programming support for procedural COBOL, Object COBOL, and Java mixed-language programming, as well as WebSphere distributed-transaction technologies.

In supporting mixed-language programs, Net Express supports calling Java code from MicroFocus COBOL as well as calling MicroFocus COBOL from Java code. In particular, Net Express supports wrapping MicroFocus COBOL within an EJB. Potentially, each approach to supporting mixed-language programs could be used, so we examined each in turn.

### 2.4.1 Wrapping COBOL Code

Wrapping COBOL code within EJBs would allow the system to be migrated quickly to a J2EE environment, although clearly not one consisting of 100% pure Java code. To implement this approach, each legacy program element must be wrapped as an EJB, and all the internal calls must be converted to invoke the new Java methods—requiring the COBOL code inside the Java code to call Java again. There are several apparent consequences to this approach:

1. Turning legacy program elements into EJBs guarantees that the legacy architecture is maintained, as the decomposition of the system remains constant and the calls between

® Windows is a registered trademark of Microsoft Corporation.
® MicroFocus Net Express is a registered trademark of MERANT.

modules remain the same. As a result, this approach is incompatible with the RSS desire to migrate to a new target architecture.

2. The majority of the modernized system, in particular the business logic, is still implemented in COBOL. This means that any maintenance problems that existed will remain and be further complicated by the problems associated with maintaining a multi-language system.
3. Modernizing the system in this manner is not conducive to incremental development and would require a big-bang deployment of COBOL-filled EJBs.

The primary advantage of this approach is that it is a relatively inexpensive way to create a componentized system, but the characteristics of this modernized system would not be very different from those of the legacy system. As a result, we eliminated this approach as a possible design contingency.

### 2.4.2 Calling Java from COBOL

The second approach using Net Express is to call Java directly from the MicroFocus COBOL program elements. This approach would allow us to invoke EJB methods directly from MicroFocus COBOL and support WebSphere distributed transactions. As this approach appears to satisfy all of our requirements, we decided to construct a model problem to evaluate this design contingency.

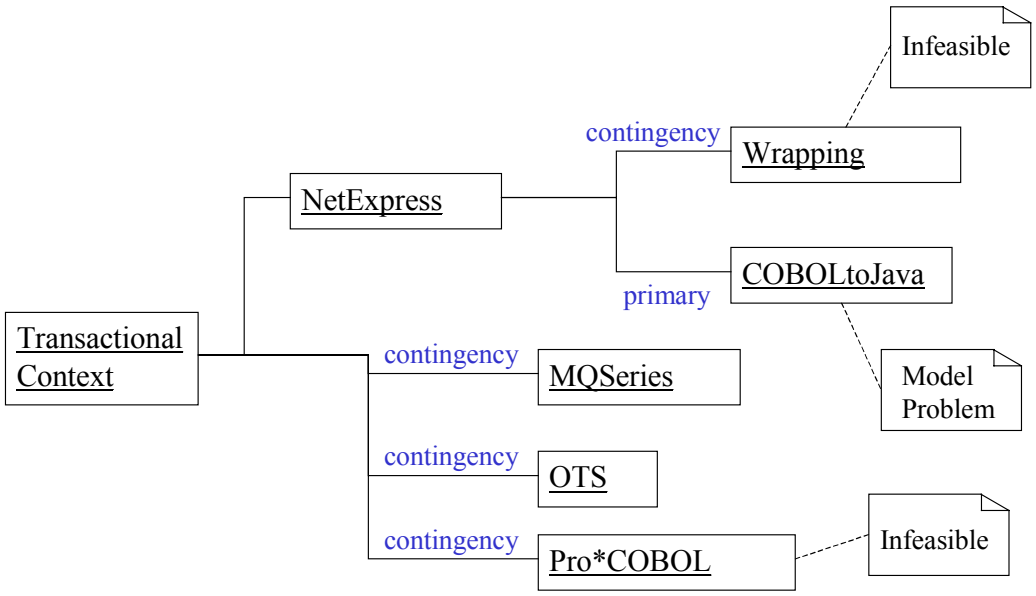



*Figure 5: Contigency Plan*

Figure 5 illustrates the result of the contingency planning. We have eliminated two contingencies, those of wrapping COBOL code using Net Express and using Oracle Pro*COBOL. Of the remaining three contingencies, we have decided to implement a model problem using Net Express to call Java from COBOL.

# 3 Model Problem Definition

To initiate a model problem, we must first create a hypothesis in two parts that establishes the design question:

> *Hypothesis Part #1: The MicroFocus Net Express integrated development environment can be used to support mixed-language programming with Java.*
>
> *Hypothesis Part #2: The Java subroutines, invoked from MicroFocus COBOL, can interface with an EJB server and perform transactions with an Oracle 8i database.*

If the first part cannot be supported, the second part is irrelevant. Now that we have defined the design question, we must identify the *a priori* evaluation criteria that will allow us to determine if the hypothesis can be supported:

> *Criterion #1: Committed updates from both the COBOL process and EJBs are applied correctly to the database.*
>
> *Criterion #2: A roll-back operation preserves the state of the database prior to the start of the transaction.*

The final step in defining the model problem is to identify any implementation constraints on the model solution. These constraints are set by the design context and are an important part of the model problem definition. For example, without the addition of the following constraints to this model problem, both of the stated criteria can be satisfied trivially:

1. The transaction must be started from the MicroFocus COBOL program and use the Java Transaction Service (JTS).
2. The MicroFocus COBOL program and the EJB must write to the Oracle 8i database as part of the same transaction.
3. The MicroFocus COBOL program will write to the Oracle 8i database using JDBC and SQL.

Taken together, the design question, *a priori* evaluation criteria, and implementation constraints provide the definition of the model problem. The next step is for the engineer to produce a *model solution* situated in this design context.

# 4 Model Problem Solution

This section describes our experience with the setup and development of the model solution.

## 4.1 Design of the Model Solution

The first step in implementing the model solution was to identify the sequence of steps that must be followed. In particular, the model solution must

1. Start a transaction using the JTS from a MicroFocus COBOL program.
2. Write to the Oracle 8i database using JDBC and SQL.
3. Invoke an EJB method that also wrote to the same Oracle 8i database as part of the same transaction.
4. Return control to the COBOL program.
5. Either roll back or commit the database changes made by both the COBOL and EJB programs.

For purposes of evaluation, it is often convenient to start with an existing prototype. Sample programs shipped with development tools are often ideally suited for this purpose. In this case, our model solution is based upon a sample banking program that manages client accounts using EJB. The EJB application consists of a database, a Java client, two EJBs, an Account Bean, and a Transfer Bean as shown in Figure 6. The Account Bean is an entity bean that persists in a relational table, and the Transfer Bean is a session bean that withdraws funds from one account and deposits the same amount in another account in the context of a single transaction. The Java client is a simple program that accepts a request from the user and invokes the beans to perform account creations or transfers. The client uses the Java Naming Directory Interface (JNDI) to get references to different resources and the Java Transaction API (JTA) to start, commit, and roll back transactions.

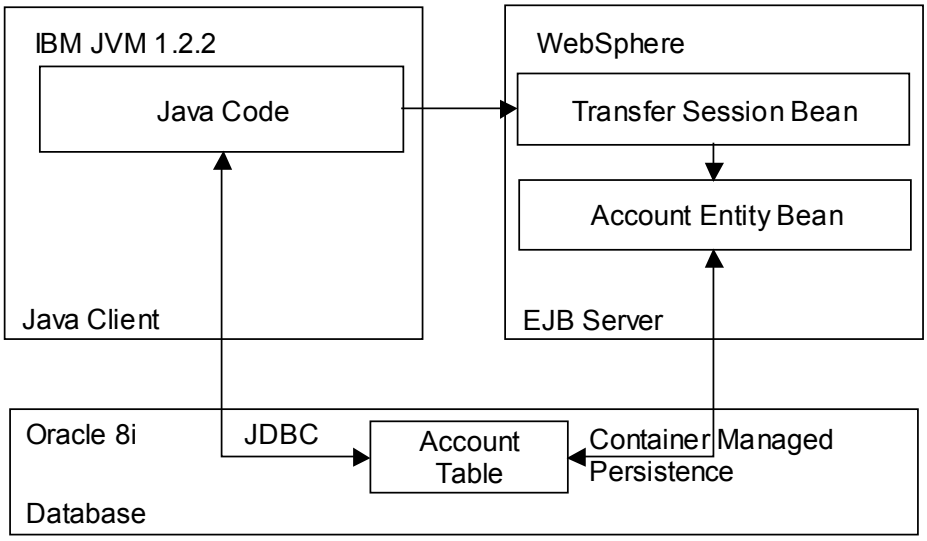



*Figure 6: Initial Architecture*

The Java client program was used to verify the operation of the banking application using EJB, transaction logic, and interaction with the Oracle 8i database.

This client program was later used to construct a test adapter as shown in Figure 7. The Java client is replaced with a combination of MicroFocus COBOL and Java code developed using Net Express MicroFocus COBOL.

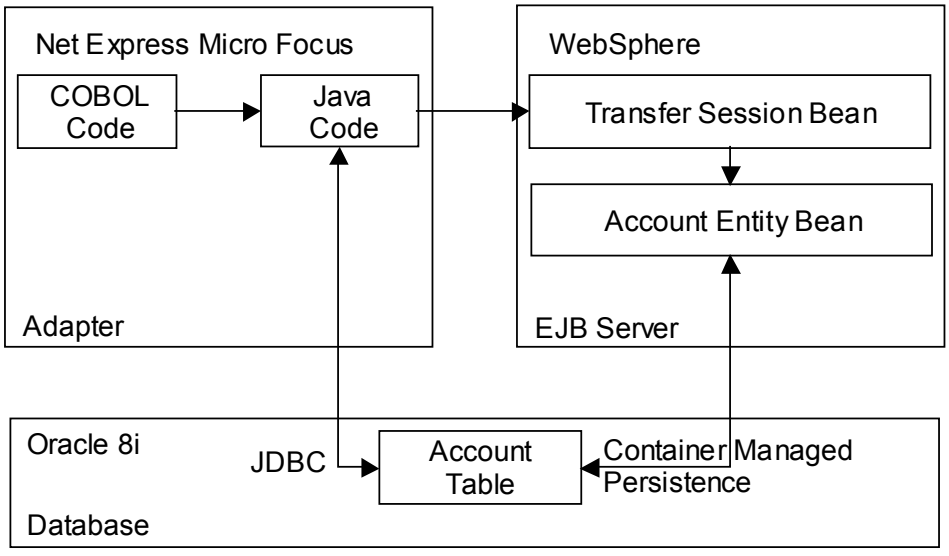



*Figure 7: Model Solution Architecture*

## 4.2 Installing the EJB Server

For our EJB server, we used WebSphere Application Server Advanced Edition (AE) for Windows. We installed the application server on a 256MB Windows 2000 machine.

The first thing we learned during the installation is that Windows 2000 is not supported by WebSphere AE but only "tolerated." Tolerated means that there is a newsgroup for people trying to install WebSphere on Windows 2000. You need to browse through the newsgroup to learn the tricks to make WebSphere work in that particular operating system (OS). For example, the software has to be installed from an "admin" account. We are not referring to any account with administrative privileges or to the default "administrator" account. It must be an account named "admin" for the installation to work. After learning this and other tricks we were able to complete the software installation in a reasonable amount of time.

## 4.3 EJB Bean Deployment

After installing the tool, we deployed two beans from the examples into the server. The deployment was quite straightforward; the only noteworthy issue we found is that the examples were configured to work with IBM's DB2®. Unfortunately, the demo version of WebSphere AE uses InstantDB, a pure Java database from IBM. We had to reconfigure the data sources of the server for the beans to work.

Creating the Java client (described in Section 3) was much more interesting. The clients in the examples are all Applets, Java Server Pages, and other Web-centered artifacts. We didn't want to clutter the model problem, so we developed a stand-alone Java client that does the minimum operations required to connect and access the application server. We noted that Java clients accessing WebSphere AE require IBM proprietary libraries to work; this contradicts the EJB standard but will probably have minimal impact over the program under consideration.

The real problems started when we changed the Java clients to access the database through JDBC in the same transaction as the data being modified by the Entity Beans. We first followed the WebSphere recommendations to get a JDBC connection from WebSphere's connection pool; this didn't work, probably because of InstantDB limitations. We then tried to create a brand new JDBC connection in the client code. This worked better, but we had data consistency problems between the beans and the client. This again turned out to be an InstantDB limitation. Finally, we decided to change the InstantDB for the Oracle 8i database. Swapping databases was not trivial, but it solved all data access problems.

The Oracle 8i database was installed using a typical installation. A SQL*Net connection to the database was defined; the Oracle initialization file had to be modified; and two users re

® DB2 is a registered trademark of International Business Machines.

quired by WebSphere had to be created so that the Account bean could be deployed. These instructions were found in the WebSphere documentation.

The Oracle JDBC thin driver was also installed. The JDBC thin driver is a 100% pure Java, type-4 driver that requires no client installation. The driver talks to the database using a 100% pure-Java presentation/session protocol. The Oracle JDBC thin driver is targeted towards applet developers and has all the functionality needed in this case. After these installations were complete, the JDBC connection in the client was modified so that it could use the JDBC driver and the SQL*Net connection.

## 4.4 Building the Test Adapter

As stated earlier, the test adapter requires the use of mixed-language programming using COBOL and Java. Therefore, our first step was to gain an understanding of how MicroFocus COBOL interfaces with Java.

We obtained an evaluation version of MicroFocus Net Express® IDE Version 3.1 and began working with the product to determine how to invoke Java methods from a COBOL program. Initially this seemed to be a trivial task, but it took much longer than we expected.

First, we examined the documentation provided with MicroFocus Net Express for information on how to invoke Java Methods from within a COBOL program. The documentation provides the mapping of COBOL types to Java types as shown in Table 1. It is currently not known how well the COBOL types defined for interacting with Java will map into the COBOL types used in the RSS. This is an area that requires further investigation.

After a thorough review of the documentation, we determined that information concerning MERANT's Java support was spotty at best. Besides mapping and setup information, the documentation contained only fragmented code examples that were incomplete and confusing and had not been updated completely to reflect the current version of the software. The documentation inaccuracies were actually discovered some time later when we browsed MERANT's Web site for additional information and determined that Version 3.1 of the software had significant improvements with respect to Java support. Next, we examined the Java demonstration programs that are provided as part of the IDE software package.

*Table 1: Java COBOL Mapping*

| Java | User defined | COBOL | Object COBOL | Description |
|---|---|---|---|---|
| Byte | jbyte | pic s99 comp-5 | pic s99 comp-5 | Signed 1-byte integer |
| Short | jshort | pic s9(4) comp-5 | pic s9(4) comp-5 | Signed 2-byte integer |
| Int | jint | pic s9(9) comp-5 | pic s9(9) comp-5 | Signed 4-byte integer |
| Long | jlong | pic s9(18) comp-5 | pic s9(18) comp-5 by ref only | Signed 8-byte integer |
| Boolean | jboolean | pic 99 comp-5 | pic 99 comp-5 | Zero value is false; non-zero is true. |
| Char | jchar | (Unicode) pic 9(4) comp-5 | N/A | All characters in Java are represented by 2-byte Unicode characters. |
| Float | jfloat | comp-1 | comp-1 (by ref only on Unix) | Floating-point number |
| Double | jdouble | comp-2 | comp-2 by ref only | Double-precision floating-point number |
| String | Mf-string | Pointer | pic x(n) | mf-jstring is a user-defined type giving the address, size and capacity of a string or buffer. For a String, the capacity is always zero. You should consider a string passed into a COBOL program as read-only, and not to be amended. For a StringBuffer, the capacity is the total size of the buffer, and the size the length of the string currently held in the buffer. |
| StringBuffer | | | | |
| Objects | N/A | Pointer | object reference | Any Java object. The pointer returned to procedural COBOL can be used with Java Native Interface (JNI) calls. |
| object[] | | Pointer | object reference to instance of class jarray | An array of Java objects. The pointer returned to procedural COBOL can be used with JNI calls. Jarray is an Object COBOL class for accessing the contents of Java arrays. |

The Java examples contained helpful information on how to call Java from COBOL and vice versa. These examples were interesting, because none of them showed how to call Java from a COBOL executable or how to pass strings from COBOL to Java. Examples typically consisted of a Java program calling a COBOL procedure linked into a Dynamic Link Library (DLL). Then, while executing the COBOL code inside the DLL, the COBOL program would invoke some Java method.

```
public class TestJava
{
 public void PassInt(int IntFromCOBOL)
 {
   System.out.println("int from COBOL: "+IntFromCOBOL);
 }
}
```

*Figure 8: Java Integer Code*

Since we could not find an example or documentation that met our needs, we developed a simple test program that passed an integer from COBOL to Java. The Java and COBOL sections of the program are shown in Figure 8 and Figure 9. Next, we tried to get these two simple programs to compile and execute. We thought that building this tiny mixed-language program was going to be a trivial exercise; the Java portion was in fact easy enough, but the COBOL portion turned out to be much more difficult than we had imagined.

We encountered several difficulties building the COBOL portion of the program. These problems ranged from builds that would compile, link, and execute without error, but only return to the command prompt when executed (not even an error log was produced), to problems with the IDE not really doing a complete rebuild when instructed. These problems were coupled with a lack of documentation on how to correctly configure the linker and the other compile parameters in the IDE to properly build a COBOL program that calls Java methods. This made this part of the task extremely difficult, but after spending some time in trial-and-error mode, we eventually got our simple program to execute correctly.

```
$set ooctrl(+p-f)
program-id. COBOLCallingJava.

class-control.
 TestJava is class "$java$TestJava"
 .
working-storage section.
copy javatypes.
  01  IntForJava        jint.
  01  JavaClassRef      object reference.

procedure division.
 display "Load Java Class"
 invoke TestJava "new" returning JavaClassRef
 display "Java Class Load Complete"
 set IntForJava to 123456
 invoke JavaClassRef "PassInt" using IntForJava
 invoke JavaClassRef "finalize" returning JavaClassRef
 stop run
 .
```

*Figure 9: COBOL Integer Code*

After discussing these problems with a MERANT representative, we were encouraged to download the various patches for the product that are available from MERANT's Web site. These patches fixed many of the problems that we were having with the IDE.

```
public class TestJava {
 public String PassInt(String StringFromCOBOL, int IntFromCOBOL) throws Ex-
ception {
  System.out.println(StringFromCOBOL+IntFromCOBOL);
  if (IntFromCOBOL==1234)
    {
     Exception e = new Exception ("Test Exception for COBOL" );
     throw e;
    }
  return("Hello from Java");
 }
}
```

*Figure 10: Expanded Integer Java Code*

Next, we expanded our test program to include the passing of a string, a return value, and the ability to catch Java exceptions in the COBOL portion of the program. The documentation provided showed how to perform Java exception handling in COBOL, but did not indicate how to pass a string from COBOL to Java. We looked through MERANT's support "Answers Lab" for information and eventually found a sample Java/COBOL program that performed string passing between COBOL and Java. It turned out that string passing was much simpler than we had expected. We had thought that string passing would be much more difficult—based on some old Net Express Version 3.0 examples we had found on MERANT's Web site and could not get to work correctly. Our updated COBOL and Java programs are shown in Figure 10 and Figure 11. We tested our new mixed-language application, and everything seemed to work correctly.

```
$set ooctrl(+p-f)
program-id. COBOLCallingJava.

class-control.
 ExceptionManager         is class "exptnmgr"
 EntryCallback            is class "entrycll"
 JavaExceptionManager     is class "javaexpt"
 TestJava                 is class "$java$TestJava"
.
working-storage section.
  copy javatypes.
  01  JavaClassRef      object reference.
  01  wsCallBack        object reference.
  01  wsIterator        object reference.
  01  IntForJava        jint.
  01  StringFromJava    pic x(100)
.

linkage section.
 01 lnkErrorNumber            pic x(4) comp-5.
 01 lnkErrorObject            object reference.
 01 lnkErrorTextCollection    object reference.
 01 lnkException              object reference.
 01 anElement                 object reference.

procedure division.
 invoke EntryCallback "new" using z"JException" returning wsCallback

 invoke ExceptionManager "register" using javaexceptionmanager wsCallback

 *>Register a CallBack to use as an Iterator (For Errors)
 invoke EntryCallback "new" using z"DispError" returning wsIterator

 display "Load Java Class"
 invoke TestJava "new"  returning JavaClassRef
 display "Test Java Load Complete"
 display "Enter a integer to pass"
 Accept IntForJava.
 invoke JavaClassRef "PassInt" using z"Int From COBOL :" IntForJava
        returning StringFromJava
 display "String Returned from Java = " StringFromJava
 invoke JavaClassRef "finalize" returning JavaClassRef
 stop run
 .
entry "Jexception" using lnkException lnkErrorNumber lnkErrorTextCollection.

 display "Error calling Java Class !"
 display "The Error from Java was:-"
 invoke lnkErrorTextCollection "do" using wsIterator
 stop run
 .
entry "DispError" using anElement.

 display "    " with no advancing
 invoke anElement "display"
 display " "
 goback
 .
```

*Figure 11: Expanded Integer Cobol Code*

Now that we had determined how to pass data successfully from a COBOL program to a Java program and handle Java exceptions from within COBOL, we were ready to begin building a test adapter. The test adapter was constructed from the Java client described in Section 3. Similar to the Java client, this adapter uses the Account Bean, Transfer Bean, JNDI, and JTA, except that it is controlled and instantiated via the COBOL portion of the program.

The Java portion of our adapter required Version 1.2.2 of the Java Virtual Machine (JVM) developed by IBM. We discovered that our version of Net Express only supported the JVM Version 1.1.8, so we called MERANT to ask if there was a patch that would allow us to run the JVM Version 1.2.2. Within a few days, MERANT sent us a beta patch set.

We installed the patch files, rebuilt our adapter program, and ran the program. During the program execution we received the obscure error shown in Figure 12.

```
Error calling ()V constructor for instance of class UltimateBankingApplica-
tion
```

*Figure 12: Error Return From Test Adapter*

To determine the cause of the error, we included print statements in the class constructor of the Java portion of our adapter and exception handlers to print out a stack trace. The only information that we could determine from this approach was that the error was occurring during the construction of `ivjInitContext`, as shown in Figure 13.

```
ivjInitContext = new InitialContext(properties);
```

*Figure 13: Statement Causing Error*

For some unknown reason, we could not get the Java stack trace option to print amplifying information in particular situations.

Next, to isolate the error we decompiled the class files associated with the `InitialContext()` constructor using Jad (a Java decompiler) and then recompiled these classes with debug information. During this process we discovered that the default internal class path that was used when running the JVM from the Java command was different from the internal class path used by MERANT when the COBOL program loaded the JVM. This difference turned out to be the cause of the error shown in Figure 12. The class path values that were missing are shown in Figure 14.

```
C:\IBMJDK\jre\lib\ext\rmiorb.jar;
C:\IBMJDK\jre\lib\ext\iioprt.jar;
```

*Figure 14: Missing Class Path Values*

We added the two missing jar files to the class path of the IDE build configuration for the test adapter and rebuilt out test adapter. When executed, our test adapter no longer failed constructing `ivjInitContext,` but we discovered that strings were not being passed to Java correctly.

Even though our program that passed a string and an integer to Java seemed to work correctly, we determined through our test adapter that COBOL strings must be NULL terminated before they are passed in a Java invocation, and that Java methods receiving strings from a COBOL program must strip off any trailing spaces. Additionally, we noticed that any COBOL string that is used as a return value for a Java method invocation must be cleared before the Java method is invoked. We did not notice this issue with our simple test program, because we: were only passing one string; were not performing string compares; and only invoked the Java method once instead of repeatedly.

We made the modifications described above to the COBOL and Java portions of the program and the adapter executed correctly interacting with EJBs and Oracle 8i.

# 5 Evaluation

Once the model solution has been completed, it is the job of the engineer and the architect to define the *a posteriori* evaluation criteria. These criteria most often include all of the *a priori* criteria plus criteria that are discovered as a by-product of implementing the model solution.

Perhaps the biggest surprise encountered during the implementation of the model solution was the difficulty we encountered passing simple types between MicroFocus COBOL and Java. Most of these difficulties stemmed not from deficiencies in the COBOL compiler, but from shortcomings in the product documentation. Nevertheless, as a result of our experience, we added an *a posteriori* evaluation criterion to test Hypothesis #1 (The MicroFocus Net Express integrated development environment can be used to support mixed-language programming with Java.):

> *Criterion #3: Simple data types can be exchanged between MicroFocus COBOL and Java, and Java exceptions can be handled in the MicroFocus code.*

Now that we had completed the model problem and defined our *a posteriori* evaluation criteria, we could evaluate the model problem solution. Both Criterion #1 and Criterion #2 are easily satisfied by the solution. Criterion #3 was at least partially satisfied, in that we were able to communicate both integers and strings (these being the most critical data types) and provide exception handling in the COBOL code. As a result, the Net Express contingency of calling Java from MicroFocus COBOL was adopted as the primary contingency.

We also identified other areas of concern during the execution of the model problem, including performance and scalability, but as the model problem was not designed to evaluate these qualities, we did not add these as criteria for evaluating this model solution.

# 6 Legacy System Assumptions

The solution described in this report for maintaining transactions between the legacy and modernized systems assumes certain characteristics of the legacy system. These assumptions are documented in this section of this report. While this approach may not be the only solution for maintaining transactions, deviation away from these assumptions concerning the legacy system could invalidate the solution described in this report.

## 6.1 MicroFocus COBOL

MERANT supports mixed-language programming with Java through a Java domain in Object COBOL. The Java domain provides the capability to declare Java classes inside a COBOL program, as well as to send and receive messages from Java classes. The Java domain support works by creating a COBOL proxy object for each Java object, as shown in Figure 15.

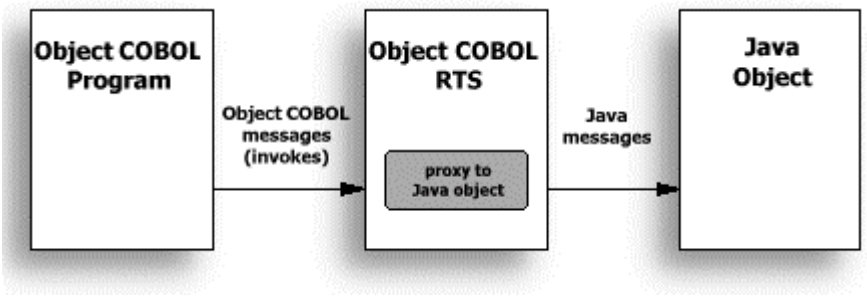



*Figure 15: Java Proxy*

The Java class itself, which is declared in the COBOL portion of the program, is a proxy for the static methods of the Java class. Mixed-language Java/COBOL programs can be either procedural or object-oriented COBOL, but they must conform to the following:

- The environment variable `cobjvm` must be set to the desired JVM.
- The system path must be set so that the `jvm.dll` can be located.
- The `mfcobol.jar` file must be included in the Java class path.
- The command `ooctrl(+p-f)` must be included in the COBOL portion of the program. This command adds type information to invoke statements, which the COBOL runtime system needs to convert data correctly between the COBOL and Java domains. This command also prevents the compiler from converting the method names that it invokes to lowercase (Java method names are case sensitive).
- Programs must be linked with the multi-threaded runtime system.

## 6.2 Java Transaction Service

The model problem implemented in this report uses Java Transaction Service APIs for programmatic transaction demarcation. Figure 16 shows a segment of Java code that creates a user transaction and then uses it to begin and commit a transaction. Operations on the database would normally be inserted between the calls to the `begin()` and `commit()` transaction methods.

```
import javax.transaction.*;

//transaction staff
UserTransaction ut = null;
ut = (UserTransaction)ivjInitContext.lookup("jta/usertransaction");

ut.begin();
  :
  :
  :
ut.commit();
```

*Figure 16: Transaction Demarcation Using JTS*

Most likely, Java utility methods to perform equivalent JTS calls will be created and invoked directly from the MicroFocus COBOL code.

## 6.3 JDBC

The model problem assumes the use of Oracle's JDBC thin driver, which is a Type 4 driver written completely in Java. This driver connects directly to Oracle using Java sockets without the need for a JDBC-specific middle tier and can only connect to a database if a Transparent Network Substrate (TNS) Listener[2] is up and listening on TCP/IP sockets. Type 4 drivers are typically database specific and provided only by the database vendor.

## 6.4 SQL

The model problem assumes the use of standard SQL.

[2] Transparent Network Substrate (TNS) Listener is a server process designated to listen for incoming connections to client applications using SQL*Net Version 2.

# 7 Summary and Conclusions

Model problems are an effective component-based software engineering technique for evaluating design contingencies. In this report, we developed a model problem to evaluate the feasibility of maintaining transactional integrity from COBOL to EJBs using MicroFocus COBOL, WebSphere, and Oracle 8i.[3] We were able to create and demonstrate a model problem that satisfied the *a posteriori* evaluation criteria for the model problem, increasing our confidence in the viability of the design option.

As a result of this model problem, the system architect identified this design solution as the principal design contingency. Becoming the principal design contingency does not guarantee that the solution will be adopted, but typically the principal contingency will receive the most resources to further verify the viability of the approach and reduce design risk. For example, we identified some additional risks during the implementation of the model solution, including how well the COBOL types defined for interacting with Java map into the COBOL types used in the RSS. In addition, we did little to verify the performance, robustness, and scalability of this approach. These attributes of the design solution must be considered further.

Although the principal contingency is not always adopted, it is an important step nonetheless. As a result of this decision, other design solutions are potentially starved of evaluation resources. If, for example, we had selected the OTS approach, we may never have applied the resources to identify the mixed-programming language approach as a viable design option. Time is also an important factor. As time passes, the design becomes more entrenched in the principal contingency as this design solution becomes a design constraint in other model problems and as engineering expertise is acquired in the requisite technologies. Eventually, the cost of replacing the principal design solution with a contingency becomes prohibitive.

[3] Information concerning some of the features of these products is provided in Appendix A.

# Appendix A

This appendix provides a brief description of the commercial software products used in the construction of the model problem.

## NetExpress MicroFocus

MicroFocus Net Express is an integrated development environment for developing procedural COBOL/Object COBOL-based applications. Net Express has built-in support for the following types of applications:

- mixed-language programming: provides support for procedural COBOL, Object COBOL, and Java mixed-language programming
- Web-based applications: includes a tool set to create, develop, build, and test Web applications and a personal Web Server
- Windows-based applications: includes support for the development of applications that use the Microsoft Windows user interface
- component-based/distributed applications: provides support for EJBs and Microsoft OLE automation products (COM/DCOM and ActiveX)
- Distributed Transaction Processing: supports EJBs, Microsoft Transaction Server, and WebSphere distributed-transaction technologies

## WebSphere Application Server

The WebSphere Application Server is an application server incorporating the following technologies:

- HTTP server that includes administration GUI and support for Lightweight Directory Access Protocol (LDAP) and Simple Network Management Protocol (SNMP)
- management and security controls for user-, group-, and method-level policy and control
- database access using JDBC for DB2 Universal Database and Oracle
- Java servlets, Java Server Pages, and XML for the display and construction of dynamic Web content
- EJBs server for implementing EJB components that incorporate business logic: allows the integration of EJB and CORBA components to business applications and includes full support for both Session Beans and Entity Beans (container-managed and bean-managed persistence)
- support for distributed transactions and transaction processing

## Oracle 8i Database

The Oracle 8i Database is a relational database. Here are just a few key features:

- JDBC and SQLJ for Java applications
- Oracle JServer: Java VM in the database
- data security
- object relational database support
- backup and recovery
- content management
- data warehousing
- transaction processing
- parallel-server, data management, and SQL
- national language support

# References

**[Dorda 00]** Comella-Dorda, Santiago; Wallnau, Kurt; Seacord, Robert C.; & Robert, John. *A Survey of Legacy System Modernization Approaches* (CMU/SEI-2000-TN-003, ADA377453). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. Available WWW: <URL: http://www.sei.cmu.edu/publications/documents/00.reports/00tn003.html>.

**[Wallnau 01]** Wallnau, Kurt; Hissam, Scott; & Seacord, Robert C. *Building Systems from Commercial Components* (ISBN: 0201700646). New York, NY: Addison-Wesley, 2001.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. AGENCY USE ONLY | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| (Leave Blank) | August 2001 | Final |

| 4. TITLE AND SUBTITLE | 5. FUNDING NUMBERS |
|---|---|
| Maintaining Transactional Context: A Model Problem | F19628-00-C-0003 |

6. AUTHOR(S)

D. Plakosh, S. Comella-Dorda, G. Lewis, P. Place, R. Seacord

| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) | 8. PERFORMING ORGANIZATION REPORT NUMBER |
|---|---|
| Software Engineering Institute<br>Carnegie Mellon University<br>Pittsburgh, PA 15213 | CMU/SEI-2001-TR-012 |

| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | 10. SPONSORING/MONITORING AGENCY REPORT NUMBER |
|---|---|
| HQ ESC/XPK<br>5 Eglin Street<br>Hanscom AFB, MA 01731-2116 | ESC-TR-2001-012 |

11. SUPPLEMENTARY NOTES

| 12A DISTRIBUTION/AVAILABILITY STATEMENT | 12B DISTRIBUTION CODE |
|---|---|
| Unclassified/Unlimited, DTIC, NTIS | |

13. ABSTRACT (MAXIMUM 200 WORDS)

Due to their size and complexity, modernizing enterprise systems often requires that new functionality be developed and deployed incrementally. As modernized functionality is deployed incrementally, transactions that were processed entirely in the legacy system may now be distributed across both legacy and modernized components.

In this report, we investigate the construction of adapters for a modernization effort that can maintain a transactional context between legacy and modernized components. One technique that is particularly useful in technology and product evaluations is the use of model problems—focused experimental prototypes that reveal technology/product capabilities, benefits, and limitations in well-bounded ways.

This report describes a model problem used to verify that such a mechanism exists and could be used to support the modernization of a legacy system. In this report, we describe a model problem constructed to verify the feasibility of building this mechanism. We also discuss the results of our investigation including the problems we encountered during the construction of the model problem and workarounds that were discovered.

| 14. SUBJECT TERMS | 15. NUMBER OF PAGES |
|---|---|
| modernization, legacy system, transaction, COBOL, EJB | 42 |

16. PRICE CODE

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| Unclassified | Unclassified | Unclassified | UL |