Sneed and Majnar, "A Case Study in Software Wrapping,"
Proc. of the International Conference on Software
Maintenance, IEEE COMPUTER Society Press, November 1998, pp. 86-93.

A Case Study in Software Wrapping

Harry M. Sneed / Rudolf Majnar
SES Software-Engineering Service GmbH
Email: Harry.Sneed@t-online.de

Background

Object-oriented technology has become the predominant software development paradigm of the 1990's. More and more data processing user departments have started projects to introduce the technology and more and more universities are now teaching it. The languages C++, SmallTalk and JAVA have become the languages of choice.[1] The COBOL standards committee has even proposed an object-oriented variant of COBOL and there are already three compilers for Object COBOL.[2] The explosion of internet and intranet applications together with the introduction of CORBA and ACTIVE-X as alternative means of coupling distributed objects have had a profound impact on business information systems. In Germany, there is hardly any major data processing user which is not in the process of introducing object technology.

One such data processing shop is the German Savings and Loan Organization (Sparkasse) which has established a common task force to support the member banks in migrating from their current conventional mainframe based centralized systems to more flexible object-oriented distributed systems. The task force is part of the Sparkassen Information Systems Company (SIZ) located in Bonn whose role is to coordinate the software development among the member banks. The member banks are regional financial institutions with their own computer centers and their own data processing staffs. There are currently seven such regional computing centers in Germany equipped with IBM-390 architectures, IMS, ADABAS or DB-2 databases, IMS-DC or CICS online teleprocessing monitors and the programming languages Assembler, PL/I, COBOL and NATURAL.

Migration Strategies

There are alternative strategies for introducing object technology into an existing data processing organization.[3] One is to start from scratch and redevelop all of the business applications without any consideration of the existing software. This can be done all at once - the big bang approach - or one application at a time - the incremental approach.[4] In both cases it is only necessary to migrate the data. Another strategy is to reuse programs of the existing systems by converting them to object-oriented programs such as PL/I in C++ or COBOL in OO-COBOL.[5] A third strategy is to wrap components of the existing software and to invoke them from the distributed environment. Thus, the three possible migration paths under consideration by the Sparkassen Organization are

- Redevelopment
- Transformation and
- Encapsulation

There is much to be said in favor of redevelopment. It gives the developers a free hand in coming up with an entirely new solution independent of the existing one. The only restriction is the current data which has to be retrofitted into the new objects. The disadvantage of this approach is the fact that every function has to be re-implemented and tested in a new language in a new environment. Not only is this costly, but it can take a long time. Besides that, there is also a greater risk involved that the application will fail to meet requirements.[6]

The transformation approach, i.e. reengineering, appears to be promising since it involves the reuse of code. However, there are also problems here. Much of the older code of the Sparkassen is in Assembler, making transformation extremely expensive. The PL/I and NATURAL code alsopresents a major challenge to conversion. A pilot project is underway to transform PL/I code into JAVA, but this is no easy task. The best case here is COBOL which can be transformed into the IBM Object COBOL language for MVS. There are also tools available to support this transformation under

VISUAL AGE. For COBOL applications the object-oriented recycling of programs is definitely an option. This option has been treated in a previous paper by this author.[7]

The encapsulation approach, i.e. wrapping, is another way to reuse existing software without moving it to the new environment. It remains with minor changes in its native environment and is made accessible to the other distributed software components in the new object-oriented environment which can invoke it via an application program interface (API). In this way, existing transactions, modules and procedures can be included as services in an object-oriented system.[8]

In support of the latter approach the Sparkassen Organization has sponsored a pilot project under the title "Encapsulation of existing Software Components" - EEC. The goal of this project is to explore and test means of encapsulating existing software in new distributed object-oriented application systems. This paper is a report on the experience made so far in that project.

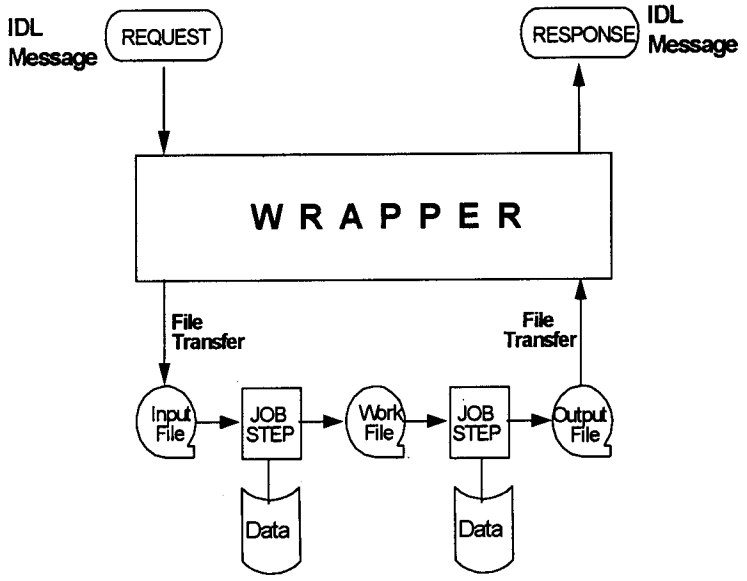
Levels of Encapsulation

The first decision to be made in any wrapping project is to decide upon the level of encapsulation. In the EEC project five levels of granularity were considered

- job encapsulation,
- transaction encapsulation,
- program encapsulation,
- module encapsulation and
- procedure encapsulation [9]

Job encapsulation entails the wrapping of batch type job control procedures which can be invoked from a remote location. The transaction files used in the batch job can also be provided via a data stream from the invoking program. The IBM-MQ Series file transfer product lends itself to this form of wrapping. (See Figure 1)

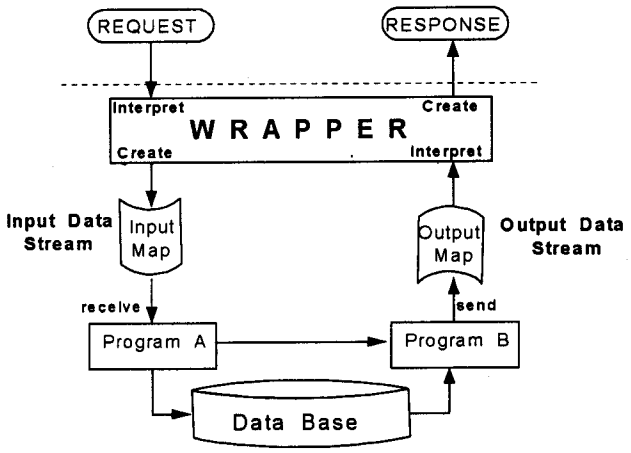
Figure 1: Job Encapsulation



Transaction encapsulation involves the invocation of an online transaction by passing a message from a distributed client. The host system receives the message via the wrapping shell and processes it as if it were coming from a host terminal. The output

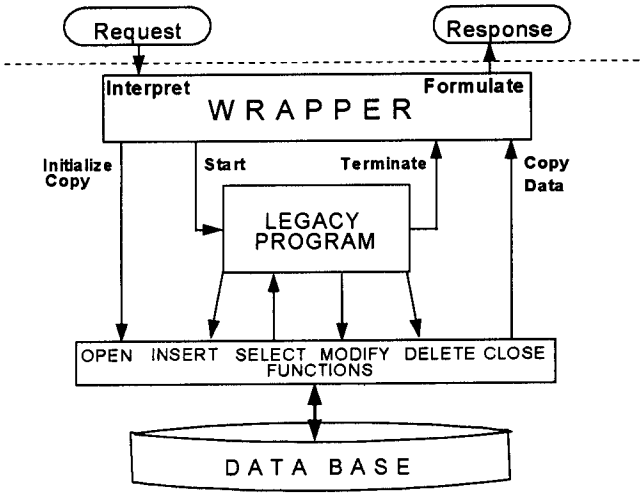
messages are collected and dispatched back to the client. In this way, legacy transactions can be nested in the new distributed transactions. This is now supported to some extent by the IBM CICS TP-Monitor. (See Figure 2)

Figure 2: Transaction Encapsulation



Program encapsulation occurs when a host batch program is invoked from a remote location. The program receives its parameters from the client and also one or more of its input files. The program processes its transactions as if it were running in a batch job except that one or more of its output files are routed back to the client. (See Figure 3)

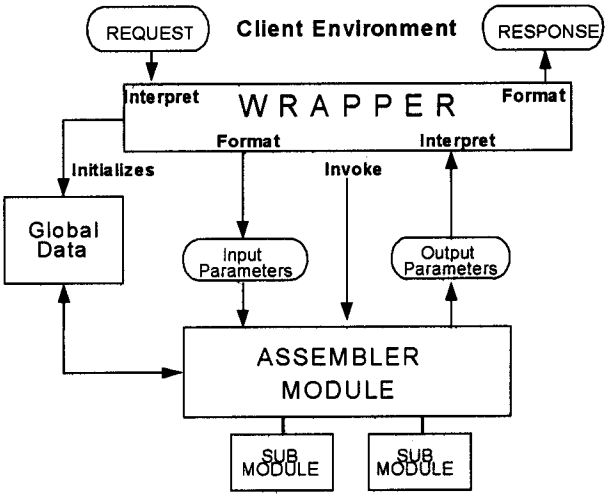
Figure 3: Program Encapsulation



Module encapsulation is the easiest to implement. In their native mode modules in a host language like COBOL or PL/I are called with a parameter list. They use the input parameters and set the output parameters.

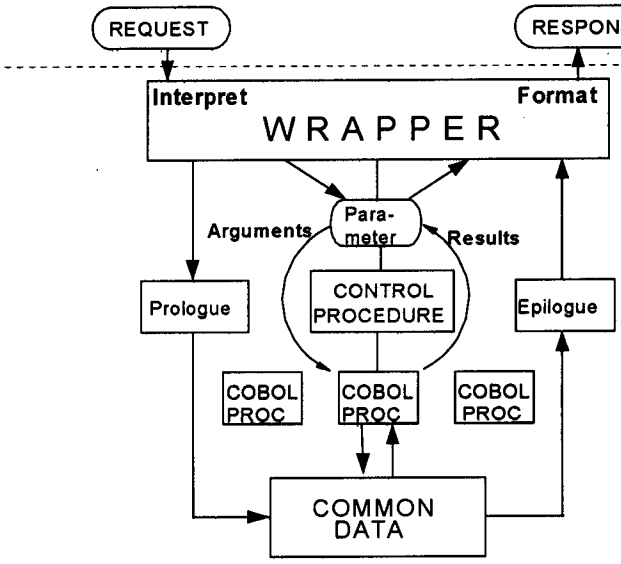
In wrapped mode, the only difference is that the module is called by the wrapper shell. (See Figure 4)

Figure 4: Module Encapsulation



Procedure encapsulation is the most difficult to implement since it entails dissecting a legacy program in order to selectively invoke individual procedures embedded in the program. Often, a program as a whole does not fit into the newly designed distributed business process. However, parts of it do. Therefore, the task is to dynamically include those parts within the work flow. Technically, this involves separating the reusable embedded procedures from the others and providing them with their own entry points and parameter lists. As such, they can be invoked like modules. (See Figure 5)

Figure 5: Procedure Encapsulation



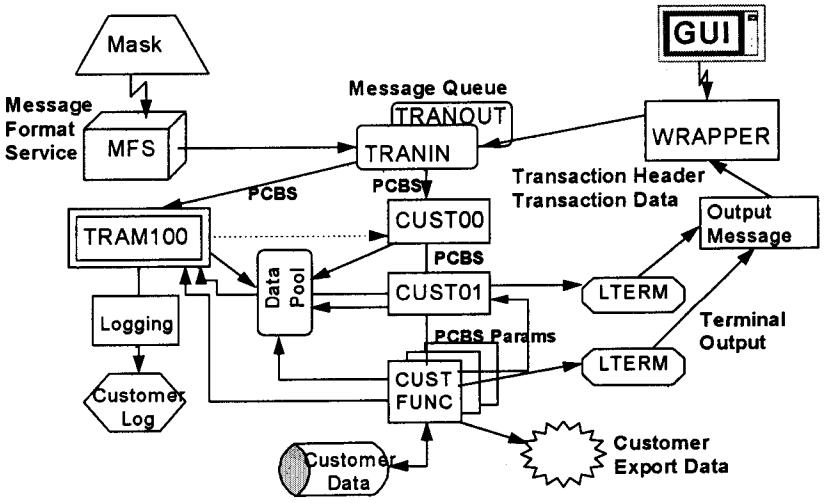
Pilot Project in Distributed Processing

For the pilot project, a host transaction was selected which opens a checking account for a bank customer. The process for enrolling a new customer has been removed from the IBM-Host and re-implemented on a UNIX server, using an ORB - Visibroker from Visigenic - to connect the interface applets written in JAVA executing on PC-Workstations under MS-Windows with the server objects implemented in C++. The server objects, in all 26 different data objects per customer combined to form a single unified business object - customer - were created from IMS database segments exported from the host at runtime.

The need for encapsulation grew out of the fact that the legacy host applications are highly integrated. The Sparkassen applications not only access the data of other applications directly but they also invoke transactions in neighboring applications in a multi threaded mode. Thus, when an application is distributed it still retains its ties to the other applications remaining on the host. In the case at hand, the customer enrollment application requires the account application to open a new checking account. Since the new customer enrollment application is now distributed, this entails a remote procedure call from the responsible Unix server back to the host. For this the remote calling facility of the IBM CICS transaction monitor had to be used.[10]

The host TP-Monitor receives the message from the Unix server and starts a communication procedure. The communication procedure picks up the appropriate IMS database pointers and calls the wrapper shell passing it the user message and the database pointers. It is the task of the wrapper software to build up the massage queue, to convert the incoming messages from the external IDL format to the internal Assembler format and to load and execute the host transaction program. After the transaction has been executed and the checking account opened, a confirmation is sent back to the server and the distributed process continued. Otherwise, an error message is returned and the distributed process rolled back. (See Figure 6)

Figure 6: Encapsulation of Customer Transaktions



Wrapper Implementation

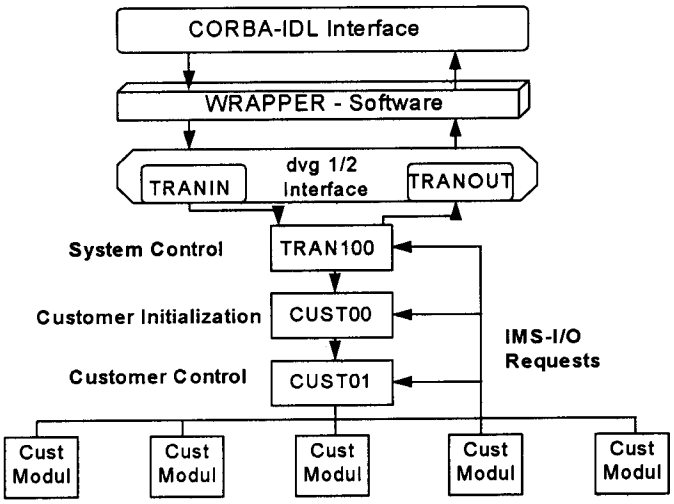
The wrapper software was implemented with two main procedures in MVS-C. In the first procedure - CWRAP - the input message queue is build up, the messages checked and converted to the internal format of the IMS Message Format Service, and the transaction control program called. In the second procedure - CSTUB - terminal input requests from the transaction program are intercepted and fulfilled by taking the next message from the wrapper input queue. This amounts to a

simulation of the host terminal input. CSTUB also intercepts the terminal output messages and routes them to the wrapper output queue, thus simulating the host terminal output.
When the host transaction program is finished, control is passed back to CWRAP which then converts the output messages into the external IDL format and returns them to the server.

In the study, it came out that wrapper shells always have to be customized to the particular terminal processing utility used on the host. This is also confirmed in the literature.[11]

The interface to the distributed system can be made uniform for all applications, but the interface to the host legacy program must be adapted to the requirements of that particular program. Thus, there is no such thing as a standard, generic wrapper. Wrappers are the hinge between the interfaces of the external distributed environment which of course can and should be standardized, and the interface of the internal legacy software which are by the nature of legacy systems individual. The interface conversion procedures will always be dependent on the existing program interfaces, however the other procedures of the wrapper as well as the overall architecture can be standardized in a common program framework. (See Figure 7)

Figure 7: Customer Wrapper Architecture



User Program Adaptation

Unfortunately, it is generally not possible to wrap programs without modifying them. If they are batch programs their input/output files have to be rerouted. If they are online programs their input/output maps have to be rerouted. The inputs are taken from the input message queue of the wrapper. The outputs are placed in the output queue. In the case of this pilot project, the user program was written in Assembler with DLI macros for accessing the IMS database as well as for accessing the user terminals. The database macros were scattered throughout the modules of the program, however, the terminal access macros were concentrated in the control module. Therefore, it was possible to restrict the adaptations to this module. Each DLI terminal operation was placed in comment and replaced by a link macro to CSTUB.

In addition, some of the other initialization routines of the control module had to be removed since these conflicted with the initialization procedures of the CICS communication shell. As it turned out most of the testing effort was devoted to correcting errors caused by the adaptation of the user transaction control module. This will probably always be the case when the user software is badly implemented and poorly documented. For this reason, it is well worth considering reengineering the

user programs before wrapping them. Not only will this reduce the wrapping effort, but it will also reduce the future maintenance effort. It should not be forgotten, that wrapped software also has to be maintained, both in its wrapped and its original form.[12]

Testing the wrapped transaction

The total effort for the pilot wrapping project involved no more than 40 person days. Of that 10 days were for analysis and design, 13 days for the implementation of the wrapper, 5 days for adapting the user program and 11 days for testing and debugging.

The testing of a wrapped transaction is no trivial matter, since the host program is only one link in a long chain which includes the client user interface programs, the server object management programs, the ORB, the host/server communication link and the wrapper. The recommended strategy and the one followed here is to test bottom-up in five steps.[9]

The first test should be of the adapted user program in a test frame where the interfaces can be manipulated and controlled. The second test should be of the wrapper software alone using a test driver to simulate the server and a test stub to simulate the wrapped user program. The third test should be of the wrapper software together with the wrapped program, both of which run under a test driver in batch mode. The fourth test should be of the whole transaction starting with the client program linked to the server and the server sending a message to the host communication link which invokes the wrapper. This integration test confirms the links between the remote programs and ensures that the interactions are performed correctly. The fifth and final test is a systems test of several transactions being transmitted one after another to validate the reentrancy of the wrapper and the wrapped software.

Lessons Learned

The purpose of this pilot project was to demonstrate that legacy assembler programs executing online transactions can be wrapped and accessed by distributed objects. This goal was achieved. However, in achieving the goal three important lessons were learned.

First, it became obvious that in all cases except for existing subprograms with a coherent calling interface, the user programs to be wrapped have to be adapted. This entails either preprocessing the target programs after each new version release, inserting extra conditions in the code, or maintaining a separate copy for wrapping purposes.

Secondly, it became clear that the weakest link in the chain is the server to host communication. This is an inherent bottleneck due to the weakness of current file transfer utilities and to the necessity of character conversion from ASCII to EBCDIC. This also requires keeping the user messages in character mode so that they can be converted easily.

Thirdly, it was learned how time consuming it is to test distributed systems with a three tier architecture. There are many dependencies in the long chain of software components as well as dependencies on the hardware. If any one component fails the other components cannot be tested. There is still much to be done to improve the testability of three tier architectures. Nevertheless the efforts of the OMG with the CORBA standardization as well as the investment made by Microsoft in the Active-X development indicate that this will probably be the future computing architecture.[13]

References

[1] Yourdon, Ed.: "Language Wars" in Application Development Strategies, Vol. 9, No. 8, Cutter Information Corp., August, 1997

[2] MacFarland, D.: "The Future of COBOL" in American Programmer, Vol. 9, No. 9, Sept. 1996, p. 2

[3] Mattison, R.: The object-oriented Enterprise, McGraw-Hill, New York, 1994, p. 351

[4] Brodie, M./Stonebraker, M.: Migrating Legacy Systems, Morgan Kaufmann Pub., San Francisco, 1995, p. 41

[5] Sneed, H.: "Migration of procedurally oriented COBOL Programs in an objectoriented Architecture" in Proc. of 8th JCSM, IEEE Press, Orlando, 1992, p. 105

[6] Ewusi-Mensah, K.: "Critical Issues in Abandoned Information Systems Development Projects" in Comm. of ACM, Vol. 40, No. 9, Sept. 1997, p.75

[7] Sneed, H.: "Object-oriented COBOL Recycling" in Proc. of 3rd WCRE, IEEE Computer Society, Monterey, 1996, p. 169

[8] Gossam, S.: "Accessing Legacy Systems" in Object-Expert, March 1997, p. 58

[9] Sneed, H.: "Encapsulation Legacy Software for Use in Client/Server Systems" in Proc. of 3rd WCRE, IEEE Computer Society, Monterey, 1996, p. 104

[10] Crownhart, B.: IBM's Workstation CICS, McGraw-Hill, New York, p. 141

[11] Mowbray, T./Zahavi, R.: The Essential CORBA, John Wiley & Sons, New York, p. 231

[12] Etzkom, L./Davis, C.: "Automatically identifying Reusable OO-Legacy Code" in IEEE Computer, Oct. 1997, p. 66

[13] Fayad, M.E./Schmidt, D.C.: "Object-oriented Application Frameworks" in Comm. of ACM, Vol. 40, No. 10, Oct. 1997, p. 32