Encapsulation of legacy software: A technique for reusing legacy software components

Article *in* Annals of Software Engineering · May 2000

DOI: 10.1023/A:1018989111417 · Source: DBLP

Harry Sneed
ANECON GmbH
220 PUBLICATIONS **2,590** CITATIONS

Annals of Software Engineering 9 (2000) 293–313

Encapsulation of legacy software: A technique for reusing legacy software components

Harry M. Sneed
Prellerweg 5, D-82054 Arget, Bavaria, Germany
E-mail: Harry.Sneed@T-Online.de

The following paper reviews the possibilities of encapsulating existing legacy software for reuse in new distributed architectures. It suggests wrapping as an alternative strategy to reengineering and redevelopment. It then defines the levels of granularity at which software can be encapsulated before going on to describe how to construct a wrapper and how to adapt host programs for wrapping. Some wrapping products are discussed and the state of the art summarized. The advantage of wrapping over conventional reengineering is the low cost and even lower risks involved. This is the driving force in the search for improved wrapping technology.

1. Motivation for encapsulation

Legacy software components can be jobs, transactions, programs, modules or procedures within existing application systems which are more than five years old. Usually these systems are running on a mainframe and are based on an outdated technology such as hierarchical or networked database systems and transaction-oriented teleprocessing monitors with fixed panels. Although the technology with which they have been implemented is out of fashion, the application systems themselves are performing critical business functions in an acceptable and reliable manner. It took a lot of time and effort to develop these systems and specially to test them. Therefore, it is irrational to discard them completely, just because they do not meet current technical requirements. Typical candidates for encapsulation as business objects are batch jobs, online transactions and user subroutines as depicted below.

On the other hand, there is a definite need to reengineer existing systems in order to satisfy the needs of modern business practices. It is not possible to restructure business processes without restructuring the computer process behind them [Taylor 1995]. If business processes are distributed to ensure more flexibility and to delegate responsibility then the supporting computer systems must also be broken up and distributed. Very often this leads to a new hardware architecture with client and server computers as well as a central host computer. On the software side it leads to a new user interface with a different work flow control and to a distributed relational or object-oriented database system, i.e., the underlying hardware is replaced as well as the software frontend and backend [Brodie and Stonebraker 1995].

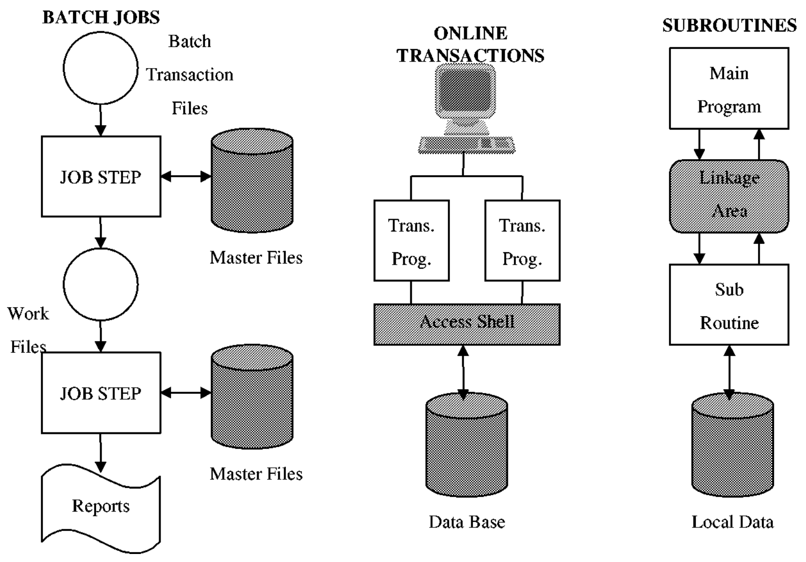


Figure 1. Legacy software components.

The question which comes up here is whether it is necessary to replace all of the existing software. Some will say yes, if you are going to replace the user interface and the database system, you might as well replace it all. Others will say no, you should keep as much of the existing software as possible and to reuse it in the new environment. The correct answer depends on the quantity, complexity and quality of the existing software as well as on the time and resources of the user. If the user has tight time or budget constraints, he may have no other choice but to reuse the existing programs. On the other hand, if the business logic of the old software is only a small layer of code between the presentation and access logic, then it might as well be replaced together with the presentation and access layers. So there is no cut and dried answer to this question. It is context dependent [Mattison 1994]. In practice, it depends on the ratio of business logic relative to the presentation and access logic.

In order to justify the effort involved in reusing the existing programs, their business logic functions must be a significant part, i.e., there must be enough business logic available to warrant reuse. In addition, the programs must meet a minimum quality standard as far as reliability, efficiency and maintainability are concerned. Finally, the programs should be complex enough so that rewriting them will be no simple task. If these conditions are all met, then it is definitely advisable to reuse them. Even after the decision has been made to reuse legacy software there is still the question as to how to reuse it. There are two basic alternatives [Sneed 1998]:

– conversion, and
– encapsulation.

Conversion entails porting the code from the old environment to the new one and may entail the translation from one language to another. The existing code for the

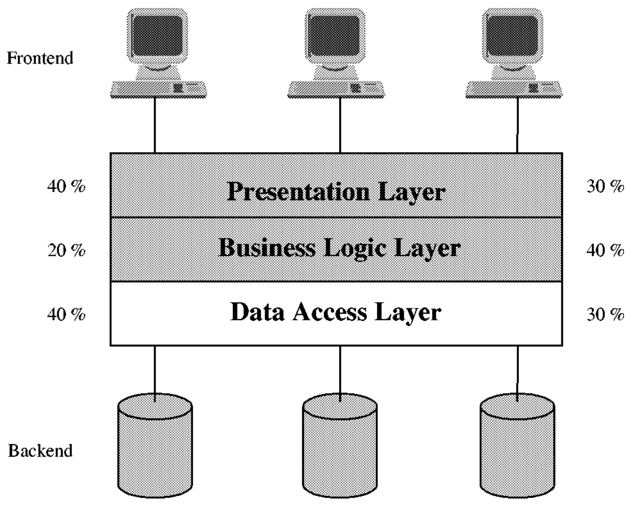


Figure 2. Legacy software layers.

business logic is integrated statically with the new code for the presentation and access logic. This may appear to be the cleanest approach, but it is the most costly one and the one with the greatest risk. If the old code has to be significantly reworked, it will become more like a new development, only with other constraints.

Encapsulation means leaving the code in its current environment and connecting it dynamically to the new presentation and access layers. Here too some changes will have to be made to the code but they are minimal. The interfaces have to be reengineered, otherwise, the code remains as it was. The main costs involved are in connecting the distributed software. Since so little is changed, there is hardly any risk. Thus, though it may not be the cleanest solution, encapsulation is definitely the most economical one, at least in the short range. The conversion alternative has been handled in other papers by this author as well as by many others. The focus here is on the encapsulation alternative which is becoming increasingly popular in industry [Jacobson 1992].

2. Wrapping technology

Software encapsulation is based on the technology of wrapping. The notion of a “wrapper” was first introduced by Thomas Dietrich of IBM at the 00PSLA Conference in 1988. When asked what to do with the existing legacy software, in a new object-oriented architecture, his answer was to wrap it [Dietrich 1989]. Since then there have been dozens of papers written on the subject in the object-oriented literature, but hardly any in the field of reverse- and re-engineering [Yourdon 1997]. One of the best technical discussions of the subject is to be found in the book “The Essential CORBA” by Mowbray and Zahari. According to these authors, an object wrapper provides access to a legacy system through an encapsulation layer. The encapsulation

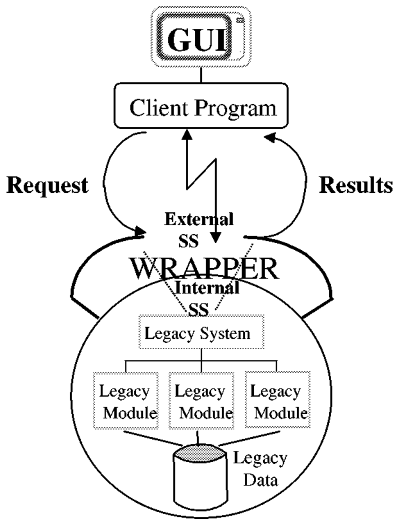


Figure 3. Wrappers as gateways between client and server programs.

exposes only the attributes and operations desired by the software architect. The same authors go on to describe seven techniques for implementing a wrapper:

- remote procedure calls,
- file transfers,
- sockets or docking,
- application program interfaces,
- script procedures,
- macros, and
- common headers.

These techniques can be implemented independently or in combination with one another to build a connection between the requester of a service and the service provider [Mowbray and Zahavi 1994].

In his book "Migration to Object Technology" Ian Graham defines a wrapper as a software controller layer which allows object-oriented programs to access conventional ones as if they were objects. The wrapping of existing software components plays a major role in Graham's SOMA – Semantic Object Modeling Architecture. However, Graham warns that "implementing wrappers is not as easy as it may sound." The wrapper software has to dynamically adapt the incoming request to the interfaces of the wrapped software primarily because of data type incompatibility. This is really a case of down casting. Old mainframe programs use packed and hexadecimal data types as well as special map constructs such as PF keys and map field attributes which cannot be represented in modern client languages [Graham 1995].

In an article in Object Expert Gossain [1997] describes five alternatives to wrapping legacy applications for reuse in a client/server architecture:

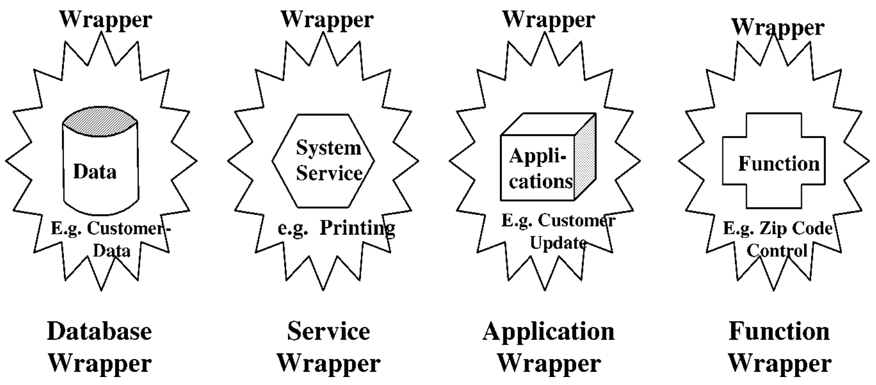


Figure 4. Types of software – wrappers.

– remote procedure calls,
– application program interfaces,
– teleprocessing transactions,
– file transfers, and
– data tables.

Gossain differentiates between wrappers with semantic content and wrappers without. Wrappers with semantic content perform some kind of data conversion. As a framework for implementing wrappers Gossain suggests using Gammas' bridge pattern [Gamma 1995].

In general, four kinds of wrappers have been identified in the literature depending on what is being wrapped [Orfali *et al.* 1996]:

- database wrappers,
- system services wrappers,
- application wrappers,
- function wrappers.

Database wrappers are gateways to existing databases. They allow client applications implemented in a modern object-oriented language to access data stored in a legacy database.

System service wrappers provide a customized access to standard system services such as printing, sorting, routing and queuing. It is possible for a user program to invoke such services without knowledge of their internal interfaces.

Application wrappers encapsulate batch processes or online transactions. They allow new client applications to include the legacy components as objects which can be called to perform certain tasks such as producing a report or updating a file.

Function wrappers offer an interface to invoke individual functions within a wrapped program. Not the program as a whole but only certain parts of it can be invoked from the client application. This amounts to a limited access.

All wrapper use some kind of message passing mechanism to connect themselves to their clients. As a rule the wrapper is in the same address space as the wrapped object. On the input side, it receives incoming requests, reformats them, loads the object and invokes it with the reformatted arguments. On the output side, it takes the results from the wrapped object, reformats them and sends them back to the requester. This is, in essence, what wrapping is all about.

3. Levels of encapsulation

Legacy databases can be accessed at that level at which the database system operates. If one is dealing with file systems one will get records. If one is dealing with hierarchical systems like IMS, one will get segments and with network systems like IDMS records. Only when dealing with relational databases or partial relational databases like ADABAS is it possible to access individual data attributes.

Legacy software has many levels of granularity. A typical legacy application will consist of batch processes and/or online-transactions. Both batch processes and online-transactions include one or more programs. Programs are made up of one or more statically or dynamically linked modules. These separately compilable units will consist of one or more internal procedures. The procedures have a unique name and are made up of a number of instructions. They may also have their own entry point with parameters.

Since it does not make sense to wrap single instructions, the lowest level of granularity is the procedure and the highest level is the application. However, since applications are generally just logical collections of processes performed at different time intervals, the real highest practical level of granularity is the process. Thus we can conclude that there are in effect five levels at which one can access legacy software applications [Sneed 1996]:

– at the process level,
– at the transaction level,
– at the program level,
– at the module level, and
– at the procedural level.

At the *process level* a job is started after having allocated and filled its input files via file transfers from the client application. To this end a job control or command procedure is generated by the wrapper. When the job is finished, the output files are taken over by the wrapper and forwarded back to the client. This is the simplest form of encapsulation.

At the *transaction level* requests from the client application are converted into transactions of a teleprocessing monitor. However, instead of receiving panels from a terminal the transaction processing programs are fed a data stream from the client

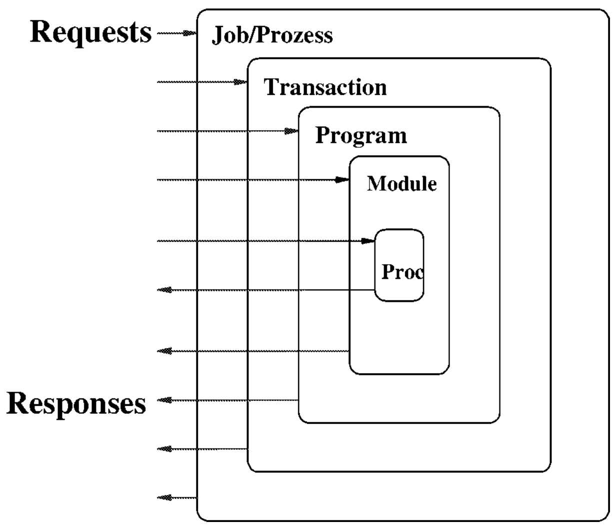


Figure 5. Levels of encapsulation.

application. The output messages produced by the transaction programs have to be intercepted by the wrapper and redirected back to the client application rather than going to an end terminal. This form of wrapping has become relatively simple since modern transaction processors have solved the problem of host to client communications and taken care of exception handling as well as the tasks of restart and recovery, commit and rollback.

At the *program level* batch programs are called via an application program interface. The inputs which drive the program logic are simulated by a wrapper which substitutes them with data streams from the client application. Program outputs are also captured and returned to the client. These could be files, displays or reports.

At the *module level* existing modules are loaded and executed using their standard linkage interface. The only difference to the normal call is that the parameters are passed by value rather than by reference, i.e., the parameter values are stored in the address space of the wrapper after having been received from the client. Afterwards, the output parameter values are forwarded back to the client.

At the *procedure level* an internal procedure is invoked as if it were a separately compiled module. This entails building a parameter interface and possibly setting global values prior to calling the procedure. This can become the most challenging form of wrapping.

4. Constructing a wrapper

Wrapping legacy software is normally done in three steps:

– first, the wrapper should be constructed,

– secondly, the target programs should be adapted,

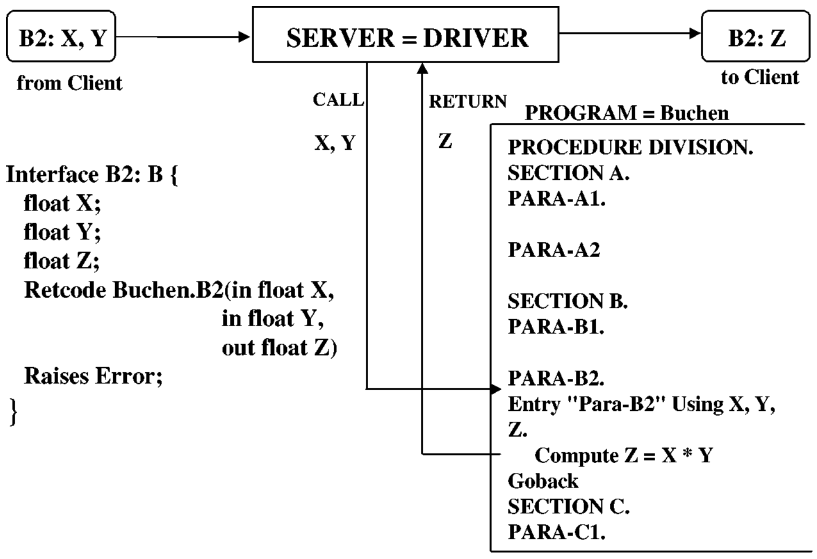


Figure 6. Procedural wrapping.

– thirdly, the interaction between the wrapper and the target programs should be tested.

The first step then is to construct a wrapper, if there is none already available. A wrapper is a software shell between the middleware software, e.g., ORB, DCOM, TP-Monitor, and the user software, e.g., job, program, module or procedure. It receives messages from the client application, converts them to an internal format and invokes the target software. It also intercepts the outputs of the target software, converts them to an external format and sends them back to the client application. As such the wrapper itself will consist of several event driven modules and two interfaces. The interfaces are:

– the external public interface, and
– the internal private interface.

The modules are [Winsberg 1995]:

– the control module,
– the message handling module,
– the interface conversion module,
– the input/output emulation module, and
– the exception handling module.

4.1. External interface

The external interface is the interface used by the client. It will normally consist of a message header and a body. The header will contain control information such

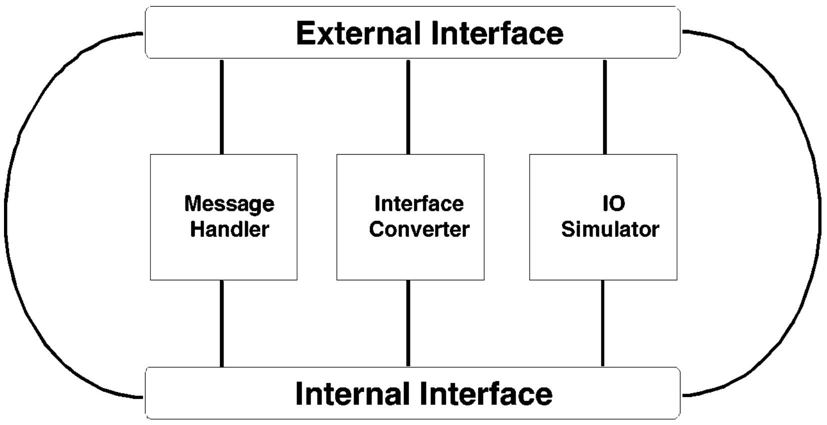


Figure 7. Modules of a wrapping-framework.

```
#ifndef SIZSS_IDL
#define SIZSS_IDL
 module SIZSS
 {
  interface SIZ_Interface: SOM_Objects
  {
struct User_Message
   {
struct Message_Header    // FESTFORMATIERTER KOP
     {
          long      Message_Id; // NACHRICHTENKENNZEICHEN
          long        Message_Lng; //  NACHRICHTENLAENGE
          char         Message_Type;  // NACHRICHTENTYP
        char       Terminal_Type;  // TERMINALTYP
          char         Message_Time[6]; // NACHRICHTENZEIT (HHMMSS)

          char         Terminal_Id[8];  // TERMINALKENNZEICHEN
          char         User_Id[8]; // NACHRICHTENSENDE
          char         Tran_Code[8]; // TRANSAKTION-/PROGRAMMNAME
          char         Method-Id[32];  // METHODENKENNZEICHE
          char         Func_Code[4];  // FUNKTIONSTYP (CICS/IMS)
          char         Func_Key[2];  // FUNKTIONSTASTE (CICS)
          char         Ret_Code[2];  // FUNKTIONSSTATUS (CICS/IMS)
          short      Feld_Nr;  // FELDANZAHL
          char         User_Bytes[8]; // RESERVIERT FÜR ANWENDE
     }; // End of Message_Header
          struct Message_Body   // FREIFORMATIERTER RUMPF
     {
           long            String_Lng; // ZEICHENFOLGELAENGE
          string           User_Data[String_Lng]; // ZEICHENFOLGE
          //       string   :=     {<Feldkz> = <Feldwert> \}
          //                                            (Feldanzahl)
          //       Feldkz   :=     char[*]
          //       Feldwert  :=    'String' / Decimal Value
     }; // End of Message_Bod
    }; // End of User_Message

  }; // End of Interface SIZ_Interface
 } // End of Module SIZSS
#endif
```

Figure 8. Sample of an external interface.

as the sender identification, the transaction code, the date and time, the type of target software and the name of the job, program, transaction, module or procedure to be invoked. The body will contain the arguments in the case of an input message and the results in the case of an output message. These parameters are usually ASCII character, strings separated by some kind of delimiter such as a back slash. In a CORBA or DCOM environment this interface will be specified with the IDL language.

```
WRAP ********** Generated Copy Member ************
WRAP  01 xm059-PARAMETER.
WRAP     02 xm059-P1.
WRAP       03 xm059-P1-TT PIC 99.
WRAP       03 xm059-FILLER PIC X.
WRAP       03 xm059-P1-MM PIC 99.
WRAP       03 xm059-FILLER PIC X.
WRAP       03 xm059-P1-JJ PIC 99.
WRAP       03 xm059-FILLER PIC X.
WRAP     02 xm059-P2.
WRAP       03 xm059-LANG-CODE PIC 9.
WRAP     02 xm059-P3.
WRAP       03 xm059-DIRECTION PIC X.
WRAP     02 xm059-P4.
WRAP       03 xm059-DAY-NAME PIC X(10).
WRAP ***** End of Generated Copy Member ************
```

Figure 9. Sample of an internal interface.

4.2. *Internal interface*

The internal interface is the interface known to the server software. Therefore, it is also dependent on the type and language of the encapsulated artifact. If the artifact is a job, the internal interface will be a job control procedure which can be interpreted to execute that job. If the artifact is a transaction, a program, a module or a procedure, the internal interface will be a parameter list in the language of the target software constructed according to the conventions of the target software, e.g., in the case of a COBOL program it will be a linkage section using call by reference. Normally, the internal interface will be extracted from the target source as a COPY or INCLUDE member. The parameters will be data types of the target language, so that a conversion from the incoming ASCII strings will be unavoidable.

4.3. *Message handler*

The message handler is responsible for queueing the incoming and outgoing messages. Since requests may arrive faster than they can be processed, it will be necessary to place them in an input queue from whence they are then processed. On the other side, outputs from the wrapped software, such as records in a file, may be generated faster than they can be sent back to the client. So the outputs must also be queued. for this purpose the wrapper will need to maintain a variable size buffer.

4.4. *Interface converter*

The interface converter is charged with the task of converting the external interface into the internal one and vice versa. For this it must be customized. In simple cases there will be a 1 : 1 relationship between the parameters of the two interfaces. In more difficult cases there may be a 1 : n relationship between the interfaces, meaning that the converter will have to duplicate parameters. In the most difficult cases, there can be an $m : n$ relationship between the two interfaces, causing the converter to have to associate values.

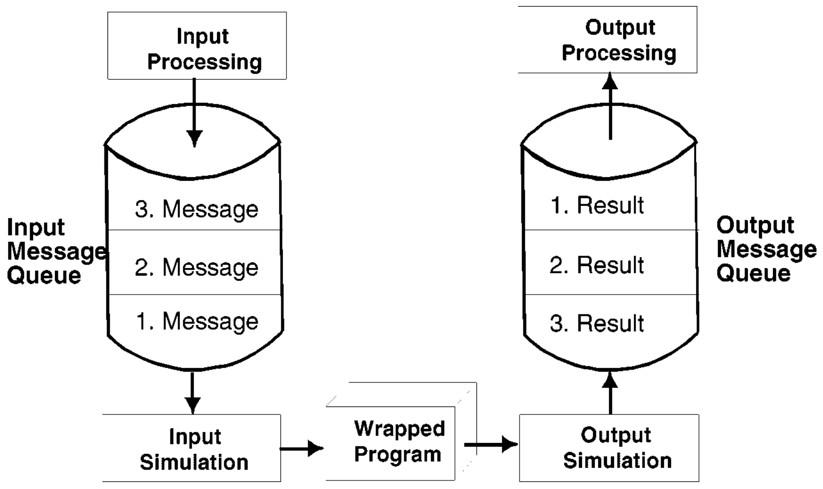


Figure 10. Structure of the message-handler.

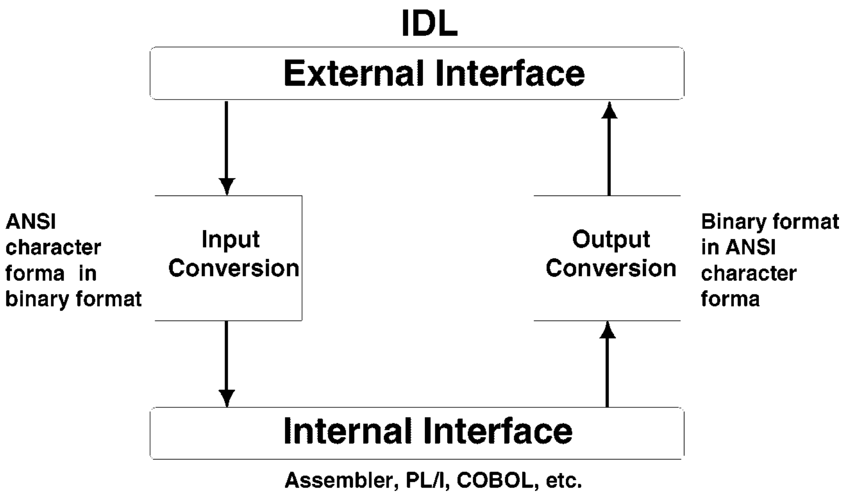

IDL
External Interface
ANSI character forma in binary format
Input Conversion
Output Conversion
Binary format in ANSI character forma
Internal Interface
Assembler, PL/I, COBOL, etc.

Figure 11. Structure of the interface-converter.

4.5. *IO-emulator*

The IO-emulator intercepts the input/output operations of the wrapped object. In case of an input operation, it fills the input buffer with the parameters of the external interface. In case of an output operation, it copies the content of the output buffer into the parameters of the external interface. As such, the IO-emulator emulates the input/output functions of the original environment in such a way that the encapsulated software is not affected. It should be capable of emulating both files and teleprocessing maps, so that the wrapped program receives its input data from the input buffer of the wrapper rather than from the file or teleprocessing panel and writes its output data

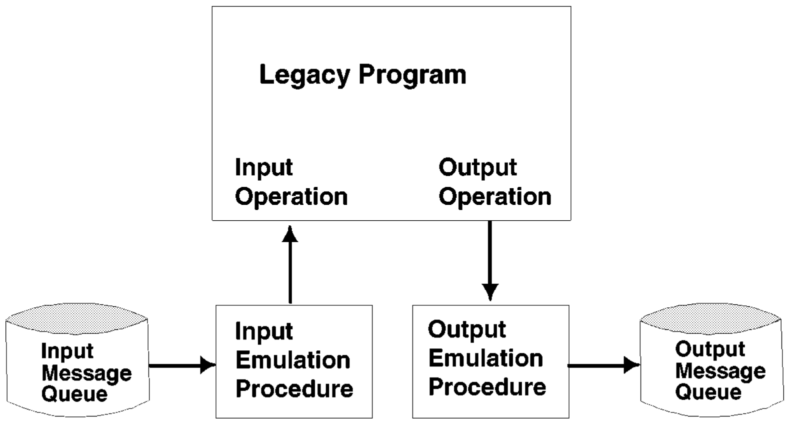


Figure 12. Structure of the IO-emulator.

onto the output buffer of the wrapper rather than into a file or onto a panel. The IO-emulator is highly dependent on the type of program being wrapped. It is one thing to emulate a sequential file input, it is something else to emulate an IMS input message. The developer of the IO-emulator must be very familiar with the product being emulated.

5. Adapting a program for wrapping

It would be ideal if programs could be encapsulated without making any changes to them whatsoever. However, even in the case of module wrapping some change has to be made. Since the programs to be wrapped should also continue to operate in the normal mode, it is unlikely that they can be adapted manually. The risk of error is too high and the adaptation has to be repeated after every change to the program. So if wrapping is to be done on a wide scale, the program adaptation has to be automated. This will require four tools:

– a transaction wrapper,
– a program wrapper,
– a module wrapper, and
– a procedure wrapper.

5.1. Transaction wrapper

The transaction wrapper processes online transaction processing programs to convert the panel input and output operations into calls to the wrapper. For instance, in CICS, the RECEIVE MAP macro has to be replaced by a CALL “MSGINP” USING MAP. The same applies to the SEND MAP macro. In fact, by replacing all of the

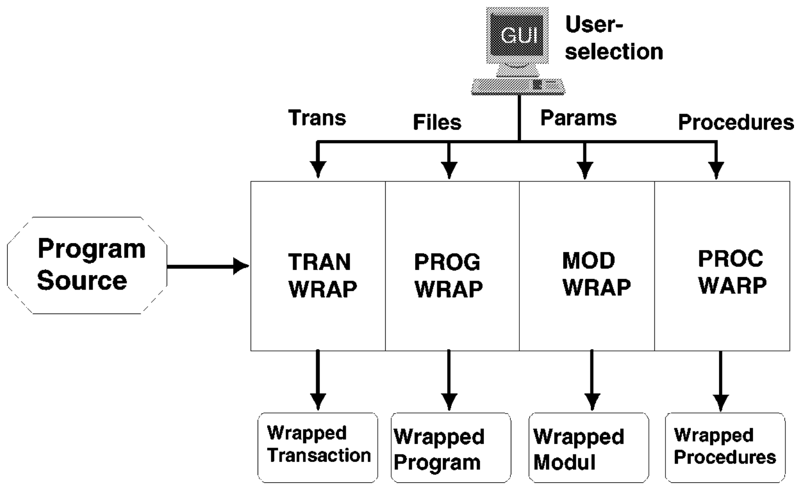


Figure 13. Software program adapter.

```
           RE-100.
WRAP  *     EXEC CICS RECEIVE MAP    ('DBRIM7F')
WRAP  *                   MAPSET ('DBRIS7')
WRAP  *                   INTO   (DBRIM7DI)
WRAP  *     END-EXEC.
WRAP        MOVE 'RC' TO X-CICS-FUNCTION
WRAP        MOVE DBRIM7DI TO X-CICS-MAP
WRAP        EXEC CICS LINK PROGRAM ('XTPINP')
WRAP                     COMMAREA (X-CICS-PARAM)
WRAP                       LENGTH (X-CICS-PARAM-LNG
WRAP        END-EXEC.
WRAP        MOVE X-CICS-RETCODE TO EIBRES
WRAP        EVALUATE TRUE
WRAP          WHEN X-MAPFAIL
WRAP          GO TO VV-860
WRAP          WHEN X-PF1
WRAP          GO TO VV-710
WRAP          WHEN X-PF2
WRAP          GO TO VV-720
WRAP          WHEN X-PF3
WRAP          GO TO VV-730
WRAP          WHEN X-PF10
WRAP          GO TO VV-800
WRAP          WHEN X-PF11
WRAP          GO TO VV-810
WRAP          WHEN X-PF12
WRAP          GO TO VV-820
WRAP          WHEN X-CLEAR
WRAP          GO TO VV-840
WRAP          WHEN X-ANYKEY
WRAP          GO TO VV-850
WRAP        END-EVALUATE.
```

Figure 14. Transaction wrapping.

CICS macros by CALL statements, the program can become entirely independent of the CICS environment.

There are some difficulties involved here such as the simulation of the program function keys and the CICS response code, but there are no problems which cannot be solved by a tool. Both CICS and IMS/DC programs can be wrapped by replacing their special preprocessing macros without affecting the program logic.

```
             MOVE 1 TO PAGE-COUNT.
             MOVE PAGE-COUNT TO PAGE-COUNT-NO.
             MOVE HEADER-PRT-LINE TO PRT-LINE.
WRAP         CALL ``XFILOUT`` USING PRT-LINE.
WRAP   *     WRITE PRT-LINE AFTER ADVANCING  PAGE.
       *
       * Read orders until end of order-file
            READ-ORDERS.
WRAP         CALL ``XFILIN`` USING X-RETCODE, ORDER-REC.
WRAP         IF X-RETCODE NOT = ZERO
WRAP             GO TO TERMINATION.
WRAP   *     READ ORDER-FILE
WRAP   *         AT END GO TO TERMINATION.
       *
       * Read Custome -data with key = cust-no
           READ-CUSTOMER.
             MOVE ZERO TO ERROR-TYPE.
WRAP         CALL ``XFILIN`` USING X-RETCODE, CUSTOMER-REC.
WRAP         IF X-RETCODE NOT = ZER
WRAP             MOVE 1 TO ERROR-TYPE
WRAP             GO TO REPORT-ERROR.
       *     READ CUSTOMER-FIL
       *         INVALID KEY MOVE 1 TO ERROR-TYP
       *         GO TO REPORT-ERROR.
```

Figure 15. Program wrapping.

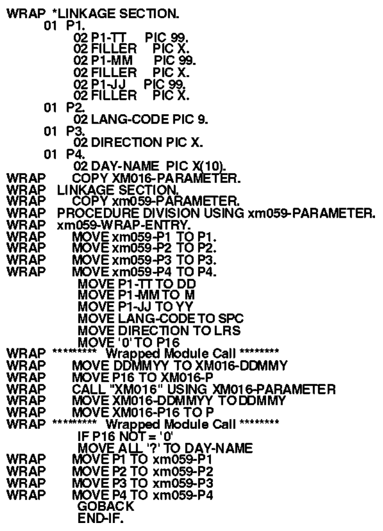
5.2. *Program wrapper*

The program wrapper processes batch processing programs by converting file operations into calls to the wrapper. Here the READ and WRITE statements become CALL statements using the record buffer as a parameter. Instead of receiving a record from the file, the record is provided by the wrapper and output records are rerouted to the wrapper. Therefore, it is also unnecessary to open and close the files.

The only problem here is in handling the exception conditions set by the IO-system. End of file and key errors must be simulated by return codes. Otherwise, the program remains unchanged.

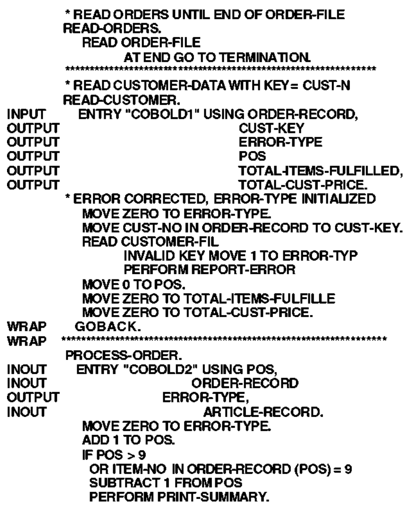
5.3. *Module wrapper*

The module wrapper may be required to adapt the parameter interfaces of a called subprogram. Such subprograms are normally called by reference, meaning that their parameters are addresses pointed to somewhere in the address space of the run unit. If the parameters are coming from a client application on another computer, this could present a problem since they are coming in as a contiguous data string. The solution is to alter the interface of the module being wrapped in such a way that all of the parameters being processed are grouped together in one structure and to pass the address of that structure as a pointer pointing to a message buffer in the wrapper. This entails an adaptation of the program entry statement and the creation of a new data definition in the linkage area [Sneed 1997a,b].



```
WRAP  *LINKAGE SECTION.
       01  P1.
             02 P1-TT     PIC 99.
             02 FILLER    PIC X.
             02 P1-MM      PIC 99.
             02 FILLER    PIC X.
             02 P1-JJ     PIC 99.
             02 FILLER    PIC X.
       01  P2.
             02 LANG-CODE PIC 9.
       01  P3.
             02 DIRECTION PIC X.
       01  P4.
             02 DAY-NAME  PIC X(10).
WRAP       COPY XM016-PARAMETER.
WRAP   LINKAGE SECTION.
WRAP       COPY xm059-PARAMETER.
WRAP   PROCEDURE DIVISION USING xm059-PARAMETER.
WRAP   xm059-WRAP-ENTRY.
WRAP       MOVE xm059-P1  TO P1.
WRAP       MOVE xm059-P2  TO P2.
WRAP       MOVE xm059-P3  TO P3.
WRAP       MOVE xm059-P4  TO P4.
             MOVE P1-TT TO DD
             MOVE P1-MM TO  M
             MOVE P1-JJ TO YY
             MOVE LANG-CODE TO SPC
             MOVE DIRECTION TO LRS
             MOVE '0' TO P16
WRAP  *********   Wrapped Module Call ********
WRAP       MOVE DDMMYY  TO XM016-DDMMY
WRAP       MOVE P16 TO XM016-P
WRAP       CALL "XM016" USING XM016-PARAMETER
WRAP       MOVE XM016-DDMMYY  TO DDMMY
WRAP       MOVE XM016-P16 TO P
WRAP  *********   Wrapped Module Call ********
             IF P16 NOT = '0'
             MOVE ALL '?' TO DAY-NAME
WRAP       MOVE P1 TO xm059-P1
WRAP       MOVE P2 TO xm059-P2
WRAP       MOVE P3 TO xm059-P3
WRAP       MOVE P4 TO xm059-P4
             GOBACK
             END-IF.
```

Figure 16. Module wrapping.



```
          * READ ORDERS UNTIL END OF ORDER-FILE
          READ-ORDERS.
              READ ORDER-FILE
                       AT END GO TO TERMINATION.
          ****************************************************************
          * READ CUSTOMER-DATA WITH KEY =  CUST-N
          READ-CUSTOMER.
INPUT         ENTRY "COBOLD1" USING ORDER-RECORD,
OUTPUT                                    CUST-KEY
OUTPUT                                    ERROR-TYPE
OUTPUT                                    POS
OUTPUT                                    TOTAL-ITEMS-FULFILLED,
OUTPUT                                    TOTAL-CUST-PRICE.
          * ERROR CORRECTED, ERROR-TYPE INITIALIZED
              MOVE ZERO TO ERROR-TYPE.
              MOVE CUST-NO IN ORDER-RECORD TO CUST-KEY.
              READ CUSTOMER-FIL
                       INVALID KEY MOVE 1 TO ERROR-TYP
                       PERFORM REPORT-ERROR
              MOVE 0 TO POS.
              MOVE ZERO TO TOTAL-ITEMS-FULFILLE
              MOVE ZERO TO TOTAL-CUST-PRICE.
WRAP        GOBACK.
WRAP      ******************************************************************
          PROCESS-ORDER.
INOUT        ENTRY "COBOLD2" USING POS,
INOUT                               ORDER-RECORD
OUTPUT                       ERROR-TYPE,
INOUT                                 ARTICLE-RECORD.
              MOVE ZERO TO ERROR-TYPE.
              ADD 1 TO POS.
              IF POS > 9
                OR ITEM-NO  IN ORDER-RECORD (POS) = 9
                SUBTRACT 1 FROM POS
                PERFORM PRINT-SUMMARY.
```

Figure 17. Procedure wrapping.

5.4. Procedure wrapper

The procedure wrapper is the most difficult tool to implement, since it makes the most significant changes to the target program. Here, interfaces have to be created which did not exist before. Each procedure or section of the target program which has been designated to be encapsulated as a method will have to have its own entry point with a parameter list as well as its own exit point. This requires a data usage analysis of all arguments used and results produced by this section of code. Then the names of these variables have to be placed in the parameter list. The generated entry statement is inserted at the beginning of the procedure and the generated exit statement at the end. In that way, it is possible to invoke this particular procedure without affecting the remainder of the program [Jacobson and Lindstrom 1991].

Procedure wrapping may appear to be too complicated to try, but if the tool is working correctly, there is no additional effort. Besides that, procedure wrapping comes closest to providing fine grain methods similar to what is expected from a class in an object-oriented environment. Therefore, one should not exclude this possibility. It is primarily a problem of using the right tool.

6. Software wrapper products

Software wrappers are highly dependent on the environment in which they are deployed. They have to be customized to simulate the local operating system, the teleprocessing monitor and the file management system. Therefore, there is no such thing as a standard wrapper product. Most of the wrapper products on the market are provided by a vendor who also provides the operating system and/or the TP-Monitor. `Soft Bench` from HP, `Object Broker` from BEA and `Component Broker` from IBM are typical examples. However, it is possible to construct a framework and to customize only those components which are dependent on the user environment as is the case with ObjectWrap. The following excerpt outlines examples of current wrapper products.

6.1. Soft Bench

`Soft Bench` is an object oriented development environment from Hewlett Packard for C++. It in intended to primarily support the development of client applications, but it also supports the access to legacy applications. On the server side, the legacy programs are invoked by a wrapper which is connected via an ORB to the client programs on the work stations. There is also a special wrapper for accessing the legacy database IMAGE.

Hewlett-Packard is one of the few wrapper venders, who also offers a wrapping process. The HP wrapping process involves three steps. In step one, the user is requested to make an inventory of his existing programs to determine which of them are reusable. For this, he will need reverse engineering tools. In step two, the user should develop a new client application which offers a graphical user interface with some local business logic, but which reuses existing programs and databases on the server machine. The goal is to maximize the reuse rate. In step three, the client and server components are connected via the object wrapper OO-DCE and their interaction tested. In this way the transition from monolithic host applications to distributed objects should be shortened [Aberdeen Group 1995].

6.2. Object Broker

`Object Broker` was originally developed by DEC to fulfill the CORBA Standard. In the meantime, it has been taken over by BEA and integrated with the TP-Monitor TUXEDO. The latest name is ICEBERG.

`Object Broker` offers three mechanisms for encapsulating legacy programs and databases:

– a script language for writing user specific wrapper procedures,
– a CORBA conform server driver to load and invoke legacy programs on a host,
– prefabricated client stubs which are linked to the client applications to establish the connection to the server driver.

The script language is used to create application program interfaces which treat legacy programs as methods. The language is similar to the command line language of UNIX shell script and VMS JCL procedures. It invokes encapsulated methods by means of an EXEC command with a standard parameter list. The outputs are picked up via a file interface. The server driver which is invoked by the EXEC command converts the parameters into an application program interface for calling the legacy program. As such, it functions as a gateway between the client object and the server function. The client stubs allow the server to be called either as an EXEC task, a statically linked module or a dynamically linked program. The method used depends on the performance criteria of the system [Parodi 1996].

6.3. Object Star

`Object Star` was developed by the Antares Alliance Group for encapsulating legacy applications and legacy databases in new client/server systems. It connects both Windows-NT and UNIX client applications to host applications running under MVS. The connection is made by a CORBA-ORB, either ORBIX from Iona or SOM from IBM. The wrapper is a driver shell running on the host which loads and executes CICS and IMS transactions, passing the results back to the client. It also acts as a gateway directly to the host database system, either IMS or DB-2. `Object Star` is used within the context of migration projects and must be calibrated to satisfy local requirements [Antares Inc. 1996].

6.4. Component Broker

`Component Broker` is the latest development of IBM to connect frontend and backend applications. It is intended to succeed the original IBM request broker SOM-System Object Modeling. Therefore, its primary function is that of connecting distributed objects, but it can be extended to be used as a wrapper.

`Component Broker` has two subsystems:

– `CB Connector`, and
– `CB Toolkit`.

The `CB Connector` is an ORB which covers most of the functions specified by CORBA-2, i.e., it provides the primary services required to connect, administer and secure distributed objects. Besides it is integrated with the IBM TP-Monitor CICS allowing legacy CICS Transactions to be reused as encapsulated object. In addition, it supports the access to both IMS and DB-2 databases.

The `CB Toolkit` is a development environment running under OS/2 and Window-NT. In effect, it is an extension of Visual Age; with the facilities for generating CORBA-IDL interfaces and for providing stubs and driver. It also supports the reengineering of legacy COBOL programs in COBOL-85 to IBM object COBOL [IBM 1997].

6.5. SoftWrap

SoftWrap is a standard windows-based tool developed by the author for automatically adapting source programs for wrapping. There are four versions:

– one for Assembler programs (`ASMWRAP`),
– one for COBOL programs (`COBWRAP`),
– one for PLI programs (`PLIWRAP`),
– one for NATURAL programs (`NATWRAP`).

All four versions provide four basic functions:

– transaction wrapping (`TRANWRAP`),
– program wrapping (`PROGWRAP`),
– module wrapping (`MODWRAP`),
– procedure wrapping (`PROCWRAP`).

`TRANWRAP` replaces CICS and IMS-DC macros with wrapper calls. `PROGWRAP` replaces file inputs and outputs with wrapper calls. `MODWRAP` bundles the parameters into a single data structure. `PROCWRAP` splits the program up into independently executable procedures and provides each procedure with a callable interface. In all cases, the result is a copy of the original source which can be compiled and linked to the wrapper [Sneed 1997a,b].

7. Software wrapping research

In the last two years there has been a surge of research activities on the subject of wrapping. It should be noted, however, that the idea of encapsulating existing software components for reuse in a new architecture did not come out of the research community. It comes from industry, where it was born as a child of expediency. It is proof of the fact that the research community has yet to provide an adequate solution to the problem of reengineering. As pointed out at the beginning, it would be preferable to transform the old software into something newer and better, however, the research community has yet to deliver a transformation technology that works. So users have no choice but to resort to something like wrapping to reuse their code.

Among the first to pick up the subject as a research topic has been the Software Engineering Institute, where Kurt Wallner and others have conducted several pilot projects and reported on their results [Wallner and Wallace 1996]. They have collaborated with the OMG which is promoting wrapping as a means of object-oriented migration. The OMG refers to wrapping both in the latest CORBA standard as well as in the draft standard on business objects [Wallner and Wallace 1996]. At the University of Drexel work is going on to ensure the security of wrapped objects. AT&T is sponsoring this work on a secure Legacy Wrapper consisting of an application user interface, a legacy client and a JAVA/IDL orb on the client side with a legacy virtual machine, a legacy server and a host orb on the server side [Souder and Mancordis 1999]. There is similar work going on at Georgia Tech where wrapping is being used within the scope of a project to restore legacy systems of the DOD [Rugaber and White 1998].

In Europe, wrapping research is taking place in Italy at the universities of Naples and Bari where a pilot project was made with IBM to wrap RPG programs on an AS400 [DeLucia *et al.* 1997], as well as at the University of Munich in Germany where BMW is sponsoring a project to encapsulate business objects in existing logistic applications using a holistic reuse approach [Molterer 1999]. Another project is underway at the University of Amsterdam to wrap CICS transactions by means of program transformations [Verhoef *et al.* 1999].

It would be beyond the scope of this paper to mention all of the current research on wrapping technology taking place worldwide, a browse through the web will suffice to show how extensive it is. The research efforts mentioned can only give some indication of the type of research being conducted. It is to be expected that many positive contributions will come from the research community in the next years, but wrapping will remain primarily a domain of industry users striving to find ways of integrating old applications with new ones.

8. Summary

Encapsulation is one of the three main alternatives to dealing with legacy software systems. The other two are reengineering and redevelopment. Of these three, encapsulation involves by far the least costs and the least risks. The goal is to reuse existing programs and databases with a minimum of change. To this end, they are built into new distributed applications as self-contained construction blocks or components which can be accessed dynamically to perform functions and provide data.

This paper has described the various means by which legacy software components can be wrapped and accessed. Since the reuse of legacy software artifacts in newly developed distributed systems is evolving at an ever-increasing rate, the techniques presented here can only be seen as a momentary state of the art. Middle-ware products in the sense of CORBA and DCOM are beginning to have a greater impact on the market. They will certainly have an even greater impact on the means by which existing applications can be integrated with new ones. So one can expect a significant change to the way existing software can be encapsulated.

What will not change is the nature of existing software itself. It will remain a hodgepodge of online transactions, job steps, file processors, database accesses, subroutines and business procedures. Thus, the techniques presented in this paper for adapting such artifacts and providing interfaces to them should still be valid for a time to come. This being the case, it is well worth the effort of learning them.

References

Aberdeen Group (1995), "Hewlett-Packard's C++ SoftBench 5.0," Aberdeen Group Report, Boston, MA.

Antares Inc. (1996), "ObjectStar – A Product for Wrapping Legacy Databases," Antares Alliance Group, Dallas.

Brodie, M. and M. Stonebraker (1995), *Migrating Legacy Systems*, Morgan Kaufman, San Francisco, CA.

DeLucia, A., G.A. DiLucca, and A.R. Fasolini (1997), "Migrating Legacy Systems Towards Object-Oriented Platforms," In *Proceedings of IEEE-ICSM-97*, Bari, Italy, October, p. 122.

Dietrich, W.C. (1989), "Saving a Legacy with Objects," In *Proceedings of OOPSLA-90*, Addison-Wesley, Reading, MA.

Gamma, E. *et al*. (1995), *Design Patterns*, Addison-Wesley, Reading, MA.

Gossain, S. (1997), "Accessing Legacy Systems," In *Object Expert*, London, March.

Graham, I. (1995), *Migrating to Object Technology*, Addison-Wesley, Workingham, GB.

IBM (1997), "A Survey of Object-Oriented Technology on MVS/ESA," IBM International Technical Support Center Report GG24-2505-00, Poughkeepsie Center, New York.

Jacobson, I. and F. Lindstrom (1991), "Reengineering of Old Systems to an Object-oriented Architecture," In *Proceedings of OOPSLA-91*, ACM Press, New York.

Jacobson, I. (1992), *Object-Oriented Software Engineering*, Addison-Wesley, Reading, MA.

Leeb, G. and S. Molterer (1999), "Developing Reengineering and Reusing Enterprise Software Using a Business Object Approach," to be presented at *TOOLS-99, CBESD Transition Workshop*, Santa Clara, July.

Mattison, R. (1994), *The Object-Oriented Enterprise*, McGraw-Hill, New York.

Molterer, K. (1999), "Erfahrung mit der Reengineering bestehender Systeme in BMW," In *Proceedings of GI-Workshop on Software Wartung & Reengineering*, F. Lehnert and F. Ebert, Eds., GI Arbeitskreis 5, Bad Honnef.

Mowbray, T. and R. Zahavi (1994), *The Essential CORBA*, Wiley, New York.

OMG (1998), "Business Object Component Architecture (BOCA)," Proposal – Revision 1.1, OMG Document bom/98-01-07, London.

Orfali, R., D. Harkey, and J. Edwards (1996), *The Essential Distributed Objects Survival Guide*, Wiley, New York.

Parodi, J. (1996), “Building Wrappers for Legacy Software Applications,” Digital Equipment Corp., Boston.

Rugaber, S. and J. White (1998), “Restoring a Legacy – Lessons Learned,” *IEEE Software 15*, 4, 28.

Sneed, H. (1996), “Encapsulating Legacy Software for Reuse in Client/Server Systems,” In *Proceedings of WCRE-96*, IEEE Press, Monterey, November.

Sneed, H. (1997a), “Software Interface Reengineering,” In *Proceedings of WCRE-97*, IEEE Press, Amsterdam, October.

Sneed, H. (1997b), “SoftWrap – ein Tool für die Kapselung vorhandener Assembler, PLI und COBOL Programme,” HMD Heft Nr. 194, Stuttgart, Germany.

Sneed, H. (1998), *Objektorientierte Softwaremigration*, Addison-Wesley, Bonn.

Souder, T. and S. Mancordis (1999), “Legacy – A Tool for Securely Integrating Legacy Systems into a Distributed Environment,” In *Proceedings of IEEE-WCRE-99*, Atlanta, October, to appear.

Taylor, D. (1995), *Business Engineering with Object Technology*, Wiley, New York.

Verhoef, C., A. Sellink, and H. Sneed (1999), “Restructuring of COBOL/CICS Legacy Systems,” In *Proceedings of 3rd European Conference on Software Maintenance and Reengineering*, Amsterdam, March, p. 72.

Wallner, K. and E. Wallace (1996), “Simulated Evaluation of the Object Management Group’s (OMG) Object Management Architecture (OMA),” *ACM SIGPLAN Notices 31*, 10, 168.

Winsberg, P. (1995), “Legacy Code – Don’t Bag it, Wrap it” *Datamation*, May.

Yourdon, E. (1997), “Distributed Computing,” *American Programmer 10*, 12.