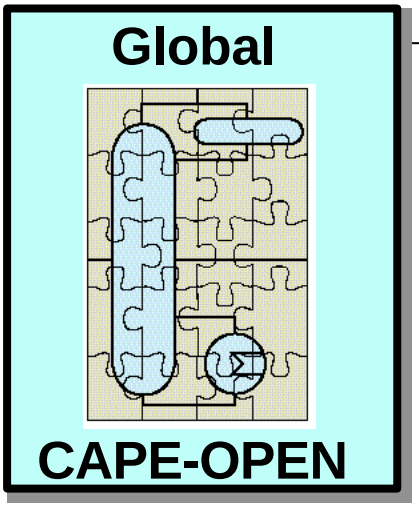
Global CAPE-OPEN

Delivering the power of component software
and open standard interfaces
in computer-aided process engineering



D822

Migration Cookbook

J. Köller **RWTH.IS**
J.C. Töbermann **RWTH.IS**

Archival Information

Reference	
Filename (if different)	
Authors	J. Köller J.C. Töbermann
Date	02.10.20022
Number of Pages	28
Version	1.1
Reviewed by (date)	**Reviewed by (name) (greyed out means not yet)**
Distribution	WP8.2
Additional Material	
Location on BSCW	

IMPORTANT NOTICES

Disclaimer of Warranty

Global CAPE-OPEN documents and publications include software in the form of *sample code.* Any such software described or provided by Global CAPE-OPEN --- in whatever form --- is provided "as-is" without warranty of any kind. Global CAPE-OPEN and its partners and suppliers disclaim any warranties including without limitation an implied warrant or fitness for a particular purpose. The entire risk arising out of the use or performance of any sample code --- or any other software described by the Global CAPE-OPEN project --- remains with you.

Global CAPE-OPEN is a collaborative research project established under BE 3512 “Industrial and Materials Technologies” (Brite-EuRam III), under contract BPR-CT98-9005

Trademark Usage

Many of the designations used by manufacturers and seller to distinguish their products are claimed as trademarks. Where those designations appear in Global CAPE-OPEN publications, and the authors are aware of a trademark claim, the designations have been printed in caps or initial caps.

Microsoft, Microsoft Word, Visual Basic, Visual Basic for Applications, Visual Studio, .NET, Internet Explorer, Windows, Windows NT, Windows 2000, Unix, Sun Microsystems, Java, Borland, Inprise, Delphi, Promula, Linar, J-Integra, Cobalt Blue, FOR_C, Anita, CO*Star are registered trademarks and ActiveX is a trademark of Microsoft Corporation.

Netscape Navigator is a registered trademark of Netscape Corporation.

Adobe Acrobat is a registered trademark of Adobe Corporation.

Contents

1 Introduction

The CAPE-OPEN (CO) standards provide a set of interfaces which allows the seamless integration of Computer Aided Process Engineering (CAPE) modules from various sources (software and equipment vendors, universities, and company generated) into process simulation environments. This enables a process engineer to 'assemble' the necessary computational tools with the minimum effort from a collection of software (in-house, commercial, and/or academic) to achieve a best-in-class solution to various CAPE-related problems.

Since the CO standards are a recent development, a lot of in-house, academic or commercial CAPE-software naturally do not comply to the CO standards yet. These systems – often, and especially in the following, called legacy systems – have consumed a lot of resources during their development and may comprise significant parts of a company's expertise. In the classical sense a legacy software has been defined by Brodie and Stonebraker as follows: "A legacy information system is any information system that significantly resists modification and evolution." [1]. Additionally, these legacy systems perform mission critical tasks and cannot be switched off without ruining the company operating the system.

Fortunately, the situation with regard to the CAPE-OPEN standard is often not as difficult as in this definition. In our domain these software systems are still maintained and improved but they do not fit into the CAPE-OPEN framework. Often, these systems have been developed using FORTRAN 77 which is not object-oriented or FORTRAN 90 if we are lucky. Therefore, it is important to combine these legacy systems with the CO standards in some way to earn the benefits offered by a CO integration without loosing the know-how captured within the legacy systems. Additionally, software migration help to preserve the investments that have been made during the development of the legacy system. A migration of the legacy system leads to this goal.

Which migration strategies, e.g. by wrapping, source-code reuse or other alternatives, are feasible depends heavily on the legacy system and its maintenance state, but of course also on the resources and further goals related to the migration project. For example many of these systems are monolithic applications without object-oriented interfaces and are written in FORTRAN. In such a case any integration with other software is difficult and a direct integration into a component based framework like CO is almost impossible. However, wrapping may be easily achieved.

To assist with migration of such legacy software to the CO standard, this document compiles migration issues, strategies, methods and tools as well as experiences. From the presented material follow guidelines and rules which are supported by actual migration experience. But the document is not intended to be exhaustive, i.e. to cover all migration opportunities with all their possible pitfalls. However, the document will be extended and revised according to further knowledge gains about best-migration-practice.

In the next section, migration goals, issues and alternative strategies are discussed in general. Section 3 presents a small decision framework giving hints and recommendations for choosing a migration path according to the actual goals and circumstances. In the further sections, alternative tools for migration support, with their requirements, their abilities, and examples, are then described.

2 Migration goals, issues and strategies

The first step in a migration project is the definition of its goals. Of course, the main migration goal in our context is that we want the legacy system to support the CAPE-OPEN standard. This means that it must match the CAPE-OPEN component framework and expose the appropriate interfaces. Additionally to the outside world the component to be developed from the legacy system must behave like it is defined in the CAPE-OPEN standards specification. To reach this overall migration goal the legacy system must then be investigated to define certain sub-goals determining how the system can be migrated to the CAPE-OPEN standard. It must be checked whether these goals may be met at all

and if so by which strategy and at which cost. From this assessment, and the gained knowledge about the legacy system, may follow a reformulation of the goals. Defining the goals and deriving an appropriate strategy is therefore obviously an iterative process.

2.1 Migration goals

During definition of the goals at least the following questions must be answered to derive a appropriate migration path and choosing a migration strategy:

- Migrate the whole legacy system or just parts of the legacy system ?
  Example: the legacy system is a library of unit operation models. The migration may then include all models or just a few, because others are clearly out-dated or are not used anymore.

- Migrate the legacy system as a whole or decompose it during migration ?
  Example: the legacy system is a unit operation model with its own thermodynamic system. The migrated system may also consist of the same intertwined unit/thermodynamic-model or of a unit operation model and a separated thermodynamic system.

- Is the (old) legacy system further used after migration and if so which system should be primarily maintained ?
  Example: A unit operation model of the legacy system should be migrated to CO standards to use it within a commercial available COSE. On the other hand the same model must be further used within simulations already set-up within an in-house simulator. In such a case the primary maintenance, e.g. bug-fixes to the core functionality, may be targeted at the legacy system, the migrated system or perhaps both systems (with the added complexity to keep both systems consistent).

- Is it a real migration or just adding some new interfaces?
  Example: If you still need the legacy system in the way it is used now a full migration might not be desirable. Then just adding the CAPE-OPEN interfaces to the system might be sufficient. A full migration can include re-implementation of the foundations of the legacy system making it impossible to use it in the original context.

- Is it just a migration or an extension?
  Example: It might be desirable just to provide the old functionality in a new shell if we just want to migrate a unit operation implementing a specific model. On the other hand it might be a goal to extend the units functionality by adding some more parameters. Another example is the addition of new calculation routines to a properties package.

- Is the current legacy system environment kept operational in the future ?
  Example: the legacy system runs on a specific machine with a specialised configuration (e.g. ancient operating systems such as MS-DOS). This machine, i.e. the system environment, is going to be shut down after migration. Then it must be assured that the migrated system runs on a new target environment.

- What is the target environment for the migrated system ?
  Examples: the target environment may be a Windows-system or a Unix-Server or a Linux-PC. The decision may depend on the type of the CO component, e.g. unit operation or thermodynamic system, the COSE environment, company-agreed hardware and middleware preferences and so on.

- Is performance a big issue ?
  Example: performance is always an issue, but in the case of an unit-operation which is just sometimes called during a whole flowsheet calculation and convergence, it is not such a big issue as it may be for a thermodynamic calculation called many times by many unit operations during each flowsheet calculation loop.

- What is the timeframe for the migration ?
  Example: If the migrated system has to be available very soon a complete re-implementation is not an option.

2.2 Migration issues

Having defined the migration goals we now need to assess the current situation of the legacy in detail to define a migration path and to choose the appropriate technologies. This may sound trivial but can be very tricky especially if knowledge about the system has been lost. During investigation of the legacy system at least the following issues have to be considered:

- Determine the exact functionality of the system. What are its features and how can this functionality be accessed by the user and/or other software systems.

- Check whether the underlying conceptual model is available and up-to-date, e.g. the model equations of a unit operation.

- Check the underlying software framework. What programming paradigm (structured, object oriented, functional programming or something completely different) has been used and what programming language(s) were chosen.

- Check whether documentation is available and up-to-date, e.g. source documentation like calling sequences, API-descriptions, user-guide, examples, manuals, etc..
  If it is not the case some documentation may have to be recaptured by specialised analysis or generation tools.

- Check which people are familiar with the legacy system (users and developers).

- Check whether source code is available and up-to-date. Also check whether the development environment is still available, e.g. included library headers as well as your compiler-linker, so you can rebuild your actual legacy system (at least in principle).
  If it is not the case, source code may be recaptured by sophisticated tools of reverse-engineering. But keep in mind that source-code obtained from reverse-engineering is very hard to understand.

- Check whether your legacy system is bound to a specific hardware/middleware configuration, e.g. due to operating system calls, special libraries, database or file access, I/O-routines, etc.. Also check if the software depends on other software systems. If yes: Do you have to migrate these systems as well (If this is possible!) or can they be kept as they are.

2.3 Migration strategies

According to the migration goals and the results of the system investigation alternative migration strategies may be applied. Each of them bears a certain set of advantages, risks and drawbacks:

- **Automated code generation**
  Code generation means here a complete re-implementation of the conceptual model by a specialised tool. From the user-supplied model equations, a new CO compliant software is automatically generated. This is distinct from a wizard, which generates a skeleton code to be further edited by a developer, e.g. placing at appropriate places copy & paste code from the legacy source code or calling functions of the maintained legacy system.

  Requirements: The actual implemented conceptual model must be known, so a good documentation may be essential or the source code must be available to re-extract it.

Advantages: No programming skills needed, because the user supplies just the model equations.

Disadvantages: It is quite hard to check the consistency of the generated system and the supplied model, i.e. just black-box tests are possible. Additionally, the code generation may not work on more complex models or may lead to very inefficient code execution. Moreover the tools required for that process might be expensive and hard to obtain.

Risks: The code generated by the tool might be too inefficient to fulfil simulation requirements.

- **Copy & paste approach**
  The functionality of the legacy system is re-implemented. The functional core of the original source code is copied into the new source code and placed into appropriate new routines. The skeleton of the new source code may be provided by a wizard-tool.

  Requirements: The legacy source code must be available and should be easy to understand. Additionally, it must be decomposable, i.e. it must be possible to extract certain pieces/modules/routines that implement a specific functionality. The legacy programming language must be the same as the one used for the skeleton. To further the understanding, documentation of high quality should be available too.

  Advantages: Results in a new, modernized system (i.e. new interfaces, complying to new standards,...) running in a almost free-to-choose environment (i.e. choice of operating system, development environment, middleware, etc.). Offers the opportunity to migrate just parts of the system and decompose the system (e.g. split a unit-operation model from its thermodynamic model). Existing documentation can be re-used. Also offers the opportunity to enhance the quality with regard to software engineering aspects, usability, documentation, future maintainability, etc. The migrated system's performance is the same as in the legacy system.

  Disadvantages: See risks, also resources needed to analyse and understand the source code as well as to implement and test the whole new system.

  Risks: The original legacy system will be normally almost bug-free in its core functionality. During re-implementation there is a high risk of introducing "new" bugs due to erroneous copy & paste actions or misunderstanding of the legacy source code (which might be quite complex to enhance performance, use special tricks of the underlying programming language, libraries or operation system, etc.). This may lead to system crashes, which are at least easy to detect, and much worse to not obvious differences between the behaviour of the old and the new implementation, e.g. slightly different calculation results.

- **Wrapping approach**
  The legacy system is not touched, but a wrapper is written, which acts as a shell around the legacy system adding the CO support. This wrapper may consist of several layers each of them providing a certain conceptual view. The legacy system is used either in compiled form (such as a library or some sort of server processs) or the source code is embedded completely as it is in the migrated system.

  Requirements: The API-Interface of the legacy system must be available and clearly understandable. Necessary calling sequences as well as preconditions to a function calls and its parameters and data structures must be known or derivable. The source code itself is not required if the compiled version of the legacy system is used. The wrapper has to start or to link dynamically the existing binaries. Therefore, the execution environment of the legacy system must still be available (e.g. operating system). In case of full source code

integration only the dependencies on other legacy systems have to be taken into account (such as databases, network infrastructure, etc.).

Advantages: The wrapping system is rather small, so it is also rather easy to implement, test and maintain. Basic functionalities of the wrapper should be easily re-useable in other migration projects using the wrapping approach. Since the underlying, i.e. the legacy, system is not touched itself, there is no risk of undesired changes in behaviour (see risks of the copy & paste approach). The programming language for the wrapper can be chosen freely. Multiple layers of wrappers allow a large amount of flexibility. Systems that are not well documented on source code level can be migrated rather easily.

Disadvantages: See risks, also no good chances to migrate just parts of the system or perform a decomposition of the system during the migration. Approach leads to additional complexity of the whole system and performance might slightly decrease due to overhead especially if multiple wrapper layers are used. Functional extensions to the legacy system are difficult.

Risks: Since the legacy system is not changed or replaced it sticks to the same, possibly quite old, environment and programming approach. This may lead to comparable high operational and maintenance cost in the future. The legacy system might depend on other legacy systems that might not be available anymore in the near future. Data conversion needed to call legacy routines can be tricky and error prone. Bridging technologies to access legacy libraries or binaries might not be available.

- **Hybrid**
  The migration strategies may be mixed according to the possible decomposition of the legacy system and the migration goals.

  Example: The main approach is wrapping, but to make the system operational on a new platform, the legacy system is also touched, to exchange for example operation specific calls. In such a case the migration project is split into two parts, the first part is a CO-unspecific migration to another platform, and the second part is the CO-migration by wrapping. This migration strategy comes in lots of flavours due to the CO-unspecific parts. It can include database migrations, platform changes, structural changes (e.g. transforming a stand-alone server to a library) and many more.

  Advantages, disadvantages, risks:
  Highly dependent on the strategy chosen. Hybrid migration itself add some complexity to the migration process but gives a great flexibility.

- **Full re-implementation**
  The legacy system is fully re-implemented from scratch using the original requirements specification (if available).

  Advantages: Since the system is completely rewritten the new system can be design and implemented using freely chosen state-of-the-art software development techniques and technologies. The new system can be optimised for its new environment thereby allowing an increase in performance. To employ this strategy no internal information about the legacy system is needed. This is useful when no source code documentation or even no source code at all exists. The only thing that id needed is knowledge about the outside behaviour (i.e. system's features) of the legacy software.

  Disadvantages: See risks. The full re-writing of a system is very resource consuming.

  Risks: Because the system is fully re-implemented it is likely (in fact sure) that there will be lots of new errors. Therefore, it will take a while to make the new system stable. Secondly, the probability of accidentally modifying a system feature present in the legacy system in an unwanted way or completely leaving it out is high.

- **No migration**
  In some cases a migration may not be feasible at all or the obstacles are too high to make the migration worth it.

3 Migration paths

We will now give some concrete pieces of advice how to act in some specific situations one might find oneself in after defining the migration goal and assessing the current situation. But before doing so we will give a short introduction into some of the relevant technologies such as middleware and programming languages. We will discuss how we can make use of the various technologies which are currently available to do legacy migration by wrapping old code either on source or on binary level. In the CAPE-OPEN domain we consider the wrapping approach to be the most useful because much of the existing legacy software is not going to be re-written. Then wrapping offers the easiest way to a new system. Nevertheless, the decision framework we will present in this section will also cover other migration strategies.

3.1 Middleware issuses

The CAPE OPEN standards are based on a component based software architecture and rely on two alternative middleware solutions: CORBA from the Object Management Group and COM from Microsoft. Pros and Cons of both alternatives are discussed in the following sections.

3.1.1 Middleware Principles

In a component based software architecture an application consists of several software pieces which are executed independently and may even reside on different computers in a network. To make this approach workable, the components must be able to communicate, i.e. to exchange data and issue method calls to each other via the component's defined interface. The technology that implements an infrastructure for such a component communication is called middleware. A middleware solution provides standardized mechanisms for the definition of component functionality and communication as well as additional services easing the use and implementation of component based software.

Basically, a middleware framework consists of four parts: An interface definition language (IDL), an interoperability protocol, an object request broker (ORB), and additional supporting services, e.g. naming or persistence services. The IDL is a language which describes all accessible data and methods of a component. It is very similar to a programming language but contains just method headers and no implementations. The interoperability protocol defines how the components communicate which each other. It defines valid message types and formats and how their content should be interpreted. The ORB is responsible for directing messages to and from different components throughout the network. It is the central piece of software in every advanced middleware because it is the logical backbone for all component communication. A general introduction about middleware issues can be found in [2].

Figure 3.1 illustrates the use of a middleware infrastructure for component communications. Here a client application uses some server functionality. The IDL is used to define the interface that a server implements, i.e. the set of services that clients may request. IDLs compile into client and server stubs. The client application calls a client stub to request a service. The client stub interfaces to the runtime system of which the ORB is a part and invokes server code that implements the requested service through the appropriate server stub. The transmission of service requests and responses between clients and servers is handled by the runtime system of the middleware platform. Thus, applications does not worry about locations of clients and servers are located on the network, differences between

hardware platforms, operating systems, and implementation languages (e.g. data formats and calling conventions).

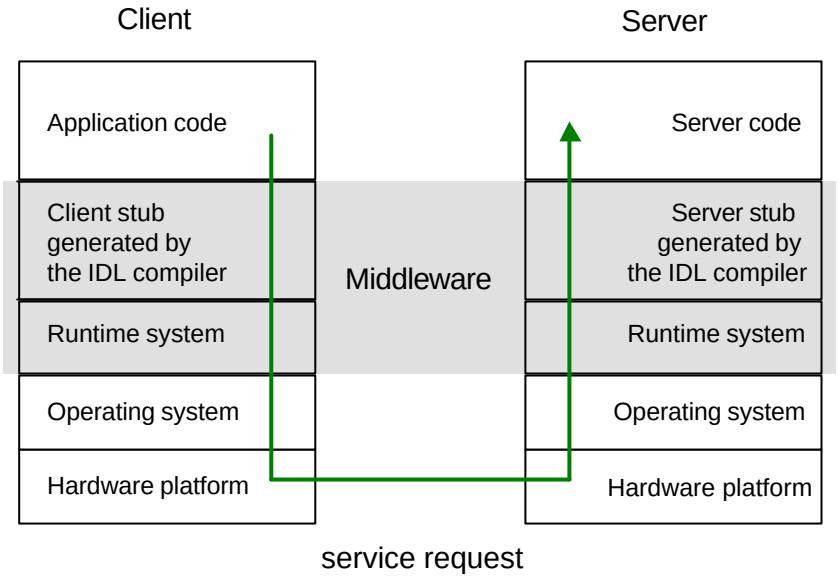


Figure 3.1: Middleware approach

As presented in this example the client and the server can be located on different computers based on different operating systems and different hardware platforms (e.g. a Windows PC and a SUN Solaris machine). Additionally, different programming languages can be used on client and server side (e.g. C++ and Java).

3.1.2 CORBA

The Common Object Request Broker Architecture (CORBA) is an open middleware standard facilitating broad interoperability between object oriented distributed components in highly heterogeneous environments [3]. It was developed by the Object Management Group (OMG) [5] which represents a wide community of software vendors, developers, and users. The OMG specified the Object Management Architecture (OMA) which includes CORBA as a key part. It is important to know that the OMA and CORBA are just open specifications and no implementations. The implementation of these standards is left to the software vendors.

The OMA is constituted of an Object Model that defines how objects distributed across a heterogeneous environment can be described and an architectural Reference Model that characterises the interaction between those objects. Therefore, these models also include the definition of the Internet Inter-Orb Protocol (IIOP) which sets a standard for internet based communication between software components for transferring messages and data. It also defines marshalling procedures on a binary level. Therefore, it is possible that CORBA implementations from different software vendors can communicate with each other. The OMA Object Model defines an object as an encapsulated entity which offers services that can only be accessed through well-defined interfaces. Clients send requests to objects to perform services on their behalf. An overview of the OMA Reference Model is given in Figure 3.2.

The central part of the OMA is the ORB, a kind of distributed object bus that is mainly responsible for facilitating communication between client and server objects through their interfaces. Several ORBs in the network from different vendors may communicate via IIOP thereby maybe connecting different hardware platforms or operating systems. Additionally, several standardized services are part of the OMA and are defined using the CORBA interface definition language (IDL). These services can be grouped in the following categories [4]:

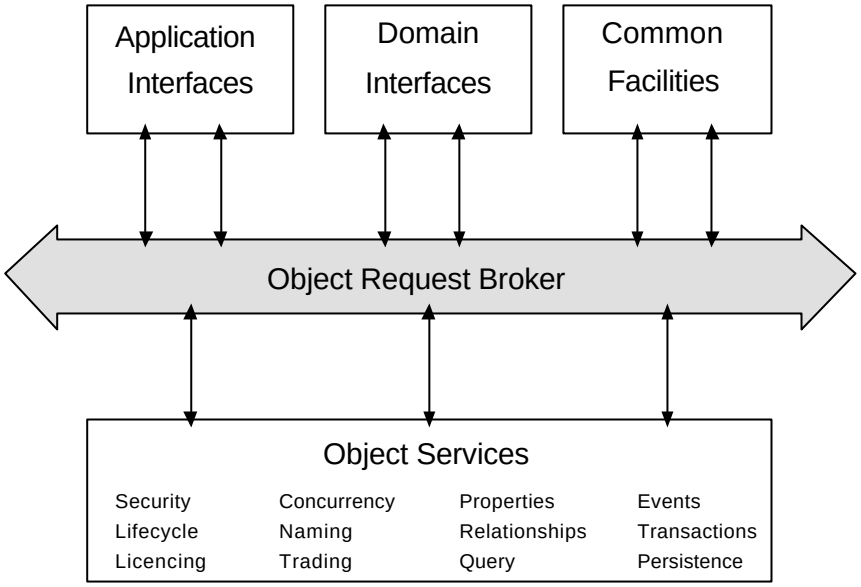


Figure 3.2 OMA Reference Model: interface categories

Object Services: These interfaces define the system-level object frameworks that extend the CORBA object bus. They are used by many application programs and provide services that are almost always necessary regardless of the application domain. Two examples of Object Services are the Naming Service which allows to locate objects based on names and the Trading Service which is a kind of yellow pages that enables to find objects based on their properties.

Common Facilities: Like Object Services these interfaces are horizontally-oriented, i.e. independent of the application domain, but they are more oriented towards the end-user application. For example database services could be found here.

Domain Interfaces: These interfaces are vertically-oriented and focus on specific application domains such as manufacturing, telecommunications, medical, and financial domains. These standard interfaces as well as the Common Facilities are designed by special task forces of the OMG.

Application Interfaces: These interfaces are specifically developed for applications and thus not standardized by the OMG.

The IDL used for defining these standard services and for all components developed outside the OMG is fully object oriented and supports multiple inheritance of interfaces. To use these IDL definitions the CORBA standard includes language bindings which are translation rules from IDL to real programming languages. Therefore, it is possible to connect components developed in different programming languages using these standardized mappings and the other CORBA mechanisms. Additionally, the language bindings facilitate the development of IDL compilers which can automatically generate all necessary code for the communication between the component implementation and the ORB. Currently, official standardized bindings are available for C, C++, COBOL, Java, Smalltalk, Ada, Python, and IDLScript [3]. There are also some bindings available for other languages (e.g. FORTRAN) which have not been standardized by the OMG so far and therefore should not be used. Hybrid approaches can be used for integrating languages that no mapping exists for. For example one could provide a C++ wrapper for a FORTRAN program and then use the wrapper to access CORBA (see below).

Due to its strengths in the integration of software in heterogeneous environments and its robustness CORBA has gained a growing importance in large enterprise applications. It has proven itself useful for complex and mission critical applications and is therefore employed in many e-commerce applications as well as in the banking sector. The importance and maturity of CORBA is stressed by the fact that a wide range of stable and high-performance commercial and non-commercial CORBA

implementations is currently available. There are also integrated tools in the market supporting rapid prototyping with CORBA (JBuilder, Delphi and C++Builder from Inprise)

3.1.3 COM

Microsoft's Component Object Model (COM) and its distributed version DCOM are the foundation of Microsoft's compound document technology OLE (Object Linking and Embedding) [6]. In the beginning, OLE was a proprietary integration solution for Microsoft Office products, but now it has become very popular. Based on OLE there is another Microsoft component standard which has become very popular recently especially in the area of web-based applications: ActiveX. Many tools available today are based on these technologies which allows them to integrate with each other seamlessly on the GUI level. Moreover, the next generation COM+ has been made available in Windows 2000, which offers advanced object services and targets the integration of systems developed in different programming languages.

In contrast to CORBA COM is no open specification but a binary interface standard that is directly connected to the Windows operating system. This makes COM and its derivatives a platform dependent technology not very suitable for heterogeneous environments. There are however some DCOM implementations available for other operating systems but these products have not gained much acceptance in the market so far. But on the other hand this strong relation to Windows has facilitated a quite efficient implementation for reasons that are explained below.

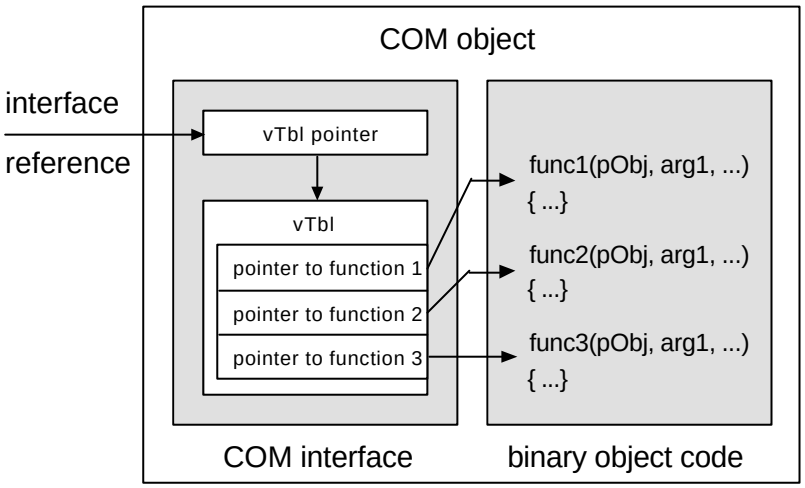


Figure 3.3: COM objects and interface references

The central idea of COM is to construct a binary interface that is constituted of a data structure called virtual function table, or v-table (vTbl). This v-table contains pointers to the functions within the COM object. A reference to a COM interface implies that you have a pointer to a pointer to a v-table. Since COM provides no language mappings, the developer of a component is responsible for providing these tables. They match very closely with v-tables known from Visual C++, but it may be very difficult if not impossible to directly implement COM interfaces in other languages like FORTRAN. However, Visual Basic, Visual C++, Visual J++, Delphi and others offer powerful tools that support COM such that less effort is required for these languages. Regarding FORTRAN, a hybrid can be built by wrapping FORTRAN with Visual C++ or Visual Basic. In this implementation model there is no explicit piece of software handling all object requests like in CORBA. All the COM ORB functionality is built into the Windows operating system. This kind of implementation is obviously quite efficient but is very proprietary.

On a higher level the functionality of a COM component is described by the Microsoft IDL (MIDL). The MIDL whose syntax strongly resembles C++ is an object oriented language which features single

inheritance only. But in contrast to COM a component is allowed to implement several interfaces which can be used for simulating multiple inheritance.

Microsoft already defines many standard COM interfaces which can be used to provide standard behaviour of application components. Most of them are closely related with OLE and define common services such as Structured Storage or compound document facilities. For application development it is necessary to create custom interfaces to define the services a component wants to provide.

In spite of its conceptual weaknesses COM plays a very important role in completely Windows based environments. Strong tool support is offered by various vendors hiding the inherent complexity of COM to a developer. Especially with Visual Basic COM component development is very easy. Additionally, no necessary software is needed for using COM as it is shipped as part of the Windows operating system. But when choosing COM as technology one should keep in mind that there are very limited options for interfacing non Windows systems.

3.1.4 Middleware related Migration issuses

Using COM, CORBA or both

As already discussed COM and CORBA have their specific strengths and weaknesses. COM is a good choice in a purely Windows based non-distributed environment. It offers good tool support for different programming languages and is efficient. CORBA on the other side has its strengths in the integration of very heterogeneous IT-landscapes. It is available for most operating systems and many programming languages. Its concepts are well designed, but it does not integrate as seamlessly as COM into the Windows environment thereby making it less runtime efficient.

Therefore, the best solution may be a mixed strategy which combines the best of all worlds. To reach such a solution so called bridging software is needed exposing a component written for a specific middleware system to another system (e.g. make a CORBA component available to COM or vice versa). To integrate COM into CORBA is not an easy task, but there are some products available (e.g. Iona Orbix COMet) which offer bridging capabilities for that area. COM-CORBA bridging is tackled extensively in [7] and the Global CAPE-OPEN bridging report [8] that ships with the CAPE-OPEN COM-CORBA bridge available online at the CO-LaN web-site [9].

Middleware and programming languages

Most modern programming languages, e.g. C++, Java, Delphi, Visual Basic, are suitable to develop software components for a middleware environment. Most state-of-the-art integrated development environments offer direct support for such middleware projects. We will now very briefly discuss some of the possible combinations of programming languages and middleware approaches.

COM-C++
Among others Microsoft Visual Studio and Borland C++ Builder offer good support for developing COM-C++ application. Generally speaking COM-C++ implementations can be made very runtime efficient but require profound experience in both C++ and COM development. This is due to the fact that the developer operates on a very low abstraction level compared to other languages.

COM-Delphi
Although not being as runtime efficient as C++ compiled Borland's Delphi code is rather efficient. The Delphi IDE offers good support for COM component development and is suitable for less experienced COM developers as much of COM's complexity is hidden away by the Delphi COM classes. Delphi is a good compromise between efficiency and development complexity.

COM-Visual Basic
Visual Basic as included in Microsoft Visual Studio is very easy to learn and to use. Most of the complexity of COM is hidden away which is why almost no detailed low-level knowledge about COM

is needed for COM-VB development. VB can be definitely recommended for COM rapid prototyping but has the drawback of not being very runtime efficient. Additionally, from a software engineering point of view VB is not suitable for larger projects because it is not really object-oriented but just object-based.

COM-Java
This combination is not recommendable because COM is a native Windows technology (although there is a Unix implementation by Software AG) whereas Java is platform independent. Therefore, COM is not part of the JDK. Nevertheless, there are some commercial technologies that facilitate Java-COM integration such as Microsoft Visual J++ or J-Integra [10].

CORBA-C++
Similar to COM CORBA-C++ applications are very runtime efficient for the price of a rather complex development. Fortunately CORBA-C++ development is not as complex as in the COM case due to the cleaner conceptual design of CORBA. Therefore, C++-CORBA development adds not very much complexity to usual C++ development. An advantage of C++-CORBA is that most CORBA ORBs support C++ language mappings and are available for multiple software platforms. This makes code migrating between different operating systems easier.

CORBA-Delphi
In version 6.0 Delphi offers good support for CORBA. Similarly to the COM-Delphi combination this offers a good compromise between ease of implementation and performance.

CORBA-Java
The combination of Java and CORBA is in principle highly recommendable because both technologies fit together very well. CORBA-Java applications are very easy to implement (comparable to the COM-VB situation). With current just-in-time compilers performance is also good but of course worse than in the C++ case. Nevertheless for migration tasks this combination can be a little bit painful. This is due to the fact that integrating non-Java code into a Java application is tricky. We will come back to this later.

Other languages
Because both COM and CORBA IDL´s follow an object oriented approach non-object oriented languages such as FORTRAN77 are not suitable in a middleware environment. Migration to CAPE OPEN requires then a re-implementation in another programming language, migration by wrapping or a hybrid migration approach.

3.2 Language integration

As we have seen in the previous sections migration will often require us to integrate pieces of code that have been implemented in different programming languages if we do not want to re-implement the whole system. Each of these languages may have it's own data structures and data types and may have different calling conventions for procedures, subroutines or functions. But if we want these pieces of software to interoperate we need mechanisms that can cope with these problems. In this document we will take a look at the two integration technologies: bridges and cross-compilers.

Cross compilers are tools that translate source code written one language (e.g. FORTRAN77) to another language (e.g. C). They mechanically translate one programming language into another without user interaction. A large set of commercial and non-commercial cross-compilers are currently available, e.g. [11,12,13]. They are not limited to FORTRAN to C or C++ translation but have been implemented for all major programming languages. Therefore, using them for integrating legacy code into a new framework is an approach applicable to a wide range of legacy systems. One serious drawback of these tools is that the source code generated by them is often not human readable and therefore not maintainable. Additionally, depending on the quality of the cross compiler performance losses might occur.

The alternative to cross compiling for language integration is the use of bridging technologies. Here we use a piece of software which can be an executable, a library or a piece of source code or macros respectively to call legacy routines that reside in a compiled library in our new source code. The bridging approach clearly separates old and new code thereby facilitating the maintenance of both sub-systems. We will now briefly introduce some bridging technologies.

cfortran

cfortran is a simple C-header file with a set of macros providing a machine independent framework for bridging between C/C++ and FORTRAN [14]. It enables the developer to call FORTRAN routines and access complex FORTRAN data structures in C/C++ source code and vice versa. It can be used on a variety of platforms including most UNIX derivatives and windows. It allows the integration of static libraries thereby providing very efficient access to FORTRAN routines. In the CAPE-OPEN project context it has been used successfully for a prototypical IK-CAPE thermo migration. Additional information about FORTRAN-C/C++ integration can be found in [15].

Dynamic Link Libraries

Windows' Dynamic Link Libraries (dll's) are the standard technology on the different Windows platforms to integrate various pieces of software that have been implemented in different programming languages. Virtually, they allow arbitrary combination of all programming languages whose compilers support the generation of such dll's. However, using these dll's directly can sometimes be tricky due to the calling conventions and subtly different data structures. For example it is possible to integrate to call a FORTRAN in VB directly without using any additional bridging technology. But the advanced tools such as the ObjectBrowser in VB cannot be used for these plain dll's because the lack some metadata about the methods it contains. Therefore, the CapWizard (cf. Section 5.2) system implements the automated generation of metadata to ease the use of FORTRAN in VB. Similar technologies exist on Unix platforms.

.NET

The .NET framework is one of the last developments on the windows platform. Currently, a first official release is available from Microsoft [15]. The .NET framework is very interesting with regard to migration because it offers a common execution environment for multiple languages. This so called Common Language Runtime (CLR) is a virtual machine with features that are similar to the JAVA virtual machine as specified by Sun Microsystems. Using Visual Studio .NET and the remaining parts of the .NET framework it is possible to integrate pieces of software written in different languages to form an complete software system in an absolute seamless manner. This is due to the fact that all .NET compilers generate metadata about a program that allows the linker to calls methods and convert data structures automatically. Another great advantage of the .NET framework is that distributed computing and the implementation of web services come almost for free and require only very little additional coding. As with Java executables .NET programs can be executed on any platform supporting the .NET framework. Currently, supported platforms are Windows NT/2000/XP, Windows CE and freeBSD. More ports to other operating systems are expected to be done soon. There are compilers for various programming languages available: C++, C#, VB, COBOL, FORTRAN, Python, Perl, SML, Oberon and others. Some of these implementations are still in beta while others such as VB, C++ and C# are available as commercial systems.

JNI

The Java Native Interface (JNI) is a technology to allow calls to platform dependent (native) libraries from Java programs [16]. Using JNI and a C++ wrapper you can call any library implemented in an arbitrary programming language. To achieve this you have to write a wrapper in C++ which then calls the actual package to be migrated. In more detail the process consists of the following steps (source code example can be found in the annex). First, you start to implement the Java program that should expose the legacy system as CAPE-OPEN component. You have to declare those methods as native that will do the C++ calls to your legacy libraries. Using the javah tool a C++ header file is generated for the methods declared as native. You then implement C++ code for this header file which then calls the actual legacy implementation. This of course bears the same problems as calling a dll or other type of shared library directly. This C++ code is compiled into a library which then is called by the Java runtime automatically. JNI is available on all platforms that have a JDK implementation.

If performance is a big issue, then overheads by wrapping may be become a problem. Also performance gains due to a modern implementation may be promising and may lead to a re-implementation decision. On the other hand a re-implementation for example in Visual Basic, which is a widely known and used language, but leads to comparably slow code execution, may reduce the performance gain significantly. Additionally, language bridging (except for the .NET framework which was specifically designed for that purpose) may create data conversion errors that are had to detect. However the wrapping approach seems to be the most promising approach in the CAPE-OPEN world as most of the software is well implemented, efficient and stable. Therefore, it is a good idea to preserve this code for future use. Moreover, the code will not be used in the CAPE-OPEN framework alone which is why preserving the code in its original environment definitely makes sense.

4 How to migrate to CAPE-OPEN

As explained in the previous section we consider the wrapping approach to be the most promising in the CAPE-OPEN domain. However there are still many options or concrete paths how to actually perform the migration. In this section we will now give several pieces of advice on how to act in a specific migration scenario. These scenarios are based on the technologies presented before and will be presented in terms of a set of question and typical situations including suggestions on how to proceed from there. The scenarios are not mutually exclusive and neither are the advices given there.

The first two steps of a migration have been presented already: situation analysis and goal definition. By now you should know exactly how your legacy system behaves, how it is structured, what it depends on, and what artefacts exist (source code, documentation, etc.). Additionally should now what the renewed system should look like: What components should be implemented, what platforms it should support and other things. Based on this situation the following questions will help you to define your migration path and the appropriate technology.

4.1 Source Code

Whether the source of the legacy system is available is one of the most important questing when determining the migration strategy. If source code is not available wrapping the system in its original environment is the only option. If wrapping is not possible (e.g. when the machine the system is installed used on will not be available anymore and is not compatible to another machine) then reverse engineering tools such as disassemblers or tools that generate source code from executable binaries can be a solution. However, the source code generated by these tools is often not human readable and should be is hard to include into a migration project. There are also tools that do no full source code reverse engineering but that just extract header information from a library. This is useful if you are wrapping a binary (see below). Another problem can be the availability of these reverse engineering tools for the platform you need. If you can neither wrap nor reverse engineer your legacy systems you should go for a full re-implementation from scratch.

If source code is available you should first check if it is complete and if it can be rebuilt on its original platform. If this is possible (and the new build really does the same thing as the legacy system) you can choose if you want to use the legacy system as binary or as source code. This decision depends on how you want to use migrated system (see also below). If you just want to use the legacy functionality or the legacy system is still maintained just wrapping the system without changing the source code at all is a good idea. This is due to the fact that you will not introduce new bugs by manipulating the source code. Additionally, this is the fastest method. But one must keep in mind that it must be possible to integrate the legacy code with the wrapper code using the technologies mentioned in Section 3.2. However, integrating the legacy source with the wrapper or the migrated source directly (or using a cross-compiler) eases the deployment of you migrated application and may increase performance.

4.2 Layers for Wrapping

We now take a look at how a migration by wrapping can be performed and how the resulting system is structured. In principle, a wrapped legacy system should consist of at least four software layers each of them providing a separate API. These four layers are shown Figure 4.1. The thickness and structure of these layers strongly depends on the complexity of the legacy application and the way the migrated system is going to be used. Depending on that each of these layers can consist of multiple sub-layers introducing new abstraction levels.

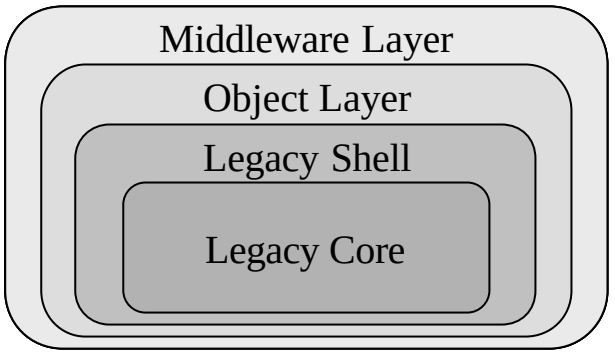


Figure 4.1 Wrapper Layers

The legacy core contains the original functionality and routines of the system to be migrated. These routines can either be used in form of a legacy binary (library or stand alone server) or as legacy source code. This does not necessarily mean that the original legacy code is not altered at all. The core level might already contain a slightly changed version of the original legacy system. For example when the legacy source was available one could have decided to extract certain routines from the original source and just include them into the new system. But this level should not contain any fundamental changes to the code. This has the advantage that the core functionality is not touched at all thereby not introducing new bugs on this level. All new bugs will then only introduced on the other layers. If the core level has outside dependencies (e.g. database access) that have to be changed in the migration process. This should be done by introducing appropriate gateways in the legacy shell layer. If this is not possible due to the lack of abstraction layers in the legacy core then these layers should be carefully introduced into the core. One should keep in mind that the legacy core can consist of multiple legacy system which by definition of the migration goal should be combined in one overall new system.

The legacy shell is responsible for providing low level access to the legacy system. Again, the structure of this level strongly depends on the architecture of the legacy system itself. If the legacy system is integrated on source code level and is written in an language that supports object oriented programming this layer can be very thin or may not be needed at all. But in the CAPE-OPEN context the core layer is often coded in FORTRAN which makes it necessary to use language integration technologies like the ones mentioned in Section 3.2. It is recommendable to implement this layer using C or C++ because this opens up the a lot of options to proceed with the object layer. There is a lot of bridges to integrate C and C++ code into a system. The wrapper layer itself should have a simple API which is more or less a one-to-one mapping of the core routines. This facilitates a clean separation between the object layer and the legacy core although it would be possible to integrate those two in one overall layer. Therefore, the shell layer is also not responsible for combining several legacy functions or subsystems into one shell function. However the shell layer is responsible for providing basic functionality for the legacy core itself if it is needed. This can be the case if a source level migration to another platform is needed where e.g. database access has to be modified or calls to the operating system must be performed. In this case the shell layer provides the appropriate functions which can be called from the modified legacy core using a language integration technology.

So far we have we have just looked at the wrapping layer for migration of legacy libraries. In case of a stand alone legacy application which is to be migrated into an component based framework things are more complicated. If the system is available as source one can try to extract the functionality needed from the system and then migrate it like a library. But this can be dangerous because the dependencies in the code have to be investigated thoroughly. These dependencies tend to be poorly documented but are very important. Reverse engineering tools provide good help here. But even in the case of a legacy application that can or should not be migrated on source level (for what reason ever) we are not lost. If the application is some sort of server accessible through TCP/IP connection (or another network protocol) the shell layer must encapsulate this network interface and provide a regular one-to-one API. But even in case of a really stand alone application there are still some options. There are so called terminal or screen wrappers/emulators which can encapsulate an application's (graphical) user interface and expose it as an API or make it accessible via scripting [17, 18, 19].

On the next abstraction level the object layer provides an object oriented API for the legacy system. Naturally, this layer must be implemented in an object-oriented (or at least object based) language such as C++, VB or Java. The object model of this layer may correspond directly to the object model that is used in the target framework's middleware layer. Therefore, the object layer could make use of the CAPE-OPEN interfaces to define an object model as a one-to-one mapping. If the migration is only performed to make the legacy systems available through the CAPE-OPEN interfaces this is a quick and easy solution. But if the system is going to be used with other API's than the CAPE-OPEN interfaces a separate object model can make sense. But a separate object model is also useful when because the object layer is responsible for mapping the plain shell API to a more comfortable one. This can for example include the introduction of new objects for data structures used in the legacy core or modifications of them. Another important issue is the user friendly configuration of the legacy system which should be coded on this level. All these things should not be moved to the middleware layer directly because it is specific for CAPE-OPEN which might not be the natural object model for wrapping the legacy. Additionally, the object layer should be independent of the middleware platform used.

Finally, the middleware layer is responsible for exposing the functionality defined in the object layer in terms of CAPE-OPEN interfaces to the outside (i.e. clients). It is specific for a given set of interfaces defined in IDL (COM, CORBA, etc.) and a middleware technology. In the skeleton implementations that are generated by most IDL compilers the objects of the object layer have to be created and their methods are called. Therefore, a mapping from the object model of the middleware to the one used in the object layer is needed.

When designing and implementing these layers existing documentation of the legacy system is crucial. When there is no complete documentation set available simple header files and clients whose source code is available are very useful to reconstruct how to use the legacy system. An example for a migration by wrapping can be found in the CAPE-OPEN Migration Methodology Handbook [20].

4.3 Do just a wrapping or a re-design?

Just wrapping a legacy systems is clearly the quickest way to migrate the system to a new environment but it is not necessarily the best. If you just wrap the system you do not perform a real migration on the legacy core. This makes sense if the legacy itself is going to be maintained in its original form or if there are many dependencies in form of other applications calling it directly. A real migration of the core would cause not only the legacy itself to be migrated but also the other clients. But if maintaining the legacy core itself if expensive or cannot be performed anymore a real migration should be performed. This includes porting the system to another platform and or re-implementing it (partially) in another language or programming paradigm (e.g. from FORTRAN 77 to FORTRAN 90/95). Of course, a re-design can be performed on source code level only. One should also think about a re-design if the way the legacy core is used changes fundamentally. An example for this would be a library that is going to be accessed by object-oriented interfaces only (such as the CAPE-OPEN interfaces).

Important scenarios for a re-design are all scenarios where the source code of the legacy core has to modified significantly. If you are touching the source code anyway you should modify it in a way that it makes it easier to maintain and to use in the future. Important scenarios for re-design are the migration of stand-alone legacy systems to component or the extraction of functionality from an existing library (e.g. modifying a unit with thermo to a unit with external thermo).

When you are doing a re-design of the legacy core the first important step is to analyse the core's architecture and to identify internal abstraction layers and dependencies. Reverse and reengineering tools are available for that purpose. The next step is to transform these abstraction layers to the target architecture. We are not covering this issue here because there are too many approaches to this to show them here. A good starting point for reading is the book "Migrating Legacy Systems" by Brodie and Stonebreaker.

4.4 What middleware and target platform should be used?

The choice of the target platform (hardware and operating system) and of the middleware platform is very important because can affect the whole migration path. As explained in Section 3.1 COM and CORBA have their specific strengths and weaknesses. In general one could say that CORBA is the more flexible platform because it supports multiple operation systems. Therefore, it is a good choice if you are planning to make a piece of software CAPE-OPEN compliant that resides on a non-Windows platform such as Linux or Solaris. Because distributed computing was one of the main design goals of CORBA it should be used if you are planning to use remote calculations. Additionally, if you are planning to use Java in your migration path on the object layer the use of CORBA as middleware platform is recommended due to the good integration of these two technologies. COM on the other hand does not integrate very well with Java although there are some commercial implementations of Java-COM bridges. Examples on how to use CORBA in connection with Java to implement CAPE-OPEN components is can be downloaded from the CO-LaN web-site.

Similar to COM which ships with the Windows operating system there are free implementations of CORBA which are very stable and runtime efficient (e.g. omniORB [21]). But CORBA has some severe drawbacks that should be mentioned. Because it is not integrated into the Windows OS directly it is not as runtime efficient as COM especially when remote calls are issued (the same holds for DCOM). Additionally, component activation is not as comfortable as in COM. In COM just registering a component in the Windows registry is sufficient to make it available for potential clients. In CORBA the servers have to be started explicitly or some sort of application server (usually a commercial tool) has to be used. Therefore, if you are planning to deploy the migrated system in a non-distributed pure Windows environment the use of COM is recommended. However, the complex development of COM components should not be underestimated. Developing a COM application in C++ is a rather tricky thing whereas development in VB is easy on the price of runtime efficiency. Examples for COM components implemented in VB can be found in the source code of the CO tester which is available for download at [9].

If you are mainly working in a Windows environment and you need remote calls or components that are running on other platforms than Windows then the use of a bridging technology is a good option. There are some commercial implementations of COM-CORBA bridges and a free implementation of a specific CAPE-OPEN bridge will be available on [9]. However, when using middleware bridging technology one should be aware that performance may go down. If it the use of remote components is expected to be rather frequent a CORBA migration should be considered.

4.5 What programming language to use?

Following the discussion in Sections 3.1.4 and 3.2 the fastest way to a migrated system is the use of the combinations of Java and CORBA or VB and COM. Because both languages are very suitable for rapid prototyping and hide away much of the complexity of CORBA or COM respectively you will have a wrapped migrated legacy system rather quickly. VB and Java should be used for the object and

middleware layer whereas C++ should be used for the shell layer (in the COM case VB can be used as well) because there a good tools especially for integrating C/C++ and FORTRAN. If performance is a big issue then C++ is a good choice for the object and middleware layer for both COM and CORBA components. But usually the C++ implementation will need more resources because development takes more time here. A good alternative which is also supports COM and CORBA is Delphi.

5 Supporting tools for a CAPE-OPEN migration

In the Global CAPE-OPEN project a number of tools have been developed to support the implementation of CAPE-OPEN components. All these tools are wizards which can produce source code skeletons for CAPE-OPEN components thereby relieving the developer from implementing all the things that occur in almost every CAPE-OPEN component in similar ways. Two of these wizards are targeting the mechanical generation of Unit Operation and Property Packages. Another wizard has been developed to support a migration by wrapping directly. The latter one will be described in more detail in this document.

5.1 The Thermo and Unit Wizards

These two wizards are implemented as plug-ins for Visual Studio 6.0. They can be used to generate skeleton implementations in VB or C++ that may be completed by the component developer. They can be downloaded from [9]. These tools support the migration process by generating parts of the object and middleware layer. They do not offer direct support for implementing the shell layer of a wrapped legacy system. Nevertheless they are useful in such a process because they generate all the "standard" CAPE-OPEN compliant code that is needed in a property package or a unit operation respectively.

5.2 The Migration Wizard CapWiz

The CapWiz System which is available for download on [9] is a web based migration wizard that mainly supports the developer in implementing the shell and the object layer of a legacy wrapper. Additionally it guides the user through the overall migration process. Currently, it is capable of analyzing arbitrary FORTRAN source code (or headers) and arbitrary CORBA IDL. Based on this analysis it generates skeleton source for the shell and the object layer. The process implemented in the CapWiz system is shown in Figure 5.1.

First, the developer uploads the FORTRAN source and the target CORBA IDL to the system. A personal project area is created for this. Then, the FORTRAN source and the CORBA IDL are parsed and translated into an XML representation. Based on these representation the developer can create associations between FORTRAN routines and methods for objects which have been defined in the CORBA IDL. These associations are create graphically by drag-and-drop. The associations are also stored in a XML file. Now these three XML files can be used for creating the various pieces of wrapper code. The code generation is performed by XSL transformations on the XML files using some pre-defined XML stylesheets. Currently, the stylesheets are capable of producing the following source code skeletons:

- cfortran macro definitions
- CORBA server skeleton code including calls to FORTRAN using the cfortran (according to association definition)
- Native omniORB server skeletons with cfortran macro calls (also according to associations)
- ODL (object definition library) code which can be used to call a FORTRAN dll directly from VB without having to define the dll imports first.

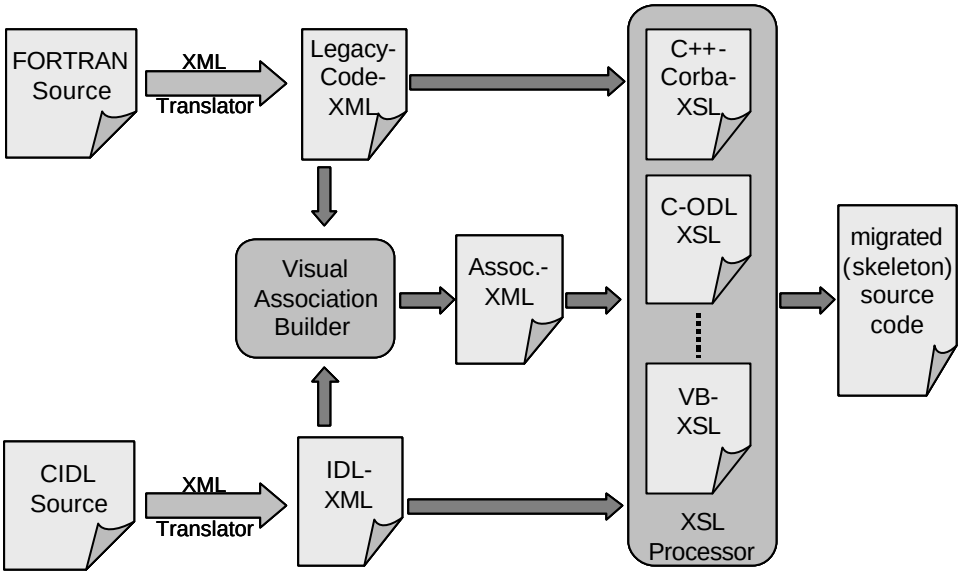


Figure 5.1 The CapWiz Migration Process

Additionally, the developer can define and upload XSL stylesheets to generate other types of source code skeletons (e.g. java source). Currently, on the input side the system is limited to CORBA and FORTRAN but COM parser and other source code parsers that would generate appropriate XML structures could be integrated easily. Because the system itself is self-explanatory we will not prsent more details here.

6 Source Code Samples

Finally, we will now present some examples for typical pieces of code needed to perform legacy migrations.

6.1 Calling a FORTRAN dll from VB or C++ directly

This sample was provided by Daniel Pinol, AEA Hyprotech.

The following piece of FORTRAN source specifies an enthalpy calculation routine.

```
LIBRARY      "HDAPack"
DESCRIPTION  'Fortran HDAPack DLL'
EXPORTS
    _ENTHAL@20         @3

SUBROUTINE ENTHAL(TEMP,PRESS,COMP,PHASE,ENTH)
    DOUBLEPRECISION PRESS,TEMP,COMP(10),ENTH
    INTEGER PHASE
```

The routine can be called in C++ using this declaration:

```
extern void _stdcall ENTHAL //"_ENTHAL@20"
  (double*Temp, double*press, double*comp, long*Phase,
   double*enthalpy);
```

C++ call:

```
ENTHAL(myTemp, myPress, myComp, myPhase, myEnthalpy);
```

In VB the declaration for the same subroutine looks like this.

```
Public Declare Sub RKEnthalpy Lib "HDAPack.dll" Alias "_ENTHAL@20" _
   (Temp As Double, press As Double, comp As Double, _
    phase As Long, enthalpy As Double)
```

VB call:

```
RKEnthalpy myTemp, myPress, myComp, myPhase, myEnthalpy
```

6.2 Calling FORTRAN from Java using cfortran

Let's assume the FORTRAN source contains the follwing definition (this example is taken from the IK-CAPE thermodynamics package)

```
SUBROUTINE T_ENTH (WHAT, PHASE, T, P, MOLES, ENTHALPY, RVAL)

CHARACTER WHAT*(*), PHASE*(*)
       INTEGER RVAL
       DOUBLE PRECISION T, P, ENTHALPY, MOLES(*)
```

Our goal is to call this function from Java source code. To do this we have to declare a wrapper class which contains this function as a static member. Additionally in the static initialization part of the class we have to load the native wrapper library "IKCThermoWrapperLib" into memory:

```
package NativeCape;
public class NativeFunctions {
   public static native int CEnthalpy(java.lang.String what,
         java.lang.String phase, double t, double p, double[] moles,
         Double enthalpy);

    static {
           try {
           // IKCThermoWrapperLib should be in searchpath
            System.loadLibrary("IKCThermoWrapperLib ");
           }
           catch (UnsatisfiedLinkError e)
           {
            System.out.println("Library not found");
           }
           catch (SecurityException e)
           {
            System.out.println("Security violation");
           }
  }

  public NativeFunctions() {
  }
}
```

Next, we must generate a C/C++ header file for this wrapper class using the javah-tool. The output will look like this:

```
#include <jni.h> //include all necessary jni routines
#ifndef _Included_IKCThermoWrapperLib
#define _Included_IKCThermoWrapperLib

#ifdef __cplusplus
 extern "C" {
#endif

JNIEXPORT jint JNICALL Java_NativeCape_NativeFunctions_Centhalpy
  (JNIEnv *,jclass, jstring, jstring, jdouble, jdouble,
   jdoubleArray, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Now, we must implement the C/C++ (here it is C++) source for the wrapper library we does the actual calls to the FORTRAN source using cfortran:

```
#include <stdio.h>
#include "IKCThermoWrapperLib.h"
#include "cfortran.h"

// first the cfortran definitions
PROTOCCALLSFSUB7(ENTHALPY, t_enth, STRING, STRING, DOUBLE, DOUBLE,
                 DOUBLEV, PDOUBLE, PLONG)
#define ENTHALPY(what, phase, t, p, moles, enthalpy, mist)
CCALLSFSUB7(ENTHALPY, t_enth, STRING, STRING, DOUBLE, DOUBLE,
  DOUBLEV, PDOUBLE, PLONG, what, phase, t, p, moles, enthalpy, mist)

// now the actual implementation of the native function calling the
// FORTRAN routine

JNIEXPORT jint JNICALL Java_NativeCape_NativeFunctions_Centhalpy
  (JNIEnv *env, jclass jclas, jstring jwhat, jstring jphase,
   jdouble jt, jdouble jp, jdoubleArray jmoles, jobject jenthalpy)
  {
   int RVAL=0;
   // for jenthalpy
   jclass jcls;
   jfieldID fid;
   jdouble enthalpy;
   //for jmoles (jdoubleArray)
   jsize len;
   jdouble *moles;

   const char *what =(*env)->GetStringUTFChars(env,jwhat,0);
   const char *phase =(*env)->GetStringUTFChars(env,jphase,0);
   // GET jenthalpy
   jcls = (*env)->GetObjectClass(env,jenthalpy);
   fid = (*env)->GetFieldID(env,jcls,"value","D");
   if (fid == 0) {
       printf("VALUE not found.");
       return -1;
   }
   enthalpy =(*env)->GetDoubleField(env,jenthalpy,fid);
   // GET jmoles
   len = (*env)->GetArrayLength(env,jmoles);
   moles = (*env)->GetDoubleArrayElements(env,jmoles,0);
   // now perform the FORTRAN call
   ENTHALPY ( what, phase, jt, jp, moles, &enthalpy, &RVAL );
   (*env)->ReleaseDoubleArrayElements(env, jmoles, moles, 0);
   (*env)->SetDoubleField(env, jenthalpy, fid, enthalpy);
   (*env)->ReleaseStringUTFChars(env, jwhat, what);
   (*env)->ReleaseStringUTFChars(env, jphase, phase);
  return RVAL;
  }
```

If the header file and the C file are compiled into a library (maybe statically linking the FORTRAN lib) the java program can be executed sucessfully.

REFERENCES

[1] M. Brodie, M. Stonebreaker, Migrating Legacy Systems, Morgan Kaufmann Publishers, 1995

[2] D. Serain, Middleware, Springer, 1998

[3] OMG CORBA web site, www.omg.org/corba, 2001

[4] S. Vinoski CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. IEEE Communication Magazine, Vol. 14, No. 2, 1997

[5] OMG web site, www.omg.org, 2001

[6] Microsoft COM web-site www.microsoft.com/com, 2001

[7] R. Geraghty et al., COM-CORBA Interoperability, Prentice Hall, 1999

[8] Global CAPE-OPEN Bridging Report, 2001

[9] CO-LaN web-site, www.colan.org, 2001

[10] Linar Ltd. web-site, www.linar.com, 2001

[11] f2c FORTRAN-C cross compiler, www.netlib.org/f2c/index.html, 2001

[12] Promula FORTRAN to C translator, www.promula.com/fortrantoc, 2001

[13] Cobalt Blue, FOR_C, www.cobalt-blue.com/fcmain.htm, 2001

[13] cfortran web-site, www-zeus.desy.de/~burow/cfortran/, 2001

[14] FORTRAN-C/C++ integration, www.neurophys.wisc.edu/comp/docs/notes/not017.html, 2001

[15] Microsoft .NET web-site, www.microsoft.com/net, 2001

[16] Sun Microsystems Java Native Interface web-site, java.sun.com/j2se/1.3/docs/guide/jni/, 2001

[17] Expect web-site, expect.nist.gov, 2001

[18] CO*Star web-site, www.clearview-software.com/webpage/products/costar.html, 2001

[19] AniTa web-site, www.april.se/english/anita.asp, 2001

[20] CAPE-OPEN Migration Methodology Handbook, available at www.global-cape-open.org/09_CO_Migration_Methodology_Handbook.pdf

[21] omniORB web-site, www.uk.research.att.com/omniORB/omniORB.html, 2001