Mainframe-Based Legacy Integration: An Industry Project

Chia-Chu Chiang

Department of Computer Science, University of Arkansas at Little Rock, 2801 South University Avenue, Little Rock, Arkansas 72204-1099, U.S.A.

Abstract

As a computing environment evolves, heterogeneous and distributed computing environments are often unavoidable. Companies are willing to retain their legacy systems and also acquire new types of systems for their daily business operations. Rewrite, facelift and convert are three common ways to address legacy integration for use in heterogeneous and distributed computing environments. Rewrite and convert may not be suitable for companies to reengineer their legacy systems because the companies have made significant investments in the systems. They are more willing to reduce the effort of their programming staff and minimize unnecessary rework. Therefore, it is essential to provide a cost effective approach to integrating legacy systems. In this paper, we present an extended facelift approach to integrating an industry mainframe-based software reengineering legacy system in a distributed heterogeneous computing environment. An experience of implementing the approach and lessons learned are also discussed.

Key Words: CORBA, Heterogeneous Software, Legacy Integration, Reengineering, Wrapper

1. Introduction

Companies that are interested in building enterprise wide applications in distributed and heterogeneous computing environments need their legacy systems to work with heterogeneous applications. These legacy systems were usually developed for centralized environments that greatly restrict system distribution and scalability. There is a definite need to reengineer existing legacy systems in order to meet the needs of modern business practices [1,2]. For those companies, one of key challenges is how to leverage their legacy systems for use in heterogeneous and distributed computing environments.

There are three common ways to integrate legacy systems into a distributed heterogeneous computing environment – rewrite, facelift and convert. The rewrite approach is to write the entire system from scratch. The cost and time requirements associated with the rewrite approach are usually unacceptable for most companies. The convert approach provides a way to first analyze the legacy systems and then transform them into completely new systems [3–5]. The conversion process is lengthy, complex, and risky which usually needs tool support to facilitate the process and minimize the cost. The facelift approach is defined to wrap a legacy system conceptually into a container and provide a relatively inexpensive means to build a user interface to access the system with less impact on the system. From an economical point of view, the facelift approach is a cost-effective way of leveraging the legacy systems. Basically, this approach benefits the companies using and maintaining the legacy systems for business operations. The main goals of this approach are: (1) preserving and leveraging the company’s legacy systems, (2) keeping the company’s legacy systems operational, and (3) facilitating the company’s legacy systems evolution. In this paper, we are going to present an experience of implementing an extended facelift approach to integrating an industry mainframe-
based legacy system for use in heterogeneous computing environments via CORBA. Although we demonstrate the approach using a very specific reverse engineering project, the approach can be useful in many practical cases such as accesses to time-consuming batch jobs.

The remainder of this paper is organized as follows. In Section 2, we give a general description of a distributed computing environment for integrating legacy systems using middleware. We then discuss the strengths and weaknesses of some typical middleware technologies such as COM/DCOM, CORBA, and Enterprise JavaBeans (EJB). We also present the research work for leveraging legacy systems through the use of middleware. In Section 3, we describe issues related to legacy integration. In Section 4, we present an architecture with an extended facelift approach to legacy integration with an industry project. We also provide our experience in designing and implementing this architecture with tool support for legacy integration. Finally, the paper is summarized in Section 5.

2. Related Work

Legacy integration was first introduced by Thomas Dietrich of IBM at the OOPSLA Conference in 1989 [6]. The objective of legacy integration is to allow legacy systems on a centralized platform to interface with other systems in a heterogeneous distributed computing environment. Since then several techniques such as remote procedure calls, file transfers, sockets, scripts and macros have been presented for legacy integration [7]. In addition, there are an increasing number of papers addressing how to integrate legacy systems for use in heterogeneous computing environments using middleware such as EJB, CORBA, COM/DCOM, and XML [8–12]. In the following subsections, we provide related work about the use of middleware for integrating legacy systems.

Harry M. Sneed [10,11] proposed a wrapping approach for legacy integration. Sneed first pointed out that software could be wrapped at five different levels: job, transaction, program, module, and procedure level via middleware such as CORBA. Sneed also presented a proposal for enterprise application integration using XML as a common interface language [12]. XML is used to exchange data not only pure data but control information between programs independently of their implementation languages and runtime environments.

Cimitile et al. [4] describe an approach to wrapping that involves the use of data flow analysis in order to discover various properties of source code. The approach focuses on the static analysis of code in order to determine appropriate decompositions of different program components as well as discovery of the formal parameters for the interfaces to these program components. The legacy program components can be independently and selectively replaced with newly developed components. The incremental migration approach reduces the risk of the migration process. The approach facilitates the coexistence of legacy program components and new components.

Jacobsen and Kramer [16] describe an approach for synthesizing wrappers based on a CORBA IDL description. In their approach, they address the problem of object synchronization within the context of the CORBA standard and define a technique based on the application of the adapter design pattern. In many ways, Jacobsen and Kramer’s approach is similar to [17]. Gannod et al. describe an approach for facilitating the integration of legacy systems into new environments. A software architecture description language, called ACME, is used as an assistant for building wrappers. First, an ACME specification of a legacy system is constructed to describe the properties of a legacy software from a software architecture point of view. The interfaces to the legacy software and its adapter are then created. The adapter allows the client to have compatible interfaces with the legacy software [18]. Secondly, a set of tools is used to generate the wrappers of the legacy software by parsing the ACME specification. The approach also shows how these wrapped components can be integrated dynamically at run time.

Recent changes to the CORBA specification have added a new type of invocation model known as Asynchronous Method Invocation (AMI) [19] to support an event-based architecture. AMI allows clients to send requests without waiting for replies. Two variants of AMI were implemented in [20]: polling and callback. The AMI polling model allows a client to send a request to the server and return immediately. The client later checks with the Poller object in the ORB core to retrieve the response. The AMI callback model allows a client to send a request with a reply handler object to a server and return immediately.
3. Concerns of Legacy Integration

There are several concerns about legacy integration:

- Communication. How should new applications in different programming languages running on different platforms communicate to the legacy applications? Is the communication mechanism synchronous or asynchronous? How can data representations be handled by the new applications due to different computer architectures such as byte ordering?

- Databases. How can new applications access the legacy databases? Are data conversions required for these legacy databases? How many programs are required to be modified due to the conversion?

- User interfaces. How can the legacy interfaces be accommodated and shared by new applications?

- Resources. How much time is needed to complete the project and how much money is required to get the project done?

4. An Industry Project

This section describes an industry project for legacy integration in which a centralized software reengineering system was integrated into a distributed and heterogeneous computing environment. First, background information for software reengineering is provided. This is followed by a description of the centralized software reengineering system. The architecture with its implementation to support the integration of the software reengineering system is discussed together with tool support.

4.1 Centralized Software Reengineering Tools

An industry system [21] was developed to support the software reengineering process. The system contains tools originally developed for centralized environments as shown in Figure 1. The application understanding tool identifies all the entities of an application, the relationship between those entities, and the effects of the proposed changes on the application. The entities may include programs, files, databases, JCL, load modules, copybooks, and data items. The portfolio analysis tool is used to determine program complexity. The code change tool provides interactive program analysis capabilities in a syntax-sensitive source editor. It also checks for syntax errors to ensure a clean compile for the code. The program testing and debugging tool allows the user to step through the code one statement or paragraph at a time, run to breakpoints, and execute entire PERFORM ranges with a single command. The program reengineering tool splits large, complex programs into callable modules. The program documentation tool is a static analysis tool that generates documentation about programs.

The system’s repository-based technology stores comprehensive information about programs and applications. This information is extracted by the Analytical Engine and stored in the Application Knowledge Repository (AKR). Together, the Analytical Engine and the Application Knowledge Repository give the facilities access to the specific information needed to perform the software reengineering task on clients’ business applications. The system runs on IBM MVS for reengineering business applications designated for the same platform. The business applications may contain programs, JCL, and load modules. In its current state, the programs may be written in COBOL, Assembly, PL/I, DB2 SQL, IDMS, and/or IMS.

The system only allows the users to access the system on an IBM platform through TSO and ISPF by submitting a job in JCL in batch mode. In today’s high-tech society, distribution and scalability are very important to a software system, especially when it is part of an enterprise information system. The system built on a centralized environment restricts IT connectivity, performance, robustness, distribution and scalability. With the widespread use of object-oriented and client-server technologies, customers are expecting the system to take advantage of these new technologies and also cooperate with their heterogeneous distributed computing environments.

![Figure 1. Existing systems workbench’s repository based architecture.](image-url)
In this project, we leverage the system to operate in a heterogeneous distributed environment.

4.2 Heterogeneous Computing Environment

In this section, we describe the application of a distributed computing architecture for modernizing the centralized software reengineering system to take advantage of client-server and object-oriented technologies and also cooperate with the customer’s heterogeneous environments.

The new architecture allows clients on Windows NT or Windows 95/98 to send requests to remote servers on IBM mainframe through CORBA compliant middleware. Two servers were implemented on the IBM mainframe: an Analyze Server and a Repository Server. The Analyze Server is responsible for serving requests by invoking the reengineering system to analyze the programs/applications in COBOL, Job Control Language (JCL), DB2 SQL, IDMS, Customer Information Control System (CICS), and IMS and store the analysis results in a repository. Once the analysis is completed, the client can issue requests to the Repository Server to query the analysis results about the programs/applications. The query results may include the structures of the programs/applications, impacts of the data items, flows of the programs, and relations among the data items.

4.3 A Distributed Architecture

An architecture using the wrapper and gateway techniques [15] is limited to our legacy systems:

- Our legacy systems usually require several hours and sometimes may require days to analyze the applications. It is not reasonable for clients to wait for the results on-line.
- Our legacy systems may need files to be created in advance before they can be invoked. In batch mode, these housekeeping tasks are specified in a JCL (Job Control Language) job and handled by the system. However, in on-line mode, the wrapper or the server need to perform these housekeeping tasks before they can invoke the functionality of the legacy systems. This may affect the performance of the wrapper and the server, and increase the system response time.

Strategies for building a thin server to reduce the response time and also allow clients to send requests without being blocked play an important role in legacy integration. Before presenting our architecture, we start discussing some additional requirements and solutions in the following subsections.

4.3.1 Requirements

In addition to the above specific characteristics of our legacy systems that an architecture needs to deal with, the following list presents the specific requirements that the architecture needs to meet:

- The architecture should allow clients to cancel and delete their own requests. In addition, the architecture should have the ability to resist unauthorized attempts at usage and deny service while still providing its services to legitimate clients.
- The architecture should provide a way to monitor the performance of serving clients’ requests.
- The architecture should reduce the coupling between the client code and server code in addition to reducing the coupling between the server code and the legacy system code.

4.3.2 Design

There is not a single way to design an architecture satisfying the above requirements. Instead, there are many possible different architectural styles, each driven by its own set of underlying assumptions and objectives. In this section, we present an architecture shown in Figure 2 to support the above requirements in the following senses:

- The design of Work Dispatchers achieves multiple purposes. First, Work Dispatchers are to reduce the server’s workload. Without Work Dispatchers, the housekeeping tasks lie on the server side. Unnecessary work on the server may degrade the server performance by increasing the response time for clients. Second, Work Dispatchers serve to resolve interface mismatch [22] between the server and the legacy systems. Third, the separation of the server and the legacy systems by adding Work Dispatchers between them reduces the coupling. Fourth, Work Dispatchers avoid clients being blocked waiting for the results.
- The addition of the Adapter to the architecture decouples the clients and the server.
- All the requests are stored in a permanent status
file and their statuses are timely updated by the server and Work Dispatchers. The Work Dispatchers can consult this file to monitor the performance of serving clients’ requests.

The design of the architecture meets the following objectives:

- Loose integration through CORBA services. The architecture supports an environment based on simple clients and sophisticated servers. It should be inexpensive to develop a new client and maintain the existing clients to take advantage of existing servers. The Adapters are designed to achieve this goal.

- Sharing programs in different programming languages. The architecture supports an environment based on a desire to share programs written in different programming languages. This goal is also achieved through CORBA services. The interfaces of the servers and clients are described and created through a CORBA IDL. The communication mechanism among heterogeneous programs is hidden from programmers.

- Sharing databases deployed on different platforms. The architecture supports an environment based on a desire to share databases created on different platforms. This goal is achieved through the uses of servers. The interfaces to access the databases are developed through a CORBA IDL to help new applications accommodate the databases.

- Separating the user interfaces to the system. User interfaces are supported among different operating systems that are difficult to be constructed as a central service. The architecture accommodates the user interfaces by masking where the new application appears to have been ported to the designated platform. In fact, it is actually running entirely on another platform. This goal is achieved through the design of the adapters that supports the communication between the servers and clients.

- Uniformity. The architecture enforces uniformity and permits implementation on diverse hardware and software to accommodate heterogeneity. This goal is achieved through the adoption of CORBA. It hides the complexities of hardware and software from programmers.

### 4.3.3 Implementation

In this architecture, the three main blocks -- Client-Side Adapter, Server, and Work Dispatcher are briefly described below. The detailed implementations of these three blocks can be found in the article [23].

#### 4.3.3.1 Server

CORBA requires that a server’s interface is expressed by an Interface Definition Language (IDL) interface. The interface definition specifies the methods the server is prepared to perform, the input and output parameters the methods require, and any exceptions that may occur during the operation. The clients only know the server’s interfaces without knowing how the server is implemented.

In order to allow a client to submit requests to a server without being blocked, the client receives a handle whenever a request is submitted to the Analyze Server or the Repository Server. The handle is then used to check the status of the request; 0 is OK, 4 is a warning, 8 might be OK, 12 is serious, and 16 is aborted. The client is free to cancel his/her request as long as the request has not been processed yet by the reengineering system. Due to the lengthy task of recovery, it is suggested that a request being processed by the reengineering system should not be cancelled. The client is encouraged to logically delete
the request from the file when the submitted request is completed or cancelled. Any deleted requests will be physically deleted for space, if the server cannot find available slots for storing the incoming requests.

4.3.3.2 Work Dispatcher

When a server receives a request from the client, the server creates a work order for the request. After that, the server returns a unique handle to the client and immediately forwards the handle to the Work Dispatcher through a named pipe (also called a FIFO because data are retrieved in a first-in-first-out manner).

The Work Dispatcher is in a loop that reads the named FIFO. If the named FIFO is empty, the Work Dispatcher goes back to sleep. As long as there is a handle, the Work Dispatcher reads the handle from the named FIFO. Then the Work Dispatcher reads the work order from the persistent file in terms of the handle. Based on the information in the work order, the Work Dispatcher dynamically creates the required data sets and invokes the appropriate load module to perform the task. The load module writes the outputs to a repository. It also writes the execution results to a data set that shows whether the task completed normally or not.

During the process, all the requests are stored in a permanent status file and their statuses are timely updated by the server and Work Dispatchers. Figure 3 shows the possible states of a request. A request is in the ‘start’ state, as the client sends the request to the server. The request enters the ‘wait’ state, as the server sends it to the Work Dispatcher. The request enters the ‘process’ state, as the Work Dispatcher invokes the appropriate function of the legacy system to perform the task. Once the request is completed, it enters the ‘done’ state. However, the client can cancel the request before the Work Dispatcher invokes a function of the legacy system to perform the task. As long as the request is in the ‘cancelled’ or ‘done’ state, the client can delete the request permanently. If the request enters the ‘deleted’ state, then its resources are freed for use by another client.

4.3.3.3 Client-Side Adapter

A client-side proxy is created when a server’s interface is compiled using the IDL compiler. Without making the client-side proxy as a separate and independent component, the client code needs to be recompiled and relinked whenever a new client-side proxy is generated. Wrapping the client-side proxy into an adapter component to the architecture decouples the clients and the Server. The Client-Side Adapter component was implemented using the Envelope/Letter idiom [24]. Implementing the Adapter in terms of the Envelope/Letter idiom offers greater flexibility, better performance, and reduced impact-of-change for code [24]. The major benefit is that we separate the client code from the client proxies by making the Adapter into a DLL executable file. The separation reduces the impact on the client code and recompilation cost of the client code.

4.3.4 Tool Support

A set of tools tailored for this architecture was built on an IBM platform with MVS/OMVS to start, shut down, query, and monitor the Analyze Server, the Repository Server, and Work Dispatchers in the background without an active TELNET/RLOGIN session. The activities of each tool are logged. This is accomplished by ‘redirecting’ the stderr and the stdout to files on the OMVS system. As the tools respond with messages to the ‘display’, the message are written to the stdout file. The error messages generated by the tools are also written to the stderr file on the OMVS system. The log files are automatically created each time the server is started. User name is prefixed to all the log files. If the old log files are to be preserved and reviewed at a later time, the old log files need to be renamed manually before the Application Server is started. All the log files should be reviewed for errors and purged by clients. The

![Figure 3. State diagram of a request.](image-url)
console log files for the started tasks are created at startup. The server generates a timestamp whenever it is initiated or shut down. The timestamp is always routed to the console. Any serious messages such as a server break down are also displayed on the operator’s console.

A global environment file contains the attributes for the environment, the servers, and the Work Dispatchers. Before a server is started, the environment variables setting will be verified. Once the tools have been installed, the global environment file will be the only place where changes will need to be made when a new environment for the servers is configured. The values of the environment variables are only local to the environment. Therefore, clients can create their own environments for their own use without affecting the others.

There are 6 JCLs provided to the operators to start, terminate and monitor the Analyze Server, the Repository Server and Work Dispatchers. For example, in Figure 4, the JCL SANLZ is provided to start the Analyze Server. The JCL invokes a program SANLZC in C on OMVS to start a process running the Analyze Server in the background mode. If any debugging information messages or error messages are generated, then they are written to stdoutSA and stderrSA, respectively. Messages that need the operator attention are routed to the operator console. A job in JCL is also provided to monitor the performance of the Analyze Server and Work Dispatchers such as the number of requests completed in the last hour, the number of requests in process, the number of requests in waiting, etc. In principal, tools were designed to be used by the operators rather than the clients.

The JCL SANLZ is provided for an operator to start the Analyze Server. BPXBATCH provides a way of invoking a C program on IBM OMVS from the JCL. The debugging messages and error messages are routed to STDOUT and STDERR, respectively. A global environment file, namely global_env_file configures the environment. Users may set up their own attributes for their own use without sharing their environment with others. An example of a global environment file is shown in Figure 5.

Whenever a server is started, a timestamp is displayed on the operator’s console and also recorded in a log file as shown in Figure 6. Similarly, when the server is shut down, another timestamp is sent to the console and appended to the log file. Figure 7 provides an example showing the Analyze Server is started on September 1, 2000 at 10:19:09 and terminated on September 2, 2000 at 03:06:30 with the status 0.

4.3.5 Experience Learned from the Project

The entire architecture reengineering project was divided into three subprojects including User Interface, Analyze Server and Repository Server. User Interface subproject focused on the interactions between the clients and the servers, the Analyze Server on handling the clients’ requests and storing the analysis results to the repository, and the Repository Server on retrieving the information stored in the repository and displaying the information on the clients’ desktops.

There were 11–13 members including managers, software engineers, and technical writers conducting the project. The implementation of the Analyze Server includes 19 files in .cpp, .h, .i, .idl, and Pascal, with a total of approximately 16073 lines. The Client-Side and Server-Side proxies in C++ are automatically generated from the IDL. The entire implementation of the Analyze Server is operational in the IBM OMVS environment. The implementation of the Work Dispatcher written in C++ operates in the IBM MVS environment.

We conducted an experiment on the response time that is critical to the customers [23]. The response time is the sum of the time a request sent to a server by a client, the time the server allocates a unique handle for the request, and the time for the handle to travel back to the client. We recorded the response time in the scenario of

Figure 4. A JCL to start the analyze server.
multiple clients, three Work Dispatchers, and a single Analyze Server. The average response time for the 60 runs is about 0.06 sec in this scenario.

Our experience indicates the facelift approach is an effective way of encapsulating legacy systems for use in heterogeneous and distributed environments compared to the rewrite and convert approaches. We took approximate 4 months to learn the fundamental use of CORBA. The implementations of the Analyze Server and the Repository Server took another 4 man-months approximately, including the unit testing. The tool development took almost 1.5 man-months. The approach also provides a low risk for reengineering legacy systems. The original system is still kept in its current environment. Since so little is changed compared to the changes made by the rewrite and convert approaches, there is hardly any risk. Thus, the approach may not be the best approach for reengineering purpose; but it is definitely more effective than the rewrite and convert approaches for short-term range. Similar claims can also be found in [4,11].

The architecture has limitations. Our AMI model does not provide any notification to the client of any

---

**Figure 5.** A global environment file.

```encore_debug=1
Recovery=0
OrbDebug=1

#AnalyzeServer
AnlzSrvPath=/u/devl/servers/VAnalyzeServer
AnlzSrvLogPath=/u/devl/LogFile
AnlzSrvLogName=anlzSrvlog
AnlzSrvIORPath=/u/devl/LogFile
AnlzSrvIORName=as.ior
AnlzSrvAddress=198.160.65.5
AnlzSrvPort=10026
AnlzSrvDebug=1

#RepositoryServer
RepositorySrvPath=/u/devl/servers/RepositoryServer
RepositorySrvLogPath=/u/devl/LogFile
RepositorySrvLogName=repositorySrvlog
RepositorySrvIORPath=/u/devl/LogFile
RepositorySrvIORName=test.ior
RepositorySrvAddress=198.160.65.5
RepositorySrvPort=10024
RepositorySrvDebug=1
RepositorySrvOrbDebug=0

#Work Manager Driver
WkrDrvPath=/u/devl/Package
HigherQualifier=RMJCCC2
CobolLoadLib1=RMJCCC2.DEVL.XALOAD
CobolLoadLib2=RMJCCC2.PROD.XALOAD
# DO NOT WRITE JCL NAME in JobCard1
# Replace the jobcards specific to your shop
JobCard1="JOB (RMJCCC292300,SRT,00),RMJCCC
SWRK;CLASS=W,*"
JobCard2=""/
TIME=1440,PRTY=6,MSGCLASS=X,NOTIFY=&SYSUID"
JobCard3=
JobCard4=
XXXXXX_start_wkrdrv_sleep=20
WkrDrvDebug=1

#Work Manager
NumOfWkrMgr=3
WorkOrderQueue=/u/devl/LogFile/workorder
LOCKFILE=/u/devl/LogFile/lock
FIFO=/u/devl/LogFile/fifo

#Analyze output file allocation
# GenericUnit: Unit name of the device that the
data set will reside on.
# VolumeSerial: Volume serial number of the
device a data set will reside on.
# SpaceUnit: Unit of the space allocation for a
data set. Values are: Cyls and Trks.
# PrimarySpace: Primary space allocation for a
data set.
# SecondarySpace: Secondary space allocation for
# a data set.
# SMS: Yes or No. Specifies yes if data set
allocation using SMS.
# StorageClass: Specifies the storage class of
# system managed storage.
# ManagementClass: Specifies the management class
# of a data set.
# DataClass: Specifies the data class of a data
# set.
GenericUnit=SRTDA
VolumeSerial=SRT822
SpaceUnit=Cyls
PrimarySpace=2
SecondarySpace=1
SMS=no
StorageClass=""
ManagementClass=""
DataClass=""
```

Figure 6. A timestamp the analyze server started.

Figure 7. Timestamps the analyze server started and terminated.

```
completed request. The client needs to check the results by sending a request with the handle to the server. Checking the execution results with the handle is definitely a disadvantage of using this architecture. Currently, an automatic way to get the client notified with the execution results via e-mail is under development. In addition, the architecture does not have the ability to resist unauthorized attempts at usage and deny service while still providing its services to legitimate clients. Also, forking child processes for executing the Work Dispatchers incurs much overhead. This can be resolved by creating threads in the next release. Finally, replacing the CORBA implementation with a new one may cause the failure of the entire system. The integration of heterogeneous CORBA implementations is still an issue in the middleware community.

One problem of using free open source for implementing systems is that the open source may not fully support the features yet. For example, one problem we encountered during the development of the project is that the open CORBA implementation we adopted has not implemented the asynchronous invocation yet. And it was very difficult for us to understand the code and add the features to it. In this project, we developed our model for asynchronous requests using the synchronous invocation provided in the CORBA implementation. It is suggested that the supported features of a free open software are fully explored and understood before the software is adopted for implementing systems.

Basically, the new CORBA specification that includes asynchronous calls on the proposed architecture should not affect the implementation of our architecture as long as the IDL is not modified. However, if this is not the case, the new IDL needs to be recompiled to generate new client-side proxy and server-side proxy. The implementations of the client-side Adapters and the servers are then recompiled with the new proxies.

5. Conclusion

Legacy systems were usually developed for centralized environments. These systems are usually inflexible and non-scalable. In addition, with the advent and widespread use of object-oriented and client-server technologies, companies expect their legacy systems to take advantage of these new technologies and cooperate with their heterogeneous systems and environments.

There are three traditional approaches to modernizing centralized legacy systems: convert, facelift and rewrite. The first approach is to convert the legacy systems from older computing environments to modern computing languages, platforms, and architectures. The convert approach usually uses tools to facilitate the conversion process. Unfortunately, the process is still tedious and risky. The second approach is to facelift the legacy systems by means of middleware without major changes in the systems. This approach is generally only a stopgap solution, rather than a strategic one. However, from an economical point of view, the second approach is a more cost-effective way of modernizing legacy systems by using middleware. The third approach is to rewrite the entire system from scratch. The cost and time requirements with the rewrite approach are usually unacceptable for most businesses.

In this paper, we presented an industry project in which an extended facelift approach was applied to encapsulate an industry software reengineering system on an IBM mainframe into a heterogeneous and distributed environment. A set of tools was developed to facilitate remote access to the legacy systems on the IBM mainframe. We think software vendors are eager to preserve their products in the market. It is to be expected that more and more companies will be more willing to use the facelift approach for a short-term solution to migrating their legacy systems into a new architecture rather than using the rewrite and convert approaches due to the low cost and risk of the facelift approach. Therefore, we presented our experience of implementing an extended facelift approach and lessons learned from the project in this paper.

References


Manuscript Received: Mar. 1, 2006
Accepted: Feb. 5, 2007