



RENAISSANCE

*Method and tool support for the
evolution and re-engineering of legacy systems*

CLIENT/SERVER MIGRATION

Guidelines for Migration from Centralised to Distributed Systems

©RENAISSANCE Consortium 1997

Version 1.0, First published April 1997

The RENAISSANCE project is partially funded by the European Commission under the Framework Initiative (ESPRIT 22010). The objective of the project is to develop a systematic method to support the re-engineering of legacy systems. Further information about the project is available on the World-Wide-Web at URL:

<http://www.comp.lancs.ac.uk/projects/renaissance/>

The members of the RENAISSANCE Consortium are:

CAP Gemini Innovation (Mr Alain Dineur)

Bâtiment Karélian
7, chemin de la Dhuy
38240 Meylan, FRANCE
Tel: +33 476 76 47 47; Fax: +33 476 76 47 48

CAP Gemini ISM (Mr Alain Paoli)

Tour Anjou
33 Quai de Dion Bouton
92814 PUTEAUX Cedex, FRANCE
Tel: +33 1 41 26 63 36; Fax: +33 1 41 26 52 17

debis Systemhaus GEI GmbH (Mr Markus Breuer)

Pascalstraße 14
D-52076 AACHEN, Germany
Phone: +49 2408 943 0; Fax: +49 2408 943 119

INTECS Sistemi S.p. A. (Mr Giancarlo Savoia)

Via Livia Gereschi, 32
56127 PISA, Italy
Tel: +39 50 545 111; Fax: +39 50 545 200

Telesoft S.p. A. (Mr Fabio Mungo)

Via Degli Agrostemi, 30
00040 SANTA PALOMBA (Roma), Italy
Tel: +39 6 710 551; Fax: +39 6 710 553 50

Engineering - Ingegneria Informatica S.p. A. (Mr Dario Avallone)

Via dei Mille, 56
I-00185 ROMA, Italy
Tel: +39 6 522 431; Fax: +39 6 522 432 48

Lancaster University (Prof. Ian Sommerville)

Computing Dept,
Bailrigg, LANCASTER LA1 4YR, UK
Tel: +44 1524 593795; Fax: +44 1524 593608

SINTEF (Prof. Reidar Conradi)

O. S. Bragstads plass 2 F
N-7034 TRONDHEM, Norway
Tel: +47 73 593 444; Fax: +47 73 594 466

Executive Summary

The migration of legacy systems to distributed, client/server architectures is a key problem of system re-engineering. Many systems to be re-engineered have been developed on mainframe systems. Customer pressure is for more responsive, distributed systems using PCs as clients and there are major technical challenges in supporting this migration.

A method and practical guidelines to evolve legacy systems towards evolutionary systems are needed to support the complex task of system evolution. These guidelines must cover several areas, ranging from business to technical considerations. Technical recommendations are necessary to support the selection of the target architecture and of enabling mechanisms, the definition of the steps to migrate towards this solution, and practical guidelines for the use of the selected mechanisms.

The objectives of this document are to give technical guidelines for the migration of legacy systems to distributed, client/server, architectures.

The recommendations given in this document provide guidance to a system architect looking for possible target architectures with a specific focus on reuse, integration, encapsulation and migration of existing system components or parts. Issues like the extension of a legacy system architecture and the integration of legacy system components are specifically addressed.

In addition to these conceptual recommendations, a list of techniques and tools to build such evolutionary architectures is presented. This list gives practical advice about the use of these techniques to build such architectures and gives guidelines to support their selection.

Table of Contents

1 Introduction	1
1.1 Objectives	2
1.2 Rationale.....	2
1.3 Overview of the Document Structure	2
2 Distributed System Architectures.....	4
2.1 Distributed Architecture Models	5
2.1.1 Two-Tier Software Architecture Models	5
2.1.2 Three-Tier and Multi-Tier Software Architecture Models.....	7
2.1.3 Distributed Objects Architecture Model.....	9
2.2 System Integration Models.....	11
2.2.1 Data Based Integration	12
2.2.2 Service Based Integration.....	15
2.2.3 Presentation Based Integration	18
2.2.4 Component Based Integration.....	20
2.2.5 Workflow-Based Integration.....	26
2.2.6 Web Based Integration	28
3 Integration Scenarios	33
3.1 Introduction	34
3.2 Migration to Graphical User Interfaces.....	34
3.3 Migration to Web-Based User Interfaces	35
3.4 Application Extension.....	36
3.5 Application Extension to Internet	36
3.6 Application Workflow Automation.....	37
3.7 Application Workflow Integration.....	38
3.8 Application Distribution	39
4 Migration Techniques and Tools.....	45
4.1 Introduction	46

4.2 Technical Evaluation Criteria.....	46
4.2.1 Environment Constraints.....	46
4.2.2 Technical Constraints	47
4.3 Data Based Integration	51
4.3.1 Data Encapsulation and Access.....	51
4.4 Service Based Integration.....	59
4.4.1 Messaging and Queuing Middleware.....	59
4.4.2 Transaction Processing Monitors.....	63
4.5 Presentation Based Integration.....	69
4.5.1 World-Wide-Web User Interfaces to Legacy Systems.....	69
4.6 Component Based Integration.....	72
4.6.1 CORBA	76
4.6.2 OLE and DCOM	80
5 Appendix A: Link with the Renaissance Project.....	85
5.1 Covering of Requirements from Problem Focusing	86
5.2 Mapping to the Technology Selection.....	88
6 Appendix B: References	89
7 Appendix C: Glossary.....	90

List of Figures

Figure 1: Remote Presentation Two-Tier Architecture.....	6
Figure 2: Remote Data Management Two-Tier Architecture	6
Figure 3: Distributed Logic Two-Tier Architecture	7
Figure 4: Generic Three-Tier Architecture	8
Figure 5: Distributed Objects Architecture.....	10
Figure 6: System Integration Models	11
Figure 7: Simple Data Centric Architecture.....	12
Figure 8: Direct Legacy Data Access.....	13
Figure 9: Integration via Meta-Data.....	14
Figure 10: Centralised Service Based Integration	16
Figure 11: Distributed Service Based Integration	16
Figure 12: RPC based Communication	17
Figure 13: Message based Communication.....	17
Figure 14: Presentation Protocol.....	19
Figure 15: Screen Scraping.....	19
Figure 16: The Software Bus	21
Figure 17: A Typical Component-based Software Architecture.....	22
Figure 18: Workflow Reference Model	27
Figure 19: Basic Web Architecture	29
Figure 20: CGI based Web Architecture.....	30
Figure 21: Dynamic Pages Web Architecture	31
Figure 22: Web Shortcut Architecture	31
Figure 23: Migration to Graphical Interfaces	35
Figure 24: Application Extension.....	36
Figure 25: Application Extension to Internet	37
Figure 26: Application Workflow Automation	38
Figure 27: Application Workflow Integration	39
Figure 28: Component Heterogeneity, Interoperability and Granularity	41
Figure 29: Integrating Components with Data	42
Figure 30: ODBC General Architecture.....	52
Figure 31: IMS Database Access	54
Figure 32: Open DM Architecture	57
Figure 33: Messaging and Queuing Principle.....	59
Figure 34: MQSeries Interface	62
Figure 35: Three-Tier TPM Architecture	64
Figure 36: Three-Tier and TPM Architecture	64

Figure 37: Architecture of Encina Monitor	66
Figure 38: Web User Interface to Legacy Systems	70
Figure 39: Removing Dependencies on Legacy Code.....	73

List of Tables

Table 1: Benefits and Risks of Component Technology	23
Table 2: Evolution Strategies.....	34
Table 3: List of Techniques and Tools Reviewed	46
Table 4: Data Encapsulation and Access Technologies.....	51
Table 5: CORBA contact sites.....	80
Table 6: OLE Contact Sites.....	84
Table 7: Covering of Requirements from Problem Focusing.....	87
Table 8: Mapping to the Technology Selection.....	88

1 Introduction

Contents

- 1.1 Objectives
- 1.2 Rationale
- 1.3 Overview of the Document Structure

Summary

This chapter introduces the document, describes its contents and the intended readership, and provides guidance on how to read it.

1.1 Objectives

The objectives of this document are to give technical guidelines for the migration of legacy systems to distributed, client/server, architectures.

The recommendations given in this document provide guidance to a system architect looking for possible target architectures with a specific focus on reuse, integration, encapsulation and migration of existing system components or parts. Issues like the extension of a legacy system architecture and the integration of legacy system components are specifically addressed.

In addition to these conceptual recommendations, a list of techniques and tools to build such evolutionary architectures is presented. This list gives practical advice about the use of these techniques to build such architectures and gives guidelines to support their selection.

These recommendations are based on the requirements coming from applications selected for their needs for evolution and are listed in section 5.1 *Covering of Requirements from Problem Focusing*.

1.2 Rationale

The migration of legacy systems to distributed, client/server architectures is a key problem of system re-engineering. Many systems to be re-engineered have been developed on mainframe systems. Customer pressure is for more responsive, distributed systems using PCs as clients and there are major technical challenges in supporting this migration.

A method and practical guidelines to evolve legacy systems towards evolutionary systems are needed to support the complex task of system evolution. These guidelines must cover several areas, ranging from business to technical considerations. Technical recommendations are necessary to support the selection of the target architecture and of enabling mechanisms, the definition of the steps to migrate towards this solution, and practical guidelines for the use of the selected mechanisms.

1.3 Overview of the Document Structure

This next chapter of this document, *Distributed System Architectures*, gives a conceptual presentation of distributed architecture models and system integration models. The list of distributed architecture models presents the most high-level classification of architectures of distributed systems according to application software layers (or tiers). The description of system integration models extends this latter list by describing various models of integration between application software layers.

This chapter provides the necessary background information for understanding the architecture of distributed systems. This basic understanding is required by the following chapters.

The following chapter, *Integration Scenarios*, describes several architectural scenarios to build a system architecture integrating legacy system components with new components. Each integration scenario includes corresponding system architecture models and references to system integration models presented earlier.

This chapter is especially dedicated to technical decision makers which must identify and assess possible architectures and evaluate the benefits and concerns of the various solutions.

Various techniques supporting the migration of legacy systems according to the targets and paths presented before are listed and assessed in the last chapter, *Migration Techniques and Tools*. For each technique, a general presentation, practical advice of use, tool support and benefits and concerns are reviewed.

This last chapter is a collection of techniques and must be used as such if they are applicable for such or such migration scenario. This collection is not exhaustive and cannot anyway be exhaustive since new technology enablers are announced and available every month or week. However, it provides a good starting point and review of the existing, current possibilities.

2 Distributed System Architectures

Contents

- 2.1 Distributed Architecture Models
- 2.2 System Integration Models

Summary

This chapter gives a conceptual presentation of distributed architecture models and system integration models. The list of distributed architecture models presents the most high-level classification of architectures of distributed systems according to application software layers (or tiers). The description of system integration models extends this latter list by describing various models of integration between application software layers.

It provides the necessary background information for understanding the architecture of distributed systems. This basic understanding is required by the following chapters.

2.1 Distributed Architecture Models

Following the evolution of needs and available technologies, the architecture of distributed systems has evolved from simple, monolithic, models to complex, highly distributed, models. Terminology evolution followed these trends and *collaborative architectures* or *distributed architectures* now replace the more restrictive *client/server architectures* when describing system architectures.

When designing an application, its software architecture, also called logical architecture, is usually based on communicating software layers, like a *presentation layer*, in charge of the interface with the end-user, using services provided by an *application layer*, using in turn data managed by a *data management layer*.

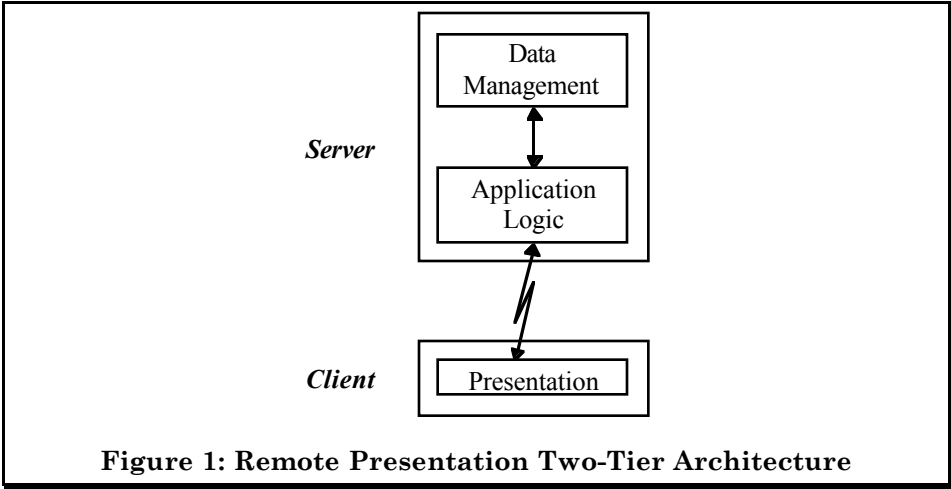
Software architectures can be considered independently of the physical architectures on which they are mapped, i.e. the physical hardware platforms. A distributed software architecture can indeed be implemented on a mainframe architecture. However, only distributed software architectures are prepared for physical distribution and implementation on a distributed physical architecture.

Depending on the number and types of layers identified or on their distribution on the target computers, application architectures can be classified according to several architecture models. We describe in this section the three main architecture models:

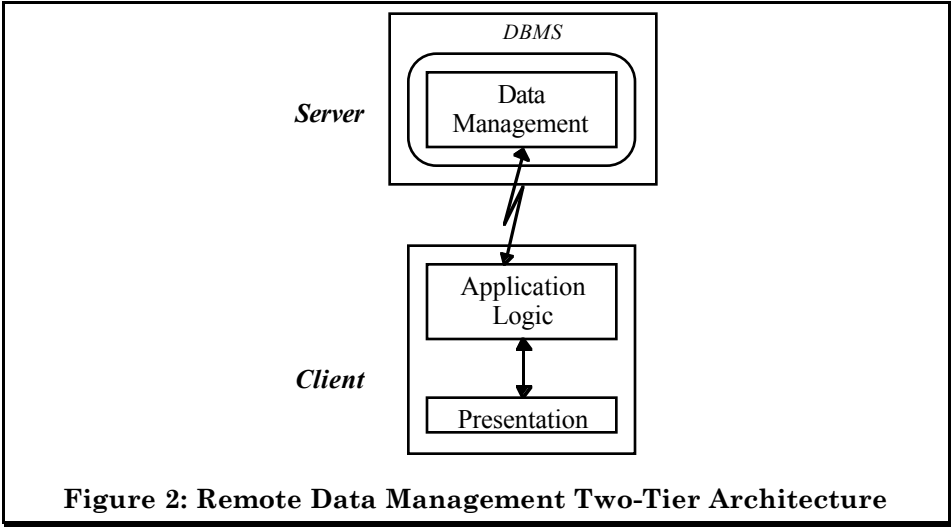
- Two-Tier Software Architecture Models
- Three-Tier and Multi-Tier Software Architecture Models
- Distributed Objects Architecture Model

2.1.1 Two-Tier Software Architecture Models

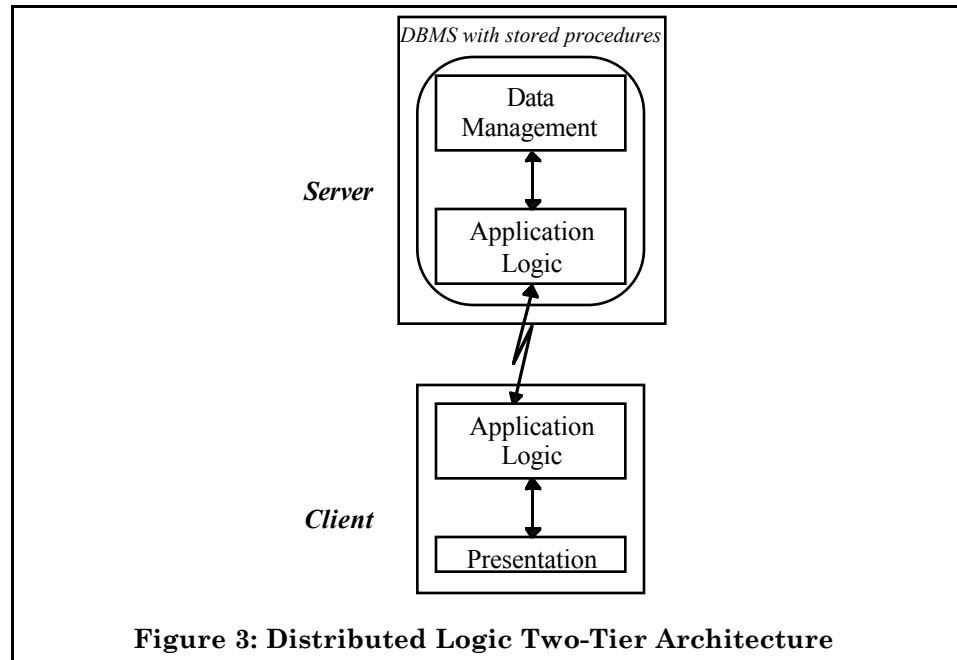
In the early 1980s, distribution was limited to the presentation layer, leaving the application logic and data on the host. The emergence of tools and technologies like terminal emulators and X-Windows allowed user interfaces with a limited intelligence, or rather control, to be deployed on users desks. This was the beginning of *two-tier architectures* also called first-generation client/server architectures with the server tier consisting of the application logic and data and the client tier consisting of the remote presentation:



With the emergence of productivity development tools like Visual Basic or PowerBuilder, application development came to the desktop allowing applications to access a centralised database. These developments were limited to small-scale or workgroup applications. This was the second generation of two-tier architectures with the server tier consisting of the database and the client tier consisting of the application logic usually tightly integrated with the presentation (*fat-client*):



The evolution of database management systems, including stored procedures, and the need to implement the main business rules close to the enterprise data, led to a new two-tier architecture where application logic is partly located on the server and partly located on the client computer:



Most two-tier architectures, especially in the latter form, satisfy most requirements for workgroup or small-scale applications. Major benefits of these architectures are:

- Large availability of development tools.
- Development tools provide and integrate all necessary technical components: database, communication middleware, ...
- Development and maintenance ease and cost-effectiveness for small-scale applications.

However these architectures suffer from the following drawbacks:

- Scalability is limited:
 - the DBMS or the server becomes a serious bottleneck in case of important traffic or large number of users.
 - the development and maintenance costs increase significantly with the complexity of the application.
- The application is tight to the development environment and vendor.
- The flexibility of the application is limited since the choice of the overall architecture, communication model, middleware, database and programming language is enforced by the development environment.

2.1.2 Three-Tier and Multi-Tier Software Architecture Models

Many problems of using two-tier architectures for medium or large-scale applications are related to the inflexibility of allocating the application logic to software layers. The application logic can roughly be composed of:

- Data Management logic: in charge of basic data management of integrity rules

- Processing Logic: implementing the business rules of the application
- User Interface Logic: in charge of basic user interaction control and more elaborated presentation management (interaction with presentation manager)

The Data Management Logic is very close to the definition of data and it is usually better to integrate it tightly with the data management system. A good way of implementing data management logic can be to implement it as stored procedures.

The User Interface Logic is also close to the presentation manager and it is better to integrate it tightly with a presentation manager. This is usually done by implementing this logic directly at the client level.

The Processing Logic is the most volatile and evolutionary part of an application and represents the best potential for effective reuse. However, in two-tier architectures, this processing logic is either tightly integrated with the data management layer or with the presentation layer. Modifying the processing logic is therefore more difficult.

In addition, when the application grows, traffic between the server and its clients usually makes the server or the network connection with the server the performance bottleneck of the whole application. Even if upgrading the server hardware or the network connection bandwidth can temporarily solve the problem, scalability remains limited.

The *three-tier architecture* model (also called *second generation client-server architectures* or *application server model*) defines a new architecture model to find answers to these problems. In this architecture model, application logic is not tightly connected with either the data management or presentation and is implemented in a separate layer. The physical distribution of these software layers can be done on several types of physical architectures, 2-tiered hardware architectures (clients and server), or 3-tiered hardware architecture (clients, application servers, database servers):

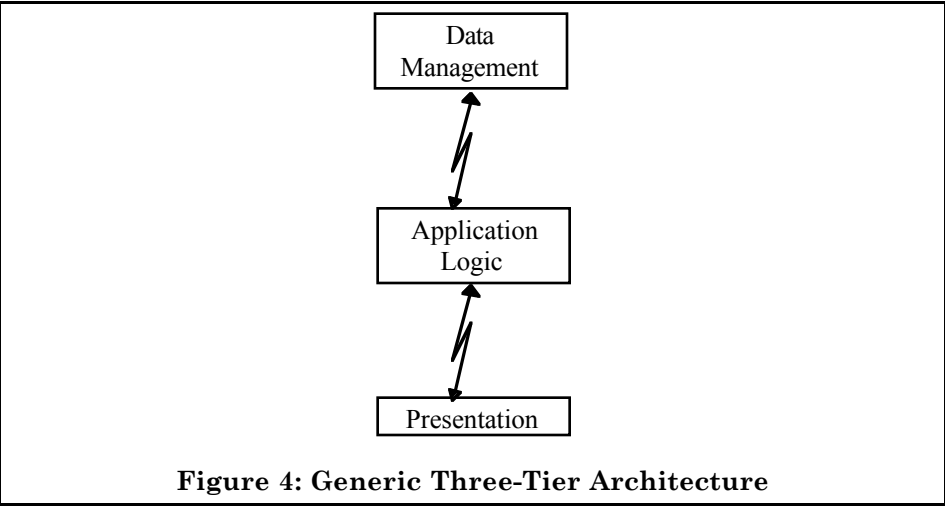


Figure 4: Generic Three-Tier Architecture

The multi-tier architecture model is an extension of the three-tier architecture model where the application layer is decomposed in turn into sub-layers.

The benefits of three-tier and multi-tier architectures are:

- Scalability,
- Transparency,
- Availability,
- Response time,
- Throughput,
- Reliability.

These benefits can be achieved by statically or dynamically distributing processing between application and database servers according to the various server loads or availabilities and client requests.

In addition, these architecture are usually less proprietary since interfaces between the various layers are also usually compatible, or gateways are available, with standard interfaces and communication protocols (e.g. ODBC, CORBA, DCE...).

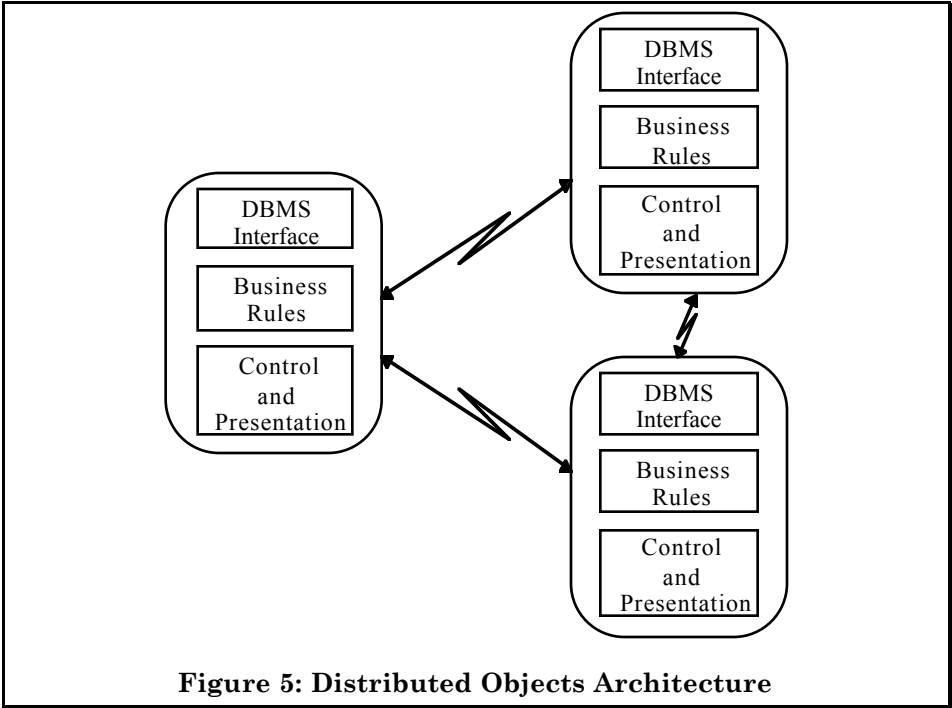
This makes three-tier or multi-tier architecture more suited for medium to large scale applications.

However, the development of such architectures may be complex because of the relative immaturity of development environments, compared to environments dedicated to the development of two-tier applications, and also because of the more complex task of architecture definition.

2.1.3 Distributed Objects Architecture Model

Architectures presented in the previous sections are based on several tiers or layers where lower layers (close to the presentation layer) use upper layers (close to the data management layer). Every layer is client and/or server of another specific layer.

In distributed objects architectures, the model is based on co-operating objects. Objects use services provided by other objects and can embed or be tightly connected with the management of their own data and own presentation. Objects can be distributed independently and communication is transparent. In a pure distributed object architecture, an application can be composed only of distributed objects. In a more general case, distributed objects may interact with more classical software layers like database or presentation management:



Distributed object architectures bring the best potential for reuse at a business level (business objects) rather than at a purely technical level (technical components).

However, distributed object technology still suffers from a relative immaturity, or at least, it is not widely used on a large scale. The emergence of Internet technologies integrated with software buses like CORBA and the emergence of object based transaction processing monitors are currently their best promoters.

2.2 System Integration Models

Distributed software architectures presented in the previous section are based on several integrated software layers. Depending on the nature of these software layers and on their dependencies and relationships, several integration techniques can be appropriate. Every integration technique defines a corresponding system integration model. This section reviews several existing system integration models:

- Data Based Integration
- Service Based Integration
- Presentation Based Integration
- Component Based Integration
- Workflow-Based Integration
- Web Based Integration

All these integration models overlap each other and they are not mutually exclusive, they are rather complementary. For instance, component based integration can be used as an encapsulation of another integration model. In addition, integration models like workflow based integration and web based integration are not self sufficient and usually rely on another, lower level, integration model:

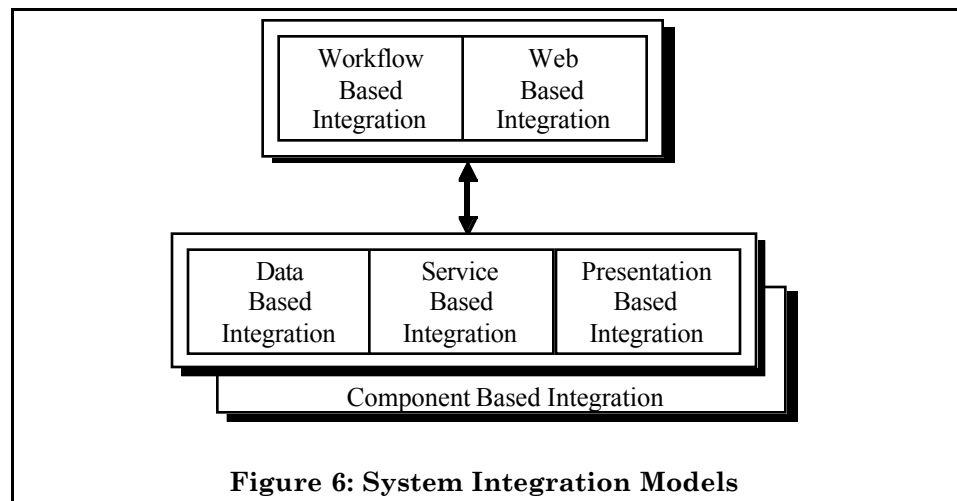


Figure 6: System Integration Models

Because of these relationships between system integration models, several models can be used when building a system.

2.2.1 Data Based Integration

2.2.1.1 Rationale

Data centric models are architectural models where communication among system components is mainly implemented through access to data stored in common archives. This typically works having components that input data information and other that use this information to elaborate other data (e.g.: a component providing a mask enabling to enter the personal data of enterprise customers and to store this information in a common archive, and other retrieving and using this information for enabling additional computations).

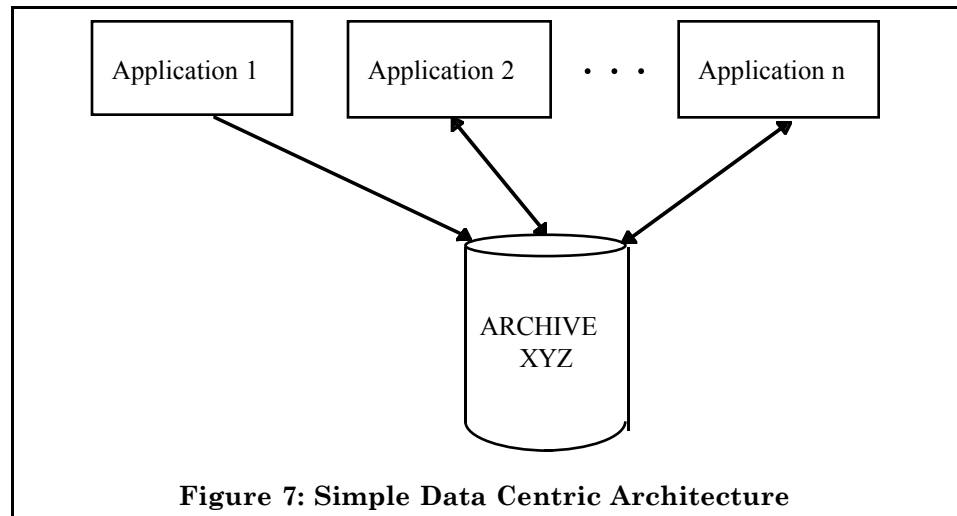


Figure 7: Simple Data Centric Architecture

The schema shown in Figure 7 provides a simplified case (in terms of number of shared data and archives) of a data centric architecture. Nevertheless, the example is not that distant from the architectural styles usually adopted, and still in place, for legacy systems.

This stated, we could define legacy system architectures as “largely data centric” in that a consistent part of the communication among the information system components (applications) is, in fact, based on shared data sources.

This kind of architectural model imposes strong constraints to the evolution of legacy application in that the evolution of an individual application must preserve it. In other terms, the evolution of a legacy application must keep into account, and preserve, the original data flow between the evolved application and the other system components in order to guarantee that data originally made available from the other system components remain accessible from the evolved application and that the evolved application continues to produce shared data in a format that is accessible from the other system components.

Data based integration models provides operational solutions to keep the data flow consistency between the different applications and, therefore, cover a central technical theme in software evolution.

2.2.1.2 General Description

A major problem with data centric systems, and in particular with legacy systems, is that it might be complex to isolate individual modules (applications) that can be extrapolated from the others without risking to affect the reliability of other components.

For this reason, the evolution of a legacy application requires that all information (data) that it shares with the other applications is precisely identified and that data flow among them is preserved.

In other terms, the evolution of the legacy application must take into account the need to integrate it with the legacy components in order to preserve the previous level of inter-relationship.

Given the possible (and desirable) technical and technological distance between the evolved and the existing applications, as well as the nature of the problem and the original integration policy, the less intrusive way to achieve the mentioned need is to integrate the evolved application at data level.

To do this there are substantially two architectural solutions based on different technical options:

The first one, that we name “direct data access” is based on the principles that legacy data, independently from their model, can be directly accessed from the evolved application through specifically designed data access software layers (data access middleware). These software layers provide translation facilities between application data access functions (used by the evolved application) and the underlying data source access primitives.

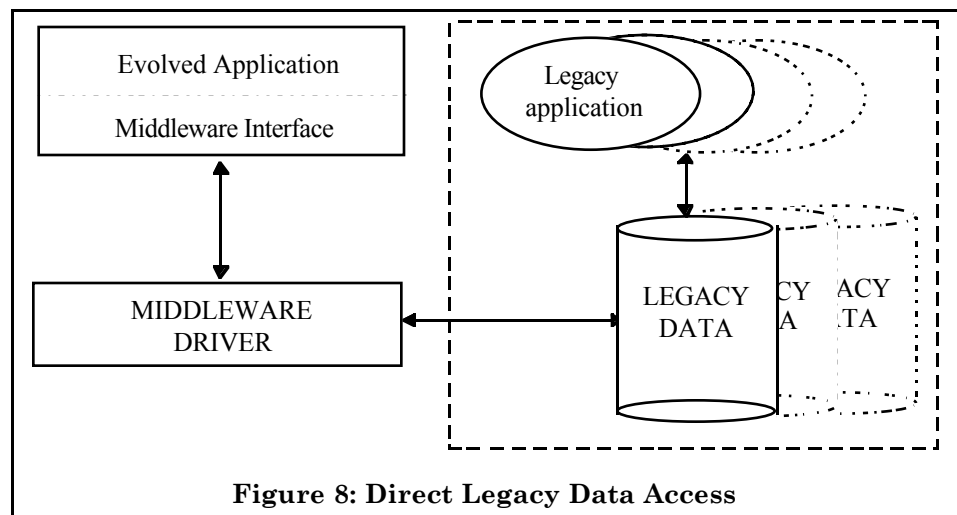


Figure 8: Direct Legacy Data Access

Figure 8 shows a sample implementation of the mentioned integration model where the evolved application invokes a middleware driver for a specific data source that may be relational or non-relational.

This integration approach allows the evolved application to access different kinds of data models and data sources in a uniform and transparent manner. Nevertheless, the adoption of such integration model requires that the evolved application “knows” which is the format and the

location (the archive, data model and structure) of the data to be accessed and takes care of the mapping between legacy data format and locally treated data format. In other terms, the adoption of this integration model requires the evolved application to keep in charge the access and manipulation modalities of both legacy and local data as well as to guarantee the two ways matching between legacy and local data.

A second architectural approach, that we can name “encapsulation”, is based on the definition of an intermediate meta-schema that, partially or completely, reflects the legacy data models. This meta-schema unifies the data models of the different legacy archives that, in this way, become simple storage devices. Depending on the distance between the legacy data models and the meta model of the meta schema, it can be necessary to implement specific data mapping features.

Using this approach, legacy data information is encapsulated within the meta data and this allows the evolved application to access them in a transparent manner and through a uniform schema.

With this respect, the evolved application is free from having to deal with structural details of legacy data. The mapping work is not done directly by the application but provided as a service to the application by a specific software layer (data mapper).

Figure 9 shows a sample implementation of this architectural solution:

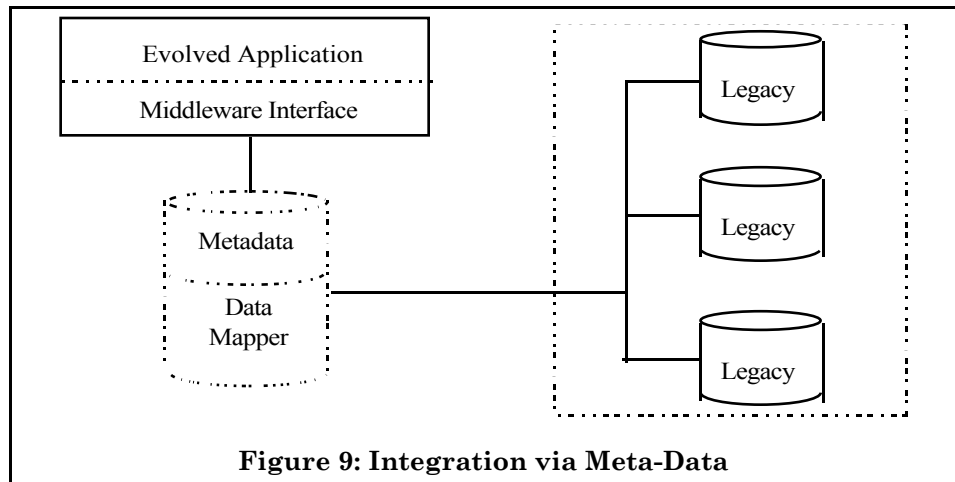


Figure 9: Integration via Meta-Data

Both the “direct data access” and the “encapsulation” integration models can be adopted regardless from the architectural solution adopted to implement the evolved application.

Moreover, the adoption of one technique, with respect to the other, is strongly influenced by the overall enterprise evolution objectives. So, if the objective concerns only a limited number of applications, costs and effort considerations may drive to choose the direct data access option that, in principle, is much more affordable.

If, on the contrary, the enterprise objective is to massively abandon the legacy operational environment, then data encapsulation option is surely

more adequate. This technique, in fact, guarantees a much higher flexibility and suitability for the following evolution projects.

2.2.1.3 Benefits/Risks of Approach

Data integration allows data flow between evolved and legacy applications to be preserved and is aimed at guaranteeing data consistency among them.

Although more costly, the “encapsulation” model seems generally more suitable to support extensive evolution strategy. Regardless from the clear advantage given by the possibility to avoid data mapping operations inside the application source (data access is virtualised at meta-schema logical), this approach allows to plan for data migration without having to change the evolved application.

Furthermore the adoption of this integration model allows to have a unified logical schema that encompasses either shared and locally treated data.

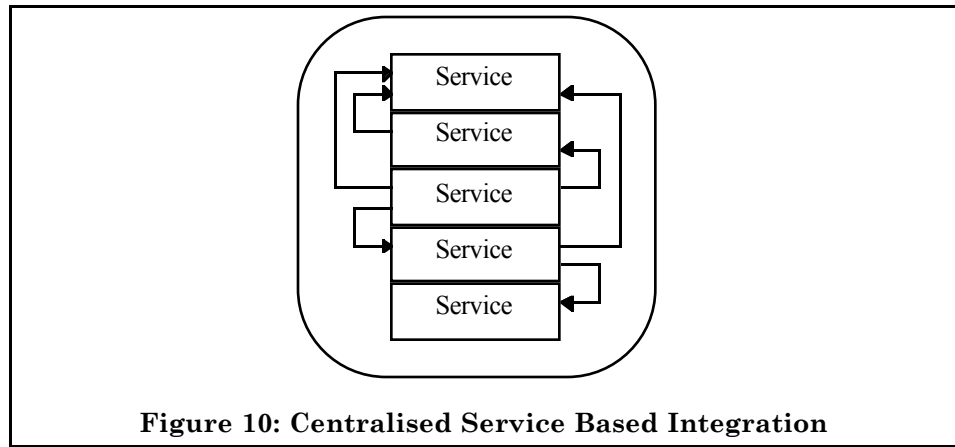
The “direct data access” model has its major advantage in the implementation effort that, compared with the “encapsulation” approach is expected to be much lower. On the other hand, this integration solution imposes strong constraints on the possibility to physically migrate data in that force to re-iterate changes to the evolved application code in order to eliminate data mapping processing. Furthermore, with this integration approach, in case we plan to physically migrate legacy archives (or even part of them), the work required to create a unified schema is still required.

As already mentioned in the previous section, a major problem accompanying the adoption of both data integration models stands in the definition of the data flow between the evolved application and the other applications and, eventually, in the definition of a conceptual schema capable to punctually reflect and map (possibly in a unified view) the structure of different legacy data sources. This is probably the major risk when performing data integration and this risk can be reduced with an accurate and detailed analysis of legacy data performed, whenever possible, by people having the proper skill of both the legacy data models and of the target one.

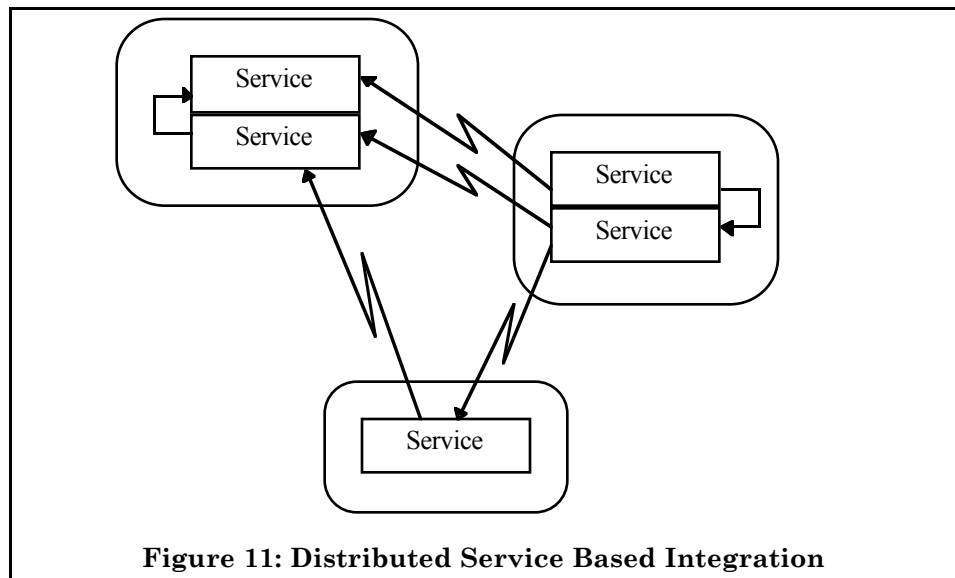
2.2.2 Service Based Integration

2.2.2.1 Rationale

Before the emergence of object technology, applications were only developed based on services (routines, procedures, functions...) calling other services to provide overall functions. Most legacy applications are still based on this unique principle. In traditional applications, services are tightly integrated with each other using linkers at application generation time. This integration model allows only centralised architectures to be built:



In distributed applications, services are offered on remote machines and distributed service based integration allows services to be defined and called independently of the location where the service is hosted, and in a way close or identical to a local call:

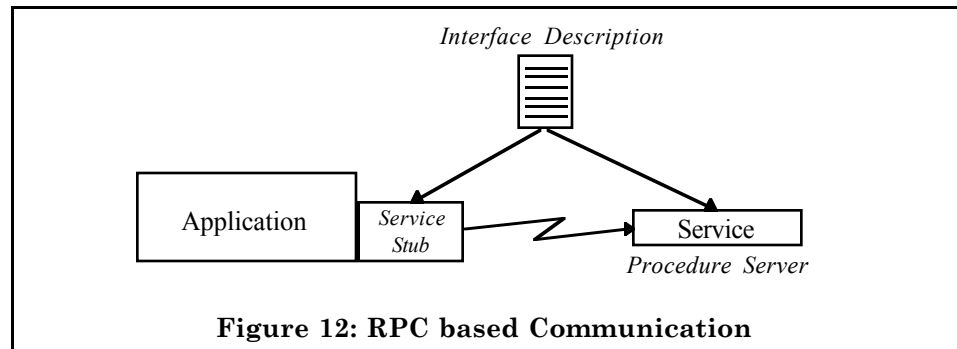


2.2.2.2 General Description

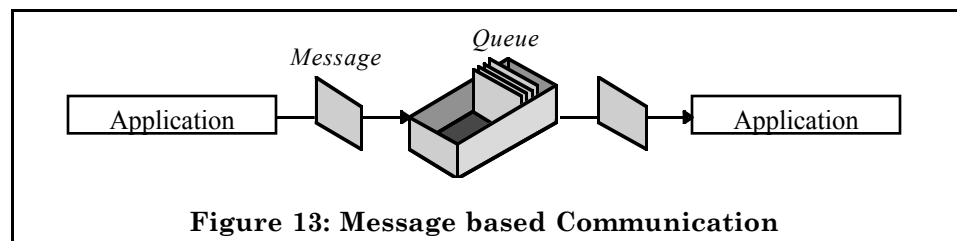
Service based integration is done through two major techniques:

Remote Procedure Calls (RPC) allows synchronous communication between applications by implementing the concept of calling a remote service as a procedure call. This is done using Procedure Servers which make it possible to implement remote procedures. Procedures interfaces are described using an Interface Description Language (IDL). These interfaces isolate the actual procedure from its callers by defining only the entry point and its required parameters. IDL allows services to be defined in a programming language neutral way. An IDL pre-compiler is in charge

of generating the interface (stub at the client level and service skeleton at the server level) which make it possible to call the procedure through the network:



Message and Queuing Oriented Middleware (MOM) allows asynchronous indirect communication between applications by exchanging messages. An application sends messages to a queue without communicating directly with another application. Other independent applications are in charge of processing messages. This allows a very loose integration between applications:



Although these two techniques differ considerably from the implementation viewpoint, they both allow applications to communicate remotely in a transparent way with regards to the location of calling and called applications or services. They also offer a way for standard applications, packages, tools or databases to offer an interface as a set of services.

2.2.2.3 Benefits/Risks of Approach

A major benefit of service based integration is that it allows applications to be developed as distributable services. Existing applications consisting of services can be open easily by providing a direct interface to these services.

As low level integration techniques, RPC and message based communication is available and a large set of hardware environments and operating systems with a unique and standard interface. This minimises the effort needed to develop an application for multiple platforms or port an existing application to a new platform.

However, there are still a lot of products implementing service based communication not necessarily compatible or inter-operable. The availability of some of them on a wider set of platforms tends to make them less proprietary. DCE as the standard RPC mechanism has not been widely adapted and there is no other standard. For message based communication, even if there are some leaders, there is no clear winner. Both techniques suffer from the wide industry battle for the middleware market.

2.2.2.4 Evolution/Future

Object based communication (as in CORBA and DCOM) seems to have more potential in the near future than RPC which is more positioned as a proprietary way to be integrated with standard packages and products. There are however still many proprietary implementations of RPC.

Message Oriented Middleware is more and more adopted as a mechanism to integrate in a safe and guaranteed way heterogeneous platforms, rather than being the architecture backbone of a system.

There is however no clear winner, compared to other integration techniques, and the battle for the middleware market brings more and more combination of integration techniques to provide solutions for all situations.

2.2.3 Presentation Based Integration

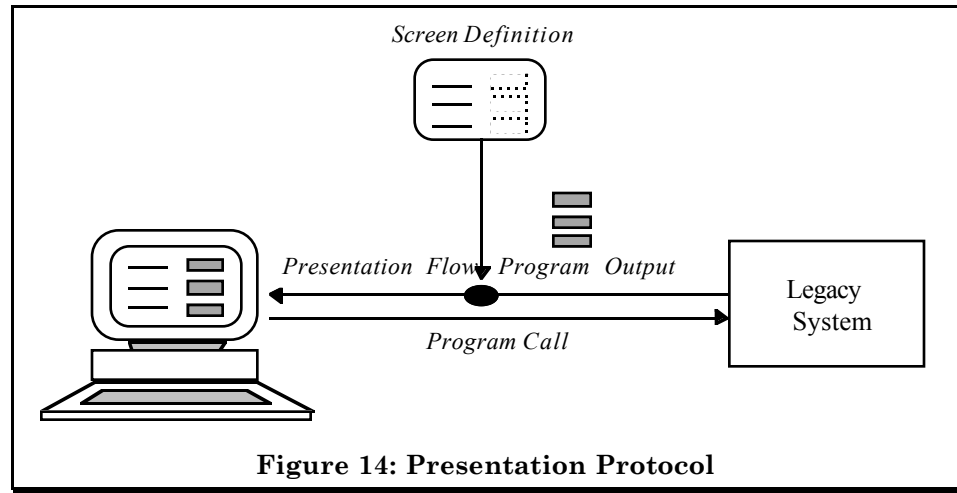
2.2.3.1 Rationale

Legacy applications are usually monolithic and running on a central host with users accessing it through dumb, character based, terminals. Remote communication between the central host and terminals is implemented using a protocol, usually proprietary. Integration with such applications can therefore be achieved easily by accessing them using this presentation based protocol.

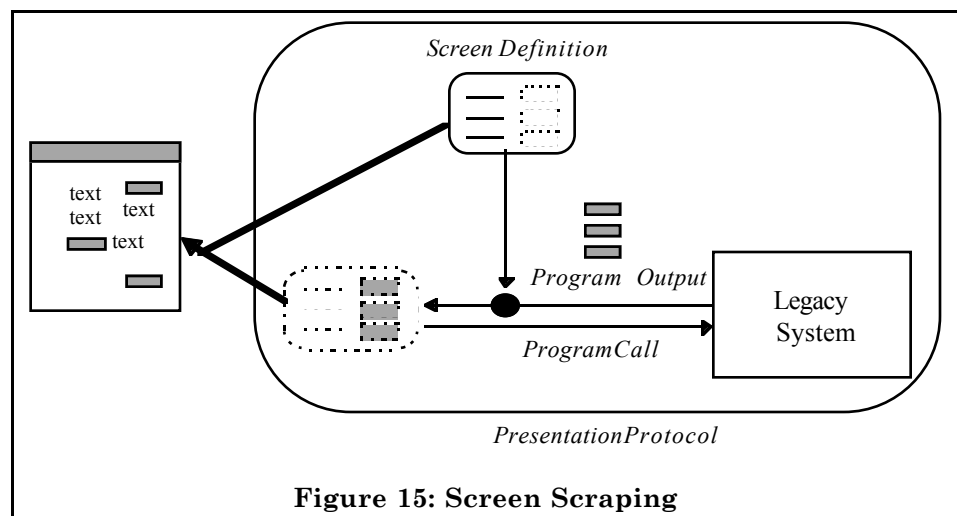
Dumb terminals can therefore be replaced by more powerful workstations on the users' desks, allowing them to reuse the existing investment by accessing legacy applications and at the same time to run more recent productivity applications or packages.

2.2.3.2 General Description

Presentation based protocols are usually based on screen definitions (screen masks) filled in with the results of a program or transaction (each field of the screen definition is filled in with a result value):



Presentation integration based techniques act as terminals and are based on the analysis of the presentation flow. By comparing the terminal screen output with the screen definition, such tools can extract and use data resulting from a program or transaction call. These techniques are the basis of *screen scrapers*. Data is then made available to another presentation routine which can format it differently, for instance to display it in a GUI environment or in an HTML based form, or use it differently, for instance as a row of data for ODBC drivers, or as variables which can be used in a script (e.g. CICS gateway for Java):



2.2.3.3 Benefits/Risks of Approach

Presentation based integration is a very lightweight solution which allows integration with an application without modifying anything of it.

This is however is short term solution for migrating only the presentation layer of an application on more modern computers, especially to allow

both old legacy applications and newer, GUI based, applications to be accessed.

There is therefore no fundamental change or better potential for evolution for the new application.

2.2.3.4 Evolution/Future

These solutions should be considered as short-term only and unless the system completely fulfils its intended objective, deeper evolution is usually needed.

2.2.4 Component Based Integration

"If you look at engineering overall, the idea of creating parts is a core process. Whether it is cars, integrated circuits, etc., it has been dramatically advanced by a mechanism of creating, manufacturing, and distributing entire subsystems of components. Yet software is the most backward engineering discipline in that it is the only one not doing it up until now."

Tom Buxton, Microsoft, 1994.

2.2.4.1 Rationale

Component technology strives to provide the levels of component plug-and-play which are enjoyed by hardware vendors. Software components should simply be plugged into a software bus from where they can use and be used by other components connected to the bus. Application development should become more a process of assembly from prefabricated components, which precludes much component development.

Recently, there has been a trend in operating system design towards microkernels. Results from research projects like Mach and Chorus have been applied to commercial operating systems, noticeably IBM's OS/2, and Microsoft's Windows NT. A microkernel-based operating system is fundamentally different from traditional monolithic systems - UNIX for example. A microkernel provides only essential services such as memory management, threading, and inter-process communication. The core services of the microkernel are extended with useful services to suit the requirements of users and applications. Examples of extended services include naming, filing, network time, system management, authentication, and transaction processing.

Microkernels publish a well-defined API from which extended services can be written. This openness promotes a third-party extensions market, and increases the level of choice of services which may be incorporated into an operating system. Microkernel-based operating systems are thus highly *modular* and easily *customised* according to the needs of users and applications. In addition, only the microkernel needs to be ported to different processors - extended services are thus *portable* across heterogeneous processors. These benefits are driving component technology for applications-software development.

2.2.4.2 General Description

Component technology aims to facilitate interoperability among components in the presence of a heterogeneous computer environment.

Figure 16 shows the software bus which masks heterogeneity to application developers and users. The software bus overlays a possibly heterogeneous network of machines. Network nodes shown in rectangular boxes may register objects and services on the bus, which may be used by clients running on any other network node connected to the bus. There are several forms of heterogeneity which component standards attempt to make transparent:

- *Programming languages and tools.* A component standard should allow interoperability between components programmed in different languages and using different tools (compilers for example).
- *Platforms.* Component technology allows applications to be distributed across different platforms (for example PC clients and Workstation servers). Hardware from many vendors running different operating systems can thus host a distributed application.

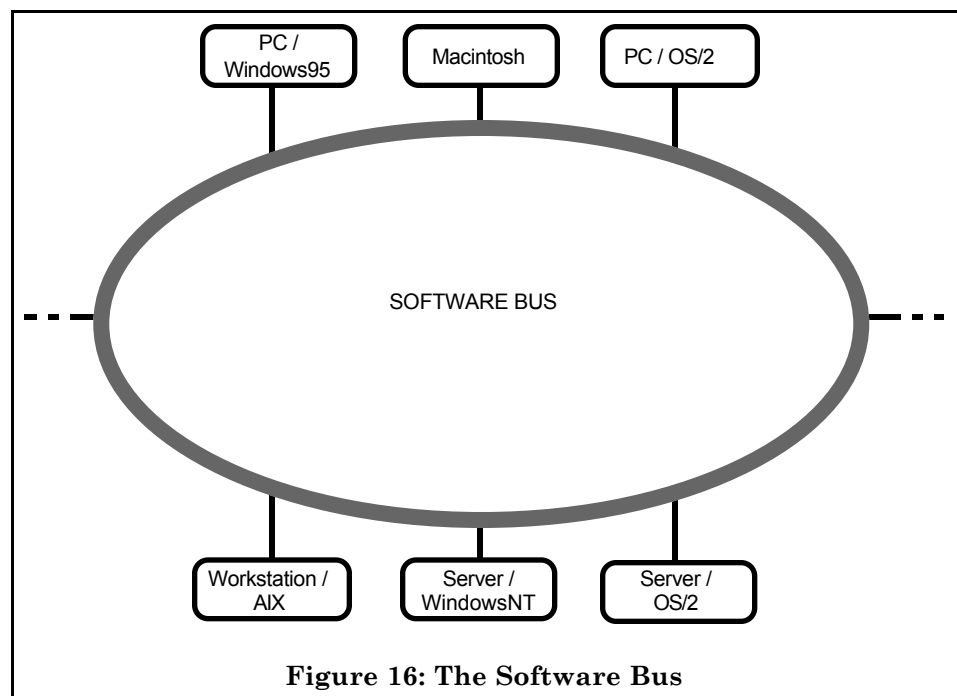


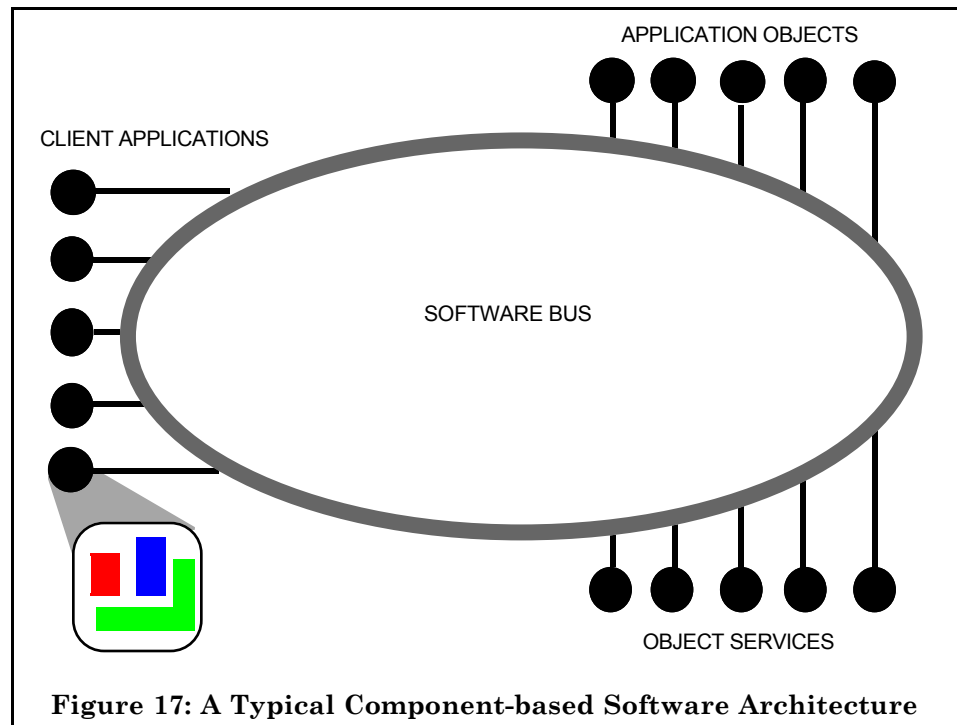
Figure 16: The Software Bus

- *Networks.* An application may span a number of networks, each of which operates according to a different network protocol.

To assist in masking the above forms of heterogeneity, component technology provides:

- *Object model.* A component object model differs from existing object-oriented programming models (for example C++, Objective C, Smalltalk) because it must ensure interoperability between components across programming languages and tools. In particular, a component object model should specify:

- standards for defining application object interfaces in a language-neutral format;
- how object implementations are organised and stored separately from their interfaces. The separation of interfaces from implementations is critical because there may be multiple versions of component implementations;
- how service requests are bound to a particular implementation.
- *Object Request Broker (ORB)*. The role of an ORB is to provide mechanisms which allow objects to transparently make requests and receive responses in a heterogeneous distributed environment. An ORB implements location and access transparency by locating an object implementation, marshalling parameters, invoking the request on the located object implementation, and returning any reply.



An ORB may also maintain an interface repository, which contains, for each object class available on the software bus, its interface. This can be used by developers and by clients to dynamically query the repository and build requests to objects which the client was unaware of when it was compiled.

In addition to the essential object model and ORB, a set of *services* is required in practice. The ORB is to component-based applications what the microkernel is to operating systems - the essential core services. An ORB provides the software bus for component software, but to be useful, this must be augmented with additional services. Naming, event management and concurrency control are examples of useful services.

Additional services are much like the extended services of microkernel operating systems. Where a need for a service is identified, it can be developed and plugged into the software bus. Services are simply components - they conform to the interface standard prescribed by the component technology's object model. The openness of component technology provides for third-party development of a range of services.

Component technology thus provides the software bus which implements the illusion of a homogeneous set of services (the application) which may actually be provided by a heterogeneous computer system.

Figure 17 shows a typical component-based application model. Application objects implement particular applications and are supported by object services (event management, transaction support and security for example). Any client application attached to the bus is able to use the services provided by application objects. Compound documents (described in Chapter 2.2.4) are documents which store collections of data managed by different applications. A compound document's constituent data may be located anywhere on the software bus. Compound documents are typical clients for a component-based client/server system.

2.2.4.3 Benefits and Risks

Component technology relies on conformance to standards to promote a number of desirable benefits (Table 1). However, component technology is relatively immature. Immaturity is the root cause for the risks identified in Table 1.

<i>Benefits</i>	<i>Risks</i>
Customisable applications	Competing standards
Reuse	Immature componentware market
Evolutionary systems	Personnel skills shortage
Open and scaleable distributed systems	
Document-centric desktop computing	

Table 1: Benefits and Risks of Component Technology

Customisable Applications

Today's software market hosts a rich set of applications at generally affordable prices. However, the cost of tailoring an application to meet the specific needs of an individual remains disproportionately high.

Consider a text processing application for example. Such applications are available today with functionality ranging from simple cursor manipulation to sophisticated word processors with features like grammar checkers, dictionaries and graphical editors. Tailoring a word processor for the scientific community, by introducing a mathematical equation editor for example appears to be a non-trivial task.

Using components, it will be possible to build customisable applications from assemblies of appropriate components. For the word processor example, scientists would use an application built from a different component configuration than that used by clerical staff. Applications

built this way will reduce redundancy too - many applications have text editors, and adoption of the Componentware paradigm will allow all applications to share one common text editor. The potential for customisable software solutions of course extends beyond the desktop to the construction of families of server applications.

Similarly to applications being realised as particular configurations of components, individual components will be tailorable by specifying parameters or setting properties. Such components are termed white-box. Mixed assemblies of black and white box components will be used to compose customisable applications.

Reuse

Object-oriented development has been accompanied by claims of dramatically reduced software development times based on object-reuse. Unfortunately, the degree of reuse realised by object technology has been disappointing. Object-oriented programming languages, such as C++ and Smalltalk have failed to deliver the expected levels of reuse primarily because of their inability to package and distribute components in binary form. Applications generally have to be written in one programming language, and often compiled using a single vendor's compiler.

Component technology may reasonably be expected to yield considerably higher levels of reuse than object technology. Componentware is a binary standard for components. A component has a well-defined language-neutral interface, which is separate from its implementation. A component's implementation, in principle, may be coded in any programming language. Component technologies provide the infrastructure for component interaction among components programmed in different languages, running in a heterogeneous distributed environment.

Evolutionary Systems

Component technology provides a means to manage scale and complexity during development and evolution of applications. Components are units of composition and configuration. Component technology is deeply rooted in object technology, and so exhibits the principles of encapsulation and abstraction. Component implementations can be changed without affecting other components which use the changed component through its interface.

Component technology offers a safe path for application evolution. Existing elements of a legacy application can be encapsulated or *wrapped* in a component interface. Where an application's module structure can be transformed into a component-based application with little effort, it is possible for the application to rapidly reap the benefits of a distributed client/server architecture.

Incremental evolution is supported by the capability of components to maintain the crucial existing business knowledge encapsulated in a legacy application. A critical COBOL module for example, may simply be wrapped and packaged as a component to be deployed in the evolved application. Once the transition to a client/server architecture has been made, it is then possible to pick individual components and re-implement or redevelop them according to evolution requirements.

Open and Scaleable Distributed Systems

Systems constructed from standardised components are open and extensible. Conformance to a universal componentware standard is essential to achieve true levels of plug-and-play among components. Component applications are extensible - by extending interfaces of existing components, by substituting component implementations, and by introducing new components. Component technology provides the necessary infrastructure to provide language, tool, platform and network transparency.

Document-Centric Desktop Computing

Until recently, human users of computers have been presented with application-oriented user interfaces. Users have had to explicitly use separate applications to perform real-world tasks which are closely related. The use of multiple applications does not capture the semantic connection between such tasks.

For example, a software developer might develop a C++ class framework using UNIX development tools. The developer would then document the framework using a word processor on a PC perhaps. Part of the documentation would be the C++ class headers of the framework. Typically, the developer would have to save the C++ files in some platform-independent format, and load them into the word processor, where the class specifications could be incorporated into the document. There are two problems with this model of work. First, the relationship between the C++ source files and the associated documentation is lost. Second, because of this decoupling, inconsistencies can arise between the source code and documentation.

Componentware aims to address such issues with *compound documents*. A compound document is a *container* which can store a collection of heterogeneous data, where each data item is managed by a different application. Users are thus presented with a single document containing related data for a given task, as opposed to related data being distributed across many files.

Competing Standards

There are currently two dominant component technology standards: CORBA and OLE/DCOM. During recent years, there has been much confusion over which standard an organisation should adopt. It has been unclear which standard is most suitable to a particular organisation's needs, and which standard is expected to survive. OLE/DCOM originated as a desktop component technology and gained notable success with compound documents. CORBA has been designed to support component-based applications in the presence of heterogeneity.

Today, both OLE/DCOM and CORBA are converging in terms of functionality and platforms they support. The two standards are no longer mutually exclusive - cross standard interoperability is in progress, and so the choice of standard is longer a critical decision. OLE/DCOM and CORBA are still dynamic, but tending towards stability.

Immature Componentware Market

A component market has emerged for OLE/DCOM which has built on the success of Visual Basic extension components. CORBA has yet to be embraced by a third-party component industry, but it is expected that a market will appear shortly. For both standards, there are a number of non-technical factors which must be resolved before markets can thrive. Such factors are business-oriented and govern how components can be purchased, licensed and distributed.

Personnel Skills

Evolving applications from centralised mainframe technology to distributed client/server systems requires a shift in personnel skills. Development and maintenance staff need to understand the component standards and how to use them. Administrators need to know how to manage object-based distributed systems - this is a problem which is more complicated with distributed and multi-vendor client/server technology.

Developing component-based applications requires a different mindset to traditional software development. Component development is difficult because developers must trade applicability of components with generality. In addition, components must be correct and reliable before they can be deployed in applications. Component development thus requires greater skill than application development by assembling components.

2.2.4.4 Evolution

Component technology is a relatively immature client/server technology. It is expected that the key standards OLE/DCOM and CORBA will stabilise and be compatible with each other.

Adoption of these standards by third party vendors will cause an explosion of componentware. There exists a healthy componentware market for OLE/DCOM today. A similar market for CORBA should emerge in the near future.

Integration of component technology with other client/server technologies such as database servers, transaction processing and groupware will be an important direction of evolution to allow widespread usage of components across many application domains.

2.2.5 Workflow-Based Integration

2.2.5.1 Rationale

Competing globally, reducing the cost of doing business, and rapidly developing new services and products are the requirements of today's business market, including the software market. Enterprises try to address these new requirements by constantly reconsidering and restructuring the way they do business. This involves changing their information systems and applications to support these evolving business processes. Workflow technology which can support, control and improve structured work facilitates change by providing a methodology and supporting software for both process support and control. This allows for

process improvement as well as ensuring the quality of execution of existing business processes.

The aim of the workflow paradigm is to increase efficiency by concentrating on the routine aspects of work activities. Workflows separate work activities into well-defined tasks, roles, rules, and procedures which regulate most of the work both in manufacturing and the office, and provide a formal description of the knowledge necessary to perform such activity. In this sense, they formalise the current skill in doing the business, and, even if this is affected by all the cognitive-science-already-known problems of eliciting and structuring the knowledge, they provide a good basis from which to start to think aloud about the process. From this perspective, the workflowed process (a market-centred description of an organisation's activity, including the software development process, which is a part of the overall business process) can be deeply investigated and simulations can be studied, in order to enforce, improve, or even redesign the process itself. The field addressed by workflow technology comprises reengineering and automating business processes.

2.2.5.2 General Description

To cope with the many Workflow Management Systems provided by independent software vendors and the lack of standard, the Workflow Management coalition has been formed. As part of the definition of standards, the following model has been defined as a standard architecture for workflow enabled systems:

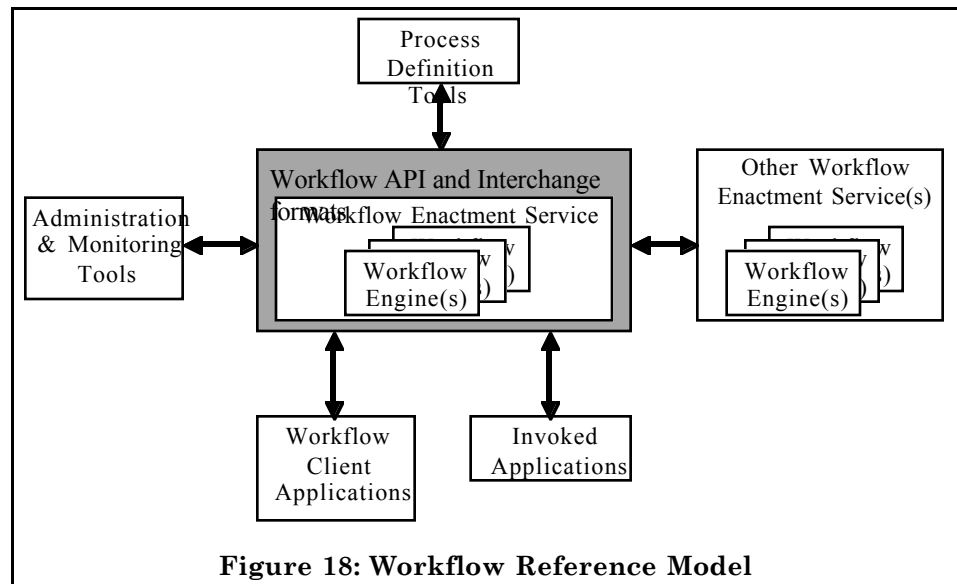


Figure 18: Workflow Reference Model

Workflow engines are in charge of executing processes and distributing work (tasks) to actors. In this distribution of work, some tasks may be realised manually, or with an assistance from applications, or automatically by invoking applications. In the above model, the interface with Invoked Applications is the basis for workflow based integration.

One important use of workflow to support integration between parts of a software system (some of which can be legacy) is that it can automate sequences of steps needed for inter-operation between the parts. The invocation of sequences is driven by the underlying business process. For example, before presenting a user with information, the workflow system may have invoked a long sequence of complex processing, involving many different parts of the software system, in order to obtain the information. A simple user operation may then result in an equally complex set of operations to restore a consistent system state. Seen from the user's point of view, the system appears as an integrated whole.

A legacy application can evolve to or be involved in a workflow based system by:

- capturing the business processes the legacy system supports,
- modify the business processes, if needed,
- model the processes using process definition tools,
- link these processes with legacy system components (programs, transactions...) for those currently support by the legacy system,
- implement these processes.

The link between the workflow system and legacy systems is normally done using more basic integration techniques, and especially using service based integration techniques.

2.2.5.3 Benefits/Risks of Approach

Moving to a workflow based system needs to define clearly the business process. This can be a complex task with many implications on the organisation. The target system will however be very suitable for process monitoring and tracking and process change.

However, there is a lack of support for validating the correctness and reliability of processes and for analysing, testing and debugging processes. In addition, current workflow management systems are lacking inter-operability.

2.2.5.4 Evolution/Future

The work of the workflow management coalition and its adoption by product vendors will ensure a minimum level of inter-operability and standard behaviour of tools.

In addition to specific workflow management tools, more and more standard packages are providing their own workflow module, even if the scope of workflow support is often limited.

2.2.6 Web Based Integration

2.2.6.1 Rationale

Internet technologies have created a dramatic explosion and renewal of methods for developing and deploying business solutions. Where classical distributed systems failed in the support for multiple platforms, Internet technologies answer with a common user interface on most platforms. On the exploitation viewpoint, Internet based systems currently have an architecture close to a centralised one, easing the deployment of solutions

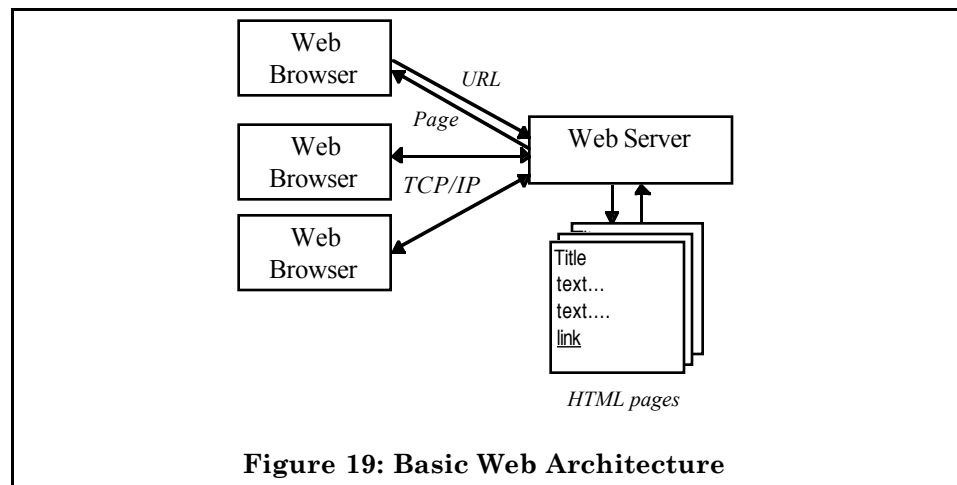
and new versions of these solutions not only within the enterprise network but also on the public Internet.

However, as the need for more complex interfaces and functions on the user desktop grow, because of the limitation of the bandwidth, and also because of limited scope of business solutions that can be built (for instance, due to the lack of or very little support for highly secured transaction environments), solutions are evolving.

This section presents the current state of possible ways to integrate a web based system with a legacy system, but as the market is evolving very rapidly, this picture can become rapidly outdated or at least limited.

2.2.6.2 General Description

Basic web based architectures consist of a client tool (web browser) sending a request for a page by specifying its Universal Resource Locator (URL) to a web browser. The web server finds and returns this page as result of the request. The contents of this page is represented using the HyperText Mark-up Language. This page can in turn contain links to other pages allowing efficient navigation or "surfing":

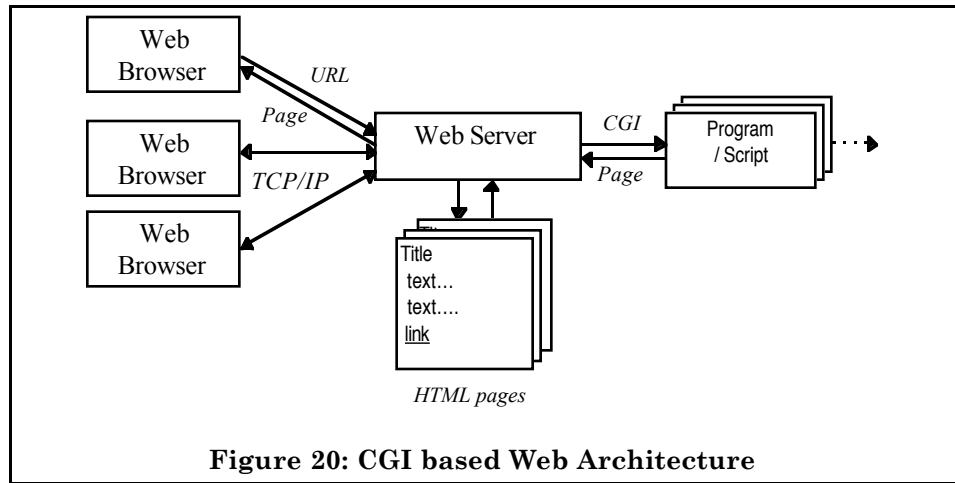


This model is however static and does not involve access to an information system. Other models exist to access an information system and are partly based on this basic model. This section reviews typical existing models:

- CGI based access
- Dynamic Pages
- Shortcut Architecture

CGI based access

CGI (Common Gateway Interface) based architectures extend the basic static model by providing means to execute a program building dynamically a page as a result of a request:



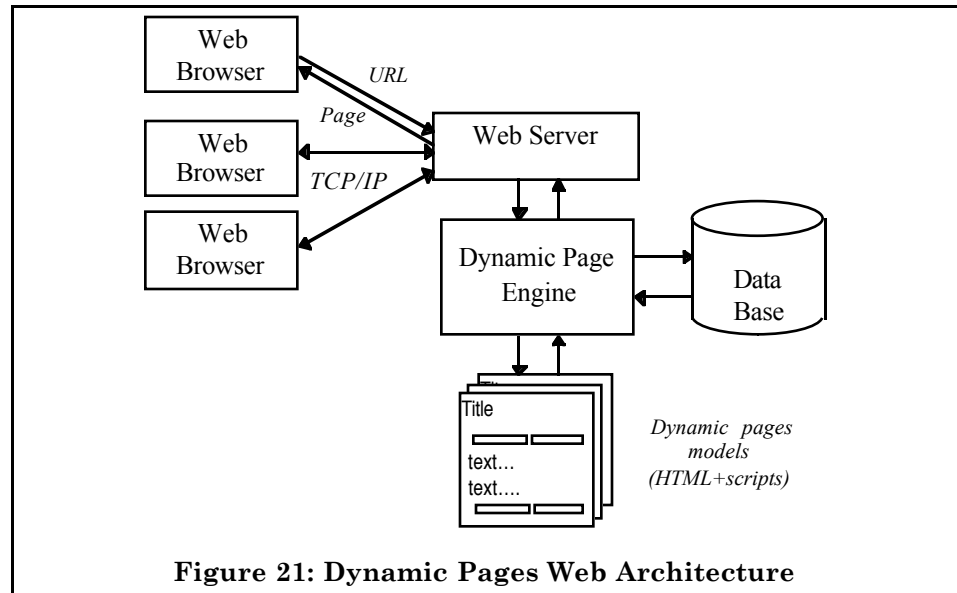
Programs and scripts called are usually developed using script languages (PERL...) allowing rapid development. These programs and scripts can then in turn access other resources and applications to build the HTML to return as result of the request.

Major drawbacks of this technique is that they are not time efficient since the web server launches a new process for each request and that there is no concept of session (e.g. connections to a database must be opened and closed for each request). In addition, building scripts or programs to generate pages is not a straight forward activity, and maintenance can be a complex activity.

Web server tools providers extended this model by providing direct access from their web servers to programs using Application Programming Interfaces (Netscape's NSAPI, Microsoft ISAPI...), thus avoiding the creation of processes for each request.

Dynamic Pages

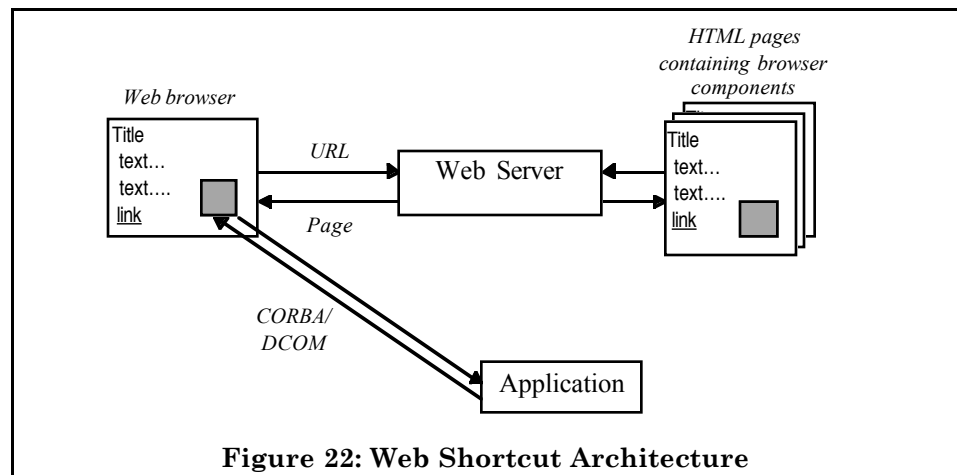
To solve some problems of CGI based access, web server providers (Netscape's LiveWire, Microsoft Active Server Pages) or independent software vendors (Allaire's ColdFusion) developed specific engines to interpret HTML pages embedding scripts (JavaScript, VBScript, CFML...) allowing basic processing or access to databases and transactions:



For every request for a dynamic page, the web server forwards it to a Dynamic Page Engine which interprets the corresponding dynamic page by executing embedded scripts and extending the page model accordingly with the results of database queries or transaction executions.

Web "Shortcut" Architecture

A more recent web architecture, object based, consists of shortcutting the connection with the web server and establishing a direct connection between the web browser and an application using another communication channel. Access to a web server is provided but only to start the dialogue and install the components on the client side to continue processing directly between the browser and the application.



Two main similar implementations of this architecture model exist:

Format Style not defined

- Java components (applets) in the browser communicating with the application using CORBA (Sun and Netscape solution)
- Active X components in the browser communicating with the application using DCOM (Microsoft solution)

Both architectures require that the application be object based since the communication between the browser and the application uses an object based protocol.

This is the most current promising architecture, providing all benefits from Internet based applications (distribution, client platform independence) and with a potential to build complex solutions. Techniques and tools are however less mature and very little return of experience is available.

2.2.6.3 Benefits/Risks of Approach

Each architecture presented above presents its own benefits and drawbacks. Basically, the more elaborated the solution is, the more functions it can provide, but the more complex it is to be set up.

Web-based architectures and solutions are the fastest evolving area today. They indeed provide benefits for several needs:

- globalisation of business
- platform independence
- solution deployment
- integrability for existing systems

However, the emergence of new solutions every month, and even often every week, makes the choice of a solution fitting business needs and ready for future evolution very complex.

In addition, this area is the main battlefield for all major computer industry actors and the overall picture is far from being stabilised.

Tool support is evolving very rapidly but is still lacking maturity, especially compared to major 4GLs.

2.2.6.4 Evolution/Future

Web based integration techniques are still evolving very fast and they can support more and more complex architectures. All major computer industry actors are involved in the battle pushing very fast for better and better solutions, but not always heading towards the same direction.

Investments in one of these architectures could be risky until the market is more stable, but due to the weight of actors involved, it is worth it.

The current evolution trends include inter-operability with current integration techniques (transaction systems, messaging systems).

3 Integration Scenarios

Contents

- 3.1 Introduction
- 3.2 Migration to Graphical User Interfaces
- 3.3 Migration to Web-Based User Interfaces
- 3.4 Application Extension
- 3.5 Application Extension to Internet
- 3.6 Application Workflow Automation
- 3.7 Application Workflow Integration
- 3.8 Application Distribution

Summary

This chapter describes several architectural scenarios to build a system architecture integrating legacy system components with new components. Each integration scenario includes corresponding system architecture models and references to system integration models presented earlier.

It is especially dedicated to technical decision makers which must identify and assess possible architectures and evaluate the benefits and concerns of the various solutions.

3.1 Introduction

An integration scenario presents how basic integration mechanisms can be used to build a typical system architecture including an integration with an existing system.

This chapter focuses on evolution strategies requiring modification of the architecture of the application, and especially in the way its components communicate:

<i>Evolution Strategy</i>	<i>Characteristics and coverage in this document</i>
Maintenance	Bug-Fixing, adaptations, etc. without changing the structure or architecture. <i>Not addressed in this document because no modification to the structure and architecture of the application is done.</i>
Re-vamp	The user interface of a system will be updated to a more advanced technology, i.e. character mode to GUI. The general structure (and architecture) of the software will not be changed.
Re-structure	The structure of the software will be changed. The underlying hardware will not be changed. <i>Not addressed in this document since it is rather in the scope of re-engineering tools rather than being a migration to a distributed environment.</i>
Re-architecture	The structure of the system will be changed as well as the underlying hardware.
Re-design with re-use	A new system will be created integrating re-usable assets of the existing system.
Re-placement	The existing system will be replaced by a new developed system without regarding the existing one. <i>Not addressed in this document because there is no modification to the structure and architecture of the application. Scenarios and System Integration Models can however be used to build the architecture of the new application</i>

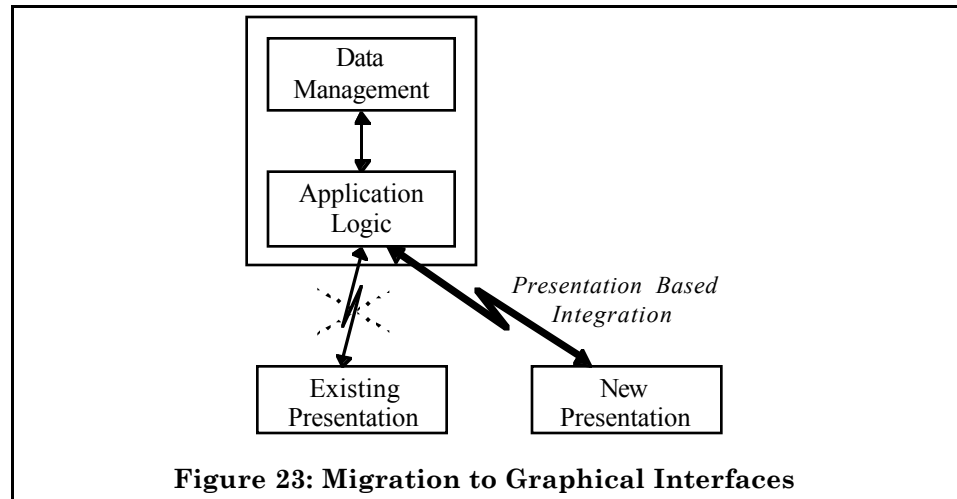
Table 2: Evolution Strategies

Integration scenarios listed describe how a legacy application and new components can be integrated, but the corresponding models can also be used if a new application is being developed without direct connection or reuse of a legacy application.

3.2 Migration to Graphical User Interfaces

Migration to Graphical User Interfaces belongs to the Re-Vamp evolution strategy.

It consists of replacing or adding a new user interface to an existing application:



This integration scenario is based on the use of Presentation Based Integration (see section 2.2.3). Instead, or in addition to character based user interfaces, access to the application can be done through similar corresponding graphical user interfaces.

Benefits and concerns of this integration scenario are similar to those applicable to presentation based integration:

- No modification is needed to the existing system.
- There is no fundamental change to the user interface and users can be rapidly trained to the new system.
- This is a short-term evolution and does not provide better evolution potential for the new system.

3.3 Migration to Web-Based User Interfaces

Migration to web based user interfaces belongs to the Re-Vamp evolution strategy and is very similar to migration to a graphical user interface.

This integration scenario is based on the use of Presentation Based Integration (see section 2.2.3). Instead, or in addition to character based user interfaces, access to the application can be done using a web browser. Browser pages can be screen equivalent HTML pages or forms, or even more sophisticated web user interfaces including Java graphical applets or ActiveX controls.

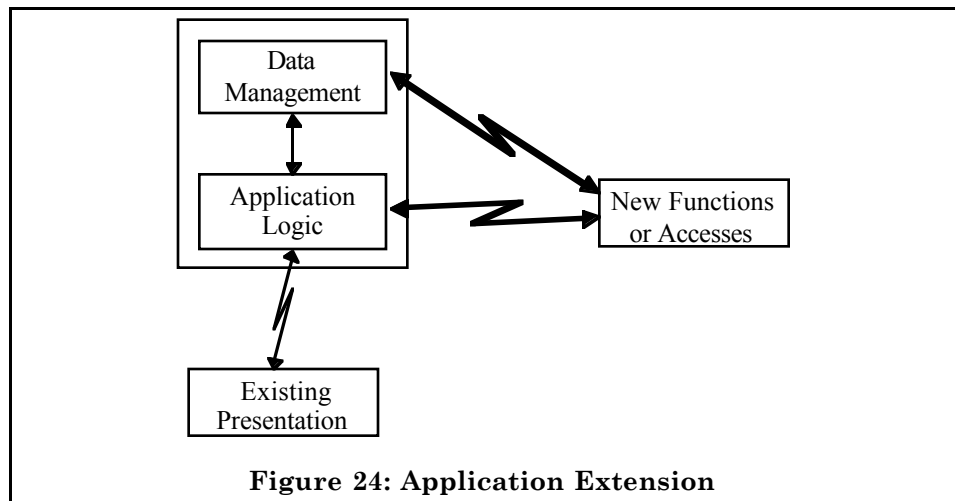
Benefits and concerns of this integration scenario are similar to those applicable to presentation based integration:

- No modification is needed to the existing system.
- There is no fundamental change to the user interface and users can be rapidly trained to the new system.
- The system can be accessed by most client platforms, more precisely by all platforms for which a web browser exist.
- This is a short-term evolution and does not provide better evolution potential for the new system.

- This type of migration is cheap for simple data-centric applications, but it is expensive if the application logic needs to be re-implemented.

3.4 Application Extension

The Application Extension integration scenario fits into the Re-Architecture evolution strategy. The architecture of the legacy application is not modified. It is rather extended to provide new functions or accesses. The legacy system can usually still be used and provides at least the same functions with the old interfaces:



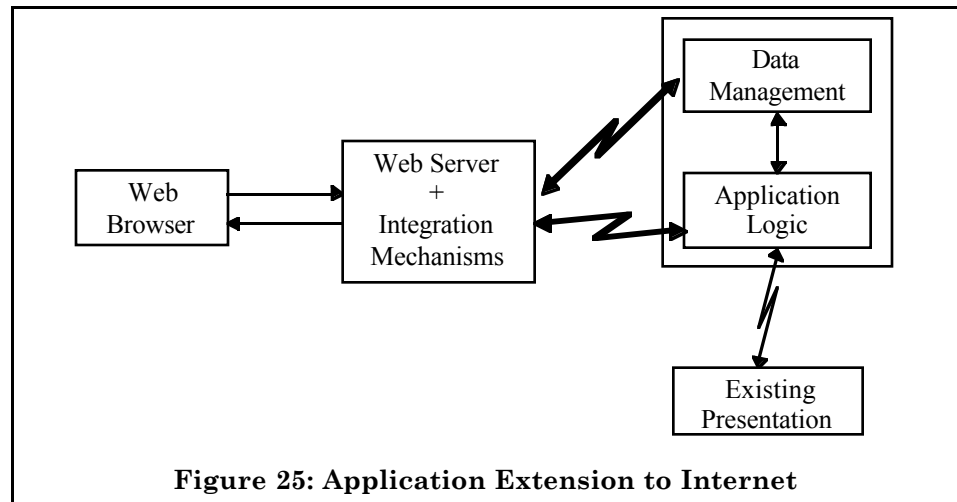
This extension is done using gateways or integration mechanisms:

- **Data Based Integration** can be used to provide a direct access to data still used by the legacy application. This approach can however be risky since integrity rules which can be included in the application logic part of the legacy system are not used by the new functions. This can result in inconsistency. It is therefore recommended to access data only in a read-only way. Such extension can also be complex to implement since documentation about data and its use is often missing and could only be retrieved by analysing the existing legacy code.
- **Service Based Integration** can be used to directly access programs or transactions in the legacy system. Therefore, new functions are using safe entry points to the legacy system (compared to data based integration).

3.5 Application Extension to Internet

Application Extension to Internet is a specific case of Application Extension. The legacy application is still running and can be accessed using the existing interfaces but a new web based access is built. In

addition to a web server, web access requires gateways with the existing systems to be used:



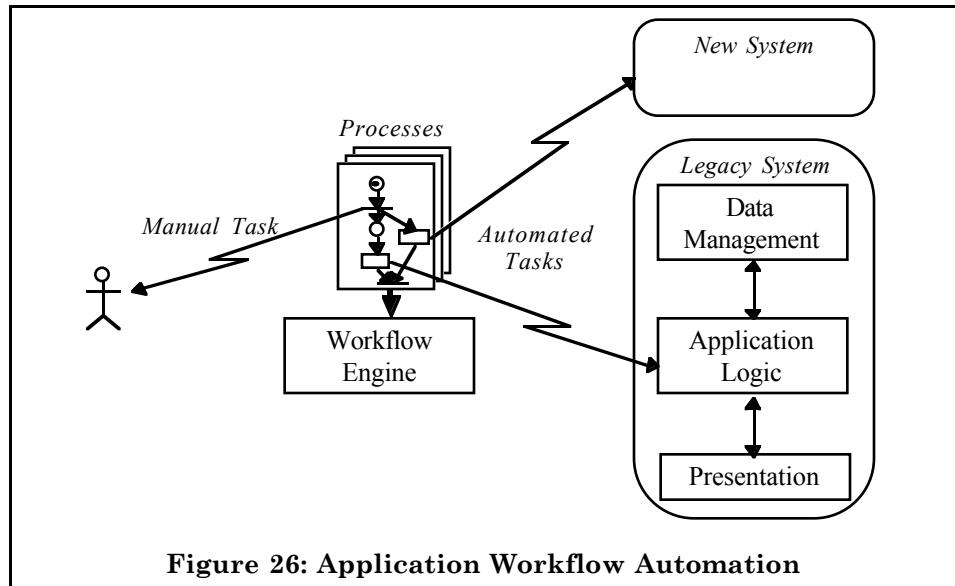
This integration scenario is based on a combination of web based integration mechanisms (see section 2.2.6) and other integration mechanisms.

Benefits and drawbacks of this integration scenario are:

- The existing legacy system can still be used.
- Wide access to system through Internet.
- Client platform independence.
- Scalability since the web integration mechanisms define an intermediate application server which can be resized or duplicated.
- The web server and web browsers can be used to access other services, especially to access new services developed incrementally to replace part or all the legacy system.
- Powerful web integration mechanisms are still relatively immature, not widely used and not standardised yet.

3.6 Application Workflow Automation

Application Workflow Automation is an integration scenario where a workflow system co-ordinates existing application components. The workflow system becomes the new access interface to the application and old access interfaces can be hidden and made more simple:



This integration scenario is based on workflow based integration mechanisms, usually combined with other mechanisms, such as service based integration mechanisms.

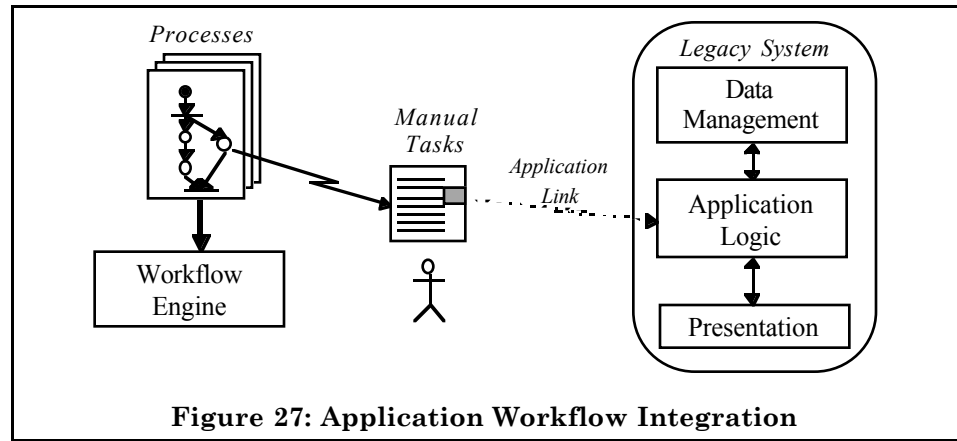
A typical use consists of automating calls to a sequence of transactions to perform a task.

The processes executed by the end user can therefore be made more simple, thanks to workflow management system support.

In addition, since the processes are clearly defined, this brings a good framework for migrating incrementally task support from an old system to a new system.

3.7 Application Workflow Integration

Application Workflow Integration is an integration mechanism where access to existing applications is provided to users to support tasks defined in workflow processes. When the workflow engine executes processes and distribute tasks to actors and applications, a task context is sent to users with a description, data and links to applications or application components supporting the task:



When a user must complete a task, he can launch the application supporting his task from this task context. Therefore, this integration scenario defines a glue between applications or application components.

This integration scenario is based on workflow integration techniques.

The major benefit of this approach in addition to benefits of workflow is that it is a lightweight integration with the existing system and it provides a guidance to users for its use.

3.8 Application Distribution

3.8.1.1 Rationale

Distributed systems promote many properties not found in the centralised mainframe model. Openness, scalability and fault tolerance are three such properties which are used to evaluate client/server technology in Chapter 4. In addition, resource sharing and concurrency are inherent in distributed architectures. The distributed architectural models introduced in Chapter 2 enhance modularity and promote a separation of concerns over centralised counterparts.

3.8.1.2 Distribution without re-Architecting

Evolving a centralised application to a distributed system generally requires that a new architectural model is adopted. However, there are scenarios where re-architecting is not necessary to distribute an application. Such scenarios have been covered earlier in this Chapter: application revamping (migration to GUI or Web-based interfaces), and application extension - with Internet access for example. Revamping and extension effectively distributes function without change to the existing architecture. Revamping and extension however, do not realise the benefits of openness, scalability, and fault tolerance. To gain these properties, re-architecting is necessary.

3.8.1.3 Re-architecting with Component Technology

Component technology provides the means to develop distributed applications in the presence of heterogeneity. Application development is

thus shielded from different languages and tools used to implement components, platforms used to host components, and network protocols used by interconnected networks. These benefits suggest component technology to be appropriate for distributed systems.

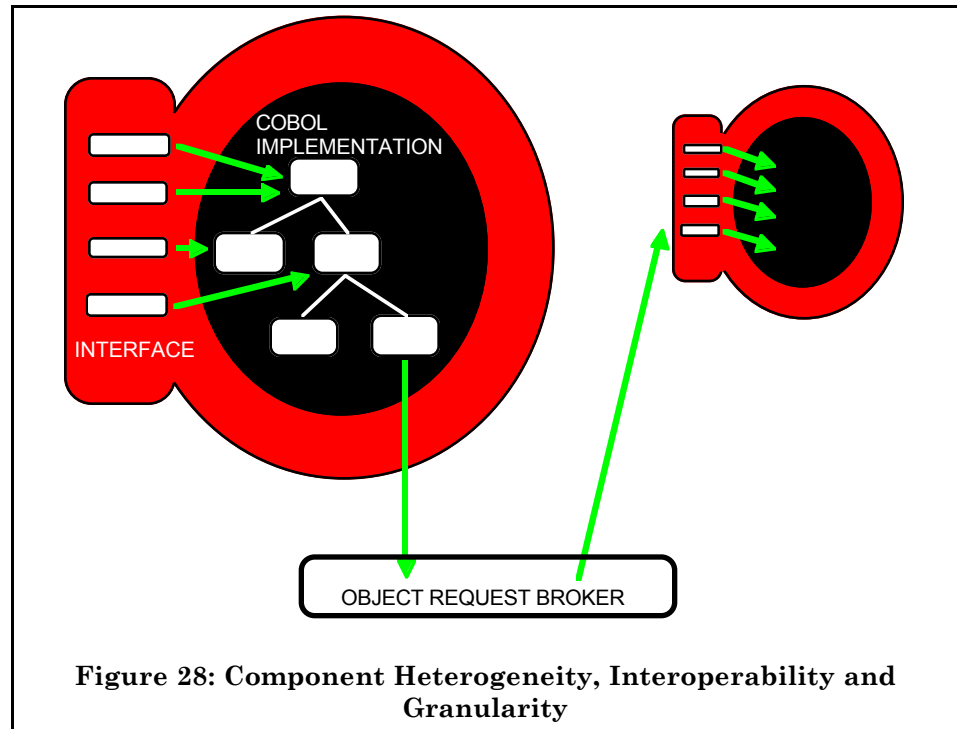
Component technology clearly promotes openness and scalability. Further, fault tolerance can be achieved with some additional effort. Component technology is attractive not only as a model for development of new applications, but as a model which can be used to integrate legacy system elements. This is achieved by a technique called *encapsulation*.

Encapsulation

Encapsulation is where legacy software elements are wrapped inside an interface which conforms to a component technology standard. The result is a component which can be plugged into the software bus, but which is implemented by legacy code.

Encapsulation provides for a *rapid yet safe* transition to a component-based client/server architecture. The transition is rapid because old code is reused - it is simply bound to a new interface. The transition is safe, because existing code is not thrown away - it is simply re-employed in the new architecture.

Component encapsulation is thus an important technique to ensure critical business knowledge which is buried in legacy software survives evolution. For example, a critical COBOL module from a closed mainframe application may simply be encapsulated and packaged as a component to be deployed in the evolved, open and distributed application. Once the transition to the new architecture has been made, it is then possible to pick individual components and re-implement or redevelop them according to evolution requirements.

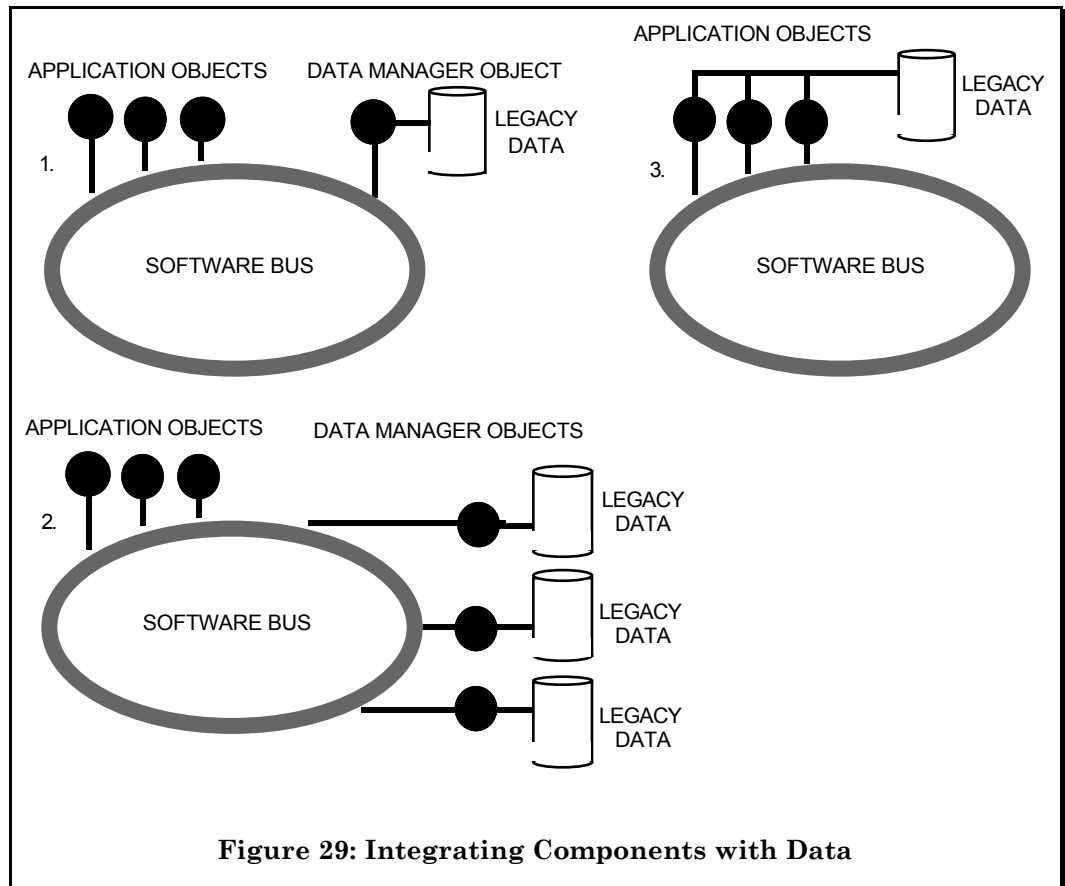


The granularity of a component is determined by a design or migration decision. At one extreme, it is possible to encapsulate an entire legacy application in a single component. Alternatively, one might use components to perform relatively simple computations. Whatever the granularity, a component may only be accessed through its interface.

Figure 28: Component Heterogeneity, Interoperability and Granularity

shows a coarse grained component - a legacy application, implemented as a collection of COBOL programs. The component's interface represents entry points to the programs. This component is able to inter-operate with a new fine grained component implemented in Smalltalk for example.

3.8.1.4 Re-architecting using Combined Integration Mechanisms



Component technology may not be appropriate for all distributed applications. Further, where other technology is necessary (for example database servers and TP monitors), a sensible model is likely to be a combination of the integration models introduced in Chapter 2.2. Currently, component technology is well suited to application development or evolution using encapsulation, where the application is structured according to object-oriented principles. However, consider a data management application where application components require access to a database. Data management can be integrated with a component-based solution in three ways (

):

1. The database can be encapsulated in a data manager component. All data requests now pass through the data manager, rather than directly to the database. In this case, fault tolerance can be achieved transparently by the data manager. It is the responsibility of the data manager to synchronise updates of data replicas hosted ON different machines.
2. The database can be partitioned and distributed across several data manager components. This is similar to the first approach, but adopts a more fine-grained view of data. The application can be

distributed according to which components use which data managers. Some fault tolerance is exhibited by this approach without replication. Providing each data manager and database pair is hosted by a different machine, a crash of any one of those machines will not affect availability of the other data managers and their data.

3. The database is a network resource, but not connected to the component software bus. In this approach, the database is accessed directly by any application components using appropriate distribution middleware.

It is often common practise for organisations to restrict direct database access to application programs. In this situation, a database interface is used by applications to enforce indirect database access. For approaches one and two, this interface can simply be used to provide a component interface to the data manager component(s).

Distributing transaction processing systems is probably best achieved using current TP monitor software. Today's TP monitor products (discussed in Chapter 4.4.2) are mature and well understood. They not only provide support for transactional integrity, but offer implementation tools for distributing applications such as: service integration mechanisms (RPC and MOM), system management, monitoring and load balancing. Transaction services will emerge as components for component technology solutions; indeed a transaction service has been specified for CORBA, but TP monitor products are available today and are associated with less risk.

3.8.1.5 Distributed Platform Considerations

Clients and servers place different requirements on platforms. Clients handle end-user interaction and make server requests. Compound documents are the heart of component-based client/server systems, and so clients will need to manage several concurrent programs (applications). The operating system should provide mutual protection to concurrently executing programs - a fault in one should not affect other applications.

Depending on the nature of a server, it may be used by hundreds of clients. Client requests should be managed concurrently and according to priority. If a request cannot be executed immediately, because a resource is unavailable for example, the server should not block, but rather it should attempt to process other requests. Pre-emptive multitasking and multithreading services should therefore be provided by the server's operating system. Server request throughput is also enhanced by a high performance file system. A high performance file system is characterised by fine-grained file locking mechanism, which locks files at the data level as opposed to the file level. Further, the operating system should support several open files.

For enterprise solutions where several network protocols are used, servers should provide a rich set of communications stacks, to enable interoperability with the greatest number of client platforms.

For large component-based systems, global directory services will enable clients to locate appropriate services. "Yellow pages" directory services, where clients specify properties of a required service, rather than the name of a particular service are now becoming available.

Authentication is a necessary service, given the open nature of a distributed client/server system. Clients should be able to prove who they are and servers should be able to determine whether clients have authorisation when requesting particular services.

System management aims to provide a global enterprise view of the system. The application may be managed by partitioning it into domains. Examples of system management services include configuration management, performance monitoring and metering (to provide for pay-as-you-use services).

Many applications require access to multi-user SQL databases. Others need TP monitors for managing transactions across servers. Database and TP extended services meet the needs of such applications.

4 Migration Techniques and Tools

Contents

- 4.1 Introduction
- 4.2 Technical Evaluation Criteria
- 4.3 Data Based Integration
- 4.4 Service Based Integration
- 4.5 Presentation Based Integration
- 4.6 Component Based Integration

Summary

This chapter describes techniques and tools relevant to be used in the context of the migration of a legacy system to a distributed architecture. Migration techniques and tools are presented according to the integration scenario they support.

4.1 Introduction

This chapter describes techniques and tools relevant to be used in the context of the migration of a legacy system to a distributed architecture. Migration techniques and tools are presented according to the integration scenario they support:

<i>Integration Scenario</i>	<i>Technique/Tool</i>
Data Based Integration	
Data Encapsulation and Access	ODBC and JDBC, page 51 EDA/SQL, page 54 Oracle Transparent Gateway, page 56 OpenDM, page 57
Service Based Integration	
Messaging and Queuing Middleware	IBM MQSeries, page 61
Component Based Integration	
	CORBA, page 76 OLE and DCOM, page 80

Table 3: List of Techniques and Tools Reviewed

4.2 Technical Evaluation Criteria

The choice of techniques and tools is constrained by several factors:

- Business goals
- Environment constraints
- Technical constraints

In order to provide guidelines for supporting the choice of techniques and tools reviewed in this chapter, the suitability of each of them is assessed according to these factors. This chapter only considers environment and technical constraints.

4.2.1 Environment Constraints

Environment constraints are those relative to the existing implementation of the legacy system. The set of criteria used for these constraints are limited in this document to the support of:

- Hardware environments (hardware platform, operating system, network protocol...)
- Software environments (databases, programming languages...)

Therefore, techniques and tools will be evaluated according to the number of environments as listed above are supported.

This evaluation should also help the reader to answer questions like "what operating system or hardware platform is best suited for my needs?" by basing the selection of such environments on the availability of the chosen technique on them.

4.2.2 Technical Constraints

4.2.2.1 Security

Migrating to distributed technology introduces security issues which are not manifested in closed mainframe computer systems. Perhaps the most obvious threat to security in a distributed environment is manipulation of messages as they are transmitted across a network. Any message can be copied, analysed, edited and subsequently replayed.

Further, the client/server interaction model is itself exposed to security threats. For example, a program might be installed as a server, and receive confidential information from clients who use the server believing it to be authentic. Similarly, clients may request services from servers which they are not authorised to request.

Security services are therefore necessary in distributed client/server systems. Security services rely on three techniques:

1. *Cryptography*. Messages can be encrypted to conceal private information where it is exposed in parts of the system like communication channels. Message encryption relies on the use of a particular encryption key - a receiver can only decrypt the message if it knows the inverse key. In addition, encryption can support authenticating communication between two processes - a process which decrypts a message successfully using a particular inverse key can assume that the message is authentic if it contains some expected value.
2. *Authentication mechanisms*. Authentication is the means by which the identities of clients and servers are reliably established. The mechanism used to achieve this is based on the possession of encryption keys - from the fact that a process possesses the appropriate secret encryption key, the process thus has the identity that it claims. Authentication services typically rely on encryption. Kerberos is a popular authentication protocol, implemented by the DCE network operating system amongst others.
3. *Access control mechanisms*. Access control mechanisms are concerned with ensuring that access to resources is available only to that subset of users that are currently authorised to do so.

Security services are best provided by operating systems, as extended microkernel services. System and application software can then make use of a uniform set of services, rather than each operating according to their own proprietary mechanisms.

4.2.2.2 Scalability

Scalability is a measure of the ease of which a system can be scaled to effectively meet users' needs. The system and application software should not need to be changed when the scale of the system increases. Distributed system design is based on the philosophy that no single resource is assumed to be in restricted supply. Rather, as the demand for a resource grows, it should be possible to extend the system to meet it.

Where increasing demand is placed on servers of a client/server system, two techniques can be used to increase system throughput.

1. *Horizontal scaling.* The hardware architecture of the system can be augmented with new server machines. Introducing new machines on which to run additional process servers however increases the complexity of the software architecture. For example, replicated data and synchronisation issues have to be addressed. Transaction processing monitors and network operating services can be used to manage these issues.
2. *Vertical scaling.* Where a personal computer is used as a server machine and demand for the services managed by the personal computer exceeds the processing capacity of the personal computer, it can be upgraded to a multi-processor server. An *asymmetric server* uses a single master processor on which to run the operating system. The server's processing is managed by the master processor distributing tasks to the remaining slave processors. On a *symmetric server*, all processors are treated as equal and applications are split into threads which can run on any available processor. However, symmetric servers require that applications are written according to a multi-threaded parallelism model. Such applications are rare, but include Oracle 7 and Sybase.

4.2.2.3 Openness

An open system is one which supports resource sharing, and can be extended with new resources without disruption to existing services. Openness is achieved by specifying and documenting the key software interfaces of a system and making them available to developers. Microkernel operating systems are an example of open systems, where the kernel provides essential services, which can be supplemented with further services according to the needs of applications and users.

Open distributed systems are based on the provision of a uniform inter-process communication mechanism and well-defined interfaces for access to shared resources. Such systems can be constructed from heterogeneous hardware and software.

4.2.2.4 Fault Tolerance

Distributed systems potentially provide for tolerance against hardware and software faults. Component redundancy (of both hardware and software) and software recovery are necessary to achieve fault-tolerance.

Availability is closely related to fault tolerance. A fault in a centralised system generally means that the whole system is made unavailable. A fault in a distributed system does not necessarily mean that the whole system will be affected. If a data server process crashes, either because of a software or hardware fault, only those clients which need access to the data server are affected; all other clients and servers may proceed as normal.

In the above example, the data server could be replicated on another node. In this case, clients which use the crashed data server are unaware of the crash, and their requests can simply be directed to the replica. Individual servers that are essential to the continued operation of critical applications can be replicated in this way. However, replication increases complexity, and TP software is necessary to manage consistency across replicated data.

The additional complexity introduced through replication may not be justified. For many applications, small periods of unavailability are inconvenient, but can be tolerated. Designers must trade fault tolerance with increased complexity.

4.2.2.5 System Management

Management issues are complex and problematic for distributed client/server systems. The logical and physical distribution inherent in client/server architectures, coupled with heterogeneity and multi-vendor products makes system management difficult and expensive. For a distributed system, nodes (machines), networks and applications must be managed effectively in order for the system to provide service.

It is not sensible to have a separate system management database and administrator for each application which runs on a distributed system. This approach leads to system administrators having to learn a number of system management products, much duplicated data and effort, and difficulties in correlating output from the products. To combat these problems, *open distributed management (DSM) platforms* have started to emerge.

Open DSM Platforms

An open distributed system management platform (open DSM) is responsible for:

1. Managing multi-vendor devices and applications on the network.
2. Running management applications.
3. Inter-operating with other managing stations.
4. Providing an integrated view of managed components.

A common model for open DSM platforms relies on a manager and a collection of agents. Each agent is responsible for monitoring and responding to changes in a particular piece of the system (a device or an application for example). Agents report to the manager component with real-time data. The manager is responsible for filtering the data it receives from agents, and presents the resulting information to system administrators. Sophisticated managers can analyse their data to identify trends and issue requests to agents to respond to changing system conditions.

An important property of open DSM platforms is that the key interfaces used to exchange data adhere to industry standards, for example SNMP and CMIP.

Open DSM platforms typically provide a number of management tools:

- *Performance monitoring.* Agents gather statistic regarding resource utilisation levels. Resources include CPUs, disks, memory, networks, file systems, application processes, and server transactions. Administrators may define conditions at which some automated action can be invoked, or an alarm can be raised to indicate the condition to the administrator.
- *Inventory management.* An inventory for both hardware and software entities is managed. For software, an inventory tool should track

which software is installed on which machine. Changes in hardware and software can be detected and brought to the attention of system administrators.

- *Configuration management.* Configuration management tools collect information from any managed system. Changes in the client./server system can be tracked over time. Such information is useful for maintenance and tuning to remove processing bottlenecks.
- *Security.* Security tools monitor access to resources and manage which processes can access which resources. It is the network operating system's responsibility to perform authentication and protection, but security management tools provide a friendly interface to maintaining user and group information.
- *Software distribution and installation.* A key problem in distributed client/server systems is managing the explosion of software across several machines. Software distribution and installation tools allow administrators to download, update, track, and de-install software on any machine on the network.
- *Software licensing.* Licensing tools enable metering and enforced use of licensed software.
- *Fault management.* Dealing with faults in the presence of different network protocols, operating systems, and databases add much complexity to fault management. Fault management tools help by reducing the separation of database, network and system management functions to make the process of correlating failure symptoms easier.

System Management Standards

SNMP (Simple Network Management Protocol) and CMIP (Common Management Information Protocol) are standard management protocols which define the structure of data maintained by managed devices. Both protocols conform to the manager/agent model described above.

SNMP is an Internet protocol which is very simple and has been widely adopted for network management. SNMP performs relatively simple management of network devices, allowing attributes to be set and retrieved from a managed device, and limited event management. CMIP is the OSI protocol which plays a similar role to SNMP, but offers richer functionality.

CORBA looks a promising technology to perform comprehensive *system* management - SNMP and CMIP are very limited in this respect. CORBA provides a modern and natural protocol for representing managed entities, defining their services, specifying instance data, and invoking methods via an ORB. CORBA further supports dynamic discovery of entities which when introduced, need to be managed. Using CORBA, distributed system management simply becomes another service on the ORB.

4.3 Data Based Integration

4.3.1 Data Encapsulation and Access

We can basically consider two basic techniques to access legacy data (see also section 2.2.1) from an evolved application:

- direct data access
- data encapsulation

The two techniques, that require substantially different architectural models, are both enabling to preserve the proper level of interaction between the evolved and the legacy applications in terms of flow of data.

As also mentioned in section 2.2.1, the adoption of one technique, with respect to the other, is strongly influenced by the overall enterprise evolution objectives. So, if the objective concerns only a limited number of applications, costs and effort considerations may drive to choose the direct data access option that, in principle, is much more affordable.

If, on the contrary, the enterprise objective is to massively abandon the legacy operational environment, that data encapsulation option is surely more adequate. This technique, in fact, guarantees a much higher flexibility and suitability for the following evolution projects.

Several middleware technologies exist that are specifically designed to support the adoption of the over-mentioned techniques. These technologies differ for the specific option they are conceived for although some of them are suitable to support both.

In this section we will briefly describe some of them. Table 4 summarises some information about the considered technologies as well as the techniques to which they are applicable.

<i>Technology</i>	<i>Provider</i>	<i>Direct data access</i>	<i>Encapsulation</i>
ODBC	Microsoft	yes	
JDBC	JavaSoft	yes	
EDA/SQL	Info. Builders	yes	yes
Oracle Transparent Gateway	Oracle		yes
OpenDM	Siemens Nixdorf		yes

Table 4: Data Encapsulation and Access Technologies

4.3.1.1 ODBC and JDBC

Basic Principles

Open Database Connectivity (ODBC) is an Application Programming Interface (API) for programs that use SQL to access data. ODBC is a multi-database API because an ODBC program can operate with heterogeneous databases and SQL DBMSs without requiring source code changes. Microsoft created ODBC by extending a Call Level Interface from

the SQL Access Group (now part of X/Open). The American National Standards Institute and International Standards Organization has adopted an updated version of that CLI as part of the SQL-92 standard. ODBC 3.0 aligns with that standard.

ODBC, a component of the Windows Open Services Architecture (WOSA), is conceptually similar to the Windows print model, where the application developer writes to a generic printer interface and a loadable driver maps that logic to hardware-specific commands. This approach virtualises the target DBMS because the person with the specialised knowledge to make the application logic work with the database is the driver developer and not the application programmer.

Figure 30 shows the general architecture of ODBC:

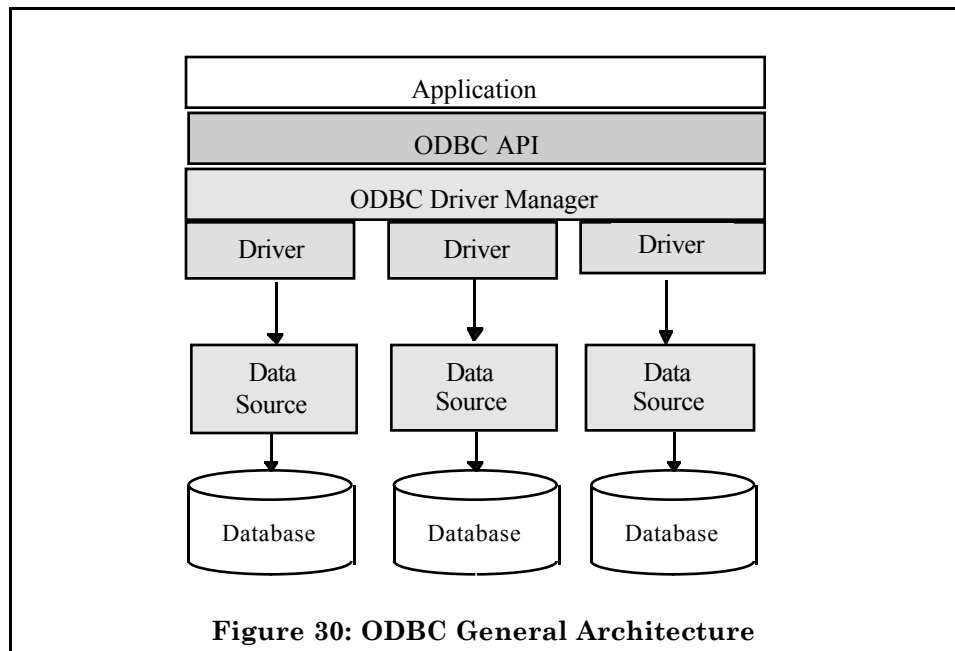


Figure 30: ODBC General Architecture

The ODBC API is a collection of function calls (*such as SQLGetInfo, SQLFetch, and others*) that interact with the ODBC Driver Manager (*a DLL usually found in the user's windows/system directory*), which fields the application calls to the database, sits at a layer above one or more single- or multiple-tier drivers. The job of the multiple-tier drivers is to take the ODBC calls passed from the application and translate them into equivalent calls or protocol in the format expected by the target database. It loads and unloads drivers, performs status checking, and manages multiple connections between applications and data sources. A data source in this case might be an Oracle database or a Microsoft Access MDB file. Single-tier drivers sit directly above a data source and process ODBC calls and SQL statements, that is, they process the SQL and retrieve the data themselves. Borland's Paradox and Microsoft Access are examples of these single-tier drivers.

Note that ODBC can be used with any database for which an ODBC driver is available, this can include object-relational DBMSs as well as non-relational DBMSs such as IBM's IMS.

A recently released product, specifically dedicated for Java applications, is JDBC. Java Database Connectivity (JDBC), by JavaSoft, is a DBMS-independent SQL API for Java applets and applications. JDBC uses SQL queries and provides Java classes that abstract the data access process.

The general architecture of JDBC, as well as the general product scope, is very similar to ODBC. In fact, JavaSoft provides a JDBC "driver manager" that allows vendor drivers for specific DBMSs to be plugged in. JavaSoft provides a special driver, the JDBC-ODBC Bridge that allows JDBC to leverage the database connectivity provided by the existing array of ODBC drivers.

JDBC API is implemented via a driver manager that can support multiple drivers connecting to different databases. JDBC drivers can either be entirely written in Java so that they can be downloaded as part of an applet, or they can be implemented using native methods to bridge to existing database access libraries.

As for ODBC, JDBC can be used with any database for which a JDBC driver is available, this can include object-relational DBMSs as well as non-relational DBMSs.

Areas of Use

These technologies enable the support of direct data access integration technique independently from the architectural model adopted to implement the evolved application. The choice of ODBC or JDBC specifically concerns with the technical peculiarities of the evolved application.

Benefits

The major benefit of ODBC and JDBC is that they enable uniform access to multiple format DBMS products. In the evolution context, this enable relevant legacy data access directly from an evolved application and, consequently, it allows the pre-existing data flow between this and the legacy components to be preserved.

Generally speaking, the adoption of such technologies enables the reduction of data-integration time in that the construction of the necessary interface to preserve the original data-flow can be limited to shared data only.

Risks

One of the main problems with the adoption of ODBC and JDBC technologies in the evolution context stands in the fact that ODBC applications must operate on data sources having direct knowledge of location, network libraries, client libraries, database name, and similar unique identifiers. Moreover the application must take care, wherever this is necessary, of any data mapping feature. This may represent a considerable constraint if decisions to migrate data are taken in that requires to modify the application source code.

A general risk to be considered when adopting these technologies stands in the identification of the data to be accessed and in the provision of

proper mapping features. To reduce these risks a detailed data analysis is the only possible remedy.

Suitability

The suitability of these technologies is constrained by the following factors:

Environment factors: Although the tools work for a wide choice of commercial DBMSs, they are not applicable if specific drivers are not available.

Technical factors: Their usage impose considerable constraints on the adaptability of the concerned application in that the access and manipulation logic is embedded in the source code.

Practical Advice of Use

The adoption of these technologies is recommended in the context of software evolution when the organisation does not plan to have a massive re-engineering of the information system and, moreover, when the organisation do not plan to migrate legacy data.

4.3.1.2 EDA/SQL

Basic Principles

EDA/SQL from Information Builders Inc. (IBI) is a middleware product enabling complete transparency access and join to data located in more than 60 database structures (both legacy - sequential, hierarchical - and relational) on more than 35 operating platforms.

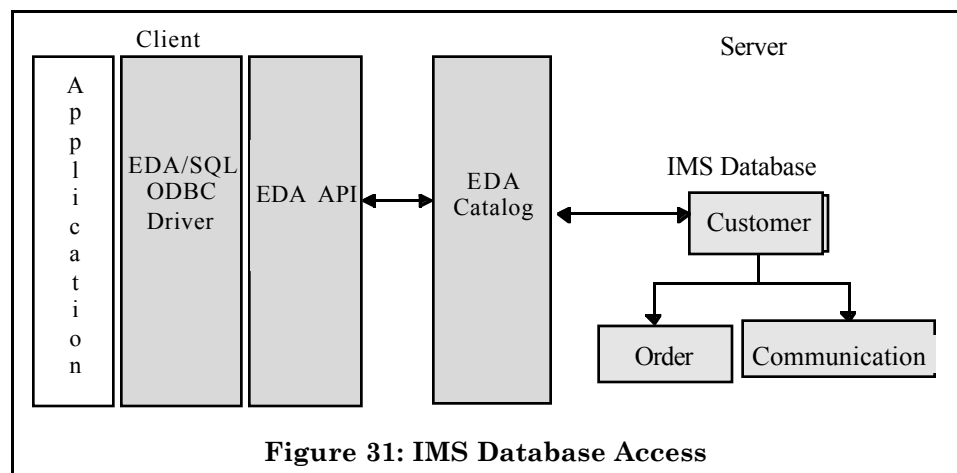


Figure 31: IMS Database Access

EDA/SQL provides an ODBC driver to access IMS data. Figure 31 shows an example of a simple IMS data structure.

Notice that the Customer segment, IMS's name for the entity, has two child segments, Order and Communication. There can be many Order segments and many Communication segments for a single Customer. This hierarchical structure doesn't conform to a relational model of tabular data. This is where EDA/SQL's ODBC driver and API/SQL API step in. The EDA/SQL ODBC driver translates ODBC API calls into IBI's

API/SQL calls. In Figure 31, the application could view the three segments as three tables. The application generates SQL to join the Customer table and Order table to return the columns desired. Any ORDER BY or GROUP BY processing as well as aggregate and scalar functions are resolved on the mainframe, taking advantage of its horsepower.

Internally, IBI uses a mainframe-based dictionary called "EDA catalogue" to determine what specific IMS access mechanisms should be used to resolve the query. The query is translated into IMS calls referencing the appropriate IMS segments.

Additionally, to speed access the IMS database administrator may provide a logical view of the two segments as one relational table, although this is highly denormalised. To ensure acceptable performance, the IMS database administrator must know how applications will access the IMS data in order to tune the EDA catalogue entries.

IBI recently enhanced EDA/SQL to support INSERT, UPDATE, and DELETE statements against IMS databases accessed through ODBC, thus making it more attractive for more robust client/server applications. However, due to the nature of IMS, some core SQL DDL statements are not supported by the EDA/SQL ODBC driver. These include table level statements such as CREATE TABLE, DROP TABLE, and ALTER TABLE. EDA/SQL cannot execute these statements dynamically because the internal architecture of IMS requires the use of specialised administration tools on the mainframe.

Areas of Use

EDA/SQL enables both direct data access and encapsulation integration technique independently from the architectural model adopted to implement the evolved application.

Benefits

The major benefit of EDA/SQL is that this enables uniform access to multiple format DBMS products. In the evolution context, this enables relevant legacy data access either directly from an evolved application or through a meta-schema whose implementation is supported by EDA.

Major benefits can be classified as follows:

- The product is widely open offering direct access to over 60 databases, support for over 300 tools, applications and utilities, supports over 35 hardware and operating platforms, supports for more than 12 API standards (Oracle, Informix, Sybase, IBM, Microsoft, X/Open, Novell, Lotus, DCE, and more), supports over 30 networking protocols, strategic commitment to formal technical standards (ANSI, SQL, ISO CLI, X/Open XA, OSF DCE).
- Native attitude to access legacy data including CICS, IMS/TM, IMS/DC, VTAM, IMS, DBCTL or BMP, IDMS DML or IDMS SQL.
- EDA provides powerful catalogue-based metadata management services for data management.

- Leading companies, including Oracle, Informix, Microsoft, IBM, and Open Environment Corporation, are using EDA for their middleware needs in their own products.

Risks

The main problem with EDA can be that the product does not provide a real DBMS to build the meta-schema.

Suitability

No particular constraint, but the environments for which the product is available hold.

Practical Advice of Use

No particular advice.

4.3.1.3 Oracle Transparent Gateway

Basic Principles

The Oracle Transparent Gateway to EDA is a complete, ready-to-use solution combining proven technologies from Information Builders and Oracle Corporation. The Oracle Transparent Gateway to EDA gives Oracle applications transparent access to mainframe relational data such as DB2 and non-relational legacy data including VSAM, IMS, ADABAS, SUPRA, TOTAL, ISAM, FOCUS, SAP.

Oracle Transparent Gateway to EDA comprises four tightly integrated components:

- EDA Server and data driver
- Oracle Transparent Gateway
- Oracle server (current implementation is for Oracle 7)
- SQL*NET

Oracle Transparent Gateway for EDA requires an IBM or compatible mainframe running MVS XA or MVS ESA. The transparent gateway and EDA components must run on MVS. The Oracle7 server can reside on any Oracle-supported platform. SQL*NET provides the network support between the client and Oracle7 server for any Oracle-supported protocol on both platforms, and support for both APPC LU6.2 and TCP/IP between a remote Oracle sever and the gateway on MVS.

Multi-site queries, joins, and views may be performed in a single SQL statement. Data may also be migrated from the mainframe to an Oracle7 database on the network using a single SQL statement.

Areas of Use

The product supports the encapsulation integration technique independently from the architectural model adopted to implement the evolved application.

Benefits

The adoption of this technology supports access to multiple data source and facilitates data migration towards Oracle DBMS.

Risks

No particular risk

Suitability

The main constraint is represented by the proprietary technology (Oracle) although this is one of the most commonly adopted when migrating data.

Practical Advice of Use

No particular advice

4.3.1.4 OpenDM**Basic Principles**

Open Database Middleware (OpenDM) is an open software product designed to solve different problem areas such as federation of heterogeneous database systems, data migration, and development of DBS independent applications.

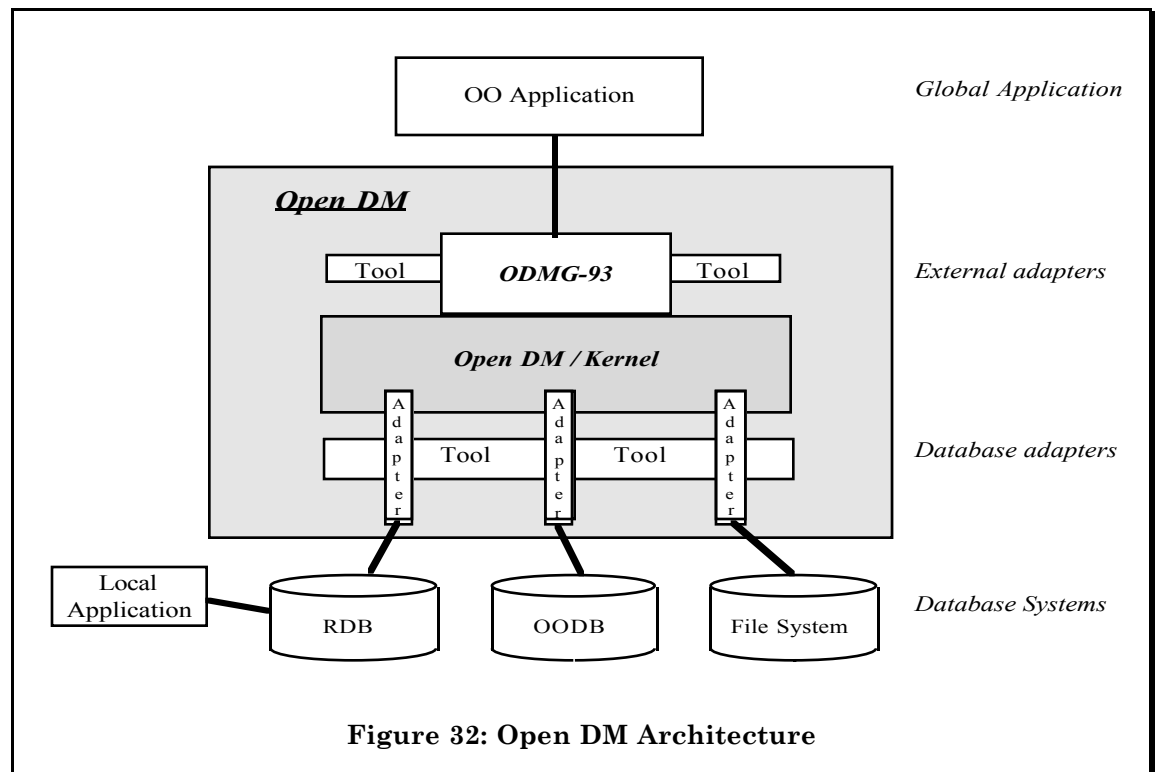


Figure 32 schematises the general architecture of Open DM.

OpenDM/ODMG-93 enables OpenDM configuration providing OO interfaces conformant to the industrial standard ODMG-93 independent from the underlying database system.

OpenDM/ODMG-93 also provide a toolset enabling to design database applications and to compile ODMG-93 C++ binding for ODL, OML, and OQL.

The OpenDM kernel is based on a nested object model which naturally supports multiple schemata and schema derivation and integration. The kernel provides database functionality like data dictionary, object management and caching. Heterogeneous database systems are interpreted by specific adapters (mappers). Automatic schema transformation from ODMG-93 C++ ODL to SQL92 is possible.

Currently available or under development adapters encompass the following database systems: File system emulation, Adabas, Oracle 7, ODBC, O2.

Areas of Use

This technology enables the planning of application evolution and data migration towards object technology.

Benefits

The main benefit consist in the possibility to work now for future system evolution towards object oriented. As a consequence this brings a number of well known advantages that are typical of OO systems. Within the evolution scenario, the major one is, probably, the possibility to model within the meta-schema also the semantic of the represented data.

Risks

The major risk is represented by the low maturity status of the technology that has, so far, too many bits that are still under implementation.

Suitability

The usage is very constrained by the limited operational environment supported as well as by the running environment.

Practical Advice of Use

This is a promising technology probably worth to be experimented within a limited context before applying it to support extensive projects.

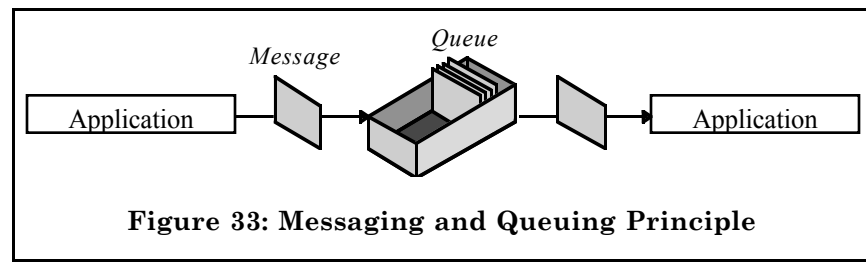
4.4 Service Based Integration

4.4.1 Messaging and Queuing Middleware

4.4.1.1 Basic Principles

Messaging and Queuing, also called commercial messaging and queuing, is a successful combination of two information passing technologies:

- *Messaging* is an asynchronous method for passing information between programs. Contrarily to synchronous communication, messaging is time-independent and does not require involved programs to be running or communication paths to be established. As a basic technology, messaging can also be used to implement synchronous communication.
- *Queuing* is a method for passing information indirectly through message queues instead of directly from one program to another one. Communicating programs do not have to bother about and establish connections. They rather connect to queue managers which free them from the underlying complexity of networking.



Messaging and Queuing middleware allows system to be developed according to an asynchronous, event driven model. An application sending a message in a queue is not blocked and can continue its processing. This removes time dependencies between applications.

4.4.1.2 Areas of Use

Commercial messaging can be seen as the basis to implement a software architecture where programs communicate through queues instead of directly. As developed before, this brings several benefits, but implementing communication using messaging and queuing is a complex issue which should not be underestimated. Experienced architects are needed to cope with the complexity of asynchronous versus synchronous messaging.

Many current uses of commercial and queuing systems are more simple and implement gateways between heterogeneous platforms. Typical examples of use include data communication and asynchronous transfer between a central server and a department server, for instance to send updates from a department or local database to a central database.

Indeed, in this latter case, reliability is more important than real-time update.

In the framework of the evolution process, commercial and queuing middleware major interest comes from the fact that products implementing it run on a large set of hardware platforms, operating systems and can be used from a variety of programming languages or commercial products. Commercial and messaging implementations run on most legacy systems platforms and can be used by existing languages and developers. Thanks to the loose integration mechanism and the large set of potential solutions it provides, messaging and queuing middleware is an ideal way of connecting to legacy systems.

4.4.1.3 Benefits

The benefits of messaging and queuing middleware are multiple::

- Isolates applications and application developers from network complexity.
- Runs on top of a variety of communication protocols.
- Scalability: servers can easily and transparently be added to cope with an increase of users or messages.
- Time-independence between applications: applications do not need to be running at the same time to co-operate.
- Allows a variety of communication styles, from asynchronous to synchronous.
- Guaranteed and unique message delivery.
- Concentrate communication by grouping messages transmission over networks.
- Application location transparency by implementing communication through queues, independently of applications.
- Loose application integration: applications are integrated through queues and messages.

4.4.1.4 Risks

Implementing messaging and queuing solutions beyond the scope of simple gateways between platforms, to implement more complex architectures, is a complex architecture which require different way of thinking of an application architecture.

Messaging and queuing usually provide a number of factor to tune the overall performance. However, administration and tuning can be a complex activity.

4.4.1.5 Suitability

Hardware Environment

One of the strength of is that several products, among which IBM MQSeries and DEC messageQ, run on a large set of platforms and operating systems, ranging from mainframes to desktop computers, on a variety of network protocols, including TCP/IP and SNA.

Software Environments

Major products implementing messaging and queuing middleware provide interfaces with main programming languages (C, COBOL...) and with external products (IMS, CICS...).

Security

Products usually implements several authentication mechanisms. For instance, MQSeries allows application to either send messages based on the current user identity, based on the original sender identity in case of a message to be forwarded, or with another user identity. Administration tools, allow authorisation to use such or such facility to be defined for all users.

Scalability

Messaging and queuing products act as an independent layer between a sender (client) and a receiver (server) responsible for communication. Products are designed to handle large numbers of queues and messages.

In addition, due to the nature of communication, the number of applications processing queues (servers) can be increased as needed.

Openness

Interfaces to messaging and queuing systems are usually relatively simple by the number of basic services they provide. However, in complete implementations, the number of options and parameters can be very large.

Fault Tolerance

Messaging and queuing systems are very robust. They guarantee unique message delivery. To prevent problems, messages can be defined as permanent (they will be saved in a persistent storage area will waiting for delivery).

System Management

In a product like IBM MQSeries, there is no management tool included. However, the messaging system generates system management events which can be used by system management tools. Basic system management utilities can be downloaded free from IBM development site or more complete (and expensive) management tools supporting MQSeries.

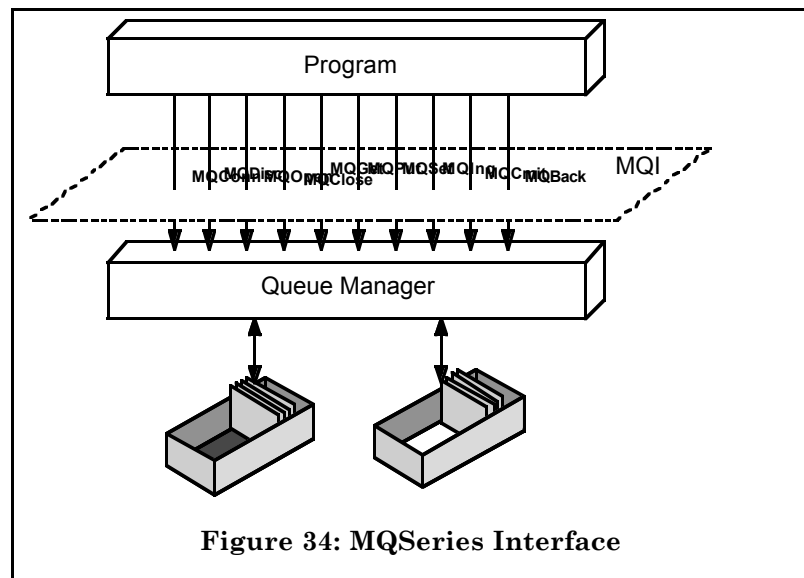
4.4.1.6 IBM MQSeries

MQSeries is IBM's implementation of commercial messaging and queuing middleware. MQSeries is the leading messaging and queuing product. MQSeries consists of the following components:

- *Queue Managers* provide all message queuing services to applications which are connected to it. It manages all local queues, communication with remote queue managers and allows programs to get messages from or put messages in queues.
- *Messages* contain data exchanged between programs. They consist of a header and of an unformatted raw data part.

- *Queues* store messages put by applications until they are retrieved by other applications.
- *Channels* are in charge of exchanging messages between remote queue managers.

The interface between an application and MQSeries is done by calling a Queue Manager through the MQ Interface (MQI):



The MQI consists of a limited set of calls:

- **MQConn:** Connects to a Queue Manager
- **MQOpen:** Opens a queue and prepare its use. Access modes are specified, and most security checks are performed at this stage.
- **MQSet/MQInq:** Sets/inquires a Queue (or Queue Manager) attributes.
- **MQGet:** Gets a message from a (local) Queue. Messages can be retrieved with a variety of options, especially in a blocking or non-blocking way.
- **MQPut/MQPut1:** Puts a message in a Queue (local or remote). MQPut1 is a convenient and more efficient shortcut for Open + Put + Close.
- **MQCmit:** Indicates that a new syncpoint has been reached and commits all operations (get and put) until the last syncpoint.
- **MQBack:** Indicates that a new syncpoint has been reached and roll-back all operations (get and put) until the last syncpoint.
- **MQClose:** Stops using Queue.
- **MQDisc:** Disconnect from Queue Manager.

In addition to basic messaging, MQSeries implements several functions:

- Integrity

- Persistence
- Information Expiration
- Notification
- Event and Triggering
- Security/Authentication

4.4.1.7 Other Commercial Messaging and Queuing Products

Apart from IBM MQSeries, there are a number of other messaging and queuing products:

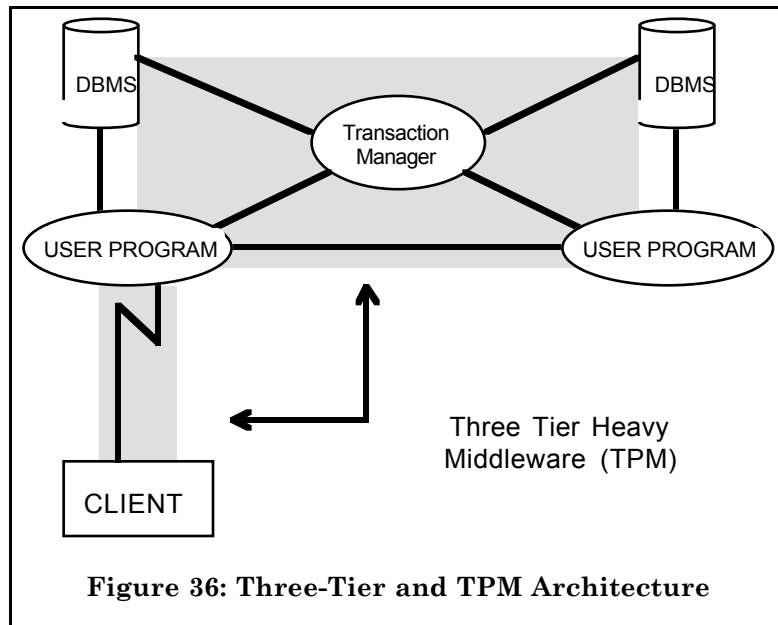
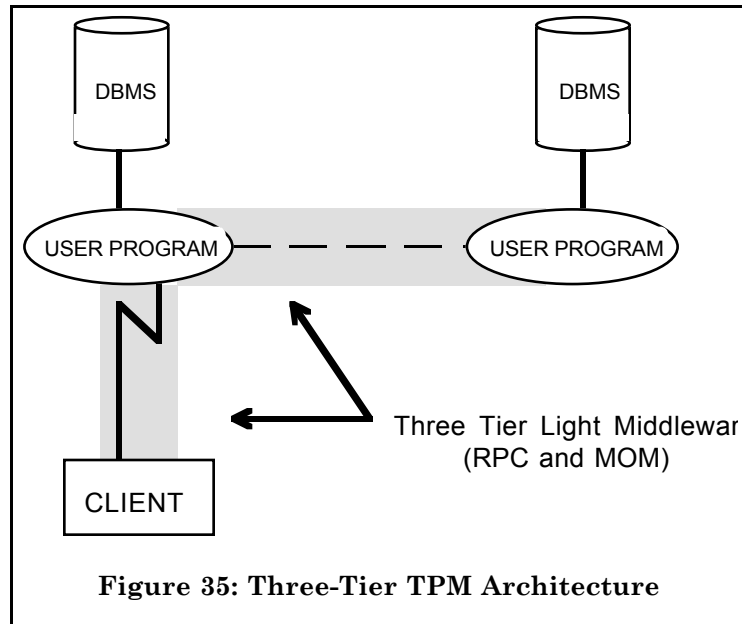
- DEC / messageQ (now taken over by BEA)
- NetWeave Corp. / NetWeave DS
- Momentum Software Corp. / X-IPC
- Verimation / VCOM
- PeerLogic / Pipes Platform
- Transaccess / Netwise

These products are either standalone products or extensions to existing solutions to provide integrated middleware solutions including Transaction Processing Monitors, Object Request Brokers and Messaging Oriented Middleware.

4.4.2 Transaction Processing Monitors

4.4.2.1 Description of Transaction Processing Monitors

Transaction processing (TP) and transaction monitors represent an important part of general client/server computing, offering a set of facilities that are indispensable for building commercial-strength applications. As client/server applications mature and become the basis for mission-critical solutions, it becomes apparent that for such solutions to be effective, a component is required in the system that manages the interactions between all the participants. This is the traditional role of the online transaction processing (OLTP) monitor, in which transactions, rather than being mere business events, represent a philosophy of application design that guarantees robustness in a distributed system. Transaction processing monitor (TPM) technology not only reduces system costs, but also protects existing investments in mission-critical mainframe production applications and data as well as CICS programming skills that are present in thousands of M.I.S..



Transactions provide a simple model of success or failure. A transaction either commits (i.e., all its actions happen), or it aborts (i.e., all its actions are undone). This all-or-nothing quality makes for a simple programming model.

The next four properties (also referred as ACID properties) describe the key features of transactions:

- *Atomic*: All or nothing, either all the actions happen or none do.

- *Consistent*: The transaction as a whole is a correct transformation of the database.
- *Isolated*: Each transaction runs as though there are no concurrent transactions.
- *Durable*: The effects of committed transactions survive failures.

Database and TP systems automatically provide these ACID properties. They use locks, logs, multi-versions, two-phase-commit, on-line dumps, and other techniques to provide this simple failure model. All the programmer needs to do is to write consistent programs and bracket them with BEGIN and COMMIT. If anything goes wrong, the programmer can call ROLLBACK much like the quit function in a text editor. This simplicity is especially important in client/server computing and distributed databases, where the transaction may have done work at many nodes. COMMIT makes all the changes at all the nodes durable; ROLLBACK undoes all the changes.

4.4.2.2 Transarc's Encina Monitor

Encina Monitor, one of the prominent TP monitors for non mainframe environments, is a function-rich, three-tier-heavy technology with uniquely deep integration of DCE and powerful MVS). Encina Monitor enables users to optimise their distributed application environment and to distribute their applications across many server platforms, various client platforms (including the World Wide Web browsers) and multiple DBMSs. Users of Encina Monitor utilise the distributed security features of DCE, extended transactional RPC technology and DCE directory services. Encina is the only TP monitor that works directly with C++. Encina Monitor is a flagship product of Transarc Corp., a subsidiary of IBM and it is part of a family of related products. Encina Monitor both enhances the DCE technology and also depends on it but, however, unlike DCE, Encina Monitor is a proprietary technology.

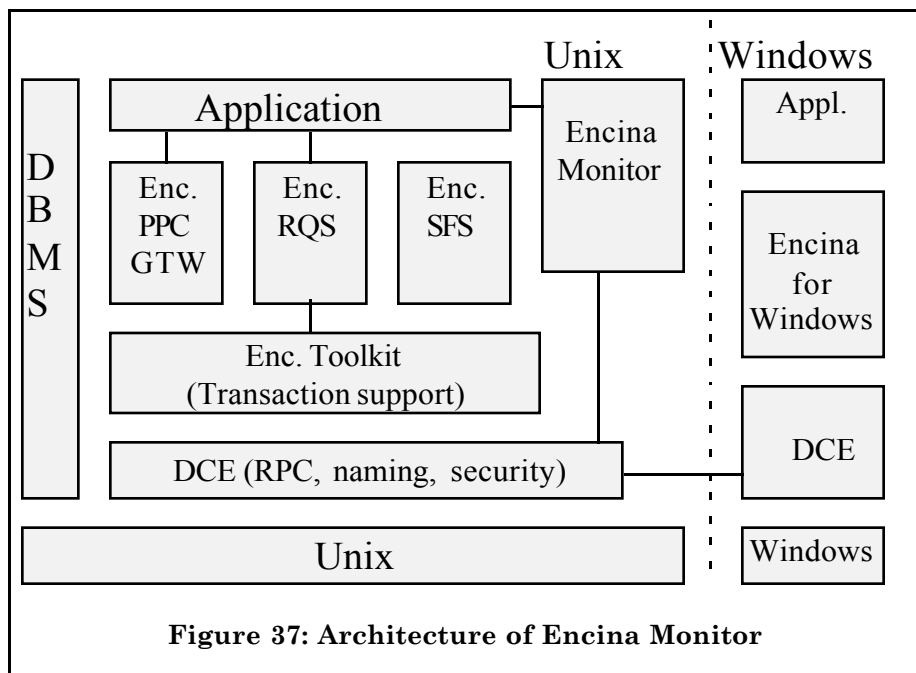


Figure 37: Architecture of Encina Monitor

Encina and DCE were developed as vendor-targeted technologies. DCE security, DCE naming or the Encina Toolkit are used in many vendor products. So, there are 600 API calls in the DCE and Encina Monitor libraries, and this means power and flexibility.

A key feature of Encina Monitor is its *integration with MVS* Transarc and IBM developed the CPI-C bridge to MVS, which is one of the most-powerful examples of UNIX-to-MVS connectivity technologies. The PPC Gateway permits two-way transactional interaction between the UNIX and MVS environments that can be originated or controlled from either side. This is one of the product's strongest features.

4.4.2.3 Other Commercial TP monitors

Portable (non mainframe) TP monitors have evolved into general-purpose, production-oriented middleware subsystems. These monitors include:

- NCR's TopEnd,
- IBM's Open CICS (on AIX, HP/UX, Windows NT and other platforms)
- BEA's Tuxedo
- Siemens Nixdorf's UTM
- UniKix's UniKix

4.4.2.4 Role of TP Monitors in the Evolution Process

The general benefits of TP monitors are the following:

- Transactional three-tier application partitioning
- Transaction integrity across different DBMSs

-
- Improved online TP performance and systems resource utilisation
 - Distributed security
 - Location-transparent procedure calls
 - Choice of interoperability models (e.g., RPC, messaging and peer-to-peer)
 - Desktop connectivity
 - Mainframe connectivity
 - Non relational transactional data access
 - Cross-platform portable middleware

The main problem for the data migration, is usually not the use of a DBMS vs. a TP monitor, but when is necessary to use a DBMS supplemented by a TP monitor. A response is that a TP monitors can help in many ways:

- They provide integrity support (three-tier heavy) for transactions that involve multiple programs or components that may be distributed across one or many systems. This enables a high level of flexibility in design of applications.
- They facilitate the implementation of sophisticated application designs by providing nested transactions, chained transactions, access to legacy MVS applications (such as via a CICS gateway), or asynchronous processing (all of the major distributed TP monitors now support a queuing option).
- They provide integrity support for heterogeneous DBMS applications. i.e., those that use more than one DBMS.
- They reduce administrative complexity, particularly for large and dynamic applications of more than 100 users or 100 application programs. TP monitors have their own administration facilities and collect statistics, such as the number of times an application is invoked, enabling smarter tuning than is achievable through DBMS and operating system tools alone. Applications running under a monitor can often be managed using common Unix debugging, change management and system management tools, unlike DBMS stored procedures.
- They reduce CPU overhead, response times and CPU cost for large applications. Although modern multithreading DBMSs, such as Oracle v.7, do not have as many server processes as were required under earlier DBMS architectures, they have not matched monitors in their ability to reduce communication overhead between front-end requester systems running presentation work and back-end servers. In general, a TP monitor can deliver more consistent response times and better load balancing, particularly if there are multiple server systems.
- They improve application up-time, using a TP monitor's automatic failover and restart features. To implement equivalent functions with

a DBMS-only solution would require extensive application programming or additional middleware products, such as an RPC communication package. TP monitors also help isolate business logic (codified in application programs) from presentation logic and DBMSs, facilitating modular, extensible application designs and promoting code reusability and DBMS independence.

4.5 Presentation Based Integration

4.5.1 World-Wide-Web User Interfaces to Legacy Systems

4.5.1.1 Basic Principles

World Wide Web techniques provide access to various systems through a unique tool, the web browser, and from almost anywhere through the public Internet. Therefore, these techniques allow an existing system to be integrated at the presentation level.

Two major techniques exist to interface with the presentation of a legacy system:

HTML/Java terminal emulators

HTML and Java based terminal emulators emulate screen-based terminals by displaying the contents of terminal pages in the web browser window. Therefore, they offer the same type of user interface display and control.

HTML/Java page/form generation from screen/panel definition

HTML/Java page/form generation from screen/panel definition (CICS, IMS...) allows the flow of information usually sent to a terminal to be analysed and reformatted to display this information using mechanisms of the graphical user interface. This technique fits into the revamping techniques (see section 3.3, *Migration to Web-Based User Interfaces*).

In both cases, interpretation and modification of the presentation flow can be done (see Figure 38):

- at the browser level (① on Figure 38), using an add-on component with the browser (browser *plug-in* which must be downloaded and installed manually, or embedded component such as a Java applet or an Active X control which is downloaded and installed automatically)
- at the server level (② on Figure 38), with a software component connected to the web server in charge of transforming the character flow into HTML pages.

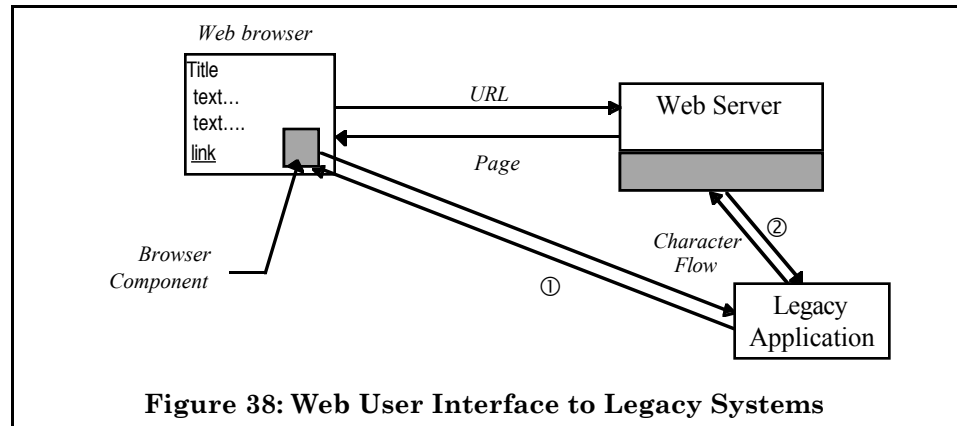


Figure 38: Web User Interface to Legacy Systems

4.5.1.2 Areas of Use

These techniques are lightweight integration techniques. They can be used when no modification wants to be done on the legacy system, especially when a decision is taken to replace character terminals with more powerful desktop computers accessing legacy applications and running new applications developed using productivity tools.

4.5.1.3 Benefits

Benefits are similar to those coming from the revamping techniques:

- The integration is light and cheap (without considering the cost of new desktop computers).
- There is no fundamental change in the user interface with the system. There is no or little need for training users.
- Old terminals can still be used to access the application, in parallel with the new access. This allows an incremental move to the new desktop platforms.

4.5.1.4 Risks

Most risks of using these techniques are rather limitations:

- This solution is a short-term solution for evolution and most limitations of the existing system are not solved.
- By becoming more open, access to the system, even if secured using firewalls and authentication, is more subject to security attacks.

4.5.1.5 Suitability

Since the legacy system is not modified and its user access is almost not modified, there is little modification to the legacy system environment or technical evaluation criteria.

Environment Constraints

The legacy application is not modified at all. The web server is usually hosted on another platform which is also in charge of the communication with the legacy system. There is no specific platform constraint for the tools used.

Most tools supporting these techniques run over a TCP/IP network. Mainframe based legacy applications can be based on more proprietary

protocols. If access to the legacy system is done through a web server hosted on an intermediate platform, this platform acts as a gateway and communication protocol converter. If the legacy system is directly accessed by a browser component, it may be necessary to add a TCP/IP communication stack on the legacy system.

Security

Communication protocols used in web techniques are much more public than legacy system protocols. Even if web based systems can be secured using authentication and crypting techniques, the door is more open for security leaks.

Scalability

The legacy system is not modified at all, so there is no impact on its scalability. The load on the legacy system remains the same with no more ability to cope with changes of scale.

4.5.1.6 Supporting Tools

The number of tools supporting these techniques has followed the growth of the interest for Internet techniques and solutions. Solutions are provided by independent software vendors and by legacy platforms providers as an easy way to open them to the Internet.

For example, IBM techniques portfolio regarding this area includes:

- *CICS Internet gateway* communicates with a CICS client to dynamically generate HTML pages
- *CICS gateway for Java* allows Java applications to be build accessing legacy applications using the External Presentation Interface (EPI) to CICS applications
- *IMS Web studio* creates CGI scripts in C++ to be connected with a web server to be able to access IMS transactions (formatting pages based

4.6 Component Based Integration

Before presenting and discussing the CORBA and OLE/DCOM component standards, we offer some practical advice for migrating to a distributed client/server architecture using component technology. In particular, we provide guidance for selecting a component standard, expose problems and suggest techniques for legacy component encapsulation, identify appropriate migration paths, and urge developers to exploit emerging componentware markets.

Introducing the Software Bus

Traditional thinking regarding the choice of component standard has argued that CORBA should be used when developing for non-Windows platforms, and DCOM is suited to Windows-only development. Recently however, management of DCOM has moved from Microsoft to Open Group, with the result that DCOM is being ported to non-Windows platforms. DCOM is thus no longer a proprietary technology, but open and scaleable. For CORBA, efforts have been made by Netscape amongst others to integrate CORBA components with desktop computing.

Rather than selecting one of the standards, it might be sensible to use both. For example, the use of OLE/DCOM for clients and CORBA for hosting server components is a possible model. This model is made possible by adopting one of several interoperability schemes. For example, the OMG have specified a COM/CORBA standard for bridging DCOM and CORBA. Implementations of this standard allow CORBA components to request services from DCOM components and vice versa.

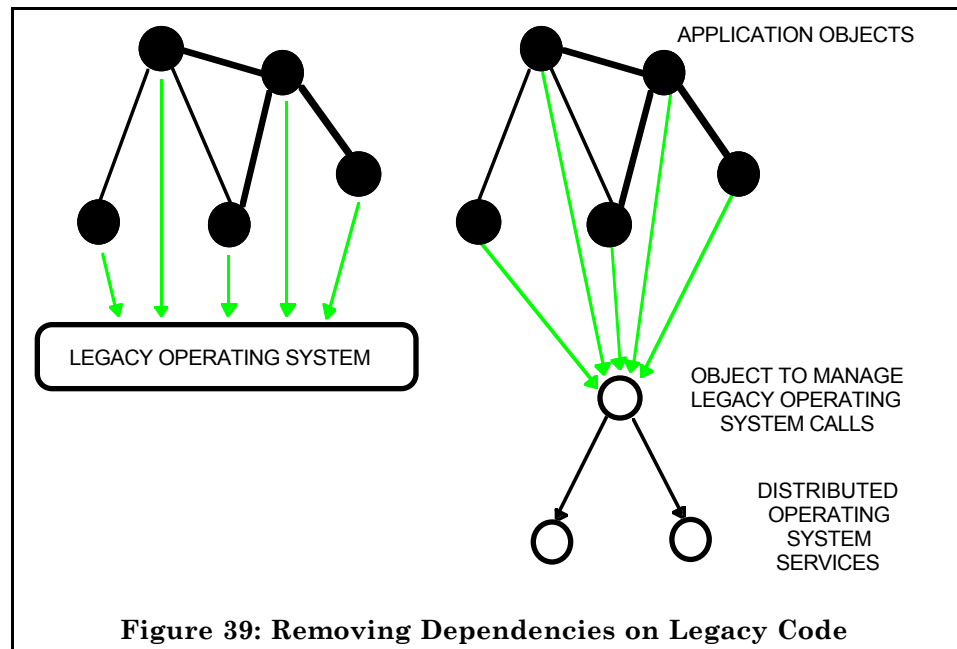
The benefits and risks of each component standard are discussed in sections 4.6.1 and 4.6.2. In general, however, if Microsoft Windows platforms are not part of the target architecture, there is no reason to use DCOM and OLE. Where PCs are to be used, several OLE/DCOM compliant packages (4GLs and compound document applications for example) and a healthy componentware market exist to build client applications.

Developing components is considerably simpler using CORBA than DCOM. Further, where legacy code is to be encapsulated in components, CORBA offers binding tools which will help bind component interfaces to implementations written in legacy programming languages. For DCOM, programming components in anything other than C++ is very difficult.

Once the software bus has been selected and installed, components can be plugged into it. Any existing services on the network should be encapsulated as components. This is to be encouraged for two reasons:

1. Encapsulating existing services with component interfaces means that the services can be used by other components which share the bus in a uniform manner.
2. Changes in services can be hidden behind the component interfaces. Where platform changes are introduced for example, applications are shielded from the changes because component interfaces provide an abstraction over the platform dependent services. This is a step towards evolutionary applications.

Encapsulating Legacy Code



Legacy code encapsulation is, in principle, a powerful technique for preserving critical business knowledge, while simultaneously providing for a relatively fast migration path from centralised to client/server technology.

However, developers must be aware of the practical constraints placed on encapsulation:

- Legacy dependencies.* Simply encapsulating existing source code with component interfaces does not solve the problem of component implementations (legacy code) making legacy operating system calls, or calls to legacy support software which is not to be used in the evolved architecture. Legacy support software might be TP monitor or database software for example. Such software may not be supported on target platforms, or it might be an evolution requirement to retire the support software. Whatever the reason, legacy code which relies on an obsolete operating system, or obsolete support software, cannot simply be encapsulated. References to obsolete software must first be removed.
- Legacy software architecture.* Component technology has been developed from object technology, and so offers an object-oriented view of the world. Components are encapsulations of state and behaviour. Where legacy applications exhibit any structure, it is highly likely to be based on a monolithic model where data is shared by functions. The task of transforming such an application to an architecture based on objects has to be assessed for feasibility.

Encapsulation first requires that discrete objects can be extracted from a legacy application.

- *Support for legacy code.* To encapsulate legacy software, it must be written in a well defined programming language. A compiler must be developed to provide a language binding between the interface definition language and the programming language of the legacy software. CORBA implementations typically provide language binding support for current object-oriented languages such as C++, Java and Smalltalk. While such language support is of little use for encapsulating legacy code, compiler-generator tools are often available for CORBA implementations, which assist developers in creating a compiler which will map interface definition code onto legacy code.

To address the legacy code dependency problem, a component (object) can be introduced to deal with legacy operating system calls. The interface of this component should reflect the capabilities of the legacy operating system. The implementation should transform the calls to equivalent service requests provided on the new software bus. All programs comprising the legacy application must be scanned for legacy operating system calls - these must be replaced with calls to the new component. It is thus the new component which should be accessed rather than the actual underlying operating system to prevent the problem occurring with subsequent operating system changes.

This transformation is shown in the figure **Error! Reference source not found.**. On the left of the figure is a legacy application which has been structured as a collection of objects, but which still makes calls to the legacy operating system. On the right side of the figure is the solution described above. In practice, automated tools are necessary to perform this process, especially for sizeable applications. Similar techniques can be used for obsolete data and TP monitor references.

Migration strategies

A constraint placed on component encapsulation is that discrete objects must first be extracted from a legacy software architecture. Depending on the ease and success of performing this activity, there are essentially three strategies to achieve migration to a component-based client/server architecture:

1. *Redesign with reuse.* The most radical approach is to redesign the legacy application according to object-oriented principles. This approach is not absolute - there may of course be opportunities for reusing elements of the legacy application, but in general adopting the new architecture means developing new software which is designed for the new paradigm, rather than encapsulating old code. This approach carries the risks associated with rewriting systems - perhaps most important is the risk of losing business knowledge embedded in the old application.
2. *Rapid migration.* On the other extreme, it might be sensible to wrap the entire legacy application into a single component and plug that into the bus. Any external interfaces provided by the legacy application must be exposed by the component interface used to encapsulate the legacy application. This is clearly an extreme

approach - no attempt is made to restructure the application in any way, and it is not possible to exploit the potential of a distributed system. The single component is bound to one machine. This approach is of course subject to the constraints placed on encapsulation introduced above.

3. *Incremental re-architecturing.* As a compromise between the first and second approaches, some effort can be made to "objectify" the application. If it is possible restructure the application towards an architecture which resembles discrete modules of data and services which operate on that data (i.e. objects), then these can be given component interfaces, and distributed across the software bus. With this approach, it is possible to start reaping the benefits of distributed computing. Similarly to approach two, each object to be encapsulated is subject to encapsulation constraints.

The first approach results in a distributed object-oriented solution. It is the ideal end product of an evolution exercise. It is however not incremental and is associated with risk. Approaches two and three are incremental approaches, which may ultimately lead towards the end result of approach one.

The second approach should be adopted when it is not possible to transform a legacy application to an object-based equivalent. Where data is shared across much of the system, restructuring may be prohibited. Some legacy application transformation may be possible, based on service extraction, rather than object extraction. Although software buses are designed to distribute objects with collections of services, they can be used to distribute services, much like RPCs are used to provide remote services.

Encapsulating an entire application inside a component interface does however offer potential to utilise underlying client/server technology. System users work with more general and flexible PCs than dumb terminals for example. Clients are thus able to exploit compound document technology and access other services which share the same software bus as the encapsulated application. Application interoperability across collections of software buses is provided for where the buses are linked. Once the transition to a software bus has been made, the monolithic component (application) can be decomposed with resulting components distributed across the software bus with little disturbance to clients.

Approach three offers an incremental evolution path which attempts to encapsulate as much of the legacy system in components as possible. This approach relies on the application being transformed to an object-based structure. Where such restructuring is possible, components which are simply encapsulated legacy code can subsequently be rewritten using new technology. Providing the component interfaces remain unchanged, changing to the new technology does not affect other components on the software bus.

Exploiting Componentware Markets

During componentware migration, developers should see what components they can buy from third-party componentware markets. It may be more cost-effective to buy a third-party component which is likely to be robust because it was designed for reuse, rather than to invest

development resources in creating a similar component. A rich componentware market is emerging for OLE/DCOM compound document applications and custom components. A similar componentware market for CORBA components is expected in the near future.

As an example, consider a legacy application which uses a text-based editor. The legacy text editor offers primitive services, but it would be advantageous to enhance the services with a spelling checker for example. Using a commercial and component-standard word processor as a replacement for the original text editor offers text editing which is reliable, comprehensible, and affordable.

Componentware markets can further be exploited by marketing components which have been developed in-house for particular applications, and which are also likely to be useful to other organisations. This can recoup some of the evolution costs. Developing componentware should also be encouraged to develop smaller, nimbler and more competitive components which plug into the software bus.

4.6.1 CORBA

CORBA (Common Object Request Broker Architecture) is a product of the OMG (Object Management Group). The OMG is a non-profit consortium, sponsored by over 500 organisations. The overall objective of OMG is to «promote the theory and practice of object technology for the development of distributed computing systems». The approach adopted by the OMG is to «provide a common architectural framework for object-oriented applications based on widely available interface specifications... conformance to these specifications will then make it possible to develop a heterogeneous applications environment across all major hardware platforms and operating systems».

4.6.1.1 Basic Principles

CORBA is a specification for a standard object-oriented architecture for distributed applications. It is composed of four elements:

1. *ORB (Object Request Broker)*. The ORB is fundamental to the CORBA architecture. Its role is to enable requests to be carried out in a heterogeneous distributed environment. When a request is made of a particular object implementation, the ORB locates the object, prepares the object for receiving the request, sends the request and returns the results back to the client. The ORB provides a number of interfaces to developers:
 - *Interface repository*. This provides developers and clients with a run-time database of all registered classes and their interfaces available on the bus.
 - *Static invocation*. A client may be compiled with a static means of invoking a particular server's services. Similarly to RPC, the client is compiled and bound to a server object's service stubs.
 - *Dynamic invocation*. Clients may discover objects at run-time, and dynamically build requests using the dynamic invocation interface.

-
- *Object adapters.* Object adapters provide a run-time environment for instantiating server objects, passing requests to them, and assigning CORBA references to them.
 - *Implementation repository.* This provides run-time information about the classes a server supports, objects instantiated on the server and their CORBA references.
2. *Object services.* A set of services which augment the ORB. Object services can be thought of useful services which application developers would typically have to write themselves, if they were not provided by CORBA implementations. Object services are presented with CORBA-compliant interfaces. Currently, the following services are specified, but not necessarily implemented by CORBA implementations:
- *Naming service.* A directory service which is inter-operable with existing directory services.
 - *Event service.* A service which allows objects to register interest in particular events. On occurrence of an event, those objects which registered an interest in the event are notified.
 - *Persistence service.* This service allows objects to be made persistent. Persistent objects live beyond the creating thread or process.
 - *Transaction service.* Using the two-phase commit protocol, this service allows groups of objects to be updated consistently.
 - *Relationships service.* This services allows relationships to be defined between objects. Users can browse the linked structures.
 - *Externalisation service.* A state capture service used to record the state of a particular object. The external representation of the object can be used to initialise another object instance elsewhere on the bus.
 - *Life cycle management service.* Support for the lifetime of an object, such as initialisation, copying, moving, and destruction. Composite objects can also be managed by this service by relying on the relationships service.
 - *Concurrency control service.* A service to enable multiple clients to co-ordinate their access to shared objects. The service is implemented by various degrees of locking to provide flexible concurrency control.

CORBA services are evolving, and so the above list is not static. Security and licensing are examples of new services which are currently being specified.

3. *Common facilities.* Additional services which an application might require. Common facilities are poorly defined at this time. This is because the majority of effort has been directed towards stabilising the ORB and object services. It is intended that common facilities will be built on top of object services, and classified according to application domain requirements - financial, or health care for example.

4. *Application objects.* Objects with CORBA interfaces which are developed outside of the CORBA architecture.

In addition to the architecture defined above, CORBA defines a particular object model. A CORBA object is an enforced encapsulation of state, with a well-defined language-neutral interface, described by an IDL (Interface Definition Language). Inheritance is supported in the form of (multiple) interface inheritance. The rules for inheritance are strict - a derived interface may only extend the operations provided by a base interface. A derived interface is thus a subtype of its base interface.

The IDL looks much like C++, in an attempt to ease component development using C++ as the implementation language. An IDL-specified interface may export methods (services), exceptions, modules, attributes, constants and types. Binding IDL-specified interfaces to implementation languages requires a compiler to map the IDL declarations to a programming language. Typical programming languages supported by CORBA implementations include C++, C, Smalltalk and Java.

CORBA 2.0 resolves a fundamental flaw in the CORBA standard. With CORBA, interoperability across implementations from different vendors was not specified. CORBA 2.0 implementations are guaranteed to interoperate.

4.6.1.2 Areas of Use

CORBA implementations are well suited to medium to enterprise scale distributed object client/server solutions. CORBA supports the technique of encapsulation, and so is well suited integrating legacy application code in a new architecture. As a consequence of supporting encapsulation, migration to CORBA can be performed incrementally.

4.6.1.3 Benefits

CORBA facilitates the development of true open client/server systems in a heterogeneous and distributed environment. The future looks assured for CORBA in this capacity, given the substantial interest from the OMG and CORBA implementation vendors.

CORBA is well documented and relatively easy to use compared to Microsoft's DCOM technology. Encapsulating legacy components is in particular much easier with CORBA, and there is rich support for programming languages.

The typical language bindings supported by CORBA implementations are of little use to business-domain legacy applications. However, implementations generally provide compiler generator tools to assist developers create their own bindings to a particular programming language. While this requires more effort than simply using a pre-developed compiler, it is nevertheless possible to encapsulate code implemented in non-supported languages.

CORBA implementations span a variety of suitable platforms for client/server architectures.

4.6.1.4 Risks

CORBA doesn't yet enjoy the levels of componentware which have emerged for Microsoft's rival component technology. We do not however see Microsoft's market advantage in this respect as a critical success factor for CORBA. Third party involvement should produce additional object services and common facilities for CORBA.

Mainframe platforms are an obvious omission to the platform supported by CORBA. A business goal of an organisation might be to increase interoperability across its disparate applications. CORBA could be used to realise the increased interoperability, allowing the legacy systems to talk to each other. However, adopting CORBA might force a move to a new platform, which in turn could force further migration, of database technology for example, if the legacy database is not supported on the new platform.

4.6.1.5 Suitability

CORBA is suited to medium to enterprise solutions, and as such is able to meet the demands of large scale interoperability - a common business objective.

There is much potential for preserving application software from the current environment of a legacy application by encapsulation of legacy code. The hardware and software architecture of the current environment must generally be replaced during client/server migration using component technology.

Technical constraints such as scalability and openness are met by definition of the CORBA standard. Although not currently available, security services are now being specified by the OMG. Implementations of the security service will be provided as part of the distributed operating system. Fault tolerance, beyond the natural availability of services in a distributed environment will be supported with the advent of replication services. CORBA is likely to be used as the infrastructure for system management in the future - with management tools being provided by ORB services.

4.6.1.6 Supporting Tools

There are a number of CORBA 2.0 implementations available. We identify some characteristics from a few CORBA implementations.

- *DSOM*. DSOM (Distributed System Object Model), from IBM was the first CORBA implementation. DSOM is packaged with IBM's OS/2 operating system and is used as the underlying component technology for OpenDoc. DSOM supports interoperability across the NetBIOS, TCP/IP, and IPX transport protocols. The SOMObjects developers toolkit includes a C++ framework to provide IDL parsing support for programming languages not supported directly (i.e. C and C++).
- *ORBIX*. Initially released in 1993, ORBIX is produced by IONA Technologies, and supported on a variety of platforms: UNIX (including HP, Sun, IBM, and DEC), Windows 3.1 and NT, OS/2, Macintosh, QNX, LynxOS and the VxWorks real-time operating system. The ORBIX IDL compiler provide language bindings for C++,

Smalltalk, Ada and Java. ORBIX can be packaged with a number of other technologies. In particular:

- ORBIX-OLE provides for automated generation of an OLE automation server and ORBIX client from an IDL specification.
- ORBIX-ISIS integrates ORBIX with the ISIS fault tolerant infrastructure to offer highly available CORBA facilities.
- ORBIX-ObjectStore integrates ORBIX with a data management system to provide persistence for ORBIX objects.
- *PowerBroker*. PowerBroker is developed by Expersoft Cooperation. Platforms include SunSoft Solaris/SunOS, HP-UX, IBM AIX, WindowsNT, and Digital UNIX for Alpha. PowerBroker is unique in its support for asynchronous message passing. Object services provided by PowerBroker include naming, event management and object life cycle. Similarly to ORBIX, PowerBroker provides for interoperability with OLE by mapping CORBA objects into OLE objects for inclusion into OLE applications.
- *ObjectBroker*. A CORBA implementation from Digital. ObjectBroker is available for UNIX (OSF/1, Ultrix/RISC, SunOS, HP-UX, IBM-AIX), Windows, Open VMS VAX/AXP, and Macintosh.

Digital Equipment Corporation	http://www.digital.com/objectbroker/
Iona	http://www.iona.com/
The Open Group	http://www.activex.org/
SunSoft	http://www.sun.com/solaris/neo/

Table 5: CORBA contact sites

4.6.2 OLE and DCOM

4.6.2.1 Basic Principles

OLE/DCOM is a set of specifications and implementations for component technology and compound documents. OLE (Object Linking and Embedding) was initially intended to be a compound document standard and implementation technology. It is for this purpose which OLE is best known. The scope of OLE has since broadened to embrace more general component technology - beyond the desktop. OLE is built on the foundation of Microsoft's COM (Component Object Model).

COM provides a language-independent binary standard for object implementations. Similarly to CORBA's object model, COM defines how components interact. In principle, components can be written by different vendors at different times, and in different programming languages on different platforms. In practice however, the COM specification has been implemented and packaged with Microsoft operating systems. There has been little adoption of the COM standard by third-party vendors. There are two fundamental differences between COM and CORBA's object model:

1. A COM object cannot inherit from another COM object. Rather, aggregation has to be used to simulate extensibility of a base object.
2. There is no separate IDL. A COM object provides a collection of interfaces - each of which provides a coherent view of the object. Clients do not reference whole objects, but instead reference a particular interface of the object. Each interface is implemented by an array of pointers to functions.

The COM standard does not support distribution, so applications based on the standard are bound to a single machine. With COM it is not possible for example, for an OLE compliant application to maintain a link to data managed by another OLE compliant application where the container and data reside on different machines. DCOM adds distribution support to COM in the form of an RPC protocol, based on DCE's RPC specification. In addition to the RPC mechanism, DCOM provides other ORB-like features such as a repository for locating particular objects, and dynamic method invocation.

When OLE was introduced it provided for limited application interoperability. An OLE container document could contain an OLE object, managed by a different OLE compliant application. When selected, a new window would be spawned for the application which managed the contained object. OLE 2.0, which we subsequently refer to simply as OLE, raises the level of integration by maintaining a single window whose appearance changes to reflect selected data items.

OLE can be decomposed into three sub-technologies: compound documents, automation and custom controls.

Compound Documents

Microsoft has achieved great success with interoperability on the desktop by providing a standard for application interoperability. Recent Microsoft applications conform to the standard, as do many third party applications which run on Windows. An application (for example Microsoft Word) equates to a component in OLE compound document technology.

Compound document technology is a set of object services built on a component technology standard. Compound document technology provides the protocols that let containers communicate with the applications responsible for data items. Protocols have to be sufficiently general in order to allow applications to be independently developed and able to communicate with each other without prior knowledge of each other.

The heterogeneous data which a compound document may contain includes text, graphics, audio, video, and data tables amongst others. Data sources (applications) are as diverse as data types. All data items must however be managed by applications which conform to the compound technology standard.

A compound document manages its data by interacting with applications responsible for the data. Interaction is two way - if a data item is shared, and it is updated by another user, the responsible application may notify the container of the event. The application might request control of the container to make the user aware of the new state. Interaction from the container, directed to a particular application, may be initiated by a user

selecting a data item for which the application is responsible. Applications take control of the container when appropriate.

End users thus work with single documents rather than applications. A container is stored as one structured file of heterogeneous data. It is relatively easy for users to adapt to this more natural paradigm, but for application developers, they are confronted with the additional complexity of requesting and relinquishing control of processing resources, and event management.

Data contained in a compound document may be referenced using one of two techniques:

1. *Linking*. The data item spreadsheet is logically linked to the compound document. Whenever the data item is updated, the change in state of the data is reflected in the compound document.
2. *Embedding*. The data item is physically embedded in the compound document. The data reference is a separate copy of the original data. Unlike linking, there is no connection between the two data - if the original data item is edited, the changes are not reflected in the copy contained in the compound document.

Automation

OLE automation allows applications to export their COM-defined interfaces for programmatic use. These interfaces may be used by user-oriented scripting languages and by programming languages. An OLE server is an application, such as Microsoft Excel which exports a number of useful interfaces. These services can be invoked by an OLE controller - another application such as Word for example, or by some third party OLE component (see *custom controls*). OLE automation thus extends beyond desktop computing to «doing work under the hood» - a computational component may use the services exported by Excel for example, without launching the application.

Programming with OLE automation is much simpler than programming directly with DCOM. However, this is a classic trade-off between complexity and performance. Programming with DCOM results in software which exhibits relatively high performance to OLE automation.

Custom Controls

OLE controls (OCXs) are objects which may be plugged together to compose applications. OLE controls replaced Visual Basic extensions (VBXs) - binary components which could be plugged into Visual Basic applications. VBXs however, were not designed to be componentware and so did not conform to a published standard. OCXs conform to the COM/OLE automation standard, and so offer greater potential for interoperability.

OCX components are now evolving to operate in the Internet environment. ActiveX is the name given to the new components. For example, functionality provided by an third party ActiveX component can be exploited by embedding the component in a Web page. The Web page must of course be accessed by an ActiveX compliant browser.

4.6.2.2 Areas of Use

Although OLE's roots are in compound document technology, OLE has evolved to a more general component technology. For the former, OLE is clearly suited to the client concerns of a client/server application. There is much Windows support for compound documents and client front-ends which conform to the OLE standard. For the latter, it is possible to use DCOM as the component infrastructure for hosting assemblies of server components.

4.6.2.3 Benefits

OLE is well supported, particularly for compound documents. The majority of recent Microsoft desktop products comply to the standard, in addition to many third party products. A healthy third-party componentware market for OLE controls has also emerged.

The strength of the OLE componentware market and the ease of integrating off-the-shelf components with 4GLs makes OLE very attractive for building clients (front-ends) of a client/server application.

OLE is an economic component technology. Run-time licenses are not required because the technology is part of Microsoft's operating systems.

4.6.2.4 Risks

COM is a complex component technology which is rarely used on its own to build components. It is more typical for it be used as the basis for higher level facilities such as OLE for example.

Although simple to use with Visual Basic and other 4GL front-end tools, writing OLE custom controls in C++ is more difficult. Microsoft released the MFC (Microsoft Foundation Classes) software, designed for use with C++ compilers for Windows to help manage the complexity.

In principle, DCOM is a language independent component standard. In practice, it is difficult to implement a DCOM component in a language other than C++. Doing so requires writing code to simulate the vtable structures generated by C++ compilers. It is not clear how legacy code (written in COBOL or PL/1 for example) could be encapsulated in DCOM.

OLE/DCOM is poorly documented.

4.6.2.5 Supporting Tools

OLE/COM is currently supported on WindowsNT and Windows95. OLE/DCOM is shipped as part of the WindowsNT 4.0 operating system. It is intended that DCOM will also be available for Windows95 and Macintosh in the near future. Beyond Microsoft operating systems, DCOM is being ported to Solaris, HP-UX, MVS and Linux amongst others.

Componentware is produced by Microsoft, other Windows tool vendors, and third parties. Internet-based component brokers have started to emerge where it is possible to purchase and download components.

There are several 4GL products aimed at front-end development of client/server systems, for example Gupta's SQLWindows, Borland's Delphi, and Microsoft's Visual FoxPro.

Microsoft	http://www.microsoft.com/
Netscape	http://home.netscape.com/
Centura (SQL Windows)	http://www.gupta.com/
Visual Components	http://www.visualcomp.com/products/
Component Source	http://www.componentsource.co.uk
Componentware	http://www.componentware.com
OLE Broker	http://www.olebroker.com

Table 6: OLE Contact Sites

5 Appendix A: Link with the Renaissance Project

Contents

- 5.1 Covering of Requirements from Problem Focusing
- 5.2 Mapping to the Technology Selection

Summary

This chapter presents the relationships between this document and the work done in the framework of problem focusing phase of the Renaissance project.

5.1 Covering of Requirements from Problem Focusing

The work leading to the production of this document followed a problem focusing phase during which requirements for the evolution of selected candidates applications were identified.

These requirements related to the migration to a distributed client/server environment are covered in this document in the following way:

Req. #	Requirement description	Requirement coverage
CS1	An overview of standard C/S architectures is needed.	
CS1.1	How does a 2-tier, 3-tier, n-tier architecture look in detail?	Section 2.1, <i>Distributed Architecture Models</i> , describes various styles of distributed architectures.
CS1.2	Which kind of architecture is suited to which kind of application?	Section 2.2, <i>System Integration Models</i> , describes several types of architectures based on the type of integration needed for the application.
CS1.3	How can legacy system elements be integrated into a C/S co-operative environment?	Integration scenarios are reviewed in chapter 3, <i>Integration Scenarios</i> and corresponding techniques providing solutions for integration with legacy system components in chapter 4, <i>Migration Techniques and Tools</i> .
CS1.4	Which operating system is suited for which kind of client?	Techniques reviewed in chapter 4, <i>Migration Techniques and Tools</i> are evaluated against their support for operating systems.
CS1.5	Which operating system is suited for which kind of server?	Techniques reviewed in chapter 4, <i>Migration Techniques and Tools</i> are evaluated against their support for operating systems.
CS1.6	How can C/S systems be scaled to different sizes?	Techniques reviewed in chapter 4, <i>Migration Techniques and Tools</i> are evaluated with a specific mention on their support for scalability.
CS1.7	How can security be guaranteed in C/S systems?	Techniques reviewed in chapter 4, <i>Migration Techniques and Tools</i> are evaluated with a specific mention on their support for security.
CS2	An overview of existing and emerging standards is needed.	
CS2.1	What's behind OSF-DCE, Message Passing, Message queuing, Distributed TP monitors, APPC, RPC, OLE/COM, CORBA/ORB, DLL, VBX and OpenDoc?	All these techniques are reviewed on the conceptual and detailed viewpoints all over this document.
CS2.2	How can the standards mentioned above be used?	All these techniques are reviewed on the conceptual and detailed viewpoints all over this document.
CS3	Methods for migrating existing applications are needed.	
CS3.1	Which policy could be used to modularise, distribute and partition an application?	Various integration and distribution scenarios are presented in this document.
CS3.2	How can existing software be encapsulated and integrated?	Various techniques (component based...) are presented to encapsulate existing components in a new, distributed, application.
CS3.3	How can emerging and de-facto standards be used for the migration of existing applications?	Standards are presented on a conceptual viewpoint in section 2.2, <i>System Integration Models</i> , and on a more detailed viewpoint in chapter 4 reviewing integration and migration techniques and tools.
CS3.4	Which policies could be used to generalise existing software to reusable components?	Componentware is addressed more specifically in sections 2.2.4 and 3.8.
CS3.5	What are the most suitable models and target environments for the existing example applications?	Models are reviewed in section 2.2, <i>System Integration Models</i> .

<i>Req. #</i>	<i>Requirement description</i>	<i>Requirement coverage</i>
CS3.6	What are the most suitable implementation environments for the existing example applications?	Techniques and tools are reviewed in chapter 4, <i>Migration Techniques and Tools</i> with a specific mention of the implementation environments supported.
CS4	Methods for data migration are needed.	
CS4.1	How can the data schema be migrated?	Data based integration and corresponding techniques are presented in sections 2.2.1, <i>Data Based Integration</i> , and 4.3.
CS4.2	How can the physical data be migrated?	Data based integration and corresponding techniques are presented in sections 2.2.1, <i>Data Based Integration</i> , and 4.3.
CS4.3	How can DB2/VSAM database on MVS be migrated to a fully relational DBMS?	Data based integration and corresponding techniques are presented in sections 2.2.1, <i>Data Based Integration</i> , and 4.3.
CS4.4	Which tool support is available?	Data based integration and corresponding techniques are presented in sections 2.2.1, <i>Data Based Integration</i> , and 4.3.
CS5	An overview about the existing componentware market is needed.	
CS5.1	Which componentware is available?	Componentware is addressed more specifically in sections 2.2.4, <i>Component Based Integration</i> , and 3.8.
CS5.2	How can they be integrated?	Componentware is addressed more specifically in sections 2.2.4, <i>Component Based Integration</i> , and 3.8.
CS6	How can work-flow technology be used for co-ordinating distributed business and operational processes?	Workflow technology is addressed in sections 2.2.5, <i>Workflow-Based Integration</i> , 3.6, and 3.7.

Table 7: Covering of Requirements from Problem Focusing

5.2 Mapping to the Technology Selection

Several evolution enabling technologies and tools were reviewed in the framework of the Technology Selection activity during the first phase of the Renaissance project. This work led to the identification of the following technologies and tools which have been assessed and are summarised in this document (chapter 4):

<i>Migration Technique</i>	<i>Coverage</i>
Process Support	Workflow technology is addressed in sections 2.2.5, 3.6, and 3.7.
Componentware	Componentware is addressed more specifically in sections 2.2.4 and 3.8.
Middleware	Various forms of middleware are presented in section 2.2, <i>System Integration Models</i> and more specific techniques in chapter 4, <i>Migration Techniques and Tools</i> .
Internet Technology	Internet technology is addressed in sections 2.2.6, 3.3, and 3.5.
Distributed Object Technology	Distributed object technology is addressed in the framework of componentware in sections 2.2.4 and 3.8.

Table 8: Mapping to the Technology Selection

6 Appendix B: References

- Adler, R. M. «Emerging standards for component software». *Computer*, March 68-77 1995.
- Bernstein, P. A. «Middleware: a model for distributed system services». *Communications of the ACM* 39(2), 86-98, 1996.
- Card, D. N. »The RAD Fad: Is timing really everything?» *IEEE Software* 12(5), 19-22, 1995.
- Coulouris and Dollimore. «Distributed Systems - Concepts and Design». 2nd edition, Addison-Wesley, 1994.
- Free On-line Dictionary of Computing, London UK, <http://wombat.doc.ic.ac.uk/cgi-bin/foldoc>
- Foody, M. «Let's talk». *Byte*, April 99-102, 1997.
- Halfhill, T. R. and Salamone, S. «Components everywhere». *Byte* January 97-104, 1996.
- Hettler, M. «New leaders of the client-server migration». *Byte*, June 124-131, 1996.
- Hollingsworth, D., «Workflow Management Coalition - The Workflow Reference Model».
- Linthicum, D. S. «Integration, not perspiration». *Byte*, January 83-96, 1996.
- Montgomery, J. «Distributing Components». *Byte*, April 93-98, 1997.
- Orfali, R. and Harkey, D. «Client/server survival guide». Van Nostrand Reinhold, 1994.
- Orfali, R., Harkey, D. and Edwards, J. «Intergalactic client-server computing». *Byte*, April, 108-122, 1995.
- Pleas, K. «OLE's missing links». *Byte*, April 99-102, 1996.
- Sneed, H. M. and Nyary, E. «Downsizing large application programs». *Journal of Software Maintenance-Research and Practice*, 6(5), 235-247, 1994.
- Udell, J. «Componentware». *Byte*, May 46-56, 1994.

7 Appendix C: Glossary

Client

A computer system or process that requests a service of another computer system or process (a server). For example, a workstation requesting the contents of a file from a file server is a client of the file server.

--

In the client-server model for communications, the client is a process that remotely accesses resources of a compute server, such as compute power and large memory capacity.

Client-Server

A common form of distributed system in which software is split between server tasks and client tasks. A client sends requests to a server, according to some protocol, asking for information or action, and the server responds. There may be either one centralised server or several distributed ones. This model allows clients and servers to be placed independently on nodes in a network, possibly on different hardware and operating systems appropriate to their function, e.g. fast server/cheap client.

Client-Server, two-tier

If most, or all, of the business logic for an application resides on the client (known as Fat Client syndrome) or on the server (known as Fat Server syndrome), the architecture is a classic 2-tier Client/Server architecture.

Client-Server, three-tier

Application partitioning, or three-tier architectures are expected to be the next generation of client-server systems. A three-tier system adds a third component (the application server) in between the current client and server. The application server maintains some data and behaviour which reflects business rules. With a two-tier system if business rules change, then all client workstations have to be upgraded. In contrast a three-tier's application server provides a central location and transparent updating of the rules with respect to the clients.

--

3-tier Client/Server architecture comes about when the business logic of the application can be moved off from either the client and the server and placed on a third tier, or application server. Once the ability exists to move business logic to separate application servers, an infinite array of possible n-tier (or multi-tier) architectures opens up.

Client-Server, n-tier

See the definition for «Client-Server, three-tier».

Common Gateway Interface (CGI)

A standard for running external programs under a World-Wide Web HTTP server. Current version 1.1. External programs are known as *gateways* because they provide an interface between an external source of information and the server.

Common Object Request Broker Architecture (CORBA)

An Object Management Group specification which provides the standard interface definition between OMG-compliant objects.

Component Software (Component Object Model, COM)

Compound documents and component software define object-based models that facilitate interactions between independent programs. These approaches aim to simplify the design and implementation of applications, and simplify human-computer interaction. Component software addresses the general problem of designing systems from application elements that were constructed independently by different vendors using different languages, tools, and computing platforms. The goal is to have end-users and developers enjoying the same level of plug-and-play application interoperability that are available to hardware manufacturers. Compound documents are one example of component interoperability.

--

COM supports a client-server model where clients are service users, OLE objects are data, and OLE servers are the applications which implement OLE objects.

Compound Document (Compound Object Model, COM)

Compound documents and component software define object-based models that facilitate interactions between independent programs. These approaches aim to simplify the design and implementation of applications, and simplify human-computer interaction. A compound document is a container for sharing heterogeneous data, which includes mechanisms which manage containment, association with an application, presentation of data/applications, user interaction with data/applications, provision of interfaces for data exchange, and more notably linking and embedding. Data can be incorporated into a document by a pointer (link) to the data contained elsewhere in the document, or in another document. Linking reduces storage requirements, and facilitates automatic transparent updates. Embedding is where the data is physically located within a compound document.

--

Compound documents are containers and composed of (possibly nested) parts, organised by end-users. A part contains one or more types, e.g. sound, text, image. End users access and manipulate parts via component handlers.

Encapsulation

The result of hiding a representation and implementation in an object. The representation is not visible and cannot be accessed directly from outside the object. Operations are the only way to access and modify an object's state.

The ability to provide users with a well-defined interface to a set of functions in a way which hides their internal workings. In object-oriented programming, the technique of keeping together data structures and the methods (procedures) which act on them.

Heterogeneous Network

A network composed of systems of more than one architecture. Contrast with homogeneous network.

Hypertext Markup Language (HTML)

A Hypertext document format used on the World-Wide Web. Built on top of SGML. "Tags" are embedded in the text. A tag consists of a "<", a "directive", zero or more parameters and a ">". Matched pairs of directives, like "<title>" and "</title>" are used to delimit text which is to appear in a special place or style.

Hypertext Transport Protocol (HTTP)

The client-server TCP/IP protocol used on the World-Wide Web for the exchange of HTML documents. It conventionally uses port 80. Version 1.0 is the current standard.

Interface

A boundary across which two systems communicate. An interface might be a hardware connector used to link to other devices, or it might be a convention used to allow communication between two software systems. Often there is some intermediate component between the two systems which connects their interfaces together.

--

The point at which independent systems or diverse groups interact. The devices, rules, or conventions by which one component of a system communicates with another. Also, the point of communication between a person and a computer.

The part of a program that defines constants, variables, and data structures, rather than procedures.

The equipment that accepts electrical signals from one part of a computer system and renders them into a form that can be used by another part.

Hardware or software that links the computer to a device.

To convert signals from one form to another and pass them between two pieces of equipment.

Internet

Generally speaking, an *internet* is any set of networks interconnected with routers. The Internet is the biggest example of an internet. The Internet is the largest internet in the world. It is a three level hierarchy composed of backbone networks (e.g. ARPAnet, NSFNet, MILNET), mid-level networks, and stub networks. These include commercial (.com or .co), university (.ac or .edu) and other research networks (.org, .net) and military (.mil)

networks and span many different physical networks around the world with various protocols including the Internet Protocol.

Intranet

A network which provides similar services within an organisation to those provided by the Internet outside it but which is not necessarily connected to the Internet. The common example is a company which sets up one or more World-Wide Web servers on an internal network for distribution of information within the company.

Language, 4th Generation (4GL)

An application specific language. The term was invented to refer to non-procedural high level languages built around database systems. The first three generations were developed fairly quickly, but it was still frustrating, slow, and error prone to program computers, leading to the first "programming crisis", in which the amount of work that might be assigned to programmers greatly exceeded the amount of programmer time available to do it. Meanwhile, a lot of experience was gathered in certain areas, and it became clear that certain applications could be generalised by adding limited programming languages to them. Thus were born report-generator languages, which were fed a description of the data format and the report to generate and turned that into a COBOL (or other language) program which actually contained the commands to read and process the data and place the results on the page. Some other successful 4th-generation languages are: database query languages, e.g. SQL; Focus, Metafont, PostScript, RPG-II, S, IDL-PV/WAVE, Gauss, Mathematica and data-stream languages such as AVS, APE, Iris Explorer.

Language, Interface Definition (IDL)

To accomplish interoperability across languages and tools, an object model specifies standards for defining application interfaces in terms of a language independent - an interface definition language. Interface definitions are typically stored in a repository which clients can query at run-time.

Language, Markup

Languages for annotation of source code to simply improve the source code's appearance with the means of bold-faced key words, slanted comments, etc. See also reformatting.

--

In computerised document preparation, a method of adding information to the text indicating the logical components of a document, or instructions for layout of the text on the page or other information which can be interpreted by some automatic system.

Legacy System

A typical computer legacy system may be 10-25 years old, have been developed using archaic methods, have experienced several personnel changes, one for which current maintenance is very expensive, and one for which integration with current or modern technology or software systems is difficult or impossible. Legacy systems require reengineering to put them in a form where they may better suit modern requirements and may evolve more efficiently.

--

A computer system or application program which continues to be used because of the prohibitive cost of replacing or redesigning it and despite its poor competitiveness and compatibility with modern equivalents. The implication is that the system is large, monolithic and difficult to modify. If the legacy software only runs on antiquated hardware the cost of maintaining this may eventually outweigh the cost of replacing both the software and hardware unless some form of emulation or backward compatibility allows the software to run on new hardware.

Middleware

Software that mediates between an application program and a network. It manages the interaction between disparate applications across the heterogeneous computing platforms. The Object Request Broker (ORB), software that manages communication between objects, is an example of a middleware program.

--

A middleware service is a general purpose service that sits between platforms and applications. It is defined by the APIs and protocols it supports. Middleware is generally not application-specific, not platform specific, distributed, and supports standard interfaces and protocols.

Object

A run-time entity that packages both data and the procedures that operate on that data.

--

In object-oriented programming, a unique instance of a data structure defined according to the template provided by its class. Each object has its own values for the variables belonging to its class and can respond to the messages (methods) defined by its class.

Object Linking and Embedding (OLE)

A distributed object system and protocol from Microsoft. OLE allows an editor to «farm out» part of a document to another editor and then re-import it. For example, a desk-top publishing system might send some text to a word processor or a picture to a bitmap editor using OLE.

Object Management Group (OMG)

A consortium aimed at setting standards in object-oriented programming. The Common Object Request Broker Architecture (CORBA) specifies what it takes to be OMG-compliant.

Object Request Broker (ORB)

ORBs are fundamental to CORBA. In a distributed environment they provide a common platform for client objects to request data and services from server objects, and for server objects to pass their responses back to clients. ORBs hide interoperability details from objects (i.e., programming language and operating system used, local or remote, etc.).

OLE Custom Controls (OCX)

An Object Linking and Embedding (OLE) custom control allowing infinite extension of the Microsoft Access control set. OCX is similar in purpose to VBX used in Visual Basic.

Open Software Foundation (OSF)

A foundation created by nine computer vendors, (Apollo, DEC, Hewlett-Packard, IBM, Bull, Nixdorf, Philips, Siemens and Hitachi) to promote open computing. It is planned that common operating systems and interfaces, based on developments of UNIX and the X Window System will be forthcoming for a wide range of different hardware architectures. OSF announced the release of the industry's first open operating system - OSF/1 on 23 October 1990.

Plug-and-Play

Hardware or software that, after being installed (plugged in), can immediately be used (played with), as opposed to hardware or software which first requires configuration.

Protocol

A set of formal rules describing how to transmit data, especially across a network. Low level protocols define the electrical and physical standards to be observed, bit- and byte-ordering and the transmission and error detection and correction of the bit stream. High level protocols deal with the data formatting, including the syntax of messages, the terminal to computer dialogue, character sets, sequencing of messages etc.

Rapid Application Development (RAD)

A loose term for any software life-cycle designed to give faster development and better results and to take maximum advantage of recent advances in development software. RAD is associated with a wide range of approaches to software development: from hacking away in a GUI builder with little in the way of analysis and design to complete methodologies expanding on an information engineering framework. Some of the current RAD techniques are: CASE tools, iterative life-cycles, prototyping, workshops, SWAT teams, time-box development, and reuse of applications, templates and code.]

Remote Procedure Call (RPC)

A protocol which allows a program running on one host to cause code to be executed on another host without the programmer needing to explicitly code for this. RPC is an easy and popular paradigm for implementing the client-server model of distributed computing. An RPC is implemented by sending request message to a remote system (the server) to execute a designated procedure, using arguments supplied, and a result message returned to the caller (the client). There are many variations and subtleties in various implementations, resulting in a variety of different (incompatible) RPC protocols.

Re-architecturing

The transformation of a system by re-engineering it on a different technological architecture, specifically from a centralised to a distributed architecture.

Re-design with reuse

The transformation of a system by redeveloping it from scratch utilising some aspects of the old system. Re-engineering software for reuse in a re-design of a system provides the chance to keep the knowledge embodied in the legacy software and to save considerable work.

Re-structuring

Re-structuring is defined as the transformation of the internal structure of a system without changing any external interfaces. It can be thought of as a black box change transparent to the end user.

--

Transformation from one representation to another at the same relative abstraction level, while preserving the subject's system external behaviour.

Re-vamping

Re-vamping is the transformation of a system by modifying or replacing its user interfaces. The internal workings of the system remain intact but appear to have changed to the end user. Can be thought of as the inverse of restructuring.

Server

A program which provides some service to other (client) programs. The connection between client and server is normally by means of message passing, often over a network, and uses some protocol to encode the client's requests and the server's responses. The server may run continuously (as a daemon), waiting for requests to arrive or it may be invoked by some higher level daemon which controls a number of specific servers.

System Object Model (SOM)

SOM is IBM's CORBA-compliant object request broker for a single address space architecture. A similar distributed system object model framework exists to allow objects to communicate across address spaces and networks.

--

An implementation of CORBA by IBM.

Transaction Processing (TP)

The exchange of transactions in a client-server system to achieve the same ends as would be performed by the equivalent single complex application.

Transaction Processing Monitor (TPM)

For mission-critical applications it is vital to manage the programs which operate on the data. TP monitors achieve this by breaking complex applications down into transactions. TPMs were invented for applications which serve thousands of clients. A TP monitor can manage transaction resources on a single server or across multiple servers.]

User Interface, Graphical (GUI)

The use of pictures rather than just words to represent the input and output of a program. A program with a GUI runs under some windowing system (e.g. The X Window System, Microsoft Windows, Acorn RISC OS, NEXTSTEP). The program displays certain icons, buttons, dialogue boxes etc. in its windows on the screen and the user controls it mainly by moving a pointer on the screen (typically controlled by a mouse) and selecting certain objects by pressing buttons on the mouse while the pointer is pointing at them.

--

The graphical user interface, or GUI, provides the user with a method of interacting with the computer and its special applications, usually via a mouse or other selection device. The GUI usually includes such things as windows, an intuitive method of manipulating directories and files, and icons.

Visual Basic

An event-driven visual programming system for Microsoft Windows, in which fragments of BASIC code are invoked when the user performs certain operations on graphical objects on-screen. Widely used for in-house applications development by users and for prototyping.

Workflow

A workflow is composed of multiple tasks / steps / activities, of which there are two types:

Simple, representing indivisible activities, and

Compound, representing those which can be decomposed into sub-activities. An entire workflow can be regarded as a large compound task.

--

The set of relationships between all the activities in a project, from start to finish. Activities are related by different types of trigger relation. Activities may be triggered by external events or by other activities. Also the movement of documents around an organisation for purposes including sign-off, evaluation, performing activities in a process and co-writing.

Workflow Management (WFM)

Workflow management is a technology that supports the reengineering and automation of business and information processes. It involves:

Defining workflows, i.e., those aspects of process that are relevant to control and co-ordinate the execution of its tasks, and

Providing for fast (re)design and (re)implementation of the processes as business/information needs change.

Workflow Management Coalition (WFMC)

A standards body formed in 1993 by a group of companies, intended to address the lack of standards in WFMSs.

Workflow Management System (WFMS)

A workflow management system provides procedural automation of a business process by management of the sequence of work activities and the invocation of appropriate human/IT resources associated with the various activity steps. Workflow products are typically client-server software products in which the work is performed within defined time-scales.