View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by The University of Nebraska, Omaha

University of Nebraska at Omaha

DigitalCommons@UNO

Student Work

6-1-2001

Software Architecture for Scalable Applications.

Sujit Chaubal

Follow this and additional works at: https://digitalcommons.unomaha.edu/studentwork

Recommended Citation

Chaubal, Sujit, "Software Architecture for Scalable Applications." (2001). *Student Work*. 3592.
https://digitalcommons.unomaha.edu/studentwork/3592

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.

Software Architecture for Scalable Applications

A Thesis

Presented to the

Department of Computer Science

And the

Faculty of Graduate College

University of Nebraska

In Partial Fulfillment

Of the Requirements of the degree

Master of Science

University of Nebraska at Omaha

By

Sujit Chaubal

June 2001

UMI Number: EP74791

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

UMI EP74791

Published by ProQuest LLC (2015). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.
All rights reserved. This work is protected against
unauthorized copying under Title 17, United States Code

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

THESIS ACCEPTANCE

Software Architecture for Scalable Applications
Sujit Chaubal

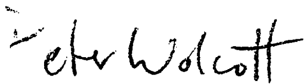
This thesis has been accepted for the Department of Computer Science and the Faculty of Graduate Studies in partial fulfillment of the requirements for completion of the degree Master of Science at the University of Nebraska at Omaha.

6/22/01

Hossein Saedian, PhD, Professor *[1] Date
Department of Computer Science

6/22/01

Hassan Farhat, PhD, Associate Professor Date
Department of Computer Science



6/22/01

Peter Wolcott, PhD, Assistant Professor Date
Department of Information Systems and Quantitative Analysis

[1] At the time of thesis defense Professor Saiedian was with the University of Kansas

Abstract

Software Architecture for Scalable Applications

Sujit Chaubal (M.S.)

University of Nebraska, 2001

Advisor: Prof. Hossein Saiedian

Software has traditionally been built based on a mix of the common architectures. Although several architectural styles have been documented, software developers repetitively solve the problems of scalability and extensibility and deal with the issues of incremental development and interoperability. Component Object Model (COM) is an advanced technology for object-based software development that facilitates interoperability and promotes extensibility. The Extensible Markup Language (XML) is the universal format for structured documents and data on the World-Wide-Web (WWW) It describes a class of data objects called XML documents, and partially describes the behavior of the computer programs which process them. This thesis describes how the COM and the XML can be integrated to implement and extend common software architecture styles to address the problems of incremental development and scalability, and shows how the various architecture styles can be modified and how the implementation of certain COM interfaces make the solution scalable and extensible. Issues related to scalability of Web applications have been discussed and architectural solutions used to scale the software have been discussed. Guidelines to building scalable and extensible applications are given and samples to adapt common architectures using COM and XML have been introduced.

Table of Contents

1 **Introduction and Problem Definition** 1

1.1 Traditional Issues in Software Development 1

1.1.1 Incremental, Extensible Development 1

1.1.2 Scalability 3

1.1.3 Interoperability - Languages / Operating Systems 4

1.1.4 Re-use 4

1.2 Problem Definition 5

1.3 Emerging Technologies 5

1.4 Organization of the Thesis 5

2 **A Survey of Architecture Styles** 7

2.1 Definition of Software Architecture 7

2.2 Classification 9

2.3 Pipes and Filter 10

2.4 Object Oriented Architecture 12

2.5 Event-based, Implicit Invocation 15

2.6 Layered Systems 16

2.7 Repositories 18

2.8 Interpreters 20

2.9 Process Control 21

2.10 Main Program / Subroutine Organizations 22

2.11 Distributed Processes 23

2.11.1 Client Server Systems 24

2.11.2 Browser Based / Internet Applications / Web Based Architecture 25

2.12 Heterogeneous Architectures 25

2.12.1 Component Based Architecture 26

2.13 Self-Evolving Software Systems 26

3 **Extending Standard Architectures Using COM** 28

3.1 Introduction to COM ... 28
3.1.1 Interoperability ... 29
3.1.2 Distributed Component Object Model (DCOM) ... 29
3.1.3 Marshalling. ... 30
3.2 The Key Word in Context Problem ... 30
3.3 Modifying the Pipe and Filter Model using COM. ... 31
3.3.1 Solving the KWIC Problem ... 32
3.4 Modifying the Layered Model Using COM. ... 34
3.5 Modifying the Object Model Using COM ... 35
3.6 Modifying the Main Program and Subroutine Architecture Using COM ... 36
3.7 Modifying the Event Based Architecture Using COM ... 36
3.8 Modifying Distributed Systems Using COM ... 37

4 Communication Between Components ... 38

4.1 COM and Data Formats ... 38
4.2 What is XML? ... 39
4.3 XML and COM ... 40
4.4 Extending Data Formats Using XML ... 40
4.4.1 XML and Character Encoding ... 42
4.4.2 XML and Binary Data ... 43
4.5 Sample for Interpreting Data ... 43
4.6 Advantages of Using XML ... 44
4.7 Disadvantages of Using XML ... 44
4.8 Revisiting the KWIC Problem ... 45

5 A Software Architecture for Incremental Applications ... 46

5.1 Problem Definition ... 46
5.2 Approach to Solution ... 46
5.3 Snap-In Architecture ... 47
5.3.1 Example: Snap-Ins in a File Converter Application ... 48
5.4 Object Discovery ... 49

5.5 Scalability and Internet Web Applications 50

5.6 Stateless Modules/Applications 50

5.7 Load Balancing by Task Distribution 50

5.7.1 Push Model for Load Distribution 51

5.7.2 Pull Model for Load Distribution 52

5.8 Distributing Databases 53

5.9 Building the Application 54

5.10 Advantages 55

6 Case Study of Building a Scalable Application 57

6.1 Meta Searcher 57

6.1.1 Problem Definition 57

6.1.2 Solution Phase I 57

6.1.3 Future Phases: Comparison Shopping for Books 62

7 Conclusion 63

7.1 Future Work 64

Bibliography 65

List of Figures

Figure 2-1: A simple representation of the pipes-and-filter architecture........ 10
Figure 2-2: Simple examples of pipes and filters........ 11
Figure 2-3: Example of a layered system applied to the ISO communications model........ 17
Figure 2-4: Example of a layered system applied to a diagramming application........ 17
Figure 2-5: Repositories........ 18
Figure 2-6: Interpreter........ 21
Figure 2-7: Feedback control system........ 22
Figure 2-8: Main program – subroutine architecture........ 23
Figure 2-9: Client-server architecture........ 24
Figure 2-10: Web architecture........ 25
Figure 2-11:Architecture for evolving software........ 27
Figure 3-1: Typical COM object........ 28
Figure 3-2: Pipe and filter solution for KWIC........ 32
Figure 3-3: Filter with interface........ 34
Figure 3-4: Sample layered system........ 35
Figure 4-1: Component requiring XML processing........ 41
Figure 5-1: SnapIn modules........ 48
Figure 5-2: Push model for task distribution........ 52
Figure 5-3: Pull model for load distribution........ 52
Figure 5-4: Horizontal splitting of a database........ 53
Figure 5-5: Vertical splitting of a database........ 54
Figure 6-1: Architecture for meta search........ 58
Figure 6-2: Data fetcher........ 58
Figure 6-3: Data parser........ 59
Figure 6-4: Integrator........ 59
Figure 6-5: Creating HTML output........ 60

1 Introduction and Problem Definition

Software systems mature with time, and software development is becoming mature as standards are being formed and as processes are being developed. The current trend stresses re-use, as developers do not wish to solve the same problems repetitively. Although millions of lines of code have been written, the architecture of the programs remains hidden in those lines. The size and complexity of the software systems being written and used are growing at a rapid pace. In large systems, importance is not only given to the algorithms used, but also to the organization of the components of the system, global control structures, communication protocols used, synchronization methods, data access methods, composition of design elements, physical distribution, scalability, performance and user interfaces.

There is no standard procedure that can guide and guarantee the software engineer that the system he or she builds, from the requirements to the implementation, is accurate.

1.1 Traditional Issues in Software Development

1.1.1 Incremental, Extensible Development

Extensible or incremental software can also be classified as a kind of reuse. It refers to the extending or modifying software without copying it or changing the existing code. A common requirement seen in software development is modular development or incremental development. Software developers need to add modules as they develop new ones. A simple example is that of a drawing application. A developer can provide his

own format for storing and retrieving images. As the clients grow, he needs to support other file formats and over a period of time support for new formats is required. Every time new code for a new file format is written, the developer has to compile his programs and release the software. This is a very tedious process and releasing new versions of software can lead to problems. Or, take the example of porting a legacy application where huge software needs to be ported, piece-by-piece, but happens to be a part of a single application. At the same time, if a problem is reported in one of the modules, the entire application needs to be rebuilt and of course needs to be redistributed. If the software becomes overly large, additional problems arise, as the memory taken up by this program will be cumbersome. In the early days of DOS, a technique of overlays was used, but this had several drawbacks and sharing data among the different overlays was not easy. Also, each developer needed to be aware of how the entire system worked.

Later in the world of Microsoft Windows, developers started using Dynamic Link Libraries (DLLs). DLL's are loaded dynamically and are loaded into memory only when the application needs a function from them. DLL's are present in many complex applications in Microsoft Windows applications. They do not provide a clean solution, however. Different versions of DLL's often cause problems. Although the DLL is loaded dynamically, where it gets loaded from cause problems as it depends on the operating system path settings and what applications are already running. Developers found solutions to all these problems, but they were never very clean. DLL's, when linked with applications directly, get loaded when the application starts. As it is linked with the application, any major change in the DLL (like changing the order of functions or the

function prototype itself) would require the recompiling and rebuilding of both the dynamic link library as well as the application. Not only do the current applications need to be rebuilt, but all other applications that were linked to the DLL do as well. DLL's are very effective for code sharing, but very often applications crash if another developer changes a DLL that is shared. Debugging such a problem is not always easy. The Microsoft Windows operating system can detect some mismatches and provide errors occasionally, but not consistently. And in case there is an error in the programming logic, it is up to the developer to locate the mistake. With the release of new DLL's, a new problem is created due to the numerous versions. A problem faced very often is that a piece of software that works on one machine does not work on another machine and the error source is related to an older version of a DLL.

1.1.2 Scalability

The world of client-server systems involves writing software for the client side and the server side. Client programs run on the client machine and server software runs on the server machine. Even if the number of clients grows, the number of server programs used remains the same, which introduces the problem of scalability. If the estimate for the number of clients is incorrect it can lead to a degraded performance. Distributed systems face the problem of one of the components being overloaded. Internet applications often perceive this problem, as the number of clients can increase to a large number. Using more powerful hardware, in terms of processor speed, network connection or hard drives, may not always solve the problem. Distributing the software across multiple machines can solve this problem, but involves managing the multiple machines.

1.1.3 Interoperability - Languages / Operating Systems

The computer industry is filled with choices in terms of platform (operating systems) and languages for development. Also available are a large number of tools that facilitate the software development process. The dominant operating systems in the market are Linux/Unix variants and Microsoft Windows (95, 98, NT, 2000). Selecting a language for development (and/or tools) depends on the application. Choices vary from using the increasingly popular Java language, C++ and Visual Basic to using scripting languages like Perl, TCL/TK, COBOL and others. Developing software on a different platform like AS/400 can totally change the options. The selection depends on company policy, platform, and the application itself.

1.1.4 Re-use

Modular development facilitates re-use. One of the major advantages of object-oriented software is its re-use. Although this is a documented re-use it is not very easy in practice. Using third-party software and tools and re-using them is not necessarily re-use. Object-oriented software is definitely more amenable to reuse, than is procedural software, but is not the ultimate solution. This is because the developer of object-based software does not necessarily keep re-use in mind when designing and defining his software. Also non object-based languages find it difficult to re-use this object-based code. Object re-use generally takes the form of re-using code that can be written in C++, Java or Smalltalk. This makes the user aware of the language of implementation and the need to know this language.

1.2 Problem Definition

There is no unique solution of solving the problems of incremental development, scalability, interoperable solutions and re-use. Although software re-use can be improved by following good software development practices, it largely depends on the people involved. Scalability is a design issue and needs to be evaluated too. Although scalability is not a requirement, good performance is. Performance requirements have to be defined in the requirements analysis but tend to be difficult to document.

The objective of the thesis is to study the common software architectures and find solutions to the problems of incremental development, scalability and interoperability.

1.3 Emerging Technologies

Microsoft's .Net technology and SOAP (Simple Object Access Protocol) are emerging technologies that attempt to solve the problems of scalability and ease the development of large software. Products for developing SOAP and .Net applications were not available at the time of the thesis defense to compile comparative information.

Java 2 Enterprise Edition (J2EE) is an advanced Java based technology meant for enterprise software development and deployment. However, the solutions are restricted to software developed using the Java language and does not accommodate legacy software or software developed in any other language.

1.4 Organization of the Thesis

I will use this thesis to present my ideas and study of architectures addressing the programming issues of communication between modules, scalability and load

distribution. The solutions I present are based on the Extensible Markup Language (XML) and Microsoft's Component Object Model (COM). Although COM is a specification and an implementation, it is well supported only in Microsoft operating systems. In other operating systems, it is possible to replace COM and use an alternative. I will not explain COM in detail, as entire books have been written about COM, but I will present a brief introduction.

This thesis is organized as follows:

In Chapter 1, an introduction to the problems encountered during the development of large software is described.

In Chapter 2, I present a survey of the common architecture styles. The styles are later modified to be scalable and extensible.

Chapter 3 provides an introduction to COM followed by a description of how the common architectures can be integrated with it.

Chapter 4 discusses issues related to communication between the components and the use of XML as a data format. It then describes how COM and XML can be used together.

In Chapter 5, I present a proposal for a software architecture addressing incremental applications using the techniques discussed in Chapters 3 and 4. Solutions to make applications scalable are also discussed.

A case study is presented in Chapter 6 to illustrate the uses and the validity of the model I have proposed in Chapter 5.

Chapter 7 summarizes the contributions of the thesis and provides open issues for further research consideration.

2 A Survey of Architecture Styles

2.1 Definition of Software Architecture

There is no precise definition for software architecture. Several textbooks attempt to define it, and though each is different, they all carry the same meaning. Several definitions of software architecture can be found. Shaw and Garlan (1996) describe software architecture as involving the description of the elements from which systems are built, the interactions among those elements, the patterns that guide their composition and the constraints on these patterns. System architecture describes its decomposition into meaningful components[1] and the interaction among these components.

Bass, Clements and Kazman (1998) define software architecture of a program or a computing system as the structure or structures of the system that comprise software components, the externally visible properties of those components, and the relationships among them.

Barry Boehm and his students (as cited in [online] Carnegie Mellon Software Engineering Institute, 'How do you define Software Architecture') at the USC Center for Software Engineering write that software system architecture comprises

- a collection of software and system components, connections and constraints.
- a collection of system stakeholders needs statements.

---

[1] Reference to the term 'Components' on its own, refers to a single module and does not mean 'Component' as part of Component Object Model (COM)

- a rationale that demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders needs statements.

There are several more definitions. In simple terms, software architecture is the high-level software system design.

To define the architecture in a diagrammatic form, we use components and connectors. Architecture can be described by drawing boxes and lines that depict the gross organization of the system. The architecture of a software system defines that system in terms of computational components and interactions among those components. Components are such things as clients, servers, databases, filters and layers in a hierarchical system. The architecture also shows the correspondence between the systems requirements and the elements of the constructed system, thereby providing some rationale for the design decision.

An architectural style defines a class or pattern of structural organizations. *Components* and *connectors* can represent most architectural styles. The components describe the major entities in the software whereas the connectors describe how these components interact with each other. Large software systems are always implemented in terms of components. They are simply too big to be designed and developed in any other way. The description of the architecture has been formalized and has resulted in Architecture Definition Languages (ADL). Examples of ADL are 'Rapide', Unicon, Dicam, Leap, Wright, and Unas, etc.

I like to define software architecture as the abstraction of the various components of the system, how these components interact and what they communicate.

Software architecture does not come into the picture when solving small problems. For example, it will be difficult to have software architecture for a program that converts Celsius to Fahrenheit.

2.2 Classification

The architecture of a software system that processes continuously varying data may try to emphasize the fact that the data is continuously varying, whereas software in which data is stored, and that play the most important role may emphasize on the data structures and how they are related. Due to the wide variety of software being developed, there cannot be a single instance that defines all possible software architectures.

The common architectures that have been accepted and discussed in the past include:

- Pipes and Filter
- Object Oriented
- Event Based
- Layered Systems
- Repositories (Blackboard)
- Interpreters
- Process Control
- Client Server (Distributed Processes)

- Main Program/Subroutine Organizations

There are several domain specific architecture styles in addition to the standard documented styles.

2.3 Pipes and Filter

The pipes and filter style consists of components that accept data, process it and produce some output. More than one such component can be connected to form a system. A filter represents the subsystem that performs the processing. Data being transferred or communicated from one filter to another is represented as being part of the pipe. This style is shown in Figure 2-1.

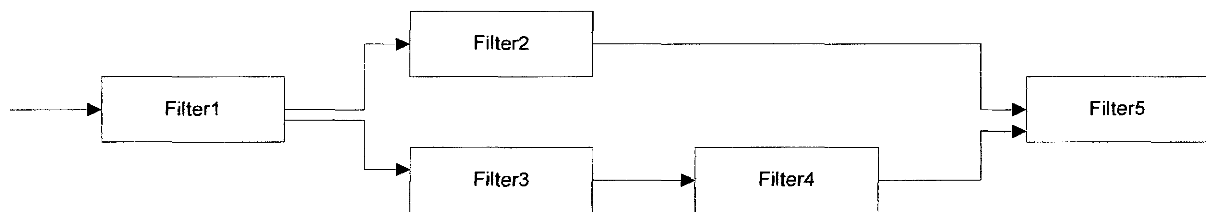


Figure 2-1: A simple representation of the pipes-and-filter architecture

The pipes-and-filter architecture concentrates on individual tasks. A filter accepts input, performs the task, and produces the required output. This kind of architecture is ideal in batch systems where there is no user interaction. One common example is that of a sort filter, which is useful when operations have to be done on files.

An example of this style is implemented in the Unix operating system. Output of any command can be redirected to another command. Each command is implemented to read data or parameters from the standard input. The operating system shell allows the

output from one command to be redirected to another. This forms a pipeline system and shows a pipes-and-filter implementation.

Parts of the traditional compiler architecture can also be viewed as a pipeline system. This is not strictly an incremental pipeline and there is global data associated with the processes. The various stages in the compiler such as lexical analysis, syntax checking or parsing, semantic analysis, code generation and code optimization are all connected by the symbol table and global memory.

Implementations of batch systems can adapt to this style of architecture. It can start off with a file being created or a file being read and an output file being created. A filter or program processes data and creates a file. The next program reads this file and processes it and creates a new one or overwrites it.

For example, a user comes to you (the programmer) and informs you that his programs are frozen. He/she wants you to write a program that will kill all his programs. A developer can write a C language program to perform the required task. We can also use the existing commands and utilities available on the Unix operating system to solve the problem. The solution is shown in Figure 2-2.

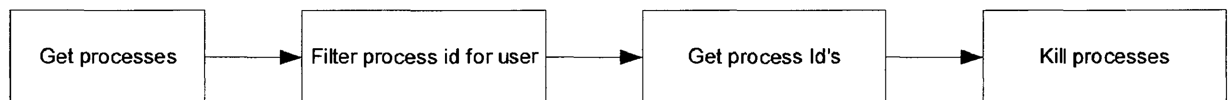


Figure 2-2: Simple examples of pipes and filters.

Advantages

1. Pipes-and-filters architecture allows the designer to understand the overall input/output behavior of the system as a simple composition of the behaviors of the individual filters.
2. It strongly supports re-use. As input and output of each filter is clearly defined, it is easy to re-use the filters that exist elsewhere. Existing filters can be joined to give quick solutions provided they agree on the input and output formats.
3. Maintenance and enhancement of these systems is easy. Individual filters can be enhanced without having any effect on others.
4. Each filter can be executed in parallel with other filters, as they ideally do not share any data and concentrate on their own task.

Disadvantages

It is difficult to apply this architecture on systems in which user interaction is required or where the pipes involved are not text files. If we do not use files as pipes, we start designing our own format of the data that is passed from one module to another. We then lose the flexibility of having the generic format and of being able to insert an available filter. This style is more suitable for batch-oriented systems and difficult to adapt to all systems.

2.4 Object Oriented Architecture

Object-oriented analysis and design is the most popular methodology used for new development. The focal point of this technique lies in identifying objects, how they are related and what operation each object performs.

A simple explanation of object-oriented architecture is that it consists of a set of classes. These classes are well arranged and consist of multiple hierarchies and a set of specification informing how these classes interact and provide the various required functions. Different methodologies present different ways in which the architecture should be represented and refining the architecture leads to the design. All techniques, however, revolve around the static and dynamic behavior of the objects in the system.

What makes an architecture object oriented? There are several texts about object oriented methodologies and programming. We can say that the high-level design in an object-oriented methodology describes the architecture of the system. This popularly boils down to being the basic "Object Diagram". Each methodology requires or suggests drawing or designing several views of the entire system. Rumbaugh, Blaha, Premerlani, Eddy, Rumbaugh, Lorenson (1991) describe the dynamic and functional model of the system that show the static and runtime view of a system. Since object diagrams are part of the analysis phase, there is a phase when we move from analysis to design. We view the high-level object diagram as something that describes the architecture of the system.

So is the architecture an object oriented architecture? The question can be answered by looking at the architecture. Also, a similar question can be asked for a program written in C++. Is the program object oriented? Using the C++ language does not automatically make the program object oriented. Similarly, having one large object that does the entire task required by the program does not make it object-oriented. If data components are represented as objects and their associated primitive operations are encapsulated in objects, the solution does make an object oriented architecture. The

objects are responsible for preserving the integrity of its representation and the representation of the object is hidden from other objects.

Advantages

Object-oriented architecture is the most popular architecture form today. Some of the advantages of using object-oriented architectures are the use of abstraction, encapsulation, extendibility and reusability. Re-usability has been emphasized by several examples of patterns using the object-oriented techniques.

Disadvantages

1. One of the disadvantages of object orientation is that the user has to know the identity of the object to use one of its methods. It has to know the actual method name and if the object is already created in memory, it needs to know the object identity.
2. If the identity of an object changes, this has to be notified to all the other objects or modules using this object.
3. It is observed that programmers and architects use too much object orientation and overlook the main features of the system.
4. In certain problems the external interfaces provided by the module can be extremely important and may be missed. Although an interface can be as simple as a text file (as in the pipes and filters architecture), a C++ programmer may want to use objects as interfaces leading to an incompatible interfaces.

Shaw and Garlan (1996) describe a cruise control system using object oriented architecture and other styles as well. It can be seen that in the case of a cruise control system, the process control architecture is better suited for the example. Contrary to the

popular belief that object-oriented architecture is the best for all types of problems, it can be seen that some problems are better designed in an alternative architecture.

2.5 Event-based, Implicit Invocation

In object-oriented systems, as well as other systems, objects or components of the system interact with each other by typically by invoking a procedure, method or routine from the other system. Another technique for interacting with other components in the system is to use techniques called implicit invocations, reactive integration and selective integration. These techniques have been used in systems based on actors, constraint satisfaction, daemons and packet switched networks. Shaw and Garlan (1996) describe the idea behind implicit invocation. Instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with it. When the event is announced, the system itself invokes all of the procedures that have been registered for the event. Thus, an event announcement 'implicitly' causes an invocation of procedures in other modules.

When a program announces an event, it is not known at that time what programs are to be affected. Thus, the programs cannot assume the order in which the modules or affected programs will be invoked.

A good example of implicit invocation can be seen in the functionality offered by a debugger. A user can place a breakpoint in the source code; when execution reaches that point, the user is given control and is allowed to perform any operation he or she wishes.

Graphical user interfaces also fall into the category of event-based systems. In Microsoft Windows, the programmer registers a function with the operating system that is invoked whenever an event occurs. In practice, several events occur and the programmer handles those that he or she is interested in. The program itself can generate events, leading to other programs or functions to be invoked. At the same time, it can call some functions explicitly. This makes it an explicit invoked system.

Advantages.

1. Re-use: because events can be used to invoke certain parts of the program, sub-events can be used and hence promote re-use.
2. System evolution: functions and more event handling can be added without affecting how existing events are processed.

Disadvantages.

1. Applications lose control: as the main program has to wait for other events, no central application exists to control the program flow. Hence, the application loses control.
2. Passing of data using global resources increases program complexity.
3. Reasoning about correctness: verification of the system can become complicated, as the effects of various event sequences have to be tested individually.

2.6 Layered Systems

A layered system is organized hierarchically, each layer providing services to the layer above it as a server and serving as a client to the layer below. In some layered systems, inner or lower layers are hidden from all except the adjacent outer layer with the exception of certain functions carefully selected for export. Each layer publishes a set of

functions that layers above it can use. A widely known example of this kind of architectural style is the layered communication protocols, as shown in Figure 2-3.

Application
Presentation
Session
Transport
Network
Data Link Layer
Physical

Figure 2-3: Example of a layered system applied to the ISO communications model

The layered architecture can be also shown in a graphic application. An example of a graphic application is shown in Figure 2-4.

Application: Systems of Diagrams: Making sense to the User.
Diagram: Knows how to handle all objects: Knows what they mean when connected.
Objects having meaning: Bitmap, Square, Circle, Flowchart Objects, State Diagram Objects, Entities etc.
Basic Graphic Object: Implementing functions of color, size, font. Supports moving, scaling etc.

Figure 2-4: Example of a layered system applied to a diagramming application

Advantages.

1. Layered systems support designs based on increasing level of abstraction. This allows implementers to partition the problem into a sequence of incremental steps.
2. They support enhancements. As the boundaries between the layers are clearly defined, enhancing a particular layer or part of the system is easier.

3. They support re-use. Like abstract data types, they allow different implementations of the same layer to be used interchangeably, provided they support the same interfaces to adjacent layers.

Disadvantages.

1. It is not easy to view all systems in the layered fashion. Primarily, large systems can be viewed in a layered fashion.
2. Some systems need closer coupling due to performance requirements. Normally each layer communicates only with its adjacent layers. Performance and architecture issues of the program may not allow this in certain programs.

2.7 Repositories

The repository style consists of two types of components, a central data structure representing the current state and a collection of independent components operating on the central data store. Figure 2-5 shows an abstract repository example. Interactions between the repository and its external components can vary significantly among various systems.

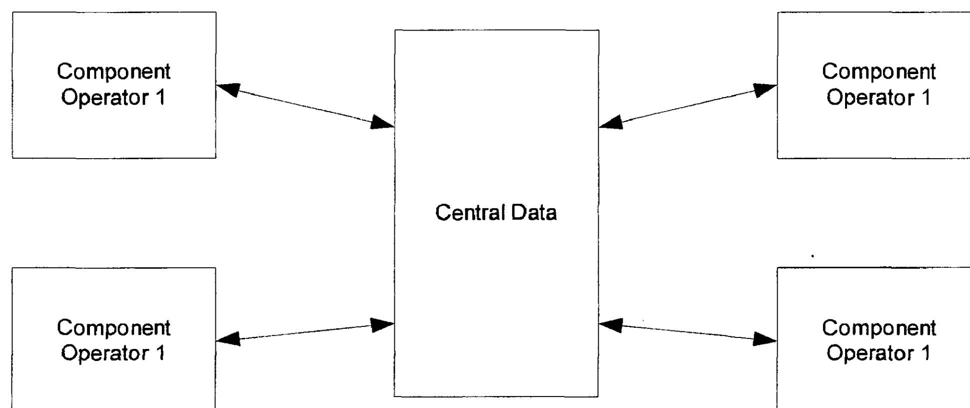


Figure 2-5: Repositories

Databases are a special case wherein the transactions in an input stream trigger a selection of processes to execute. If the current state of the central data structure is the main trigger that selects processes to execute, the architecture is called a blackboard.

The blackboard style can be found in applications in the field of artificial intelligence. It consists of the central storage of data, and knowledge pertaining to how a problem can be solved, with the data itself defining the problem. It is not necessary to solve the entire problem, but taking a step towards solving the problem is permitted. The blackboard model consists of three major components:

- Knowledge sources: separate and independent modules of problem-specific knowledge.
- Blackboard: the problem-solving state data is the blackboard, which is global.
- Control: knowledge sources that respond opportunistically to changes in the blackboard. This is the controlling module, and the level of control and strategy used is implementation dependant.

Only the knowledge sources can modify the data. Each knowledge source is responsible for knowing the condition under which it can contribute to a solution. The purpose of the blackboard is to hold computational and solution-state data needed by, and produced by, the knowledge sources. The control modules select the appropriate knowledge sources based on the latest changes to the information in the blackboard.

Disadvantages

The repository architecture trusts in all the component operators working correctly. If an operator corrupts the data or destroys the data, other components can be affected.

This architecture cannot be adopted to all problems but to only problems in which a central data repository exists.

2.8 Interpreters

In an interpreter style, a virtual machine is produced in software. An interpreter includes the pseudoprogram to be interpreted and the interpretation engine itself. The pseudoprogram includes the program itself and the interpreter's analog of its execution state (activation record). The interpretation engine includes both the definition of the interpreter and the current state of its execution. It generally consists of four components:

1. An engine to interpret the pseudocode.
2. A memory that contains the pseudocode that is to be interpreted.
3. A representation of the control state of the interpretation engine.
4. A representation of the current state of the program, which is being simulated.

Figure 2-6 shows a simplified view of an interpreter.

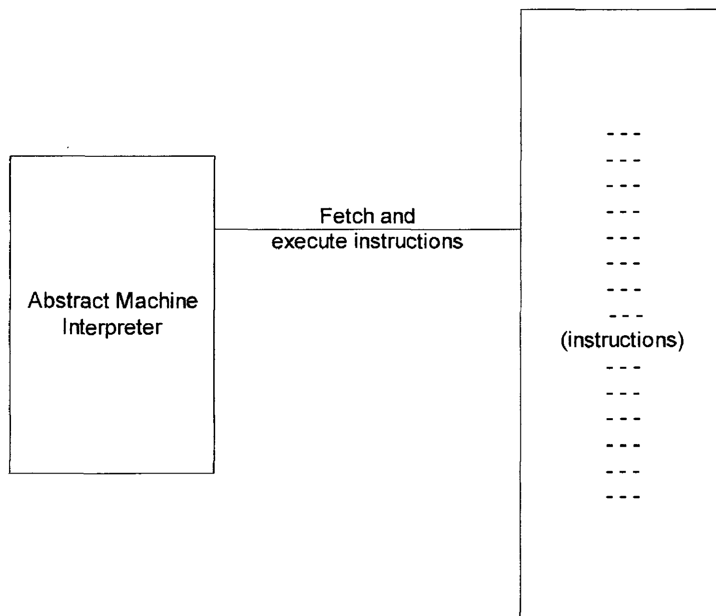


Figure 2-6: Interpreter

Interpreters are commonly used to build virtual machines that close the gap between the computing engine expected by the semantics of the program and the computing engine available in hardware. The Java Virtual Machine (JVM) can be thought of as an example of an interpreter.

2.9 Process Control

Process control software architecture involves a control-loop and a feedback. The very nature of this style indicates an iterative technique.

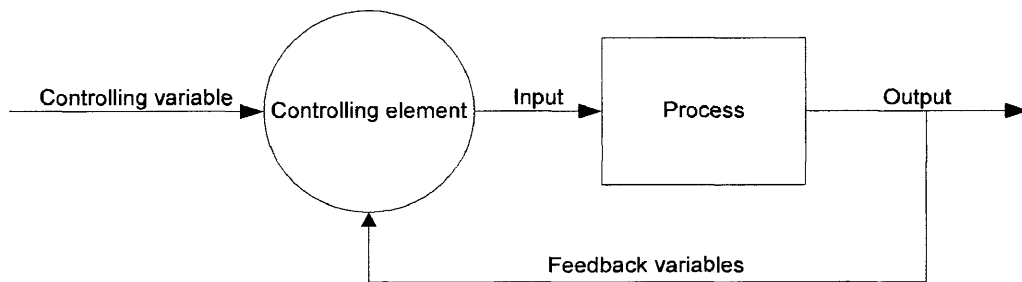


Figure 2-7: Feedback control system

There are variations to the process control architecture in which the position of the controlling element is either before or after the process. An important feature of this style is that a feedback exists in the control structure of the application (program).

The process control style is very domain specific and cannot be applied to most problems. It is implemented for problems that have a feedback element.

2.10 Main Program / Subroutine Organizations

The primary organization of many systems mirrors the programming language in which the application is written. For languages without support for modularization, this often results in a system organized around a main program and a set of subroutines. Each subroutine performs a fixed task and the main program acts as the controller for the subroutines. The main program provides a control loop for sequencing through the subroutines in an order defined by the programmer.

This architectural style is normally used for simpler problems. The systems implemented in this style normally have a single thread of control.

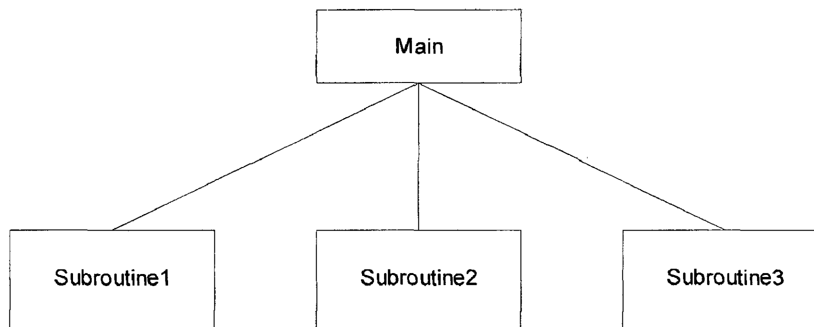


Figure 2-8: Main program – subroutine architecture

Advantages

The advantages of using such architecture style are simplicity, efficiency and permitting hierarchical reasoning. Support for such structure is available in most programming languages.

Disadvantages

The disadvantage of this approach is that as the program grows in size, it gets difficult to maintain. The presence of potentially large global data structures makes modifications of algorithms and the data structures difficult to implement.

2.11 Distributed Processes

Most distributed systems are characterized by their topological features. These include ring and star organizations. Others are better characterized based on the types of inter-process protocols that are used for communications, for example, the heartbeat algorithm.

A common form of distributed system architecture is a client-server organization. In these systems, a server represents a process that provides services to other processes (clients). Usually the server is not aware of the identity of the client or the number of clients that will access it at run-time. However, the clients do know the identity of a

server and can access it by remote procedure call. Similarly, Internet based or Web based systems can also be classified as part of distributed systems.

2.11.1 Client Server Systems

Client server systems have become extremely popular these days. The 'client' normally refers to a single user workstation that provides presentation services and the necessary computing power for connectivity and database services. The 'server' is one or more multi-user processors with shared memory providing computing, connectivity, database services, and interfaces relevant to business needs. Figure 2-9 shows an illustration of a client-server system.

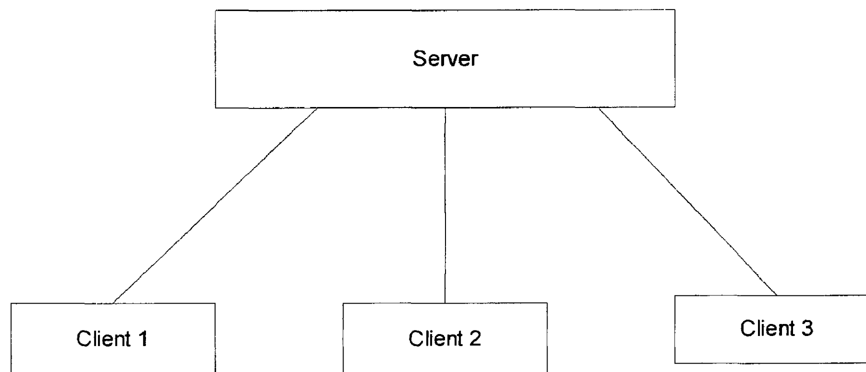


Figure 2-9: Client-server architecture

In a client server environment, the client requests services from the servers – the server processes the requests and returns the results to the client. The communication mechanism between the client and the server is a message passing inter-process communication that enables distributed placement of the client and server processes. It is a software model of computing and not a hardware definition.

2.11.2 Browser Based / Internet Applications / Web Based Architecture

Although no references have been made about the Web application style, I classify these applications as a special case of client-server applications. The client in this case is an HTML browser. What makes this architectural style unique is that the communication between the client and server is very well defined. The data passed uses the "http" protocol and is passed over a TCP/IP network.

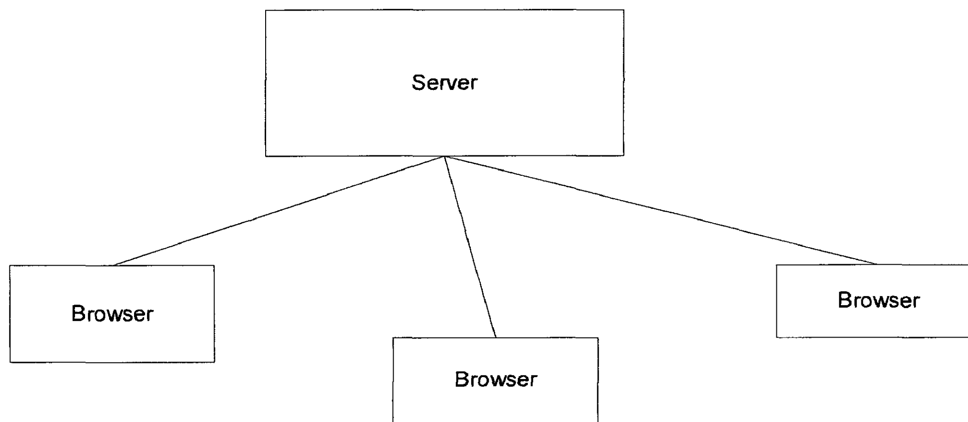


Figure 2-10: Web architecture

Typically, several hundreds (thousands or millions) of clients can use this application. Normally, no special software needs to be installed to run Web applications. The client software is available on most modern computers and is a Web browser. The user interface is also standardized and these applications are fairly easy to use.

2.12 Heterogeneous Architectures

Most large systems fall under this category. For any large system, it is difficult to design a system based on a single architecture. A large system normally consists of several

components. Each component in turn can consist of other components and is responsible for solving its part of the problem and each can have a different architecture style.

A very common classification is having a layered system. Each layer is responsible for performing tasks and communicates with the adjacent layers. The layer itself can have its unique architecture. Nowadays, multi tier systems are getting more popular. For example, a layered or hierarchical software system is designed level by level. At each level there is an object model that defines classes, objects and operations that the next higher level may reference.

2.12.1 Component Based Architecture

This architecture style is based on having several components interacting with each other. It can also be seen as a heterogeneous architecture style, as each component can have its own style. This style is normally achieved by creating the software architecture using a "divide and conquer" technique. This involves splitting the problem domain into smaller domains and solving each domain individually. Each problem domain is solved by a component.

2.13 Self-Evolving Software Systems

Self-evolving software can change over a period of time. It adjusts to external and internal states. The architecture of these systems involves a rule engine, which determines the software modules to use in case of changes to the external, internal or the input parameters. This is depicted in Figure 2-11.

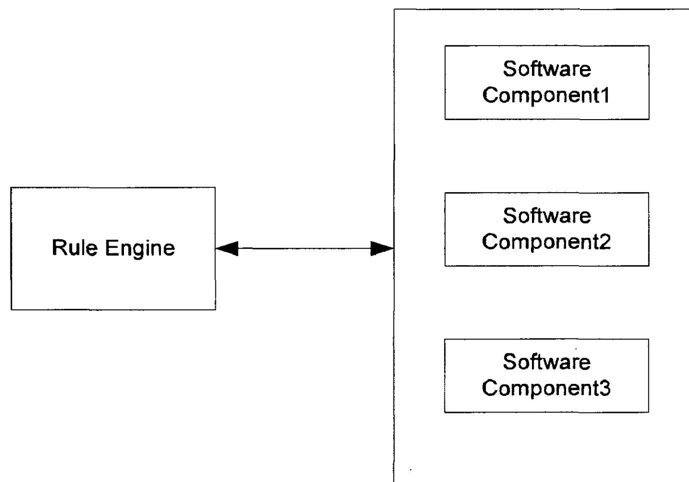


Figure 2-11:Architecture for evolving software

The rule engine is responsible for selecting the software components that are given the task of solving the requested problem. It can dynamically select a software component based on the internal and external states.

3 Extending Standard Architectures Using COM

3.1 Introduction to COM

COM can be described as the elder brother of object-oriented programming. Although COM is strictly a model description, the basic model is simple and is about interfaces.

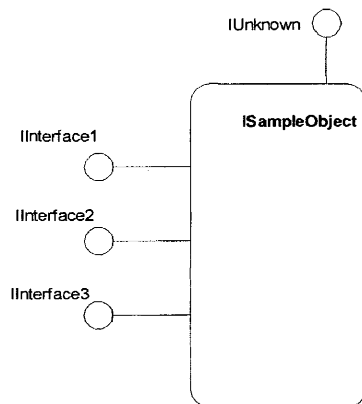


Figure 3-1: Typical COM object

Figure 3-1 shows a typical COM object. Every COM object supports the basic COM interface called IUnknown. A definition of IUnknown is as follows:

```
interface IUnknown
{
public:
  HRESULT QueryInterface(REFIID iid, void **ppvObject);
  ULONG AddRef(void);
  ULONG Release(void);
};
```

The methods, `AddRef` and `Release`, are responsible for reference counting of the object itself. The `QueryInterface` method gives access to other interfaces available within the object. A client who wishes to use an interface from the object can query the IUnknown interface and fetch the required interface.

3.1.1 Interoperability

COM objects are supported in various languages. It is presumed that COM is supported only in C++, but various languages do support COM. COM interface definitions are best described in a language independent of C++ (or Basic or Java), and are generally described using the Interface Definition Language (IDL).

Although COM is primarily used only on the Microsoft platform (Windows 95/98/NT/2000), implementations of COM are also available on Linux and its variations. SoftwareAG (http:://www.softwareag.com) provides free software called EntireX that supports COM on the Linux operating system.

3.1.2 Distributed Component Object Model (DCOM)

Programs can access COM objects in a distributed environment. However, automatic discovery of objects across different machines is not a COM supported feature. A program (or object) accessing or initiating calls is referred to as the client, whereas the object being accessed is referred to as the server. Accessing objects on other machines involves two main tasks:

1. Setting up information about the location of the server object: when the client invokes the server object, it needs to know the location of the server. The information about

the server has to be registered on the machine on which the client software runs. This information is stored in the Windows registry.

2. Non-trivial parameters have to be marshaled across object boundaries: some parameters cannot be directly transferred from one machine to another and need to be treated differently.

3.1.3 Marshalling.

Calling the interfaces of COM objects appears similar to making function calls. There can be complications if the function lies in another process. A pointer passed as a parameter may not be valid in a different process. Additional complications can arise, when the object lies on another machine and a pointer is passed. Copying data from the address space of a client program to the address space of the component is referred to as marshalling.

3.2 The Key Word in Context Problem

Parnas (1972) described the following problem in his paper in 1972.[2]

The Key Word In Context (KWIC) index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be "circularly shifted" by repeatedly removing the first word and appending it at

---

[2] The KWIC problem is explained as it will be referenced several times and various solutions addressing it will be presented.

the end of the line. The KWIC index system outputs a listing of all the circular shifts of all lines in alphabetical order

The KWIC problem is well known and has been used several times since 1972 to describe qualities of design and solution. My interest in describing the KWIC problem is to solve it using various architectural styles and enhancing it using COM and XML. After viewing the modified solution, I will describe its advantages.

3.3 Modifying the Pipe and Filter Model using COM.

In case of the pipe and filter model, each filter should communicate using an interface. The output of the filter is also available through an interface. Providing such interfaces makes it more scalable. The filter can now work with formats other than text files. A separate interface has to be introduced for binary or COM compatible data types.

The pipe and filter solution implemented using COM can be extended to a distributed solution. The filters can run on different machines. The communication layer provided by the DCOM architecture provides the pipes. The DCOM pipe and filter solution can be distributed across multiple machines.

```
interface iFilter
{
  HRESULT  ProcessData([in] BSTR inputData,
                       [out] BSTR * outputData);
}
```

The above interface has a limitation that the data has to be made available beforehand. We can model the interface to read a few blocks of data at a time.

```
interface iFilterCP
{
  HRESULT ProcessData([in] IdataStream inpData,
           [out] IdataStream *outData);
}
```

Here, the filter interface (iFilterCP) is passed the pipe interface (IdataStream), from which it should read the data.

3.3.1 Solving the KWIC Problem

We can solve the KWIC problem using the pipe and filter architecture. Figure 3-2 describes the various modules.

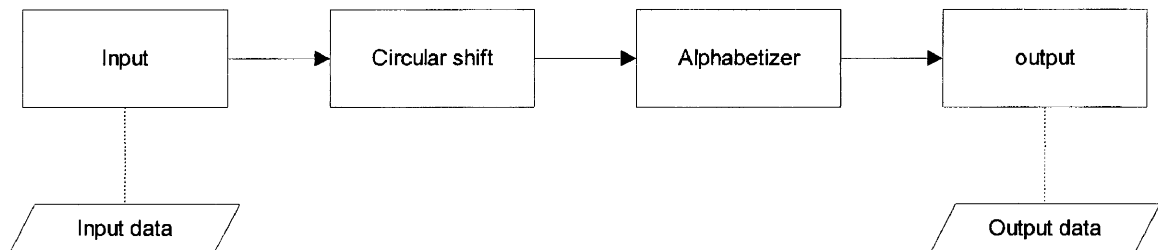


Figure 3-2: Pipe and filter solution for KWIC

In the COM enabled version of this architecture, the filters doing “Circular Shift” and “Alphabetizer” along with the “Input” and “Output” modules, communicate with each other using COM interfaces.

All components that support generic input/output interfaces can be developed. The basic architecture of the system remains the same, while the development of the filters and the way the filters interface with each other changes. Each filter component implements the same interface (iFilterCP). The definition of the interface is as follows:

```
interface iFilterCP
{
  HRESULT ProcessData ([in] IdataStream inpData,
             [out] IdataStream *outData);
}
```

There can be several designs to solve this problem. The solution depends on the design of the component and how the component is to be used. In the second design, we will define a separate interface for the input data and another interface for reading the output data. In this approach, the component that actually implements these interfaces will be required to store the data. The same component also defines the output interface. Definitions of the input and output interfaces are as follows:

```
Interface iFilterInput
{
  HRESULT InputData ([in] IdataStream inpData);
}
```

```
interface iFilterOutput
{
  HRESULT iFetchData([out] IdataStream *outData);
}
```

Both these interfaces (iFilterInput and iFilterOutput) appear similar. The input and output parameters use the parameter type IdataStream. The user defines IdataStream, which in a simple case can be a stream of characters.

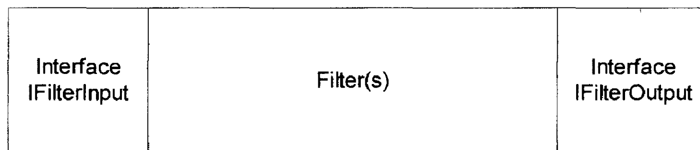


Figure 3-3: Filter with interface

One point to take note is that the actual component does not undergo any change. It is only the manner in which the component communicates with the other elements of the solution that changes.

3.4 Modifying the Layered Model Using COM.

Modifying the layered model enhances the abstraction between the layers. The local-remote transparency provided by COM can help distribute the processes across machines. Consider a hypothetical layered application example:

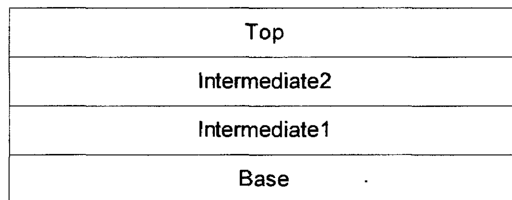


Figure 3-4: Sample layered system

The layers communicate with their adjacent neighbors. Each layer can expose interfaces to its functionality using COM. An example of such an interface is given below:

```
Interface iBase
{
    HRESULT Function1([in] long inp1, [out]long *retvalue);
    HRESULT Function2([in] iInp, [out]long *lREtVal);
}
```

3.5 Modifying the Object Model Using COM

An object model is not the same as a component object model. However, we can modify the object model to appear as a component object model. Object models are easier to adapt to COM. The driving factor for such a conversion is the requirements of the system and the existing object design. The simplest approach is to convert the existing interfaces to COM interfaces. However, some object interfaces cannot be easily converted as they reference global data or use parameters not directly supported by COM, for example, passing of data structures.

3.6 Modifying the Main Program and Subroutine Architecture Using COM

The main program and subroutine architecture can be easily enhanced using COM. The interfaces exposed by each subroutine can be seen as COM interfaces. The main program can be viewed as a controller calling various interfaces.

The conversion initially appears to be uncomplicated as each subroutine can be viewed as a component or a method in the component. However, there can be shared data between the subroutines that are not components. This data needs to be maintained by the controlling module and then passed to every subroutine component. The complexities arising from such a conversion can vary depending on the amount of global data.

3.7 Modifying the Event Based Architecture Using COM

In an event based architecture system, a client component registers its interest with a server component (or a controlling component). When the event takes place, the client component is notified of the event.

COM provides support for callback events. The ConnectionPoint interfaces are provided especially for callback event handling. The support is provided by a special interface called IConnectionPoint and IConnectionPointContainer. An analogy can be drawn with 'C' callback functions for this functionality.

The component generating the event (SourceComponent), and the component waiting for the event (SinkComponent), recognize and implement a particular interface, for example, ISinkNotification.

This interface is approved during design-time by the developers of the SourceComponent and the SinkComponent. The SinkComponent registers itself with the

SourceComponent, and passes the ISinkNotification interface pointer indicating the event it has interest in. The server (SourceComponent) maintains a list of clients interested in the events and then notifies the clients when these events occur.

3.8 Modifying Distributed Systems Using COM

Distributed systems can take advantage of COM and use its features and properties. COM's transport layer can be used for communication across machines. Another feature that can be used is that of parameter marshalling. Marshalling involves passing parameters across the network, and reconstructing them again in the target machine address-space. Modifying distributed systems involves analyzing the existing data being communicated across various machines and the processes using the data. The processes that communicate need to define interfaces to allow them to interact with each other.

However, using COM as a transport layer may not always be efficient. For example, the main focus of an on-line video conferencing application is its ability to efficiently communicate amongst machines. Adding a COM layer for communication can reduce the efficiency of the application and make it unusable.

4 Communication Between Components

Techniques used to pass data between different components or modules can vary significantly. Initially, the decision regarding data format depends on the developer. If the data is communicated among a number of people, the format of the data can be formalized. In earlier days (and even today), data used to be in a simple comma delimited format or tab delimited format.

Example (Comma delimited persons data)
Sujit Chaubal, 01-01-1971, Omaha, Computer Science
John Doe, 02-12-1982, Lincoln, Electrical Engineering

There are several problems with using such a format. As the "comma" is a delimiter, if it appears as part of the data, it has to be handled in a special manner. This leads to special cases and such variable formats can lead to confusion as well as parsing errors.

4.1 COM and Data Formats

The main data formats that are supported by COM are listed below:

Boolean
Longs
Floats
Strings (BSTR)
Variants (These are generic data types that hold another data type)
Structured Storage (for storing arrays)

Although these are the common data types, other data types can be passed by either converting them to a known format or by informing COM on how to pass these parameters (using the IMarshal interface).

4.2 What is XML?

'XML-in-10-points' (1999) [online] describes the Extensible Markup Language (XML) as the universal format for structured documents and data on the Web. The easiest way to understand XML is by looking at some samples.

Example

```
<THESIS>
    <AUTHOR>
        <NAME>Sujit Chaubal</NAME>
        <SSNO>2322221221</SSNO>
    </AUTHOR >
    <SCHOOL>University of Nebraska at Omaha</SCHOOL>
    <MAJOR>Computer Science</MAJOR>
</THESIS>
```

A tutorial about XML is beyond the scope of this thesis. W3C [online: http://www.w3c.org] has a specification of the XML format and has references to several excellent resources on the Internet that can provide this information.

XML appears similar to HTML. However, an author creating XML data has the freedom to design his/her own tags. The same information as the above example can be represented in a different XML format.

```
<T>
        <A SSNO="2322221221">Sujit Chaubal</A>
        <S>University of Nebraska at Omaha</S>
        <M>Computer Science</M>
</T>
```

4.3 XML and COM

XML is a data format while COM is an object model. But, XML can be used as a data format to communicate between components.

As XML is a relatively new specification, several new standards and protocols are being developed revolving around XML. One of them is called the Standard Object Access Protocol (SOAP). The protocol tries to formalize object invocation using XML wherein the objects are not restricted to being in the same process or in the same machine.

4.4 Extending Data Formats Using XML

Programs use various techniques to pass data between the various parts of the application. Most architectural styles are based on how data is stored, and communicated between the different components of the application. When using COM components, programmers are expected to use the standard data types for communicating data. COM components can define their own interfaces for communicating data as well.

One can also pass data using an XML format. When data elements are passed in an XML format, an extra task is added in the components code, which involves another step of interpreting the data. This is depicted in Figure 4-1.

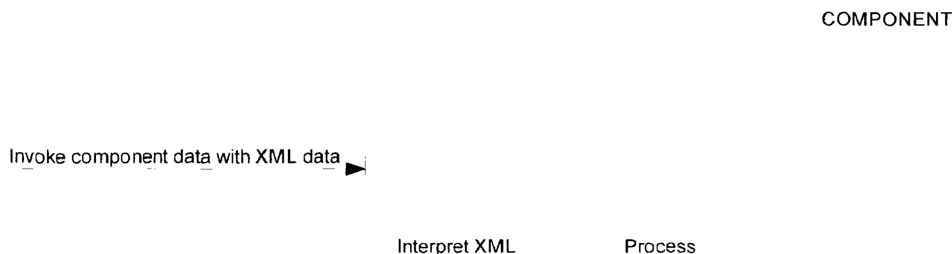


Figure 4-1: Component requiring XML processing

Although it appears trivial, the task of interpreting XML can be expensive. XML data has to be parsed to fetch the actual data set. If the size of the data is large, then time to parse the data can increase.

Example

To pass data about authors between various components; instead of passing separate fields of a formatted text string, an XML data set can be passed. The following example is a sample of an IDL (Interface Definition Language) definition that accepts an XML stream as a parameter.

```
Interface Iconference
{
HRESULT AddAuthor ([in]BSTR *xmlstream, [out] long *retval)
}
```

The input to the interface method AddAuthor is an XML string that can be of any size. This stream should be in a defined XML format. There are some standard formats of XML that are documented for particular information in some industries. As users are free to decide on the XML format, it has to be agreed upon between the two parties involved in using the data.

4.4.1 XML and Character Encoding

If there is a 4-byte Unicode byte order mark (0xFF 0xFE 0xFF 0xFE) at the start of the XML stream, it assumes the encoding to be UTF-32.

If the XML data starts with a Unicode byte-order mark [0xff 0xfe] or [0xfe 0xff], the document is considered to be in UTF-16 encoding else it is in UTF-8 format.

The following are some XML document encoding standards:

ISO-8859-1

UTF-8

ISO-8859-1

UTF-8 (using character entities)

UTF-16 (Unicode with byte-order mark)

The text in the XML data can specify a different encoding style in the header.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<- data follows ->
```

The most popular encoding format is UTF-8. It takes the first 127 characters of the Unicode standard (these happen to be the same characters as basic Latin characters), and maps them directly to single byte values. It applies a bit shifting technique using the high bits of the bytes to encode the rest of the Unicode characters. However, this makes the data unreadable.

In XML, an additional character-encoding format is available. A single character can be written by its numerical value, for example, “å”. These are especially useful for encoding “<” and “>” characters that are used for defining the tags.

4.4.2 XML and Binary Data

Binary data needs to be encoded before it can be passed along with an XML stream. Encoding the binary data increases the size of the data. Every byte may need to be encoded depending on its value. Also, this data has to be re-created by the program receiving the data, if it has to be used again.

The overhead for passing binary data is high, but it is possible to transfer binary data from one component to another.

4.5 Sample for Interpreting Data

The following code sample demonstrates how XML data can be fetched and interpreted using the Microsoft XML parser. The sample code is written in VBScript and can be executed on the command line.

```
set xmlDoc = wscript.CreateObject( "Microsoft.XMLDOM" )
bSuccess = xmlDoc.load( "http://www.someserver.com/queryusers.xml")
set root = xmlDoc.documentElement
set nodes = root.selectNodes( "USERNAME" )
for each usernameNode in nodes
  wscript.echo usernameNode.text
next
```

The ease (or difficulty) of interpreting data depends on the parser being used and the format of the XML data.

4.6 Advantages of Using XML

1. Using XML as the interface format allows flexibility. As use of XML grows and gets standardized, the input format for components will be standardized. The component can change, but the interface need not change.
2. Standard data can be passed to the components using XML, and the component can use information that it needs.
3. If extra XML data is passed and the receiver is well written, the receiver does not have to be changed.

4.7 Disadvantages of Using XML

1. As the abbreviation says, XML is an extensible format. Every time XML data is interpreted, it requires more processing. XML parsing takes more time and memory. It is not the best data format for passing parameters, if performance is a major issue.

2. XML data includes the XML tags. This increases the size of the total data stream. This may be of concern, especially when the size of the data is large and where network bandwidth is a limitation.
3. As XML is a developing standard, programmers can run into version problems.

4.8 Revisiting the KWIC Problem

In the previous Chapter, I developed a solution for the KWIC problem using the pipe and filter architecture and COM. We can now extend this solution by using XML. In this solution, the difference being that the data passed is in an XML format.

The input interface is defined as follows:

```
interface iFilterInputXML
{
  HRESULT InputData ([in] BSTR inpXMLStream);
}
```

The output interface is defined as follows:

```
interface iFilterOutputXML
{
  HRESULT iFetchData([out] BSTR *outXMLStream);
}
```

The interfaces (iFilterInoutXML and iFilterOutputXML) expect the string data to be in the XML format. All the filters have to be aware of this XML format to parse and process the data

5 A Software Architecture for Incremental Applications

5.1 Problem Definition

A typical scenario in software development is piecewise development. A basic driver module is developed, and several supporting modules need to be added on. A real life example can be a graphics package that needs to support various operations as well as various file formats. For example, initially, the software can support a single shading model, and then keep adding other shading models or rendering techniques. Similarly, the package can start by supporting GIF files and later add other modules like TIFF, BMP etc. This is a very common example.

An industry example can be an application package for an enterprise. Initially, the package can show interfaces to a payroll system. Further, it can add an accounts payable system, general ledger and other systems. Although this example is not theoretical, it is an ambitious project.

The goal lies in defining an architecture in which it is possible to replace modules selectively without affecting other modules.

5.2 Approach to Solution

There can be no generic solution to solve all the above problems. Software and products mature over a period of time. As companies release multiple versions of software, each new version tries to improvise on the previous solution.

The basis of my solution uses the Microsoft Component Object Model (COM). COM is not about defining objects or what objects do, but about how objects communicate and present themselves to the outside world.

5.3 Snap-In Architecture

The idea behind this architecture is the ability to query the availability of interfaces. The module that is developed or changed supports a standard interface. The controlling module is aware of this interface. The interface describes the new methods implemented and hence, the controlling module needs no change. An example of a simple interface is shown below:

```
interface ImoduleDescription
{
  HRESULT MethodCount([out] long lMethods);
  HRESULT MethodName([in] int nMethodNo,
          [out]BSTR *sMethodName,
          [out]BSTR *sMethodDesc,
          [out] BTSR *iconData);
  HRESULT InvokeMethod([in] long MethodCount);
}
```

This interface is similar to the published interface IDispatch. A controlling application can call the methods of the module and fetch the number and the descriptions of the methods.

We need not re-compile our controlling application or any other dependant applications because linking in case of components is dynamic and late.

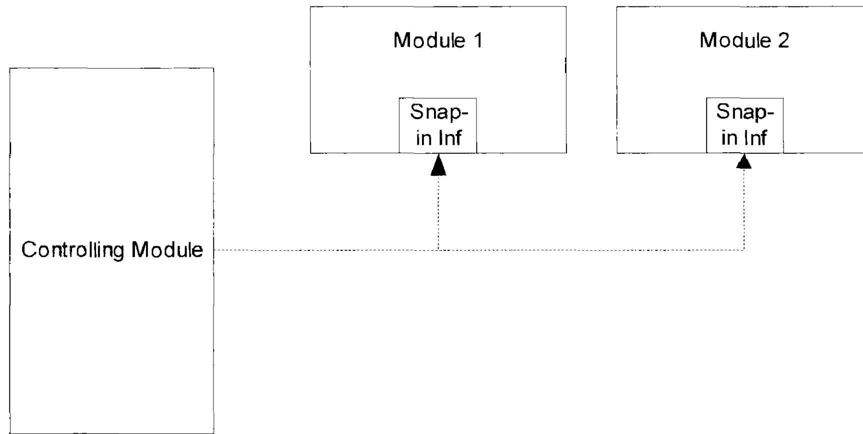


Figure 5-1: SnapIn modules

One can create snap in's at various levels. Figure 5-1 shows an example of the snap-in architecture. A simple example has a controlling module and, as sub-modules are added to the system, the controlling module involuntarily detects them and takes appropriate action.

5.3.1 Example: Snap-Ins in a File Converter Application

Problem: To convert files from various formats to a simple text format.

A stepwise solution is, to develop components for every file type supporting the same interface called IFileConverter.

```
Interface IFileConverter
{
        HRESULT getFileType([out, retval]BSTR *sFileType);
        HRESULT convert([in] BSTR sInput, [out, retval] BSTR *sConvertedtext);
}
```

In this example, the method is fixed and is known. Every new file type can be implemented in its own component. On registering the component, it will be available to the controlling module. However, the controlling module must read the presence of the newly added component. This means that the controlling module has to periodically (or every time) check, if any new components have been added to the system.

A two level generalization is present when the registered component informs the controller about the methods it can support or perform.

5.4 Object Discovery

COM provides a feature called component categories. As the name suggests, a component category specifies a standard interface that an object (or component) supports. Some standard component categories have been published. However, a user may create custom categories. All the components that support the category can be queried for. For a program to find existing components, it has to make queries using a COM component category API.

5.5 Scalability and Internet Web Applications

One of the problems faced by Web applications is handling the increasing number of users. This problem is not faced by standard corporate (intranet) applications, as the number of users is pre-determined and the software solution can be developed accordingly. Incase of single user software, this problem does not arise at all.

As the number of users on the Internet increase, it will be difficult to control the load on the servers running the particular application. Although servers can restrict the number of users accessing the application, the other users can get inaccurate results from the software.

Although there is no solution, there can be several optimizations or tricks that can be used to solve such problems.

5.6 Stateless Modules/Applications

One of the ways of getting around this problem is by having several servers accepting tasks at the same time. However, this requires a controller that forwards the requests to a particular server from a cluster of servers.

In case of Web applications, several solutions are available. This solution can be called "load balancing". Server replication is an obvious solution but may not work in every case.

5.7 Load Balancing by Task Distribution

When there are large applications to perform complex time consuming tasks, the tasks can be distributed by adding more servers or hardware that can handle the load. It also

means that the application has to be designed in such a way, that it is possible for a network of computers to solve the problem together.

This typically appears in Web applications where the number of users can grow substantially and server load can increase and deteriorate performance. To distribute tasks we can use the following models.

- Push model: When the master machine gets a task, the request is passed on to a slave machine. The master machine has a list of available slave servers to which it can redirect the task.
- Pull model: The master server maintains a list of tasks. Whenever a slave machine is ready to perform a task, it pulls a task from the master list and executes it.

5.7.1 Push Model for Load Distribution

As the number of users grows on the Internet, applications need to scale and be capable of handling more loads. Formatting HTML pages and serving them can also take a significant amount of time. As display techniques improve, people will be more creative in presenting data and hence, the amount of graphics that needs to be transmitted has grown.

1. A simple approach followed is to associate each user with a particular machine and increasing the number of machines as the number of users increase.
2. If the users data is isolated, the user and the data can be processed from a dynamically assigned machine.
3. In a typical e-commerce application, global data can be replicated and users can view this from various machines.

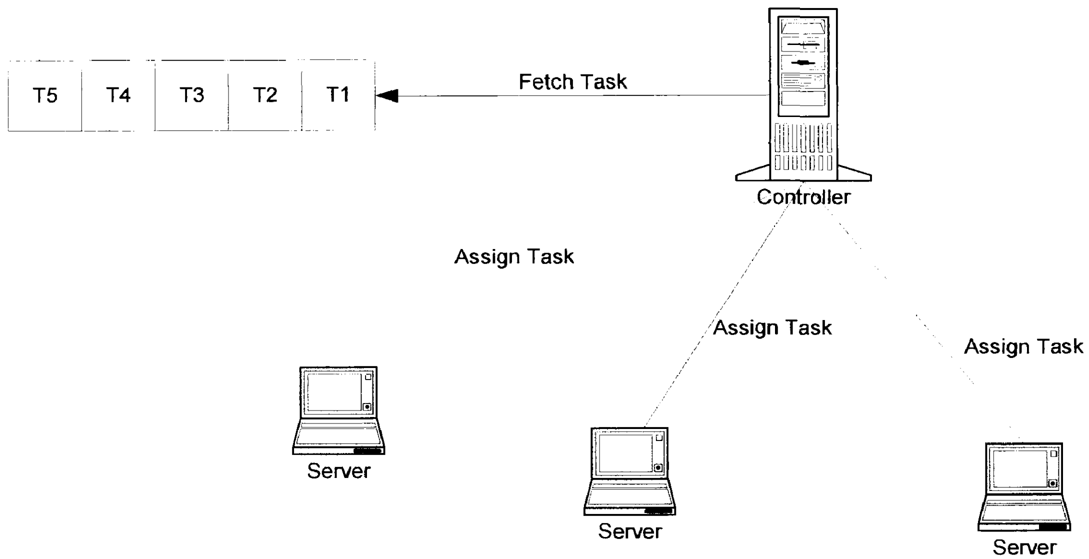


Figure 5-2: Push model for task distribution

5.7.2 **Pull Model for Load Distribution**

The pull model can be viewed as a distributed queue. A queue of tasks exists from which processes pull jobs and perform them.

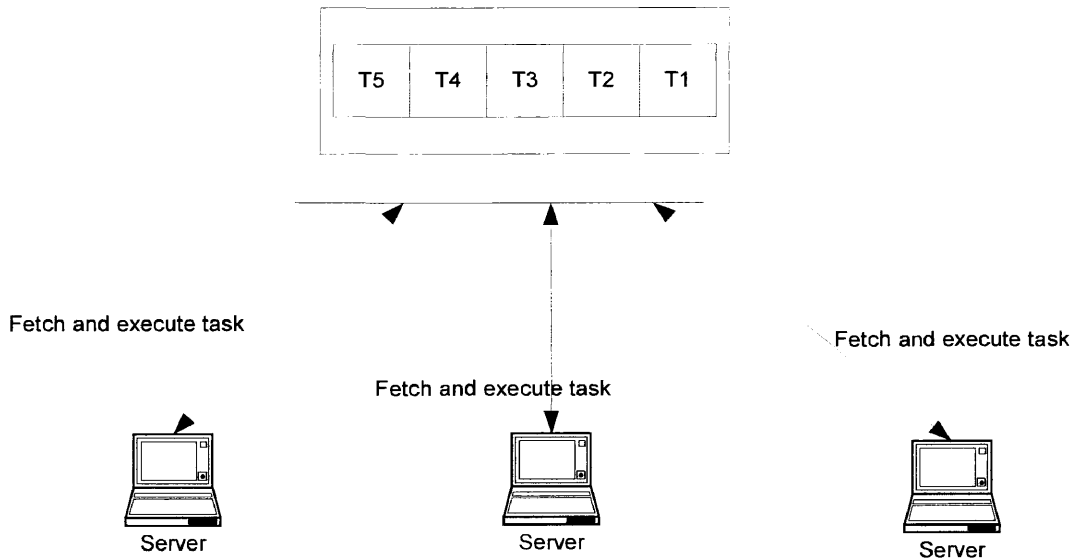


Figure 5-3: Pull model for load distribution

5.8 Distributing Databases

Although "distributed databases" is a large topic, it is one of the techniques used to make applications scalable. Ozsu and Valduriez (1991) discuss several techniques of how large data can be split up into smaller manageable units. Data can be split either horizontally or vertically.

- Horizontal splitting: this involves dividing a large table into two or more separate tables such, that the columns of the large table are distributed in the two table

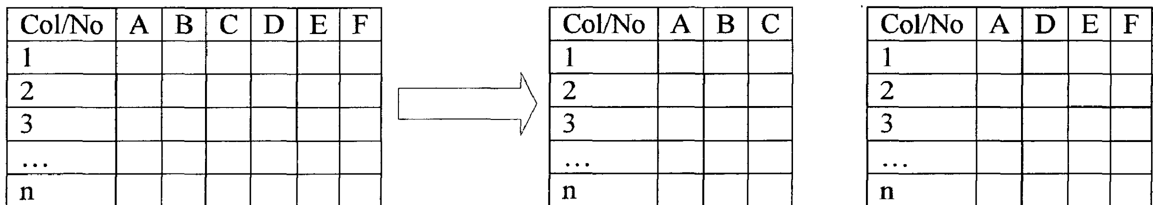


Figure 5-4: Horizontal splitting of a database

- Vertical splitting: this involves breaking a large table into two or more tables, such that half (or proportional) of the rows of information are in one table and the remaining rows in another table.

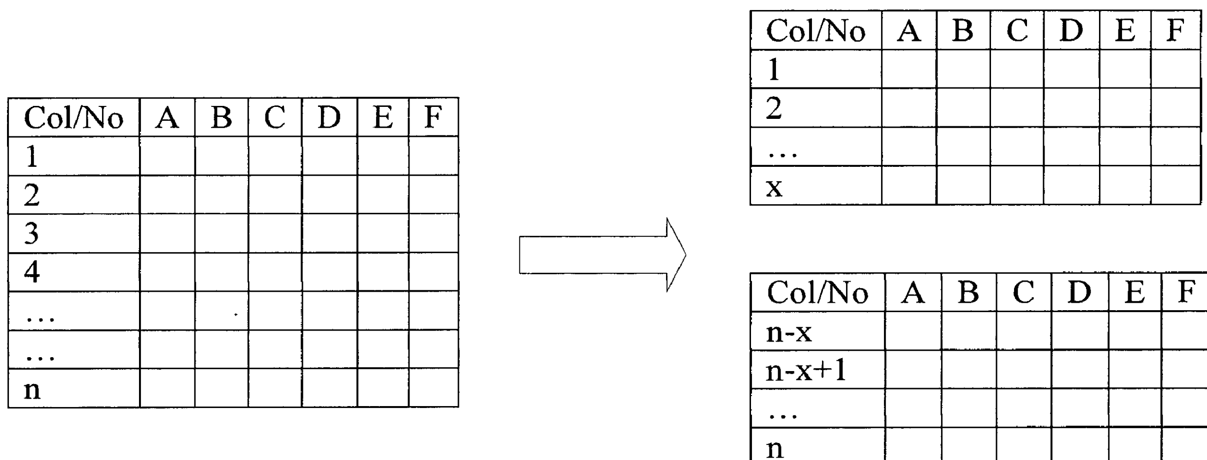


Figure 5-5: Vertical splitting of a database

Both techniques require a controller application that is aware of how the database is split and how queries are handled across the split databases.

Another technique used in applications involving a large number of users, is database caching. The data associated and required by each user is identified and copied from the master database to a local database. This database can be present on the local machine or client machine. The aim being most user actions need not access the master database. This helps reduce load on the master database and improves the user experience performance, as local database contains less information and is faster.

5.9 Building the Application

To build a scalable and extensible application, we need to understand and make use of all the techniques described in Chapters 3, 4 and 5. As every problem can have several alternatives and solutions, the techniques chosen determine the final scalability of the

application. Here are some architecture guidelines for creating scalable and extensible applications:

- Distinct interfaces: each module must have its own interfaces.
- XML interfaces: modules should understand XML and provide XML data interfaces to other modules.
- COM interfaces: COM interfaces should be exposed, so that other modules can easily access them.
- Database architecture supporting distribution: the database architecture should be such that it facilitates distribution.
- Controlling module: if a controlling module exists, it should be able to distribute tasks to different processes. It should not be limited to assigning tasks to processes on the same machine.
- Stateless application: in Web applications, modules should be stateless. This eases the task of scaling, as machines can be added to handle the load.

These guidelines help scale the applications, however, it is not always possible to follow every guideline. The architecture of the application is dependant on the requirements specification. For example, it may not be possible to have a stateless application.

5.10 Advantages

- Using COM for building modules eases black box testing of components. The components need not be tested in the language the module was written in. Testers can test at the component level easily and are not required to wait for the entire packaged software to be ready.

- As the COM interfaces have to be defined in advance, it promotes the concept of "designing before coding.".
- Use of XML as the format for passing data between module forces structure to the data being passed. It promotes an extensible software development process.
- Using techniques like snap-ins and XML for data passing aids in extensible development. Modules of functionality can be added to the solution as and when required.
- Distributing the database and using COM makes the solution scalable. COM components can be deployed on multiple machines making minimal changes.

The techniques described can be applied to the common architecture styles. The solution achieved using these techniques is scalable and extensible.

6 Case Study of Building a Scalable Application

6.1 Meta Searcher

6.1.1 Problem Definition

A meta application engine for the World Wide Web has to be built. The program should serve Web pages by fetching data from the different sources. For example, it should function as a search application, fetching results for a search term from various search engines and then present the aggregate of these results to the user. The application should be able to fetch search information, auction items, shopping data, news etc.

6.1.2 Solution Phase I

In the first phase, a meta search engine is created. Search results from the various Web sites are fetched and then integrated. In the next phases, results from different Web sites will be assembled, such as Web-auction results, comparison-shopping for books, etc. Figure 6-1 depicts the heterogeneous architecture of the meta search application.

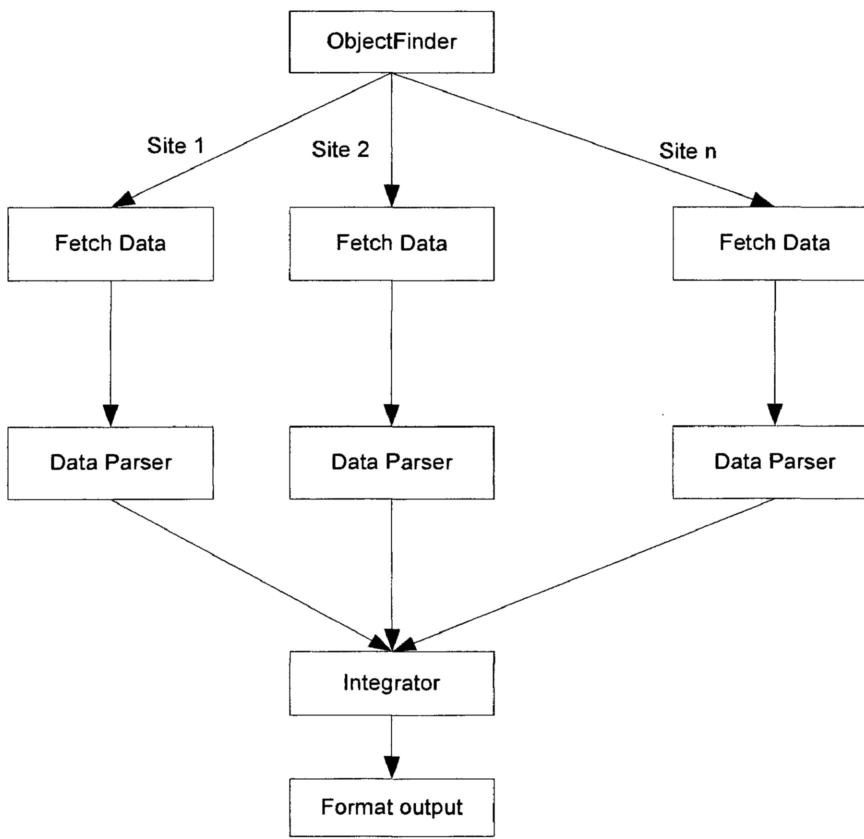


Figure 6-1: Architecture for meta search

The architecture of this application consists of the following components:

1. **Object Finder:** The object finder is responsible for querying the operating system to find the objects that support the meta search application.
2. **Data Fetcher:** The data fetcher is responsible for fetching the HTML pages from the various sources.

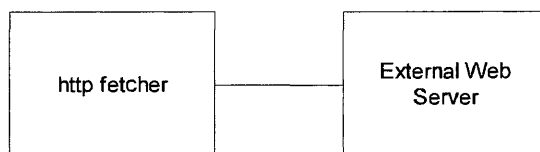


Figure 6-2: Data fetcher

3. **Data Parser:** Data parser is responsible for parsing the HTML data and extracting the query results from it. These results are then converted to an XML format.

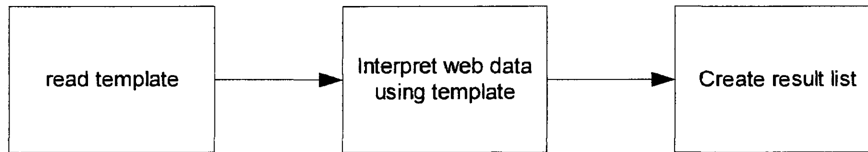


Figure 6-3: Data parser

4. **Integrator:** Figure 6-4 shows the functionality of the Integrator. It combines the data from the various sources and produces a single list of elements. The integrator uses heuristics to produce the final list of results. The results are output as an XML stream.

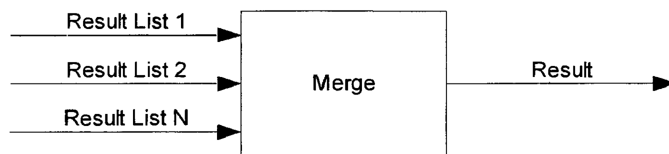


Figure 6-4: Integrator

5. **Display Results:** The component to display output takes an XML data stream of results as the input and produces an HTML page. Figure 6-5 shows how the output is produced using another XML file in Extensible Style-sheet Language (XSL) format and transforms the results to produce an HTML page.

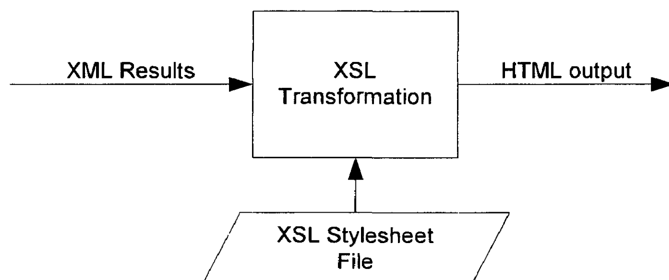


Figure 6-5: Creating HTML output

The XSL file contains information of how the XML is modified (transformed) to produce HTML data.

6.1.2.1 XML Format for Result Lists

```
<results>
<hits>NN</hits>
<engine></engine>
<resultlist>
        <item>
                <url>http://…</url>
                <title>title…</title>
                <rank>NN</rank>
                <description>xxxxxx</description>
        </ item >
        ….. (more items)
</resultlist>
</results >
```

The search data parser implements the following COM interface:

```
Interface IdataParser
{
      HRESULT SetData (BSTR inpDataStream);
      HRESULT FetchResults(BSTR *retResultStream)
      HRESULT GetParserDetails(BSTR *retParserInfo)
}
```

A new object is implemented for every search engine that is accessed. To add results from a new or different search engine, a new object has to be implemented. Each search engine interpreter needs to access search results. The COM IDL definition for representing each search result is as follows:

```
Interface IResult
{
      [propget, id(1), helpstring("property url")] HRESULT url([out, retval] BSTR *pVal);
      [propput, id(1), helpstring("property url")] HRESULT url([in] BSTR newVal);
      [propget, id(2), helpstring("property description")] HRESULT description([out, retval] BSTR
*pVal);
      [propput, id(2), helpstring("property description")] HRESULT description([in] BSTR newVal);
      [propget, id(3), helpstring("property title")] HRESULT title([out, retval] BSTR *pVal);
      [propput, id(3), helpstring("property title")] HRESULT title([in] BSTR newVal);
      [propget, id(4), helpstring("property rank")] HRESULT rank([out, retval] long *pVal);
      [propput, id(4), helpstring("property rank")] HRESULT rank([in] long newVal);
};
```

6.1.3 Future Phases: Comparison Shopping for Books

Following phase 1, the search results integrator can be expanded to fetch and return results from various online bookstores to compare prices. The overall architecture of the application remains the same. The controlling module is changed to handle the new information. New components are then implemented to parse data from the different sources.

7 Conclusion

In this thesis, I have described how advanced technologies like COM and XML can be used to make applications scalable, and given guidelines for architecting such systems. It has been necessary to follow these guidelines since the inception of the architecture design. The sample application described in Chapter 6 was built using the scalable software architecture guidelines. The application architecture is extensible and can be scaled without making changes.

In Chapter 3, I described how the common architectures could be adapted to use COM. Although the changes are not significant, they ensure that the components of the system are interoperable.

In Chapter 4, the problem of passing data between different components was addressed. An XML-based solution was suggested to pass data between components that make them extensible.

In Chapter 5, I suggested that extensible applications can be built by dynamically querying for available components and the functions they support. I also addressed the problem of building scalable Web applications. If the solution requires the use of additional hardware resources, the software should be architectured to ease this transition. Lastly, I recommended guidelines for building scalable and extensible software using COM and XML and by defining extensible interfaces.

A stepwise approach can be followed in the development of large software. The scalable architecture encourages developers to define interfaces in advance. This helps

large projects involving coordination of multiple teams. Each team will have to adhere to the interface definition and code accordingly, which will lead to fewer integration problems.

Existing software can be modified to have COM interfaces. The complexity of this process, however, depends on how the application is built. It is easier to add COM interfaces to software written in a modern language (C++, Java, and Basic, for example) as opposed to software written in Pascal or Fortran. With the latter it is not feasible, as there is no compatible compiler provided that supports COM and the inherent design of the language. COM is widely used in environments using the Windows operating system (Windows 95, 98, NT, 2000), but is not well supported on the Linux/Unix and Mac operating systems.

7.1 Future Work

A common problem faced by the developer is the changing data associated with an application. With every new version of the software, a new version of the data or object model is required. Designing object models that are extensible is a related problem. The same extensibility problem can be applied to the domain of data models of relational databases.

Relational databases store information in a fixed format. Over a period of time, requirement for the stored data can change. However, old information needs to be available in the database along with the new data. Data design patterns can be designed for the extensibility of information in databases.

Bibliography

Andrews, (March, 1991), Paradigms for Process Interaction in Distributed Programs, *ACM Computing Surveys*, Vol. 23, No1, 49-89

Andersson, J.(1998), Reactive Dynamic Architectures, *Proceedings of the third international workshop on Software architecture*, 1-4

Bass, L., Clements, P., and Kazman, R. (1998), *Software Architecture in Practice*. Addison-Wesley Publishing Company

Booch, G. (1994), *Object Oriented Analysis and Design with Applications* (Second Edition), Published by The Benjamin-Cummings Publishing Company Inc.

Box, D. (January 1998), *Essential Com*, Addison-Wesley

Carnegie Mellon Software Engineering Institute (www.sei.cmu.edu), How do you define Software Architecture, Retrieved August, 2000, from the World Wide Web: http://www.sei.cmu.edu/architecture/definitions.html

Dellarocas, C., Klein, M. and Shrobe, H. (1998), An architecture for constructing self-evolving software systems, *Proceedings of the third international workshop on Software architecture*, 29-32

Garlan, David (June 1995), Research Directions in Software Architecture, *ACM Computer Surveys* 6(27), No 2, 257-261

Garlan, D. (2000), Software architecture: a roadmap, *The Future of Software Engineering*, ACM Press, 91-101

Gacek, C. (July, 1997), Detecting Architectural Mismatches During Systems Composition, Technical Report, University of Southern California, Center for Software Engineering, USC/CSE-97-TR-506

Garlan, D., Allen, R., Ockerbloom, J. (1995), Architectural Mismatch: Why Reuse Is So Hard, *IEEE Software*, 17-26

Garlan, D., Kaiser, G., Notkin, D.(1992), Using Tool Abstraction, *IEEE Computer*, Vol 25, No 6, 30-38

Grefen P. and Wieringa, R.(1998), Subsystem Design Guidelines for Extensible General-Purpose Software, *Proceedings of the third international workshop on Software architecture*, 49-52

Krishnamurthi S. and Felleisen M. (November, 1998), Toward a Formal Theory of Extensible Software, *Proceedings of the ACM SIGSOFT sixth international symposium on Foundations of software engineering,* 88-98

Monroe, Kompanek, Melton, Garlan (January, 1997), Architectural Styles, Design Patterns and Objects, *IEEE Software*, Vol.14, No 1, 43-52

Natarajan, R. and Rosenblum, D. (November, 1998), Merging Component Models and Architectural Styles, *ACM Proceedings of the third international workshop on Software architecture*, 109-111

Luckham, D., Vera, James Sigurd Meldal (June, 1992), Three Concepts of System Architecture, *IEEE Computer*, Vol.25, No 6, 30-38

Ozsu, M., Valduriez, P. (1991), *Principles of Distributed Database Systems*, Prentice Hall

Parnas, D.L. (1972), On the criteria to be used in decomposing systems into modules, *Communications of the ACM* 5(12), 1053-1058.

Ribeiro-Justo, G.R., Cunha, P.R.F. (September, 1999), An architectural application framework for evolving distributed systems, *Journal of Systems Architecture* (45), 1375-1384

Robinson, S. and Krasilshchikov, A. (May, 1997). ActiveX Magic: An ActiveX Control and DCOM Sample Using ATL. Retrieved January , 1999, from the World Wide Web: http://msdn.microsoft.com/workshop/components/activex/magic.asp

Rogerson D.(1997), *Inside Com*, Microsoft Press

Rumbaugh J., Blaha, Premerlani, Eddy, Rumbaugh Jim, Lorenson (1991), *Object-Oriented Modeling and Design*, Prentice Hall

Shaw, M. (1994), Comparing Architectural Design Styles, *IEEE Software*, November, 27-41

Shaw, M. & Garlan, D. (April 1996), *Software Architecture: Perspectives of an Emerging Discipline*, Prentice Hall

Shaw, M. & Garlan, D. (1996), Formulations and Formalisms in Software Architecture, *Computer Science Today: Recent Trends and Developments*, Sprinver-Verlag, 307-323

W3C Website (1999), www.w3c.com, XML-in-10-points, W3C, Retrieved December 21, 2000, from the World Wide Web: http://www.w3c.org/XML/1999/XML-in-10-points

W3C Website (1997), www.w3c.com, Extensible Markup Language (XML), W3C, Retrieved December 21, 2000, from the World Wide Web: http://www.w3c.org/XML