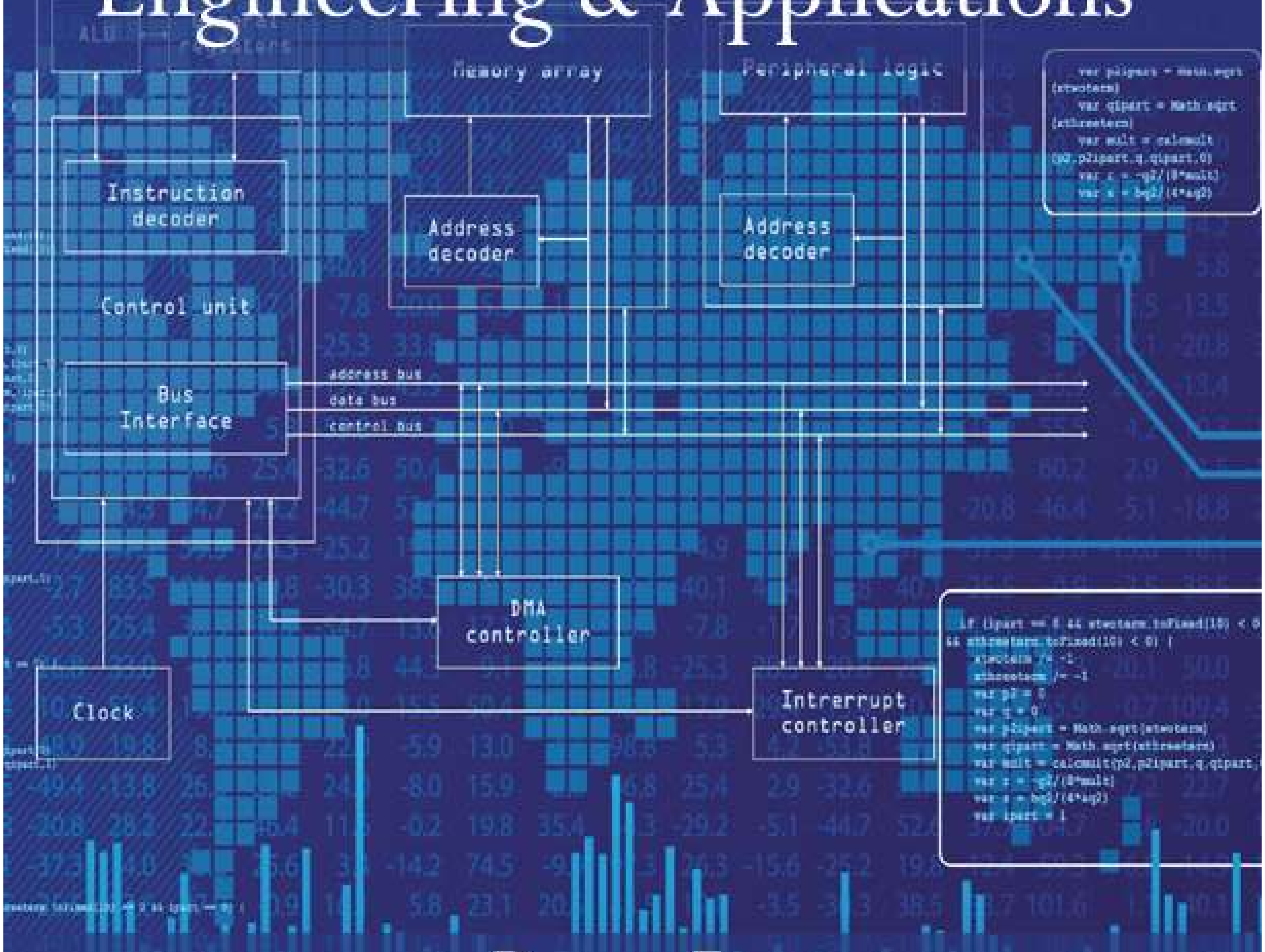
Component-Based Software Engineering & Applications



Dawson Foss

First Edition, 2012

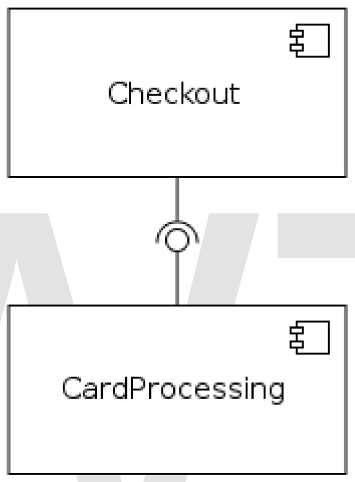
ISBN 978-81-323-1710-4

Published by:
Learning Press
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Component-Based Software Engineering



A simple example of two components expressed in UML 2.0. The checkout component, responsible for facilitating the customer's order, *requires* the card processing component to charge the customer's credit/debit card (functionality which the latter *provides*).

Component-based software engineering (CBSE) (also known as **component-based development (CBD)**) is a branch of software engineering which emphasizes the separation of concerns in respect of the wide-ranging functionality available throughout a given software system. This practice aims to bring about an equally wide-ranging degree of benefits in both the short-term and the long-term for the software itself and for organizations that sponsor such software.

Software engineers regard components as part of the starting platform for service-orientation. Components play this role, for example, in Web Services, and more recently, in Service-Oriented Architecture (SOA) - whereby a component is converted into a *service* and subsequently inherits further characteristics beyond that of an ordinary component.

Components can produce events or consume events and can be used for event driven architecture (EDA).

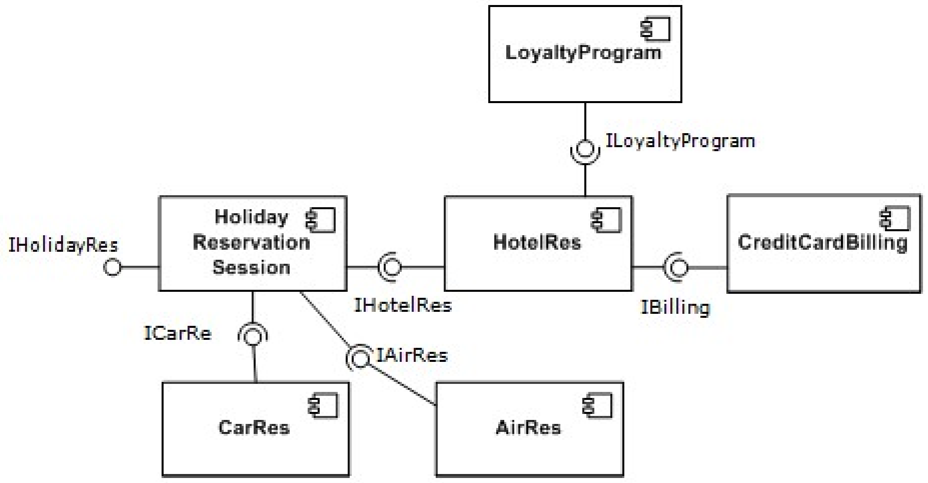
Definition and characteristics of components

An individual component is a software package, a web service, or a module that encapsulates a set of related functions (or data).

All system processes are placed into separate components so that all of the data and functions inside each component are semantically related (just as with the contents of classes). Because of this principle, it is often said that components are *modular* and *cohesive*.

With regard to system-wide co-ordination, components communicate with each other via *interfaces*. When a component offers services to the rest of the system, it adopts a *provided* interface which specifies the services that other components can utilize, and how they can do so. This interface can be seen as a signature of the component - the client does not need to know about the inner workings of the component (implementation) in order to make use of it. This principle results in components referred to as *encapsulated*.

However when a component needs to use another component in order to function, it adopts a *used* interface which specifies the services that it needs. In the UML illustrations here, *used interfaces* are represented by an open socket symbol attached to the outer edge of the component.



A simple example of several software components - pictured within a hypothetical holiday-reservation system represented in UML 2.0.

Another important attribute of components is that they are *substitutable*, so that a component can replace another (at design time or run-time), if the successor component meets the requirements of the initial component (expressed via the interfaces). Consequently, components can be replaced with either an updated version or an alternative without breaking the system in which the component operates.

As a general rule of thumb for engineers substituting components, component B can immediately replace component A if component B provides at least what component A provided, and uses no more than what component A used.

Software components often take the form of objects (not classes) or collections of objects (from object-oriented programming), in some binary or textual form, adhering to some interface description language (IDL) so that the component may exist autonomously from other components in a computer.

When a component is to be accessed or shared across execution contexts or network links, techniques such as serialization or marshalling are often employed to deliver the component to its destination.

Reusability is an important characteristic of a high-quality software component. Programmers should design and implement software components in such a way that many different programs can reuse them. Furthermore, component-based usability testing should be considered when software components directly interact with users.

It takes significant effort and awareness to write a software component that is effectively reusable. The component needs to be:

- fully documented
- thoroughly tested
    - robust - with comprehensive input-validity checking
    - able to pass back appropriate error messages or return codes
- designed with an awareness that it *will* be put to unforeseen uses

In the 1960s, programmers built scientific subroutine libraries that were reusable in a broad array of engineering and scientific applications. Though these subroutine libraries reused well-defined algorithms in an effective manner, they had a limited domain of application. Commercial sites routinely created application programs from reusable modules written in Assembler, COBOL, PL/1 and other second- and third-generation languages using both System and user application libraries.

As of 2010, modern reusable components encapsulate both data structures and the algorithms that are applied to the data structures. It builds on prior theories of software objects, software architectures, software frameworks and software design patterns, and the extensive theory of object-oriented programming and the object oriented design of all these. It claims that software components, like the idea of hardware components, used for example in telecommunications, can ultimately be made interchangeable and reliable. On the other hand, it is argued that it is a mistake to focus on independent components rather than the framework (without which they would not exist).

History

The idea that software should be componentized - built from prefabricated *components* - first became prominent with Douglas McIlroy's address at the NATO conference on software engineering in Garmisch, Germany, 1968, titled *Mass Produced Software Components*. The conference set out to counter the so-called software crisis. McIlroy's subsequent inclusion of pipes and filters into the Unix operating system was the first implementation of an infrastructure for this idea.

Brad Cox of Stepstone largely defined the modern concept of a software component. He called them *Software ICs* and set out to create an infrastructure and market for these components by inventing the Objective-C programming language. (He summarizes this view in his book *Object-Oriented Programming - An Evolutionary Approach* 1986.)

IBM led the path with their System Object Model (SOM) in the early 1990s. Some claim that Microsoft paved the way for actual deployment of component software with OLE and COM. As of 2010 many successful software component models exist.

Differences from object-oriented programming

Proponents of object-oriented programming (OOP) maintain that software should be written according to a mental model of the actual or imagined objects it represents. OOP and the related disciplines of object-oriented design and object-oriented analysis focus on modeling real-world interactions and attempting to create "verbs" and "nouns" which can be used in more human-readable ways, ideally by end users as well as by programmers coding for those end users.

Component-based software engineering, by contrast, makes no such assumptions, and instead states that developers should construct software by gluing together prefabricated components - much like in the fields of electronics or mechanics. Some peers will even talk of modularizing systems as software components as a new programming paradigm.

Some argue that earlier computer scientists made this distinction, with Donald Knuth's theory of "literate programming" optimistically assuming there was convergence between intuitive and formal models, and Edsger Dijkstra's theory in *The Cruelty of Really Teaching Computer Science*, which stated that programming was simply, and only, a branch of mathematics.

In both forms, this notion has led to many academic debates about the pros and cons of the two approaches and possible strategies for uniting the two. Some consider the different strategies not as competitors, but as descriptions of the same problem from different points of view.

Architecture

A computer running several software components is often called an application server. Using this combination of application servers and software components is usually called distributed computing. The usual real-world application of this is in e.g. financial applications or business software.

Technologies

- Pipes and Filters
    - Unix operating system
- Component-oriented programming
    - SOFA component system from ObjectWeb
    - Fractal component model from ObjectWeb
    - rCOS method of component-based model driven design from UNU-IIST
    - Visual Basic Extensions, OCX/ActiveX/COM and DCOM from Microsoft
    - XPCOM from Mozilla Foundation
    - VCL and CLX from Borland and similar free LCL library.
    - Enterprise JavaBeans from Sun Microsystems (now Oracle)
    - UNO from the OpenOffice.org office suite
    - Eiffel programming language
    - Oberon, Component Pascal, and BlackBox Component Builder
    - Bundles as defined by the OSGi Service Platform
    - The `System.ComponentModel` namespace in Microsoft .NET
    - Flow-based programming
    - MidCOM component framework for Midgard and PHP
    - Common Component Architecture (CCA) - Common Component Architecture Forum , Scientific/HPC Component Software
        - TASCS - SciDAC Center for Technology for Advanced Scientific Component Software
- Component-based software frameworks for specific domains
    - Earth System Modeling Framework (ESMF)
- Compound document technologies
    - Active Documents in Oberon System and BlackBox Component Builder
    - Bonobo (component model), a part of GNOME
    - KPart, the KDE Compound document technology
    - Object linking and embedding (OLE)
    - OpenDoc
    - Fresco
- Business object technologies
    - Newi
- Distributed computing software components
    - 9P distributed protocol developed for Plan 9, and used by Inferno and other systems.
    - CORBA and the CORBA Component Model from the Object Management Group

    - D-BUS from the freedesktop.org organization
    - DCOM and later versions of COM (and COM+) from Microsoft
    - DCOP from KDE
    - DSOM and SOM from IBM (now scrapped)
    - ICE from ZeroC
    - Java EE from Sun
    - .NET Remoting from Microsoft
    - Web Services
        - REST
    - Universal Network Objects (UNO) from OpenOffice.org
    - Zope from Zope Corporation
- Interface description languages
    - XML-RPC, the predecessor of SOAP
    - SOAP IDL from W3C
    - WDDX
    - Part of both COM and CORBA
    - Open Service Interface Definitions
    - Platform-Independent Component Modeling Language
    - SIDL - Scientific Interface Definition Language
        - Part of the Babel Scientific Programming Language Interoperability System
        - (SIDL and Babel are core technologies of the CCA and the SciDAC TASCS Center - see above.)
- Generic programming emphasizes separation of algorithms from data representation
- Inversion of Control (IoC) and Plain Old C++/Java Object (POCO/POJO) component frameworks

Chapter 2

Flow-based Programming

In computer science, **flow-based programming** (**FBP**) is a programming paradigm that defines applications as networks of "black box" processes, which exchange data across predefined connections by message passing, where the connections are specified *externally* to the processes. These black box processes can be reconnected endlessly to form different applications without having to be changed internally. FBP is thus naturally component-oriented.

FBP is a particular form of dataflow programming based on bounded buffers, information packets with defined lifetimes, named ports, and separate definition of connections.

Introduction

The FBP development approach views an application not as a single, sequential, process, which starts at a point in time, and then does one thing at a time until it is finished, but as a network of asynchronous processes communicating by means of streams of structured data chunks, called "information packets" (IPs). In this view, the focus is on the application data and the transformations applied to it to produce the desired outputs. The network is defined externally to the processes, as a list of connections which is interpreted by a piece of software, usually called the "scheduler".

The processes communicate by means of fixed-capacity connections. A connection is attached to a process by means of a port, which has a name agreed upon between the process code and the network definition. More than one process can execute the same piece of code. At any point in time, a given IP can only be "owned" by a single process, or be in transit between two processes. Ports may either be simple, or array-type, as used e.g. for the input port of the Collate component described below. It is the combination of ports with asynchronous processes that allows many long-running primitive functions of data processing, such as Sort, Merge, Summarize, etc., to be supported in the form of software black boxes.

Because FBP processes can continue executing as long they have data to work on and somewhere to put their output, FBP applications generally run in less elapsed time than conventional programs, and make optimal use of all the processors on a machine, with no special programming required to achieve this.

The network definition is usually diagrammatic, and is converted into a connection list in some lower-level language or notation. FBP is thus a visual programming language at this level. More complex network definitions have a hierarchical structure, being built up from subnets with "sticky" connections.

FBP has much in common with the Linda language in that it is, in Gelernter and Carriero's terminology, a "coordination language": it is essentially language-independent. Indeed, given a scheduler written in a sufficiently low-level language, components written in different languages can be linked together in a single network. FBP thus lends itself to the concept of domain-specific languages or "mini-languages".

FBP exhibits "data coupling", described in coupling as the loosest type of coupling between components. The concept of loose coupling is in turn related to that of Service-oriented architectures, and FBP fits a number of the criteria for such an architecture, albeit at a more fine-grained level than most examples of this architecture.

FBP promotes high-level, functional style of specifications that simplify reasoning about system behavior. An example of this is the distributed data flow model for constructively specifying and analyzing the semantics of distributed multi-party protocols.

History

FBP was invented by J. Paul Morrison in the early 1970s, and an early implementation of this technology has been in continuous production use at a major Canadian bank since that time.

FBP at its inception was strongly influenced by some IBM simulation languages of the period, in particular GPSS, but its roots go all the way back to Conway's seminal paper on what he called coroutines.

FBP has undergone a number of name changes over the years: the original implementation was called AMPS (Advanced Modular Processing System), which (as of 2008) has been in continuous production use since the early 1970s at a major Canadian bank. A number of the basic concepts were put into the public domain by IBM, by means of a Technical Disclosure Bulletin in 1971, using a very general title. An article describing its concepts and experience using it was published in 1978 in the IBM Research IBM Systems Journal under the name DSLM. A second implementation was done as a joint project of IBM Canada and IBM Japan, under the name "Data Flow Development Manager" (DFDM), and was briefly marketed in Japan in the late '80s under the name "Data Flow Programming Manager".

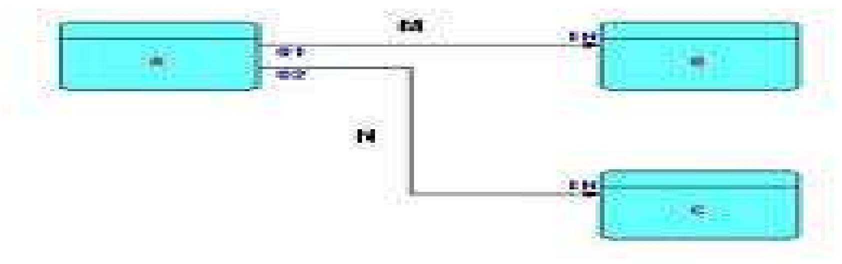
Generally the concepts were referred to within IBM as "Data Flow", but this term was felt to be too general, and eventually the name Flow-Based Programming was adopted, and a book with that title was published in 1994. The 2nd edition is now available, published by CreateSpace, a DBA of On-Demand Publishing LLC, part of the Amazon.com group of companies.

The late IBM architect, Wayne Stevens, wrote several articles describing and supporting the FBP concept, and included material about it in several of his books.

As of 2009 several companies were marketing tools based on FBP concepts, among them: Trelliswerk LLC, Proto Software, Inc., InforSense, Accelrys, and open-source Kettle and Knime. IBM also sells a tool for general data transformation called DataStage which combines FBP with parallel processing.

Concepts

The following diagram shows the major entities of an FBP diagram (apart from the Information Packets). Such a diagram can be converted directly into a list of connections, which can then be executed by an appropriate engine (software or hardware).



Simple FBP diagram

A, B and C are processes executing code components. O1, O2, and the two INs are ports connecting the connections M and N to their respective processes. It is permitted for processes B and C to be executing the same code, so each process must have its own set of working storage, control blocks, etc. Whether or not they do share code, B and C are free to use the same port names, as port names only have meaning within the components referencing them (and at the network level, of course).

M and N are what are often referred to as "bounded buffers", and have a fixed capacity in terms of the number of IPs that they can hold at any point in time.

The concept of *ports* is what allows the same component to be used at more than one place in the network. In combination with a parametrization capability, called Initial Information Packets (IIPs), ports provide FBP with a component reuse capability, making FBP a component-based architecture. FBP thus exhibits what Nate Edwards of IBM Research has termed configurable modularity.

Information Packets or IPs are allocated in what might be called "IP space" (just as Linda's tuples are allocated in "tuple space"), and have a well-defined lifetime until they are disposed of and their space is reclaimed - in FBP this must be an explicit action on the part of an owning process. IPs traveling across a given connection (actually it is their "handles" that travel) constitute a "stream", which is generated and consumed asynchronously - this concept thus has similarities to the lazy cons concept described in the 1976 article by Friedman and Wise.

IPs are usually structured chunks of data - some IPs, however, may not contain any real data, but are used simply as signals. An example of this is "bracket IPs", which can be used to group data IPs into sequential patterns within a stream, called "substreams". Substreams may in turn be nested. IPs may also be chained together to form "IP trees", which travel through the network as single objects.

The system of connections and processes described above can be "ramified" to any size. During the development of an application, monitoring processes may be added between pairs of processes, processes may be "exploded" to subnets, or simulations of processes may be replaced by the real process logic. FBP therefore lends itself to rapid prototyping.

This is really an assembly line image of data processing: the IPs travelling through a network of processes may be thought of as widgets travelling from station to station in an assembly line. "Machines" may easily be reconnected, taken off line for repair, replaced, and so on. Oddly enough, this image is very similar to that of unit record equipment that was used to process data before the days of computers, except that decks of cards had to be hand-carried from one machine to another.

Implementations of FBP may be non-preemptive or preemptive - the earlier implementations tended to be non-preemptive (mainframe and C language), whereas the latest Java implementation (see below) uses Java Thread class and is preemptive.

Implementations

After Paul Morrison retired from IBM, these concepts were implemented first in C++, using green threads (this version is currently undergoing conversion to fibers), then in Java, starting from a base developed by John Cowan - this implementation is available as Open Source on SourceForge. A C# implementation is also available at the same site. Both of these implementations use threads, so they make optimum use of all the processors on the machine running the application.

A diagramming tool, called "DrawFBP", is also available at that site - it can also be run via JWS.

Examples

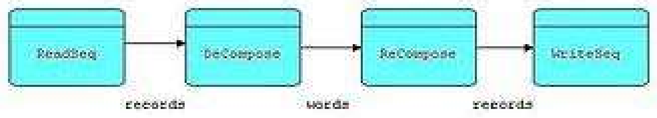
"Telegram Problem"

FBP components often form complementary pairs. This example uses two such pairs. The problem described seems very simple as described in words, but in fact is surprisingly hard to do using conventional procedural logic. The task, called the "Telegram Problem", originally described by Peter Naur, is to write a program which accepts lines of text and generates output lines of a different length, without splitting any of the words in the text (we assume no word is longer than the size of the output lines).

In conventional logic, the programmer rapidly discovers that neither the input nor the output structures can be used to drive the call hierarchy of control flow. In FBP, on the other hand, the problem description itself suggests a solution:

- "words" are mentioned explicitly in the description of the problem, so it is reasonable for the designer to treat words as information packets (IPs)
- in FBP there is no single call hierarchy, so the programmer is not tempted to force a subpattern of the solution to be the top level.

Here is the most natural solution in FBP (there is no single "correct" solution in FBP, but this seems like a natural fit):

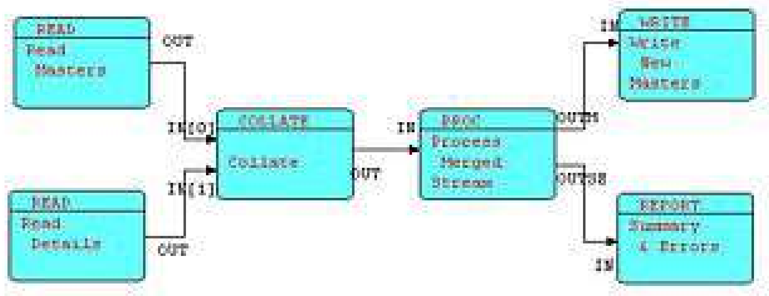


"Telegram problem"

As mentioned above, Initial Information Packets (IIPs) can be used to specify parametric information such as the desired output record length (required by the rightmost two components), or file names. IIPs are data chunks associated with a port in the network definition which become "normal" IPs when a "receive" is issued for the relevant port.

Batch update

This type of program involves passing a file of "details" (changes, adds and deletes) against a "master file", and producing (at least) an updated master file, and one or more reports. Update programs are generally quite hard to code using synchronous, procedural code, as two (sometimes more) input streams have to be kept synchronized, even though there may be masters without corresponding details, or vice versa.



Update

In FBP, a reusable component (Collate), based on the unit record idea of a Collator, makes writing this type of application much easier as Collate merges the two streams and inserts bracket IPs to indicate grouping levels, significantly simplifying the downstream logic. Suppose that one stream ("masters" in this case) consists of IPs with key values of 1, 2 and 3, and the second stream IPs ("details") have key values of 11, 12, 21, 31, 32, 33 and 41, where the first digit corresponds to the master key values. Using bracket characters to represent "bracket" IPs, the collated output stream will be as follows:

```
( m1 d11 d12 ) ( m2 d21 ) ( m3 d31 d32 d33 ) (d41)
```

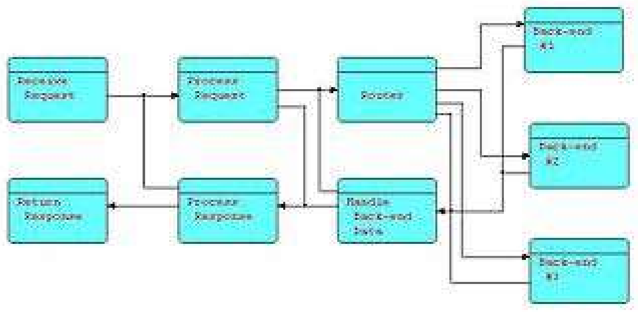
As there was no master with a value of 4, the last group consists of a single detail (plus brackets).

The structure of the above stream can be described succinctly using a BNF-like notation such as

```
{ ( [m] d* ) }*
```

Collate is a reusable black box which only needs to know where the control fields are in its incoming IPs (even this is not strictly necessary as transformer processes can be inserted upstream to place the control fields in standard locations), and can in fact be generalized to any number of input streams, and any depth of bracket nesting. Collate uses an array-type port for input, allowing a variable number of input streams.

Simple interactive network



Schematic of general interactive application

In this general schematic, requests (transactions) coming from users enter the diagram at the upper left, and responses are returned at the lower left. The "back ends" (on the right side) communicate with systems at other sites, e.g. using CORBA, MQSeries, etc. The cross-connections represent requests that do not need to go to the back ends, or requests that have to cycle through the network more than once before being returned to the user.

As different requests may use different back-ends, and may require differing amounts of time for the back-ends (if used) to process them, provision must be made to relate returned data to the appropriate requesting transactions, e.g. hash tables or caches.

The above diagram is schematic in the sense that the final application may contain many more processes: processes may be inserted between other processes to manage caches, display connection traffic, monitor throughput, etc. Also the blocks in the diagram may represent "subnets" - small networks with one or more open connections.

Comparison with other paradigms and methodologies

Jackson Structured Programming (JSP)

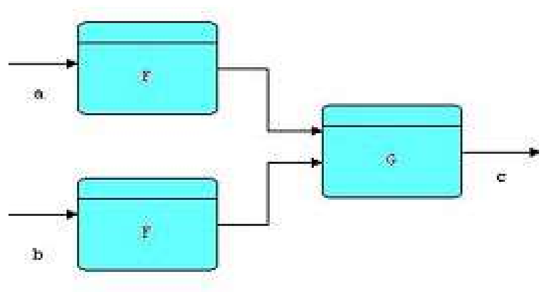
This methodology assumes that a program must be structured as a single procedural hierarchy of subroutines. Its starting point is to describe the application as a set of "main lines", based on the input and output data structures. One of these "main lines" is then chosen to drive the whole program, and the others are required to be "inverted" to turn them into subroutines (hence the name "Jackson inversion"). This sometimes results in what is called a "clash", requiring the program to be split into multiple programs or coroutines. When using FBP, this inversion process is not required, as every FBP component can be considered a separate "main line".

FBP and JSP share the concept of treating a program (or some components) as a parser of an input stream. The FBP book contains a discussion of how the concept of push-down automata may be used to design components (Chapter 23). It describes how a stack of controlling IPs may be used to control nested substreams in an FBP data stream.

Applicative programming

W.B. Ackerman defines an applicative language as one which does all of its processing by means of operators applied to values. The earliest known applicative language was LISP.

An FBP component can be regarded as a function transforming its input stream(s) into its output stream(s). These functions are then combined to make more complex transformations, as shown here:



Two functions feeding one

If we label streams, as shown, with lower case letters, then the above diagram can be represented succinctly as follows:

```
c = G(F(a),F(b));
```

Just as in functional notation F can be used twice because it only works with values, and therefore has no side effects, in FBP two instances of a given component may be running concurrently with each other, and therefore FBP components must not have side-effects either. Functional notation could clearly be used to represent at least a part of an FBP network.

The question then arises whether FBP components can themselves be expressed using functional notation. W.H. Burge showed how stream expressions can be developed using a recursive, applicative style of programming, but this work was in terms of (streams of) atomic values. In FBP, it is necessary to be able to describe and process structured data chunks (FBP IPs). In the FBP book, a notation is added for accessing the fields of an IP, and an operator, called the "mini-constructor" (μ), based on a similar function in the Vienna Definition Language, for creating an IP from a set of (perhaps modified) field values and identifiers.

Furthermore, most applicative systems assume that all the data is available in memory at the same time, whereas FBP applications need to be able to process long-running streams of data while still using finite resources. Friedman and Wise suggested a way to do this by adding the concept of "lazy cons" to Burge's work. This removed the requirement that both of the arguments of "cons" be available at the same instant of time. "Lazy cons" does not actually build a stream until both of its arguments are realized - before that it simply records a "promise" to do this. This allows a stream to be dynamically realized from the front, but with an unrealized back end. The end of the stream stays unrealized until the very end of the process, while the beginning is an ever-lengthening sequence of items.

In the FBP book (Chapter 24), these ideas are combined to allow the expression of some quite complex component logic using applicative notation.

Linda

Many of the concepts in FBP seem to have been discovered independently in different systems over the years. Linda, mentioned above, is one such. Chapter 26 of the FBP book goes into some detail about similarities and differences, but probably the major difference is that, in Linda, data is accessed associatively, whereas in FBP, IPs arriving at a particular input port are retrieved sequentially. FBP's IPs are very similar to Linda's tuples. The difference between the two techniques is illustrated by the Linda "school of piranhas" load balancing technique - in FBP, this requires an extra "load balancer" component which routes requests to the component in a list which has the smallest number of IPs waiting to be processed. Clearly FBP and Linda are closely related, and one could easily be used to simulate the other.

Object-oriented programming

An object in OOP can be described as a semi-autonomous unit comprising both information and behaviour. Objects communicate by means of "method calls", which are essentially subroutine calls, done indirectly via the class to which the receiving object belongs. The object's internal data can only be accessed by means of method calls, so this is a form of information hiding or "encapsulation". Encapsulation, however, predates OOP - David Parnas wrote one of the seminal articles on it in the early 70s - and is a basic concept in computing. Encapsulation is the very essence of an FBP component, which may be thought of as a black box, performing some conversion of its input data into its output data. In FBP, part of the specification of a component is the data formats and stream structures that it can accept, and those it will generate. This constitutes a form of design by contract. In addition, the data in an IP can only be accessed directly by the currently owning process. Encapsulation can also be implemented at the network level, by having outer processes protect inner ones.

A paper by C. Ellis and S. Gibbs distinguishes between active objects and passive objects. Passive objects comprise information and behaviour, as stated above, but they cannot determine the *timing* of this behaviour. Active objects on the other hand can do this. In their article Ellis and Gibbs state that active objects have much more potential for the development of maintainable systems than do passive objects. An FBP application can be viewed as a combination of these two types of object, where FBP processes would correspond to active objects, while IPs would correspond to passive objects.

Chapter 25 of the FBP book goes into more detail on the relationship between FBP and OOP.

Chapter 3

Pipeline (Software)

In software engineering, a **pipeline** consists of a chain of processing elements (processes, threads, coroutines, *etc.*.), arranged so that the output of each element is the input of the next. Usually some amount of buffering is provided between consecutive elements. The information that flows in these pipelines is often a stream of records, bytes or bits.

The concept is also called the **pipes and filters design pattern**. It was named by analogy to a physical pipeline.

Multiprocessed pipelines

Pipelines are often implemented in a multitasking OS, by launching all elements at the same time as processes, and automatically servicing the data read requests by each process with the data written by the upstream process. In this way, the CPU will be naturally switched among the processes by the scheduler so as to minimize its idle time. In other common models, elements are implemented as lightweight threads or as coroutines to reduce the OS overhead often involved with processes. Depending upon the OS, threads may be scheduled directly by the OS or by a thread manager. Coroutines are always scheduled by a coroutine manager of some form.

Usually, read and write requests are blocking operations, which means that the execution of the source process, upon writing, is suspended until all data could be written to the destination process, and, likewise, the execution of the destination process, upon reading, is suspended until at least some of the requested data could be obtained from the source process. Obviously, this cannot lead to a deadlock, where both processes would wait indefinitely for each other to respond, since at least one of the two processes will soon thereafter have its request serviced by the operating system, and continue to run.

For performance, most operating systems implementing pipes use pipe buffers, which allow the source process to provide more data than the destination process is currently able or willing to receive. Under most Unices and Unix-like operating systems, a special command is also available which implements a pipe buffer of potentially much larger and configurable size, typically called "buffer". This command can be useful if the destination process is significantly slower than the source process, but it is anyway desired that the source process can complete its task as soon as possible. E.g., if the source process consists of a command which reads an audio track from a CD and the destination process consists of a command which compresses the waveform audio data to a format like MP3. In this case, buffering the entire track in a pipe buffer would allow the CD drive to spin down more quickly, and enable the user to remove the CD from the drive before the encoding process has finished.

Such a buffer command can be implemented using available operating system primitives for reading and writing data. Wasteful busy waiting can be avoided by using facilities such as poll or select or multithreading.

VM/CMS and MVS

CMS Pipelines is a port of the pipeline idea to VM/CMS and MVS systems. It supports much more complex pipeline structures than Unix shells, with steps taking multiple input streams and producing multiple output streams. (Such functionality is supported by the Unix kernel, but few programs use it as it makes for complicated syntax and blocking modes, although some shells do support it via arbitrary file descriptor assignment). Due to the different nature of IBM mainframe operating systems, it implements many steps inside CMS Pipelines which in Unix are separate external programs, but can also call separate external programs for their functionality. Also, due to the record-oriented nature of files on IBM mainframes, pipelines operate in a record-oriented, rather than stream-oriented manner.

Pseudo-pipelines

On single-tasking operating systems, the processes of a pipeline have to be executed one by one in sequential order; thus the output of each process must be saved to a temporary file, which is then read by the next process. Since there is no parallelism or CPU switching, this version is called a "pseudo-pipeline".

For example, the command line interpreter of MS-DOS ('COMMAND.COM') provides pseudo-pipelines with a syntax superficially similar to that of Unix pipelines. The command "dir | sort | more" would have been executed like this (albeit with more complicated temporary file names):

1. Create temporary file 1.tmp
2. Run command "dir", redirecting its output to 1.tmp
3. Create temporary file 2.tmp
4. Run command "sort", redirecting its input to 1.tmp and its output to 2.tmp
5. Run command "more", redirecting its input to 2.tmp, and presenting its output to the user
6. Delete 1.tmp and 2.tmp, which are no longer needed
7. Return to the command prompt

All temporary files are stored in the directory pointed to by %TEMP%, or the current directory if %TEMP% isn't set.

Thus, pseudo-pipes acted like true pipes with a pipe buffer of unlimited size (disk space limitations notwithstanding), with the significant restriction that a receiving process could not read *any* data from the pipe buffer until the sending process finished completely. Besides causing disk traffic, if one doesn't install a harddisk cache such as SMARTDRV, that would have been unnecessary under multi-tasking operating systems, this implementation also made pipes unsuitable for applications requiring real-time response, like, for example, interactive purposes (where the user enters commands that the first process in the pipeline receives via stdin, and the last process in the pipeline presents its output to the user via stdout).

Also, commands that produce a potentially infinite amount of output, such as the yes command, cannot be used in a pseudo-pipeline, since they would run until the temporary disk space is exhausted, so the following processes in the pipeline could not even start to run.

Object pipelines

Beside byte stream-based pipelines, there are also object pipelines. In an object pipeline, the processes output objects instead of texts; therefore removing the string parsing tasks that are common in UNIX shell scripts. Windows PowerShell uses this scheme and transfers .NET objects. Channels, found in the Limbo programming language, and the IPython ipipe extension are other examples of this metaphor.

Pipelines in GUIs

Graphical environments such as RISC OS and ROX Desktop also make use of pipelines. Rather than providing a save dialog box containing a file manager to let the user specify where a program should write data, RISC OS and ROX provide a save dialog box containing an icon (and a field to specify the name). The destination is specified by dragging and dropping the icon. The user can drop the icon anywhere an already-saved file could be dropped, including onto icons of other programs. If the icon is dropped onto a program's icon, it's loaded and the contents that would otherwise have been saved are passed in on the new program's standard input stream.

For instance, a user browsing the world-wide web might come across a .gz compressed image which they want to edit and re-upload. Using GUI pipelines, they could drag the link to their de-archiving program, drag the icon representing the extracted contents to their image editor, edit it, open the save as dialog, and drag its icon to their uploading software.

Conceptually, this method could be used with a conventional save dialog box, but this would require the user's programs to have an obvious and easily-accessible location in the filesystem that can be navigated to. In practice, this is often not the case, so GUI pipelines are rare.

Other considerations

The name 'pipeline' comes from a rough analogy with physical plumbing in that a pipeline usually allows information to flow in only one direction, like water often flows in a pipe.

Pipes and filters can be viewed as a form of functional programming, using byte streams as data objects; more specifically, they can be seen as a particular form of monad for I/O.

The concept of pipeline is also central to the Cocoon web development framework or to any XProc (the W3C Standards) implementations, where it allows a source stream to be modified before eventual display.

This pattern encourages the use of text streams as the input and output of programs. This reliance on text has to be accounted when creating graphic shells to text programs.

History

Process pipelines were invented by Douglas McIlroy, one of the designers of the first Unix shells, and greatly contributed to the popularity of that operating system. It can be considered the first non-trivial instance of software componentry.

The idea was eventually ported to other operating systems, such as DOS, OS/2, Windows NT, BeOS, AmigaOS, MorphOS and Mac OS X (the latter being a UNIX OS).

Chapter 4

DirectSound

DirectSound is a software component of the DirectX library, supplied by Microsoft, that resides on a computer with the Windows operating system. It provides a direct interface between applications and the sound card drivers on Windows XP and earlier operating systems, enabling applications to produce sounds and music. Besides providing the essential service of passing audio data to the sound card, it provides many needed capabilities such as recording and mixing sound; adding effects to sound e.g. reverb, echo, flange; using hardware controlled buffers for extra speed; positioning sounds in 3D space (3D audio spatialization), capturing sounds from a microphone or other input and controlling capture effects during audio capture.

DirectSound also allows several applications to conveniently share access to the sound card at the same time. Its ability to play sound in 3D added a new dimension to games. It also provides the ability for games to modify a musical script in response to game events in real time, e.g. the beat of the music could quicken as the action heats up.

After many years of development, today DirectSound is a very mature API, and supplies many other useful capabilities, such as the ability to play multichannel sounds at high resolution. While DirectSound was designed to be used by games, a number of professional audio applications now take advantage of its diverse capabilities.

DirectSound3D

DirectSound3D (DS3D) is an addition to Microsoft's DirectX system which is intended to standardize 3D audio under Microsoft Windows, introduced with DirectX 3 in 1996.

DirectSound3D allows software developers to write to a single standardized audio API instead of writing code for each audio card manufacturer.

In DirectX 5, DirectSound3D has the capability of having sound cards that use third party 3D audio algorithms accelerate DirectSound3D properly, through Microsoft-approved methods. This eliminates the need for separate 3D audio libraries.

Starting with DirectX 8 onwards, DirectSound and DirectSound3D (DS3D) are together referred to as **DirectX Audio**.

Windows Vista

Windows Vista features a completely re-written audio stack based on the *Universal Audio Architecture*. Because of the architectural changes in the redesigned audio stack, a direct path from DirectSound to the audio drivers does not exist. DirectSound and other APIs such as MME are emulated as WASAPI Session instances. DirectSound runs in emulation mode on the Microsoft software mixer. The emulator does not have hardware abstraction, so there is no hardware DirectSound acceleration, meaning hardware and software relying on DirectSound acceleration may have degraded performance. It's likely a supposed performance hit might not be noticeable, depending on the application and actual system hardware. In the case of hardware 3D audio effects played using DirectSound3D, they will not be playable.

Third-party APIs such as ASIO and OpenAL are not affected by these architectural changes in Windows Vista. A solution for applications that wish to take advantage of hardware accelerated high-quality 3D positional audio is to use OpenAL. However, this only works if the manufacturer provides an OpenAL driver for their hardware.

As of 2007, a solution to re-enable hardware acceleration of DirectSound3D and Audio Effects, such as EAX, called *Creative ALchemy* was launched. Creative ALchemy intercepts calls to DirectSound3D and translates them into OpenAL calls to be processed by supported hardware such as Sound Blaster X-Fi and Sound Blaster Audigy. For software-based Creative audio solutions, ALchemy utilizes its built-in 3D audio engine without using OpenAL at all.

Realtek, a manufacturer of integrated HD audio codecs, has a product similar to ALchemy called 3D SoundBack. C-Media, a manufacturer of PC sound card chipsets, also has a solution called Xear3D EX, although it works instead by intercepting DirectSound3D calls transparently in the background without any user intervention.

XAudio 2

Because of Xbox 360 and Microsoft Windows integration, Microsoft is actively pushing developers to migrate new applications to equivalent Xbox audio APIs such as XAudio and the Cross-platform Audio Creation Tool (XACT). XAudio is an Xbox-only API designed for digital signal processing, however, XAudio 2 is a cross-platform (Windows and Xbox) common low-level audio API, intended as the replacement for DirectSound. The RTM version of the XAudio 2 library is included in the March 2008 DirectX SDK. The target platforms include Windows XP, Windows Vista and the Xbox 360. XAudio 2 provides low-level mixing and signal processing whereas high-level audio authoring and playback are available using XACT and 3D functions via the X3DAudio library. The XACT engine is a high-level audio programming library that operates through XAudio on the Xbox, as a DirectSound passthrough on Windows XP, and directly on the low-level audio renderer in the new audio stack on Windows Vista. X3DAudio is an abstracted math-driven spatialization helper library that can be replaced by custom 3D behaviors.

Xaudio 2 has special emphasis on signal processing for high-level audio APIs such as XACT. Some of its features are:

- Separation of sound data from "voice"
- Submixing (arbitrary levels and routings)
- Multi-rate processing
- Per-voice filtering (built-in, in addition to programmable DSP effects)
- Programmable voices
- Effects processing, Sample rate conversion (SRC)
- Software DSP
- Enhanced surround sound (multichannel) and explicit multichannel panning/mapping dynamically to any speaker
- Native compressed data support
    - XMA on Xbox 360
    - ADPCM and xWMA (Windows Media Audio bitstream format in a lightweight wrapper) on Windows
    - Extensible
- 3D audio handled as a separate replaceable library: XAudio 2 takes multichannel speaker volumes and X3DAudio library transforms source/listener coordinates into speaker volumes and other synthesis parameters

Windows CE

Although DirectSound support was available in Windows CE versions up to 4.2, it was removed starting 5.0 . Windows CE 6.0 also does not support DirectSound, instead favoring that applications be rewritten to use the Waveform Audio API.

Chapter 5

Common Object Request Broker Architecture

The **Common Object Request Broker Architecture** (CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together (i.e., it supports multiple platforms).

Overview

CORBA is useful because it enables separate pieces of software written in different languages and running on different computers to work with each other like a single application or set of services. More specifically, CORBA is a mechanism in software for normalizing the method-call semantics between application objects residing either in the same address space (application) or remote address space (same host, or remote host on a network). Version 1.0 was released in October 1991. CORBA uses an interface definition language (IDL) to specify the interfaces which objects present to the outer world. CORBA then specifies a *mapping* from IDL to a specific implementation language like C++ or Java. Standard mappings exist for Ada, C, C++, Lisp, Ruby, Smalltalk, Java, COBOL, PL/I and Python. There are also non-standard mappings for Perl, Visual Basic, Erlang, and Tcl implemented by object request brokers (ORBs) written for those languages.

The CORBA specification dictates there shall be an ORB through which an application would interact with other objects. In practice, the application simply initializes the ORB, and accesses an internal *Object Adapter*, which maintains things like reference counting, object (and reference) instantiation policies, and object lifetime policies. The Object Adapter is used to register instances of the *generated code classes*. Generated code classes are the result of compiling the user IDL code, which translates the high-level interface definition into an OS- and language-specific class base for use by the user application. This step is necessary in order to enforce CORBA semantics and provide a clean user process for interfacing with the CORBA infrastructure.

Some IDL language mappings are more difficult to use than others. For example, due to the nature of Java, the IDL-Java mapping is rather straightforward and makes usage of CORBA very simple in a Java application. This is also true of the IDL to Python mapping. The C++ mapping is notoriously difficult; the mapping requires the programmer to learn complex and confusing datatypes that predate the C++ STL. Since the C language is not object-oriented, the IDL to C mapping requires a C programmer to manually emulate object oriented features.

A language mapping requires the developer to create IDL code that represents the interfaces to his objects. Typically, a CORBA implementation comes with a tool called an IDL compiler which converts the user's IDL code into some language-specific generated code. A traditional compiler then compiles the generated code to create the linkable-object files for the application. This diagram illustrates how the generated code is used within the CORBA infrastructure:

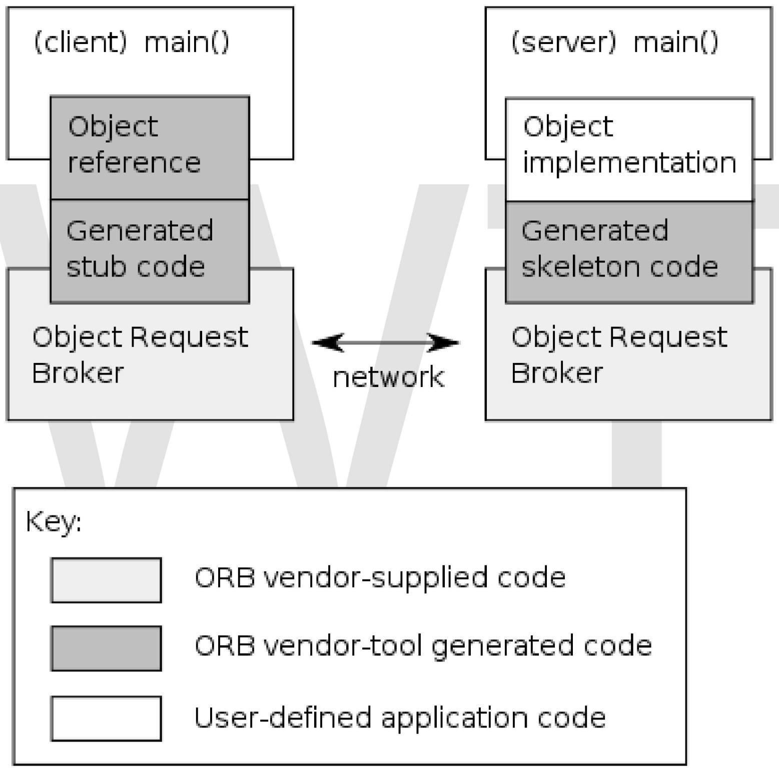


Illustration of the autogeneration of the infrastructure code from an interface defined using the CORBA IDL

This figure illustrates the high-level paradigm for remote interprocess communications using CORBA. Issues not addressed here, yet accounted-for in the CORBA specification include: data typing, exceptions, network protocol, communication timeouts, etc. For example: Normally the server side has the Portable Object Adapter (POA) that redirects calls either to the local servants or (to balance the load) to the other servers. Also, both server and client parts often have interceptors that are described below. Issues CORBA (and thus this figure) does not address, but that all distributed systems must address: object lifetimes, redundancy/fail-over, naming semantics (beyond a simple name), memory management, dynamic load balancing, separation of model between display/data/control semantics, etc.

In addition to providing users with a language and a platform-neutral remote procedure call specification, CORBA defines commonly needed services such as transactions and security, events, time, and other domain-specific interface models.

OMG trademarks

CORBA, IIOP and OMG are the registered marks of the Object Management Group and should be used with care. However, GIOP is not a registered OMG trademark. Hence in some cases it may be more appropriate just to say that the application uses or implements the GIOP-based architecture.

Objects By Reference

This reference is either acquired through a stringified URI string, NameService lookup (similar to DNS), or passed-in as a method parameter during a call.

Object references are lightweight objects matching the interface of the real object (remote or local). Method calls on the reference result in subsequent calls to the ORB and blocking on the thread while waiting for a reply, success or failure. The parameters, return data (if any), and exception data are marshaled internally by the ORB according the local language and OS mapping.

Data By Value

The CORBA Interface Definition Language provides the language- and OS-neutral inter-object communication definition. CORBA Objects are passed by reference, while data (integers, doubles, structs, enums, etc) are passed by value. The combination of Objects by reference and data-by-value provides the means to enforce strong data typing while compiling clients and servers, yet preserve the flexibility inherent in the CORBA problem-space.

Objects by Value (OBV)

Apart from remote objects, the CORBA and RMI-IIOP define the concept of the OBV and Valuetypes. The code inside the methods of Valuetype objects is executed locally by default. If the OBV has been received from the remote side, the needed code must be either *a priori* known for both sides or dynamically downloaded from the sender. To make this possible, the record, defining OBV, contains the Code Base that is a space-separated list of URLs from where this code should be downloaded. The OBV can also have the remote methods.

The OBV's may have fields that are transferred when the OBV is transferred. These fields can be OBV's themselves, forming lists, trees or arbitrary graphs. The OBV's have a class hierarchy, including multiple inheritance and abstract classes.

CORBA Component Model (CCM)

CORBA Component Model (CCM) is an addition to the family of CORBA definitions. It was introduced with CORBA 3 and it describes a standard application framework for CORBA components. Though not dependent on "language independent Enterprise Java Beans (EJB)", it is a more general form of EJB, providing four component types instead of the two that EJB defines. It provides an abstraction of entities that can provide and accept services through well-defined named interfaces called *ports*.

The CCM has a component container, where software components can be deployed. The container offers a set of services that the components can use. These services include (but are not limited to) notification, authentication, persistence and transaction processing. These are the most-used services any distributed system requires, and, by moving the implementation of these services from the software components to the component container, the complexity of the components is dramatically reduced.

Portable interceptors

Portable interceptors are the "hooks", used by CORBA and RMI-IIOP to mediate the most important functions of the CORBA system. The CORBA standard defines the following types of interceptors:

1. IOR interceptors mediate the creation of the new references to the remote objects, presented by the current server.
2. Client interceptors usually mediate the remote method calls on the client (caller) side. If the object Servant exists on the same server where the method is invoked, they also mediate the local calls.
3. Server interceptors mediate the handling of the remote method calls on the server (handler) side.

The interceptors can attach the specific information to the messages being sent and IORs being created. This information can be later read by the corresponding interceptor on the remote side. Interceptors can also throw forwarding exceptions, redirecting request to another target.

General InterORB Protocol (GIOP)

The GIOP is an abstract protocol by which Object request brokers (ORBs) communicate. Standards associated with the protocol are maintained by the Object Management Group (OMG). The GIOP architecture provides several concrete protocols, including:

1. Internet InterORB Protocol (IIOP) — The Internet Inter-Orb Protocol is an implementation of the GIOP for use over an internet, and provides a mapping between GIOP messages and the TCP/IP layer.
2. SSL InterORB Protocol (SSLIOP) — SSLIOP is IIOP over SSL, providing encryption and authentication.
3. HyperText InterORB Protocol (HTIOP) — HTIOP is IIOP over HTTP, providing transparent proxy bypassing.

VMCID (Vendor Minor Codeset ID)

Each standard CORBA exception includes a minor code to designate the subcategory of the exception. Minor exception codes are of type unsigned long and consist of a 20-bit "Vendor Minor Codeset ID" (VMCID), which occupies the high order 20 bits, and the minor code which occupies the low order 12 bits.

Minor codes for the standard exceptions are prefaced by the VMCID assigned to OMG, defined as the unsigned long constant CORBA::OMGVMCID, which has the VMCID allocated to OMG occupying the high order 20 bits. The minor exception codes associated with the standard exceptions that are found in Table 3-13 on page 3-58 are or-ed with OMGVMCID to get the minor code value that is returned in the ex_body structure.

Within a vendor assigned space, the assignment of values to minor codes is left to the vendor. Vendors may request allocation of VMCIDs by sending email to tagrequest@omg.org.

The VMCID 0 and 0xfffff are reserved for experimental use. The VMCID OMGVMCID (Section 3.17.1, "Standard Exception Definitions," on page 3-52) and 1 through 0xf are reserved for OMG use.

The Common Object Request Broker: Architecture and Specification (CORBA 2.3)

Corba Location (CorbaLoc)

Corba Location (CorbaLoc) refers to a stringified object reference for a CORBA object that looks similar to a URL.

All CORBA products must support two OMG-defined URLs: "`corbaloc:`" and "`corbaname:`". The purpose of these is to provide a human readable and editable way to specify a location where an IOR can be obtained.

An example of corbaloc is shown below:

```
corbaloc::160.45.110.41:38693/StandardNS/NameServer-POA/_root
```

A CORBA product may optionally support the "`http:`", "`ftp:`" and "`file:`" formats. The semantics of these is that they provide details of how to download a stringified IOR (or, recursively, download another URL that will eventually provide a stringified IOR). Some ORBs do deliver additional formats which are proprietary for that ORB.

Benefits

CORBA aims to bring to the table many benefits that no other single technology brings in one package. These benefits include language- and OS-independence, freedom from technology-linked implementations, strong data-typing, high level of tunability, and freedom from the details of distributed data transfers.

Language Independence

CORBA at the outset was designed to free engineers from the hang-ups and limitations of considering their designs based on a particular software language. Currently there are many languages supported by various CORBA providers, the most popular are Java and C++. There are also C-only, SmallTalk, Perl, Ada, Ruby, and Python implementations, just to mention a few.

OS Independence

CORBA's design is meant to be OS-independent. CORBA is available in Java (OS-independent), as well as natively for Linux/Unix, Windows, Sun, Mac and others.

Freedom from Technologies

One of the main implicit benefits is that CORBA provides a neutral playing field for engineers to be able to normalize the interfaces between various new and legacy systems. When integrating C, C++, Object Pascal, Java, Fortran, Python, and any other language or OS into a single cohesive system design model, CORBA provides the means to level the field and allow disparate teams to develop systems and unit tests that can later be joined together into a whole system. This does not rule out the need for basic system engineering decisions, such as threading, timing, object lifetime, etc. These issues are part of any system regardless of technology. CORBA allows system elements to be normalized into a single cohesive system model.

For example, the design of a Multitier architecture is made simple using Java Servlets in the web server and various e time, C++ legacy code can talk to C or Fortran legacy code and Java database code, and can provide data to a web interface.

Strong Data Typing

CORBA provides flexible data typing, for example an "ANY" datatype. CORBA also enforces tightly coupled datatyping, reducing human errors. In a situation where Name-Value pairs are passed around, it's conceivable that a server provides a number where a string was expected. CORBA Interface Definition Language provides the mechanism to ensure that user-code conforms to method-names, return-, parameter-types, and exceptions.

High Tune-ability

There are many implementations available (e.g. OmniORB (Open source C++ and Python implementation)) that have many options for tuning the threading and connection management features. Not all implementations provide the same features. This is up to the implementor.

Freedom From Data Transfer Details

When handling low-level connection and threading, CORBA provides a high level of detail in error conditions. This is defined in the CORBA-defined standard exception set and the implementation-specific extended exception set. Through the exceptions, the application can determine if a call failed for reasons such as "Small problem, so try again", "The server is dead" or "The reference doesn't make sense." The general rule is: Not receiving an exception means that the method call completed successfully. This is a very powerful design feature.

Compression

CORBA marshals its data in a binary form and supports compression. IONA, Remedy IT and Telefónica have worked on an extension to the CORBA standard that delivers compression. This extension is called ZIOP and this is now a formal OMG standard.

Problems and criticism

While CORBA promised to deliver much in the way code was written and software constructed, it has been the subject of much criticism.

Some of the failures were due to the implementations and the process by which CORBA was created as a standard, others reflect problems in the politics and business of implementing a software standard. These problems led to a significant decline in CORBA use and adoption in new projects and areas.

Implementation incompatibilities

The initial specifications of CORBA defined only the IDL, not the on-the-wire format. This meant that source-code compatibility was the best that was available for several years.

Location transparency

CORBA's notion of location transparency has been criticized; that is, that objects residing in the same address space and accessible with a simple function call are treated the same as objects residing elsewhere (different processes on the same machine, or different machines). This notion is flawed if one requires all local accesses to be as complicated as the most complex remote scenario. However, CORBA does not place a restriction on the complexity of the calls. Many implementations provide for recursive thread/connection semantics. I.e. Obj A calls Obj B, which in turn calls Obj A back, before returning.

Design and process deficiencies

The creation of the CORBA standard is also often cited for its process of design by committee. There was no process to arbitrate between conflicting proposals or to decide on the hierarchy of problems to tackle. Thus the standard was created by taking a union of the features in all proposals with no regard to their coherence.

This made the specification very complex, expensive to implement entirely and often ambiguous.

A design committee composed largely of vendors of the standard implementation, created a disincentive to make a comprehensive standard. This was because standards and interoperability increased competition and eased customers' movement between alternative implementations. This led to much political fighting within the committee, and frequent releases of revisions of the CORBA standard that were impossible to use without proprietary extensions.

Problems with implementations

Through its history, CORBA has been plagued by shortcomings in its implementations. Often there have been few implementations matching all of the critical elements of the specification, and existing implementations were incomplete or inadequate. As there were no requirements to provide a reference implementation, members were free to propose features which were never tested for usefulness or implementability. Implementations were further hindered by the general tendency of the standard to be verbose, and the common practice of compromising by adopting the sum of all submitted proposals, which often created APIs that were incoherent and difficult to use, even if the individual proposals were perfectly reasonable.

Working implementations of CORBA have been very difficult to acquire in the past, but are now much easier to find. The SUN Java SDK comes with CORBA already. Some poorly designed implementations have been found to be complex, slow, incompatible and incomplete. Commercial versions can be very expensive. This changed significantly as commercial-, hobbyist-, and government-funded high quality free implementations became available.

Firewalls

CORBA (more precisely, IIOP) uses raw TCP/IP connections in order to transmit data. However, if the client is behind a very restrictive firewall or transparent proxy server environment that only allows HTTP connections to the outside through port 80, communication may be impossible, unless the proxy server in question allows the HTTP CONNECT method or SOCKS connections as well. At one time, it was difficult even to force implementations to use a single standard port — they tended to pick multiple random ports instead. As of today, current ORBs do have these deficiencies. Due to such difficulties, some users have made increasing use of web services instead of CORBA. These communicate using XML/SOAP via port 80, which is normally left open or filtered through a HTTP proxy inside the organization, for web browsing via HTTP. Recent CORBA implementations, though, support SSL and can be easily configured to work on a single port. Most of the popular open source ORBS, such as TAO and JacORB also support bidirectional GIOP, which gives CORBA the advantage of being able to use callback communication rather than the polling approach characteristic of web service implementations. Also, more CORBA-friendly firewalls are now commercially available.

Chapter 6

Component Object Model

Component Object Model (**COM**) is a binary-interface standard for software componentry introduced by Microsoft in 1993. It is used to enable interprocess communication and dynamic object creation in a large range of programming languages. The term *COM* is often used in the Microsoft software development industry as an umbrella term that encompasses the OLE, OLE Automation, ActiveX, COM+ and DCOM technologies.

Overview

The essence of COM is a language-neutral way of implementing objects that can be used in environments different from the one in which they were created, even across machine boundaries. For well-authored components, COM allows reuse of objects with no knowledge of their internal implementation, as it forces component implementers to provide well-defined interfaces that are separate from the implementation. The different allocation semantics of languages are accommodated by making objects responsible for their own creation and destruction through reference-counting. Casting between different interfaces of an object is achieved through the `QueryInterface()` function. The preferred method of inheritance within COM is the creation of sub-objects to which method calls are delegated.

COM is an interface technology defined and implemented as standard only on Microsoft Windows and Apple's Core Foundation 1.3 and later plug-in API , that in any case implement only a subset of the whole COM interface. For some applications, COM has been replaced at least to some extent by the Microsoft .NET framework, and support for Web Services through the Windows Communication Foundation (WCF). However, COM objects can be used with all .NET languages through .NET COM Interop.

Networked DCOM uses binary proprietary formats, while WCF encourages the use of XML-based SOAP messaging. COM is very similar to other component software interface technologies, such as CORBA and Java Beans, although each has its own strengths and weaknesses. The characteristics of COM make it most suitable for the development and deployment of desktop applications, for which it was originally designed.

History

One of the first methods of interprocess communication in Windows was Dynamic Data Exchange (DDE), first introduced in 1987, that allowed sending and receiving messages in so-called "conversations" between applications.

Antony Williams, one of the most notable thinkers involved in the creation of the COM architecture, later distributed two internal papers in Microsoft that embraced the concept of software components: *Object Architecture: Dealing With the Unknown – or – Type Safety in a Dynamically Extensible Class Library* in 1988 and *On Inheritance: What It Means and How To Use It* in 1990. These provided the foundation of many of the ideas behind COM.

Object Linking and Embedding (OLE), Microsoft's first object-based framework, was built on top of DDE and designed specifically for compound documents. It was introduced with Word for Windows and Excel in 1991, and was later included with Windows, starting with version 3.1 in 1992.

An example of a compound document is a spreadsheet embedded in a Word for Windows document: as changes are made to the spreadsheet within Excel, they appear automatically inside the Word document.

In 1991, Microsoft introduced Visual Basic Extensions (VBX) with Visual Basic 1.0. A VBX is a packaged extension in the form of a dynamic-link library (DLL) that allowed objects to be graphically placed in a form and manipulated by properties and methods. These were later adapted for use by other languages such as Visual C++.

In 1993, when version 3.1 of Windows was released, Microsoft released OLE 2 with its underlying object model. The COM Application binary interface (ABI) was the same as the MAPI ABI, which was released in 1992. While OLE 1 was focused on compound documents, COM and OLE 2 were designed to address software components in general. Text conversations and Windows messages had proved not to be flexible enough to allow sharing application features in a robust and extensible way, so COM was created as a new foundation, and OLE changed to OLE2.

In 1994 OLE custom controls (OCXs) were introduced as the successor to VBX controls. At the same time, Microsoft stated that OLE 2 would just be known as "OLE", and that OLE was no longer an acronym, but a name for all of the company's component technologies.

In early 1996, Microsoft found a new use for OLE Custom Controls, expanding their Web browser's capability to present content, renamed some parts of OLE relating to the Internet **ActiveX**, and gradually renamed all OLE technologies to ActiveX, except the compound document technology that was used in Microsoft Office. Later that year, DCOM was introduced as an answer to CORBA.

Related technologies

COM was the major software development platform for Windows and, as such, influenced development of a number of supporting technologies.

COM+

In order for Microsoft to provide developers with support for distributed transactions, resource pooling, disconnected applications, event publication and subscription, better memory and processor (thread) management, as well as to position Windows as an alternative to other enterprise-level operating systems, Microsoft introduced a technology called Microsoft Transaction Server (MTS) on Windows NT 4.

With Windows 2000, that significant extension to COM was incorporated into the operating system (as opposed to the series of external tools provided by MTS) and renamed **COM+**. At the same time, Microsoft de-emphasized DCOM as a separate entity. Components that made use of COM+ services were handled more directly by the added layer of COM+, in particular by operating system support for interception. In the first release of MTS, interception was tacked on - installing an MTS component would modify the Windows Registry to call the MTS software, and not the component directly.

Windows 2000 also revised the Component Services control panel application used to configure COM+ components.

An advantage of COM+ was that it could be run in "component farms". Instances of a component, if coded properly, could be pooled and reused by new calls to its initializing routine without unloading it from memory. Components could also be distributed (called from another machine). COM+ and Microsoft Visual Studio provided tools to make it easy to generate client-side proxies, so although DCOM was used to actually make the remote call, it was easy to do for developers.

COM+ also introduced a subscriber/publisher event mechanism called **COM+ Events**, and provided a new way of leveraging MSMQ (inter-application asynchronous messaging) with components called **Queued Components**. COM+ events extend the COM+ programming model to support late-bound events or method calls between the publisher or subscriber and the event system.

.NET

The COM platform has *largely* been superseded by the Microsoft .NET initiative, and Microsoft now focuses its marketing efforts on *.NET.* COM was often used to hook up complex, high performance code to front end code implemented in Visual Basic or ASP.

To some extent, COM is now deprecated in favor of .NET. Since .NET provides rapid development tools similar to Visual Basic for both Windows Forms and Web Forms with just-in-time compilation, back-end code can be implemented in any .NET Language including C#, Visual Basic and C++/CLI.

Despite this, COM remains a viable technology with an important software base. As of 2009, Microsoft has no plans for discontinuing either COM or support for COM. It is also ideal for script control of applications such as Office or Internet Explorer since it provides an interface for calling COM object methods from a script rather than requiring knowing the API at compile time. The GUID system used by COM has wide uses any time a unique ID is needed.

Several of the services that COM+ provides have been largely replaced by recent releases of .NET. For example, the System.Transactions namespace in .NET provides the TransactionScope class, which provides transaction management without resorting to COM+. Similarly, queued components can be replaced by Windows Communication Foundation with an MSMQ transport.

There is limited support for backward compatibility. A COM object may be used in .NET by implementing a *runtime callable wrapper* (RCW). .NET objects that conform to certain interface restrictions may be used in COM objects by calling a *COM callable wrapper* (CCW). From both the COM and .NET sides, objects using the other technology appear as native objects.

WCF solves a number of COM's remote execution shortcomings, allowing objects to be transparently marshalled by value across process or machine boundaries.

Internet security

Microsoft's idea of embedding active content on web pages as COM/ActiveX components (rather than e.g. Java applets) created a combination of problems in the Internet Explorer web browser that has led to an explosion of computer virus, trojan and spyware infections. These malware attacks mostly depend on ActiveX for their activation and propagation to other computers. Microsoft recognized the problem with ActiveX as far back as 1996 when Charles Fitzgerald, program manager of Microsoft's Java team said "If you want security on the 'Net', unplug your computer. … We never made the claim up front that ActiveX is intrinsically secure."

As COM and ActiveX components are run as native code on the user's machine, there are fewer restrictions on what the code can do. Many of these problems have been addressed by the introduction of "Authenticode" code signing (based on digital signatures), and later by the .NET platform also. Another security measure is that, before an ActiveX control is installed, the user is prompted whether to allow the installation or not, enabling the user to disallow the installation of controls from sites that the user does not trust. It is also possible to disable ActiveX controls altogether, or to allow only a selected few.

Technical details

COM programmers build their software using COM-aware components. Different component types are identified by class IDs (CLSIDs), which are Globally Unique Identifiers (GUIDs). Each COM component exposes its functionality through one or more interfaces. The different interfaces supported by a component are distinguished from each other using interface IDs (IIDs), which are GUIDs too.

COM interfaces have bindings in several languages, such as C, C++, Visual Basic, Delphi, and several of the scripting languages implemented on the Windows platform. All access to components is done through the methods of the interfaces. This allows techniques such as inter-process, or even inter-computer programming (the latter using the support of DCOM).

Interfaces

All COM components must (at the very least) implement the standard `IUnknown` interface, and thus all COM interfaces are derived from `IUnknown`. The `IUnknown` interface consists of three methods: `AddRef()` and `Release()`, which implement reference counting and controls the lifetime of interfaces; and `QueryInterface()`, which by specifying an IID allows a caller to retrieve references to the different interfaces the component implements. The effect of `QueryInterface()` is similar to `dynamic_cast<>` in C++ or casts in Java and C#.

A COM component's interfaces are required to exhibit the reflexive, symmetric, and transitive properties. The reflexive property refers to the ability for the `QueryInterface()` call on a given interface with the interface's ID to return the same instance of the interface. The symmetric property requires that when interface B is retrieved from interface A via `QueryInterface()`, interface A is retrievable from interface B as well. The transitive property requires that if interface B is obtainable from interface A and interface C is obtainable from interface B, then interface C should be retrievable from interface A.

An interface consists of a pointer to a virtual function table that contains a list of pointers to the functions that implement the functions declared in the interface, in the same order that they are declared in the interface. This technique of passing structures of function pointers is very similar to the one used by OLE 1.0 to communicate with its system libraries.

COM specifies many other standard interfaces used to allow inter-component communication. For example, one such interface is `IStream`, which is exposed by components that have data stream semantics (e.g. a `FileStream` component used to read or write files). It has the expected `Read` and `Write` methods to perform stream reads and writes. Another standard interface is `IOleObject`, which is exposed by components that expect to be linked or embedded into a container. `IOleObject` contains methods that allow callers to determine the size of the component's bounding rectangle, whether the component supports operations like 'Open', 'Save' and so on.

Classes

A class is COM's language-independent way of defining a class in the object-oriented sense.

A class can be a group of similar objects or a class is simply a representation of a type of object; it should be thought of as a blueprint that describes the object.

A coclass supplies **concrete** implementation(s) of one or more interfaces. In COM, such concrete implementations can be written in any programming language that supports COM component development, e.g. Delphi, C++, Visual Basic, etc.

One of COM's major contributions to the world of Windows development is the awareness of the concept of **separation of interface from implementation**. An extension of this fundamental concept is the notion of **one interface, multiple implementations**. This means that at runtime, an application can choose to instantiate an interface from one of many different concrete implementations.

Interface Definition Language and type libraries

Type libraries contain metadata that represent COM types. However, these types must first be described using Microsoft Interface Definition Language.

This is the common practice in the development of a COM component, i.e. to start with the definition of types using **IDL.** An IDL file is what COM provides that allows developers to define object-oriented classes, interfaces, structures, enumerations and other user-defined types in a language independent manner. COM IDL is similar in appearance to C/C++ declarations with the addition of keywords such as "interface" and "library" for defining interfaces and collections of classes, respectively. IDL also requires the use of bracketed attributes before declarations to provide additional information, such as the GUIDs of interfaces and the relationships between pointer parameters and length fields.

The IDL file is compiled by the MIDL compiler into a pair of forms for consumption from various languages. For C/C++, the MIDL compiler generates a compiler-independent header file containing struct definitions to match the vtbls of the declared interfaces and a C file containing declarations of the interface GUIDs. C++ source code for a proxy module can also be generated by the MIDL compiler. This proxy contains method stubs for converting COM calls into Remote Procedure Calls, thus enabling DCOM.

An IDL file may also be compiled by the MIDL compiler into a type library (.TLB file). The binary metadata contained within the type library is meant to be processed by language compilers and runtime environments (e.g. VB, Delphi, the .NET CLR etc.). The end result of such TLB processing is that language-specific constructs are produced that represent the COM class defined in the .TLB (and ultimately that which was defined in the originating IDL file).

COM as an object framework

The fundamental principles of COM have their roots in Object-Oriented philosophies. It is a platform for the realization of Object-Oriented Development and Deployment.

Because COM is a runtime framework, types have to be individually identifiable and specifiable at runtime. To achieve this, **globally unique identifiers** (**GUIDs**) are used. Each COM type is designated its own GUID for identification at runtime (versus compile time).

In order for information on COM types to be accessible at both compile time and runtime, COM uses type libraries. It is through the effective use of type libraries that COM achieves its capabilities as a dynamic framework for the interaction of objects.

Consider the following example coclass definition in an IDL :

```
coclass MyObject
{
  [default] interface IMyObject;
  [default, source] dispinterface _IMyObjectEvents;
};
```

The above code fragment declares a COM class named MyObject which must implement an interface named IMyObject and which supports (not implements) the event interface _IMyObjectEvents.

Ignoring the event interface bit, this is conceptually equivalent to defining a C++ class like this:

```
class CSomeObject : public ISomeInterface
{
  ...
  ...
  ...
};
```

where ISomeInterface is a C++ pure virtual class.

Referring once again to the MyObject COM class: once a coclass definition for it has been formalized in an IDL, and a Type Library compiled from it, the onus is on the individual language compiler to read and appropriately interpret this Type Library and then produce whatever code (in the specific compiler's language) necessary for a developer to implement and ultimately produce the binary executable code which can be deemed by COM to be of coclass MyObject.

Once an implementation of a COM coclass is built and is available in the system, next comes the question of how to instantiate it. In languages like C++, we can use the CoCreateInstance() API in which we specify the CLSID (CLSID_MyObject) of the coclass as well as the interface (specified by the IID IID_IMyObject) from that coclass that we want to use to interact with that coclass. Calling CoCreateInstance() like this:

```
CoCreateInstance(CLSID_MyObject,
    NULL,
    CLSCTX_INPROC_SERVER,
    IID_IMyObject,
    (void**)&m_pIMyObject);
```

is conceptually equivalent to the following C++ code:

```
ISomeInterface* pISomeInterface = new CSomeObject();
```

In the first case, the COM sub-system is used to obtain a pointer to an object that implements the IMyObject interface and coclass CLSID_MyObject's particular implementation of this interface is required. In the second case, an instance of a C++ class CSomeObject that implements the interface ISomeInterface is created.

A coclass, then, is an object-oriented class in the COM world. The main feature of the coclass is that it is (1) binary in nature and consequently (2) programming language-independent.

Registry

In Windows, COM classes, interfaces and type libraries are listed by GUIDs in the registry, under HKEY_CLASSES_ROOT\CLSID for classes and HKEY_CLASSES_ROOT\Interface for interfaces. The COM libraries use the registry to locate either the correct local libraries for each COM object or the network location for a remote service.

Under the key HKCR\clsid, the following are specified:

```
-> Inprocserver32 =  object is to be
                 loaded into a process        +
       Path to file/object and readable name
HKCR\interface:
 example:  ISTREAM, IRPCSTUB, IMESSAGEFILTER
   connects to a CLSID. You can specify
   NUMMETHODS and PROXYSTUB(if web-object)

HKCR\typelib
  One or more CLSID can be grouped into type library.
  it contains parameters for linking in COM.
 The rest of the info in the COM parts of the
  REGISTRY, is to give an application/object
   a CLSID.
```

Reference counting

The most fundamental COM interface of all, IUnknown (from which all COM interfaces must be derived), supports two main concepts: feature exploration through the **QueryInterface** method, and object lifetime management by including **AddRef()** and **Release()**. Reference counts and feature exploration apply to objects (not to each interface on an object) and thus must have a centralized implementation.

The COM specifications require a technique called reference counting to ensure that individual objects remain alive as long as there are clients which have acquired access to one or more of its interfaces and, conversely, that the same object is properly disposed of when all code that used the object have finished with it and no longer require it. A COM object is responsible for freeing its own memory once its reference count drops to zero.

For its implementation, a COM Object usually maintains an integer value that is used for reference counting. When AddRef() is called via any of object's interfaces, this integer value is incremented. When Release() is called, this integer is decremented. AddRef() and Release() are the only means by which a client of a COM object is able to influence its lifetime. The internal integer value remains a private member of the COM object and will never be directly accessible.

The purpose of AddRef() is to indicate to the COM object that an additional reference to itself has been affected and hence it is necessary to remain alive as long as this reference is still valid. Conversely, the purpose of Release() is to indicate to the COM object that a client (or a part of the client's code) has no further need for it and hence if this reference count has dropped to zero, it may be time to destroy itself.

Certain languages (e.g. Visual Basic) provide automatic reference counting so that COM object developers need not explicitly maintain any internal reference counter in their source codes. Using COM in C, explicit reference counting is needed. In C++, a coder may write the reference counting code or use a smart pointer that will manage all the reference counting.

The following is a general guideline calling AddRef() and Release() to facilitate proper reference counting in COM object:

- Functions (whether object methods or global functions) that return interface references (via return value or via "out" parameter) should increment the reference count of the underlying object before returning. Hence internally within the function or method, AddRef() is called on the interface reference (to be returned). An example of this is the QueryInterface() method of the IUnknown interface. Hence it is imperative that developers be aware that the returned interface reference has already been reference count incremented and not call AddRef() on the returned interface reference yet another time.

- Release() must be called on an interface reference before that interface's pointer is overwritten or goes out of scope.
- If a copy is made on an interface reference pointer, AddRef() should be called on that pointer. After all, in this case, we are actually creating another reference on the underlying object.
- AddRef() and Release() must be called on the specific interface which is being referenced since an object may implement per-interface reference counts in order to allocate internal resources only for the interfaces which are being referenced.
- Extra calls to these functions are not sent out to remote objects over the wire; a proxy keeps only one reference on the remote object and maintains its own local reference count.

To facilitate and promote COM development, Microsoft introduced ATL (Active Template Library) for C++ developers. ATL provides for a higher-level COM development paradigm. It also shields COM client application developers from the need to directly maintain reference counting, by providing smart pointer objects.

Other libraries and languages that are COM-aware include the Microsoft Foundation Classes, the VC Compiler COM Support, VBScript, Visual Basic, ECMAScript (JavaScript) and Borland Delphi.

Instantiation

COM standardizes the instantiation (i.e. creation) process of COM objects by requiring the use of **Class Factories**. In order for a COM object to be created, two associated items must exist:

- A Class ID.
- A Class Factory.

Each COM Class or **CoClass** must be associated with a unique Class ID (a GUID). It must also be associated with its own Class Factory (that is achieved by using a centralized registry). A Class Factory is itself a COM object. It is an object that must expose the IClassFactory or IClassFactory2 (the latter with licensing support) interface. The responsibility of such an object is to create other objects.

A class factory object is usually contained within the same executable code (i.e. the **server code**) as the COM object itself. When a class factory is called upon to create a target object, this target object's class id must be provided. This is how the class factory knows which class of object to instantiate.

A single class factory object may create objects of more than one class. That is, two objects of different class ids may be created by the same class factory object. However, this is transparent to the COM system.

By delegating the responsibility of object creation into a separate object, a greater level of abstraction is promoted, and the developer is given greater flexibility. For example, implementation of the **Singleton** and other creation patterns is facilitated. Also, the calling application is shielded from the COM object's memory allocation semantics by the factory object.

In order for client applications to be able to acquire class factory objects, COM servers must properly expose them. A class factory is exposed differently, depending on the nature of the server code. A server which is DLL-based must export a **DllGetClassObject()** global function. A server which is EXE-based registers the class factory at runtime via the **CoRegisterClassObject()** Windows API function.

The following is a general outline of the sequence of object creation via its class factory:

1. The object's class factory is obtained via the **CoGetClassObject()** API (a standard Windows API).
   As part of the call to CoGetClassObject(), the Class ID of the object (to be created) must be supplied. The following C++ code demonstrates this:

```
IClassFactory* pIClassFactory = NULL;

CoGetClassObject(CLSID_SomeObject,
    CLSCTX_ALL,
    NULL,
    IID_IClassFactory,
    (LPVOID*)&pIClassFactory);
```

   The above code indicates that the Class Factory object of a COM object, which is identified by the class id CLSID_SomeObject, is required. This class factory object is returned by way of its IClassFactory interface.

9. The returned class factory object is then requested to create an instance of the originally intended COM object. The following C++ code demonstrates this:

```
ISomeObject* pISomeObject = NULL;

if (pIClassFactory)
{
    pIClassFactory->CreateInstance (NULL,
        IID_ISomeObject,
        (LPVOID*)&pISomeObject);

    pIClassFactory->Release();

    pIClassFactory = NULL;
}
```

   The above code indicates the use of the Class Factory object's **CreateInstance()** method to create an object which exposes an interface identified by the IID_ISomeObject GUID. A pointer to the ISomeObject interface of this object is returned. Also note that because the class factory object is itself a COM object, it needs to be released when it is no longer required (i.e. its **Release()** method must be called).

The above demonstrates, at the most basic level, the use of a class factory to instantiate an object. Higher level constructs are also available, some of which do not even involve direct use of the Windows APIs.

For example, the **CoCreateInstance()** API can be used by an application to directly create a COM object without acquiring the object's class factory. However, internally, the CoCreateInstance() API itself will invoke the CoGetClassObject() API to obtain the object's class factory and then use the class factory's CreateInstance() method to create the COM object.

VBScript supplies the **New** keyword as well as the **CreateObject()** global function for object instantiation. These language constructs encapsulate the acquisition of the class factory object of the target object (via the CoGetClassObject() API) followed by the invocation of the IClassFactory::CreateInstance() method.

Other languages, e.g. PowerBuilder's PowerScript may also provide their own high-level object creation constructs. However, CoGetClassObject() and the IClassFactory interface remain the most fundamental object creation technique.

Reflection

At the time of the inception of COM technologies, the only way for a client to find out what features an object would offer was to actually create one instance and call into its QueryInterface method (part of the required IUnknown interface). This way of exploration became awkward for many applications, including the selection of appropriate components for a certain task, and tools to help a developer understand how to use methods provided by an object.

As a result, COM Type Libraries were introduced, through which components can describe themselves. A type library contains information such as the CLSID of a component, the IIDs of the interfaces the component implements, and descriptions of each of the methods of those interfaces. Type libraries are typically used by Rapid Application Development (RAD) environments such as Visual Basic or Visual Studio to assist developers of client applications.

Programming

COM is a binary standard (also said to be **language agnostic**) and may be developed in any programming language capable of understanding and implementing its binary defined data types and interfaces.

Runtime libraries (in extreme situations, the programmers) are responsible for entering and leaving the COM environment, instantiating and reference counting COM objects, querying objects for version information, coding to take advantage of advanced object versions, and coding graceful degradation of function when newer versions are not available.

Application and network transparency

COM objects may be instantiated and referenced from within a process, across process boundaries within a computer, and across a network, using the DCOM technology. Out-of-process and remote objects may use marshalling to send method calls and return values back and forth. The marshalling is invisible to the object and the code using the object.

Threading in COM

In COM, threading issues are addressed by a concept known as "**apartment models**". Here the term "**apartment**" refers to an execution context wherein a single thread or a group of threads is associated with one or more **COM objects**.

Apartments stipulate the following general guidelines for participating threads and objects:

- Each COM object is associated with one and only one apartment. This is decided at the time the object is created at runtime. After this initial setup, the object remains in that apartment throughout its lifetime.
- A COM thread (i.e., a thread in which COM objects are created or COM method calls are made) is also associated with an apartment. Like COM objects, the apartment with which a thread is associated is also decided at initialization time. Each COM thread also remains in its designated apartment until it terminates.
- Threads and objects which belong to the same apartment are said to follow the same thread access rules. Method calls which are made inside the same apartment are performed directly without any assistance from COM.
- Threads and objects from different apartments are said to play by different thread access rules. Method calls made across apartments are achieved via marshalling. This requires the use of proxies and stubs.

There are three types of Apartment Models in the COM world: **Single-Threaded Apartment (STA)**, **Multi-Threaded Apartment (MTA)**, and **Neutral Apartment**. Each apartment represents one mechanism whereby an object's internal state may be synchronized across multiple threads.

The Single-Threaded Apartment (STA) model is a very commonly used model. Here, a COM object stands in a position similar to a desktop application's user interface. In an STA model, a single thread is dedicated to drive an object's methods, i.e. a single thread is always used to execute the methods of the object. In such an arrangement, method calls from threads outside of the apartment are marshalled and automatically queued by the system (via a standard Windows message queue). Thus, there is no worry about race conditions or lack of synchronicity because each method call of an object is always executed to completion before another is invoked.

If the COM object's methods perform their own synchronization, multiple threads dedicated to calling methods on the COM object are permitted. This is termed the Multiple Threaded Apartment (MTA). Calls to an MTA object from a thread in an STA are also marshaled. A process can consist of multiple COM objects, some of which may use STA and others of which may use MTA. The Thread Neutral Apartment allows different threads, none of which is necessarily dedicated to calling methods on the object, to make such calls. The only provision is that all methods on the object must be serially reentrant.

Criticisms

Since COM has a fairly complex implementation, programmers can be distracted by some of the "plumbing" issues.

Message pumping

When an STA is initialized it creates a hidden window that is used for inter-apartment and inter-process message routing. This window must have its message queue regularly pumped. This construct is known as a message pump. On earlier versions of Windows, failure to do so could cause system-wide deadlocks. This problem is especially nasty because some Windows APIs initialize COM as part of their implementation, which causes a leak of implementation details.

Reference counting

Reference counting within COM may cause problems if two or more objects are circularly referenced. The design of an application must take this into account so that objects are not left orphaned.

Objects may also be left with active reference counts if the COM "event sink" model is used. Since the object that fires the event needs a reference to the object reacting to the event, the object's reference count will never reach zero.

Reference cycles are typically broken using either out-of-band termination or split identities. In the out of band termination technique, an object exposes a method which, when called, forces it to drop its references to other objects, thereby breaking the cycle. In the split identity technique, a single implementation exposes two separate COM objects (also known as identities). This creates a weak reference between the COM objects, preventing a reference cycle.

DLL hell

Because the location of each component is stored in a system-wide location (the Windows registry), there can be only one version of a certain component installed. This limitation can seriously complicate the deployment of COM-based applications, due to the possibility that different programs, or even different versions of the same program, may be designed to work with different versions of the same COM component. This condition is known as DLL hell. While this condition has been known to occur with OS components, it is generally a condition created by application developers in the use of their own components. The problem can be reduced or eliminated completely by careful software versioning and regression testing.

Windows XP introduced a new mode of COM object registration called "**Registration-free COM**". This facility makes it possible for applications that need to install COM objects to store all the required COM registry information in the application's directory, instead of in the global registry, so that, strictly speaking, only that application will ever see/use it. DLL hell can be substantially avoided using Registration-free COM, the only limitation being it requires at least Windows XP or later Windows versions and that it must not be used for EXE COM servers or system-wide components such as MDAC, MSXML, DirectX or Internet Explorer.

RegFree COM

RegFree COM (or **Registration-Free COM**) is a technology introduced with Windows XP that allows Component Object Model (COM) components to store activation metadata and CLSID (`Class ID`) for the component without using the registry. Instead, the metadata and CLSIDs of the classes implemented in the component are declared in an assembly manifest (described using XML), stored either as a resource in the executable or as a separate file installed with the component. This allows multiple versions of the same component to be installed in different directories, described by their own manifests, as well as XCOPY deployment.

During application loading, the Windows loader searches for the manifest. If it is present, the loader adds information from it to the activation context When the COM class factory tries to instantiate a class, the activation context is first checked to see if an implementation for the CLSID can be found. Only if the lookup fails is the registry scanned.

Chapter 7

Dependency Injection

Dependency injection (**DI**) in object-oriented computer programming is a technique that indicates to a part of a program which other parts it can use, i.e. to supply an external dependency (i.e. a reference) to a software component. In technical terms, it is a design pattern that separates behavior from dependency resolution, thus decoupling highly dependent components.

Developers of software strive to reduce dependencies between components in software for various reasons. This leads to a new problem, though: How can a component know all the other components it needs to fulfill its purpose?

The traditional approach was to hard-code the dependency. As soon as the database driver was necessary, the component would execute a piece of code that would load a specific driver, configure it and call the necessary methods to interact with the database. If a second database must be supported, this piece of code would have to be modified or, even worse, copied and modified (violating the DRY principle).

Dependency injection offers a solution. Instead of hard-coding the dependencies, a component just lists the necessary services and a DI framework supplies these. At runtime, an independent component will load and configure the database driver and offer a standard interface to interact with the database. Again, the details have been moved from the original component to a set of new, small, database specific components, reducing the complexity of them all.

In DI terms, these new components are called "service components" because they render a service (database access) for one or more other components.

Dependency injection is a specific form of inversion of control where the concern being inverted is the process of obtaining the needed dependency. The term was first coined by Martin Fowler to describe the mechanism more clearly.

Basics

Without dependency injection, a consumer component that needs a particular service in order to accomplish a task will depend not only on the interface of the service but also on the details of a particular implementation of that service. The user component has to handle both its use of the service and the life-cycle of that service - creating an instance, opening and closing streams, disposing of unneeded objects, etc.

Using dependency injection, however, the life-cycle of a service is handled by a dependency provider rather than the consumer. The dependency provider is an independent, external component that links the consuming component and the providing component. The consumer would thus only need a reference to an implementation of the service that it needed in order to accomplish the necessary task.

Such a pattern involves at least three elements: a **dependent** consumer, the definition of its service **dependencies,** and an **injector** (sometimes referred to as a **provider** or **container**). The dependent is a consumer component that needs to accomplish a task in a computer program. In order to do so, it needs the help of various services (the dependencies) that execute certain sub-tasks. The provider is the component that is able to compose the dependent and its dependencies so that they are ready to be used, while also managing these objects' life-cycles. The provider may be implemented, for example, as a service locator, an abstract factory, a factory method or a more complex abstraction such as a framework.

The following is an example. A car (the consumer) depends upon an engine (the dependency) in order to move. The car's engine is made by an automaker (the dependency provider). The car does not know how to install an engine into itself, but it needs an engine in order to move. The automaker installs an engine into the car and the car utilizes the engine to move.

When the concept of dependency injection is used, it decouples high-level modules from low-level services. The result is called the dependency inversion principle.

Code illustration using Java

Using the car/engine example above mentioned, the following Java examples show how coupled dependencies (manually-injected dependencies) and framework-injected dependencies are typically staged.

```
public interface Engine {
    public float getEngineRPM();

    public void setFuelConsumptionRate(float flowInGallonsPerMinute);
}

public interface Car {
    public float getSpeedInMPH();

    public void setPedalPressure(float pedalPressureInPounds);
}
```

Highly coupled dependency

The following shows a common arrangement **with no dependency injection applied**:

```
public class DefaultEngineImpl implements Engine {
    private float engineRPM = 0;

    public float getEngineRPM() {
        return engineRPM;
    }

    public void setFuelConsumptionRate(float flowInGallonsPerMinute) {
        engineRPM = ...;
    }
}

public class DefaultCarImpl implements Car {
    private Engine engine = new DefaultEngineImpl();

    public float getSpeedInMPH() {
        return engine.getEngineRPM() * ...;
    }

    public void setPedalPressure(float pedalPressureInPounds) {
        engine.setFuelConsumptionRate(...);
    }
}

public class MyApplication {
    public static void main(String[] args) {
        Car car = new DefaultCarImpl();
        car.setPedalPressure(5);
        float speed = car.getSpeedInMPH();
    }
}
```

In the above example, the `Car` class creates an instance of an `Engine` implementation in order to perform operations on the car. Hence, it is considered highly coupled because it couples a car directly with a particular engine implementation.

In cases where the `DefaultEngineImpl` dependency is managed outside of the scope of the `Car` class, the `Car` class must not instantiate the `DefaultEngineImpl` dependency. Instead, that dependency is injected externally.

Manually-injected dependency

Refactoring the above example to use manual injection:

```
public class DefaultCarImpl implements Car {
    private Engine engine;

    public DefaultCarImpl(Engine engineImpl) {
        engine = engineImpl;
    }

    public float getSpeedInMPH() {
        return engine.getEngineRPM() * ...;
    }

    public void setPedalPressure(float pedalPressureInPounds) {
        engine.setFuelConsumptionRate(...);
    }
}

public class CarFactory {
    public static Car buildCar() {
        return new DefaultCarImpl(new DefaultEngineImpl());
    }
}

public class MyApplication {
    public static void main(String[] args) {
        Car car = CarFactory.buildCar();
        car.setPedalPressure(5);
        float speed = car.getSpeedInMPH();
    }
}
```

In the example above, the `CarFactory` class assembles a car and an engine together by injecting a particular engine implementation into a car. This moves the dependency management from the `Car` class into the `CarFactory` class. As a consequence, if the `Car` needed to be assembled with a different `Engine` implementation, the `Car` code would not be changed.

In a more realistic software application, this may happen if a new version of a base application is constructed with a different service implementation. Using factories, only the service code and the Factory code would need to be modified, but not the code of the multiple users of the service.

However, this still may not be enough abstraction for some applications, since in a realistic application there would be multiple Factory classes to create and update.

Framework-managed dependency injection

There are several frameworks available that automate dependency management by delegating the management of dependencies. Typically, this is accomplished by a Container using XML or "meta data" definitions. Refactoring the above example to use an external XML-definition framework:

```
    <service-point id="CarBuilderService">
        <invoke-factory>
            <construct class="Car">
                <service>DefaultCarImpl</service>
                <service>DefaultEngineImpl</service>
            </construct>
        </invoke-factory>
    </service-point>
/** Implementation not shown **/

public class MyApplication {
    public static void main(String[] args) {
        Service service =
(Service)DependencyManager.get("CarBuilderService");
        Car car = (Car)service.getService(Car.class);
        car.setPedalPressure(5);
        float speed = car.getSpeedInMPH();
    }
}
```

In the above example, a dependency injection service is used to retrieve a `CarBuilderService` service. When a Car is requested, the service returns an appropriate implementation for both the car and its engine.

As there are many ways to implement dependency injection, only a small subset of examples is shown herein. Dependencies can be registered, bound, located, externally injected, etc., by many different means. Hence, moving dependency management from one module to another can be accomplished in a plethora of ways. However, there should exist a definite reason for moving a dependency away from the object that needs it because doing so can complicate the code hierarchy to such an extent that its usage appears to be "magical". For example, suppose a Web container is initialized with an association between two dependencies and that a user who wants to use one of those dependencies is unaware of the association. The user would thus not be able to detect any linkage between those dependencies and hence might cause drastic problems by using one of those dependencies.

Benefits and drawbacks

One benefit of using the dependency injection approach is the reduction of boilerplate code in the application objects since all work to initialize or set up dependencies is handled by a provider component.

Another benefit is that it offers configuration flexibility because alternative implementations of a given service can be used without recompiling code. This is useful in unit testing because it is easy to inject a fake implementation of a service into the object being tested by changing the configuration file.

One drawback is that excessive or inappropriate use of dependency injection can make applications more complicated, harder to understand and more difficult to modify. Code that uses dependency injection can seem *magical* to some developers, since instantiation and initialization of objects is handled completely separately from the code that uses it. This separation can also result in problems that are hard to diagnose. Additionally, some dependency injection frameworks maintain verbose configuration files, requiring that a developer understand the configuration as well as the code in order to change it.

Another drawback is that some IDEs might not be able to accurately analyze or refactor code when configuration is "invisible" to it. Some IDEs mitigate this problem by providing explicit support for various frameworks. Additionally, some frameworks provide configuration using the programming language itself, allowing refactoring directly. Other frameworks, such as the Grok web framework, introspect the code and use convention over configuration as an alternative form of deducing configuration information. For example, if a Model and View class were in the same module, then an instance of the View will be created with the appropriate Model instance passed into the constructor-

Criticisms

A criticism of dependency injection is that it is simply a re-branding of existing object-oriented design concepts. The examples typically cited (including the one above) simply show how to fix bad code, not a new programming paradigm. Offering constructors and/or setter methods that take interfaces, relieving the implementing class from having to choose an implementation, is an idea that was rooted in object-oriented programming long before Martin Fowler's article or the creation of any of the recent frameworks that champion it.

Types

Fowler identifies three ways in which an object can get a reference to an external module, according to the pattern used to provide the dependency:

- *Type 1* or *interface injection*, in which the exported module provides an interface that its users must implement in order to get the dependencies at runtime.
- *Type 2* or *setter injection*, in which the dependent module exposes a setter method that the framework uses to inject the dependency.
- *Type 3* or *constructor injection*, in which the dependencies are provided through the class constructor.

It is possible for other frameworks to have other types of injection, beyond those presented above.

Chapter 8

Distributed Component Object Model and ActiveX

Distributed Component Object Model

Distributed Component Object Model (**DCOM**) is a proprietary Microsoft technology for communication among software components distributed across networked computers. DCOM, which originally was called "Network OLE", extends Microsoft's COM, and provides the communication substrate under Microsoft's COM+ application server infrastructure. It has been deprecated in favor of the Microsoft .NET Framework.

The addition of the "D" to COM was due to extensive use of DCE/RPC (Distributed Computing Environment/Remote Procedure Calls) – more specifically Microsoft's enhanced version, known as MSRPC.

In terms of the extensions it added to COM, DCOM had to solve the problems of

- Marshalling – serializing and deserializing the arguments and return values of method calls "over the wire".
- Distributed garbage collection – ensuring that references held by clients of interfaces are released when, for example, the client process crashed, or the network connection was lost.

One of the key factors in solving these problems is the use of DCE/RPC as the underlying RPC mechanism behind DCOM. DCE/RPC has strictly defined rules regarding marshalling and who is responsible for freeing memory.

DCOM was a major competitor to CORBA. Proponents of both of these technologies saw them as one day becoming the model for code and service-reuse over the Internet. However, the difficulties involved in getting either of these technologies to work over Internet firewalls, and on unknown and insecure machines, meant that normal HTTP requests in combination with web browsers won out over both of them. Microsoft, at one point, attempted and failed to head this off by adding an extra http transport to DCE/RPC called *ncacn_http* (Network Computing Architecture, Connection-based, over HTTP). This was later resurrected to support a Microsoft Exchange 2003 connection over HTTP.

Alternative versions and implementations

The Open Group has a DCOM implementation called *COMsource*. Its source code is available, along with full and complete documentation, sufficient to use and also implement an interoperable version of DCOM. According to that documentation, COMsource comes directly from the Windows NT 4.0 source code, and even includes the source code for a Windows NT Registry Service.

The Wine Team is also implementing DCOM for binary interoperability purposes; they are not currently interested in the networking side of DCOM, which is provided by MSRPC. They are restricted to implementing NDR (Network Data Representation) through Microsoft's API, but are committed to making it as compatible as possible with MSRPC.

TangramCOM is a separate project from Wine, focusing on implementing DCOM on Linux-based smartphones.

The Samba Team is also implementing DCOM for over-the-wire interoperability purposes: unlike the Wine Team, they are not currently interested in binary-interoperability, as the Samba MSRPC implementation is far from binary-interoperable with Microsoft's MSRPC. Between the two projects, Samba and Wine, tackling interoperability from different angles, a fully interoperable implementation of DCOM should be achievable, eventually.

j-Interop is an open source (LGPL) implementation of MSRPC purely in Java, supporting DCOM client applications in Java on any platform communicating with DCOM servers.

J-Integra for COM is a mature commercial pure Java implementation of the DCOM wire protocol allowing access to COM components from Java clients, and Java objects from COM clients.

EntireX DCOM is a commercial implementation by Software AG for AS/400, BS2000/OSD, Windows, Unix (AIX, HP-UX, Linux, Solaris), z/OS, and z/VM.

Procedure

To access DCOM settings on a computer running Windows 2000, Windows XP and earlier, click **Start** > **Run**, and type "**dcomcnfg**". (Click **NO** for any warning screens that appear.) To access DCOM settings on a computer running Windows Vista or later, click **Start**, type "**dcomcnfg**", right-click "**dcomcnfg.exe**" in the list, and click "Run as administrator".

This opens the Distributed COM Configuration Properties dialog.

ActiveX

ActiveX is a framework for defining reusable software components in a programming language independent way. Software applications can then be composed from one or more of these components in order to provide their functionality.

It was introduced in 1996 by Microsoft as a development of its Component Object Model (COM) and Object Linking and Embedding (OLE) technologies and is commonly used in its Windows operating system, although the technology itself is not tied to it.

Many Microsoft Windows applications — including many of those from Microsoft itself, such as Internet Explorer, Microsoft Office, Microsoft Visual Studio, and Windows Media Player — use ActiveX controls to build their feature-set and also encapsulate their own functionality as ActiveX controls which can then be embedded into other applications. Internet Explorer also allows embedding ActiveX controls onto web pages.

ActiveX controls

Active X controls, small program building blocks, can serve to create distributed applications working over the Internet through web browsers. Examples include customized applications for gathering data, viewing certain kinds of files, and displaying animation.

Active X controls are comparable with Java applets: programmers designed both of these mechanisms to allow web browsers to download and execute them. They also differ:

- Java applets can run on nearly any platform, while ActiveX components officially operate only with Microsoft's Internet Explorer web browser and the Microsoft Windows operating system.Malware, e.g. computer viruses and spyware, can be accidentally installed from malicious websites using ActiveX controls (drive-by downloads).

Programmers can write ActiveX controls in any language which supports COM component development, including the following languages/environments:

- C++ either directly or with the help of libraries such as ATL or MFC
- Borland Delphi
- Visual Basic

Common examples of ActiveX controls include command buttons, list boxes, dialog boxes, and the Internet Explorer browser.

History

Faced with the complexity of OLE 2.0 and with poor support for COM in MFC, Microsoft rationalized the specifications to make them simpler, and rebranded the technology as ActiveX in 1996. Even after simplification, users still required controls to implement about six core interfaces. In response to this complexity, Microsoft produced wizards, ATL base classes, macros and C++ language extensions to make it simpler to write controls.

Starting with Internet Explorer 3.0 (1996), Microsoft added support to host ActiveX controls within HTML content. If the browser encountered a page specifying an ActiveX control via an `OBJECT` tag, it would automatically download and install the control with little or no user intervention. This made the web "richer" but provoked objections (since such controls only ran on Windows) and security risks (especially given the lack of user intervention). Microsoft subsequently introduced security measures to make browsing including ActiveX safer . For example:

- digital signing of installation packages (Cabinet files and executables)
- controls must explicitly declare themselves safe for scripting
- increasingly stringent default security settings
- Internet Explorer maintains a blacklist of bad controls

ActiveX in non-IE applications

It may not always be possible to use Internet Explorer to execute ActiveX content (e.g. on a WINE installation), nor may a user want to.

- FF ActiveX Host can run ActiveX controls in Mozilla Firefox for Windows.
- Mozilla ActiveX Control was last updated in late 2005, and runs in Firefox 1.5.
- MediaWrap for Firefox was last updated on 12 June, 2008, and will run in Firefox 1.5 to 3.5.*.

Other ActiveX technologies

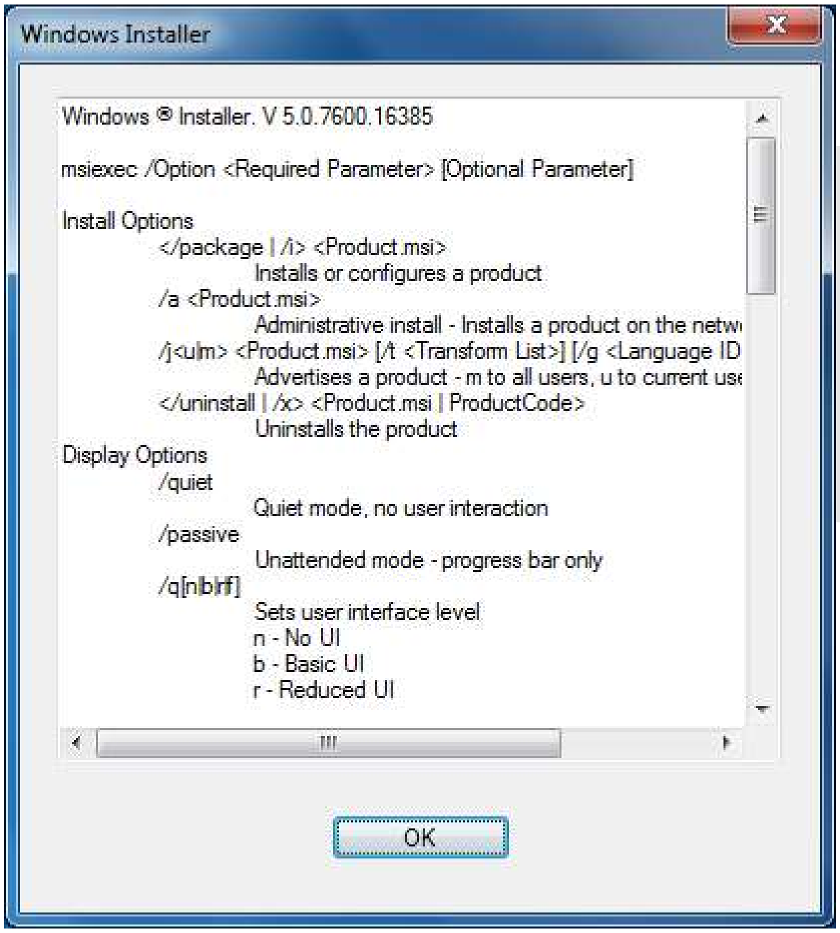
Microsoft has developed a large number of products and software platforms using ActiveX objects. Some remain in use as of 2009:

- ActiveX Data Objects (ADO)
- Active Server Pages
- ActiveMovie, later renamed DirectShow
- Active Messaging, later renamed Collaboration Data Objects
- Active Scripting, a technology for scripting ActiveX objects
- ActiveX Streaming Format (ASF), renamed Advanced Streaming Format, then to Advanced Systems Format

Chapter 9

Windows Installer

Windows Installer



```
Windows ® Installer. V 5.0.7600.16385

msiexec /Option <Required Parameter> [Optional Parameter]

Install Options
        </package | /i> <Product.msi>
                Installs or configures a product
        /a <Product.msi>
                Administrative install - Installs a product on the netw
        /j<u|m> <Product.msi> [/t <Transform List>] [/g <Language ID
                Advertises a product - m to all users, u to current use
        </uninstall | /x> <Product.msi | ProductCode>
                Uninstalls the product
Display Options
        /quiet
                Quiet mode, no user interaction
        /passive
                Unattended mode - progress bar only
        /q[n|b|r|f]
                Sets user interface level
                n - No UI
                b - Basic UI
                r - Reduced UI
```

Default window (after running msiexec.exe)

Original author(s)	Microsoft
Stable release	5.0
Operating system	Windows
Type	Installer

License	Proprietary

The **Windows Installer** (previously known as **Microsoft Installer**) is a software component used for the installation, maintenance, and removal of software on modern Microsoft Windows systems. The installation information, and often the files themselves, are packaged in *installation packages*, loosely relational databases structured as OLE COM Structured Storages and commonly known as "MSI files", from their default file extension. Windows Installer contains significant changes from its predecessor, Setup API. New features include a GUI framework and automatic generation of the uninstallation sequence. Windows Installer is positioned as an alternative to stand-alone executable installer frameworks such as older versions of InstallShield and Wise (later versions of both products are based on Windows Installer) and NSIS.

Microsoft encourages third parties to use Windows Installer as the basis for installation frameworks, so that they synchronize correctly with other installers and keep the internal database of installed products consistent. Important features such as *rollback* and *versioning* depend on a consistent internal database for reliable operation.

Logical structure of packages

A package describes the installation of one or more full *products* (Windows Installer does not handle dependencies between products) and is universally identified by a GUID (the PackageCode property). A product is made up of *components*, grouped into *features*.

Products

A single, installed, working program (or set of programs) is a product. A product is identified by a unique GUID (the ProductCode property). A product is not the same as a package: a single MSI package might install multiple different products. For example, an MSI might install French and English versions of a program, each of which is a different product.

Features

A *feature* is a hierarchical group of components—a feature can contain any number of components and other features (a feature contained in another feature is called a "subfeature"). Many software packages only involve one feature. More complex installation programs usually display a "custom setup" dialog box at run time, from which the end user can select which features to install or remove.

The package author defines the product features. A word-processing program, for example, might provide features for the main program executable, the program's help files, and optional spelling checker and stationery modules.

Components

A *component* is the minimal part of a product—each component is treated by Windows Installer as a unit: the install developer cannot, for example, use a condition to specify to install just part of a component. Components can contain files, directories, COM components, registry keys, shortcuts, and other data. The end user does not directly interact with components.

Components are identified globally by GUIDs, thus the same component can be shared among several features of the same package or multiple packages, ideally through the use of Merge Modules (although, for this to work correctly, different components should not share any sub-components).

Key paths

A *key path* is a specific file, registry key, or ODBC data source that the package author specifies as critical for a given component. Because a file is the most common type of key path, the term *key file* is commonly used. A component can contain at most one key path; if a component has no explicit key path, the component's destination directory is taken to be the key path. When an MSI-based application is launched, Windows Installer checks the existence of these critical files or registry keys (that is, the key paths). If there is a mismatch between the current system state and the value specified in the MSI package (e.g., a key file is missing), then the related feature is re-installed. This process is also known as *self-healing* or *self-repair*. No two components should use the same key path.

Setup phases

User interface

The user interface phase typically queries the target system and displays an installation wizard and enables the user to change various options that will affect the installation.

However, the user interface sequence should not make any changes to the system. Three reasons for this are as follows.

1. A user can install an MSI package in quiet mode, bypassing this phase entirely, by running the msiexec.exe command-line utility with the /qn (or /qb or /qr) option and specifying on the command line all the information that the wizard would normally gather. Therefore, any actions that occur in the user interface sequence will not be performed during a silent installation.
2. Similarly, clicking the *Remove* button in the Add or Remove Programs panel runs a product's uninstaller with a basic user interface, again with the result that any actions that occur in the user interface sequence will not be performed.

3. Actions that make system changes should not be scheduled in the user interface sequence as the user interface sequence runs with user privileges, and not with elevated privileges, as described in the following section.

Actions in the user interface sequence of a normal installation are defined in the InstallUISequence table. Similarly, there is an AdminUISequence in which can be placed dialog boxes and actions to display and perform from within an administrative installation wizard.

Execute

When the user clicks the *Finish* or *Install* button in a typical MSI installation wizard, installation proceeds to the Execute phase, in which software components are actually installed. The Execute phase makes system changes, but does not display any user-interface elements.

Execute phase happens in two steps:

Immediate mode. In this phase, Windows Installer receives instructions, either from a user or an application, to install or uninstall features of a product. The requests cause the execution of *sequences* of *actions*, which query the installation database to build an internal *script* describing the execution phase in detail.

Deferred mode. In this phase, the script built in immediate mode is executed in the context of the privileged Windows Installer service (specifically, the LocalSystem account). The script must be executed by a privileged account because of the heterogeneity of the scenarios in which a setup operation is initiated—for example, elevated privileges are necessary to serve on-demand installation requests from non-privileged users. (In order to run with elevated privileges, however, the package must be deployed by a local administrator or advertised by a system administrator using Group Policy.)

Execute sequence actions for a normal installation are stored in the InstallExecuteSequence table. An MSI database can also contain AdminExecuteSequence and AdvtExecuteSequence tables to define actions to perform for administrative and advertised installations.

Rollback

All installation operations are transactional. For each operation that Windows Installer performs, it generates an equivalent undo operation that would undo the change made to the system. In case any script action fails during deferred execution, or the operation is cancelled by the user, all the actions performed until that point are rolled back, restoring the system to its original state. Standard Windows Installer actions automatically write information into a rollback script; package authors who create custom actions that change the target system should also create corresponding rollback actions (as well as uninstallation actions and uninstallation-rollback actions). As a design feature, if applied correctly this mechanism will also rollback a failed uninstall of an application to a good working state.

Other features

Advertisement

Windows Installer can *advertise* a product rather than actually installing it. The product will appear installed to the user, but it will not actually be installed until it is run for the first time by triggering an entry point (by means of a Start menu shortcut, by opening a document that the product is configured to handle, or by invoking an advertised COM class). A package can be advertised by an administrator using Group Policy or other deployment mechanism, or by running the msiexec executable with the /jm (for per-machine advertisement) or /ju (for per-user advertisement) switch. It should also be noted that some MSI packages authored in Installshield may prevent the use of these and other Native MSI features.

What is not known by most is that the user must have administrator privileges to complete this advertised installation. In most workplaces, end users are not administrators and this method of distribution will fail. Microsoft created a workaround via Group Policies to "Elevate user privileges" during MSI installations. Administrators have since seen this GPO setting as a security hole since any MSI would automatically gain administrator privileges.

Installation on demand

Similar to advertisement, it consists in the installation of *features* as soon as the user tries to use them. Again, depending on how the features need to be installed, the end-user may have to be an administrator on the PC for the feature to install.

Administrative installation

An administrative installation creates an uncompressed source image for a product, typically to be used for installing or running an application from a network location. An administrative installation is not a typical installation, in that it does not create any shortcuts, register COM servers, create an Add or Remove Programs entry, and so on. Often an administrative installation enables a user to install the product in such a way that its features run from the uncompressed installation source.

Administrative installations are also useful when creating a Windows Installer patch, which requires uncompressed images of the earlier and current versions of a product in order to compute binary file differences. An administrative installation is performed by running the msiexec executable with the /a switch.

Custom Actions

The developer of an installer package may write code to serve their own purpose, delivered in a DLL. This can be executed during the installation sequences, including when the user clicks a button in the user interface, or during the InstallExecuteSequence. Custom Actions typically validate product license keys, or initialise more complex services. Developers should normally provide inverse custom actions for use during uninstallation.

Msiexec provides a way to break after loading a specified custom action DLL but before invoking the action.

Merge Modules and Nested Executables

A Windows Installer package may contain another package to be installed at the same time. These are ideally provided as a .msm file component, but may also be a separate executable program which will be unpacked from the installer package during the InstallExecuteSequence and can be run immediately. The file can then optionally be deleted before the end of the InstallExecuteSequence, and so is ideal for using with older installers.

Miscellaneous

Windows Installer allows applications to run directly from a network share, without the need for a local copy (*run from source*); it can repair broken installations by restoring damaged or deleted files, registry entries and application shortcuts; it supports per-user installation of applications; it can resolve component identifiers into paths, allowing applications to avoid hard-coded file paths; and it natively supports *patches* (.msp files made out of patch creation properties) and other customizations of packages through manipulations (*transforms* or .mst files) of a package's relational database. Version 2.0 onwards, it supports digital signatures and version 3.0 onwards, delta compression for patches.

It is also unique among installation software frameworks for Windows in that it is highly transparent. The full API and all command-line options are documented; packages are freely viewable and editable, both with free tools and programmatically (as opposed to the proprietary and even weakly encrypted packages of InstallShield); and the format for file archives is the well documented cabinet file format.

Windows Vista

Windows Installer 4.0, which was shipped with Windows Vista, incorporates new capabilities to take advantage of Vista's User Account Control architecture. MSI packages can be marked as not requiring elevated privileges to install, thus allowing a package to install without prompting the user for Administrator credentials. Windows Installer also works in conjunction with the Restart Manager; when installing or updating an application or system component with "full" user interface mode, the user will be displayed a list of affected applications that can be shut down, and then restarted after files have been updated. Installer actions running in silent mode perform these application restarts automatically. System services and tray applications can also be restarted in this manner.

Developing Installer Packages

Creating an installer package for a new application is non-trivial. It is necessary to specify which files must be installed, to where, with what registry keys. Any non-standard operations can be done using Custom Actions, which are typically developed in DLLs. There are a number of commercial and freeware products to assist in creating installers; for example Visual Studio, InstallShield, Installaware, Wise Installer, Advanced Installer and WiX. To varying degrees, the user interface and behaviour in less common situations such as unattended installation, may be configured. Once prepared, an installer package is "compiled" by reading the instructions and files from the developer's local machine, and creating the .msi file.

The user interface (dialog boxes) presented at the start of installation can be changed or configured by the setup engineer developing a new installer. There is a limited language of buttons, textfields and labels which can be arranged in a sequence of dialogue boxes. An installer package should be capable of running without any UI, for what is called "unattended installation".

ICE validation

Microsoft provides a set of Internal Consistency Evaluators, or ICEs, that can be used to detect potential problems with an MSI database. The ICE rules are combined into CUB files, which are stripped-down MSI files containing custom actions that test the target MSI database's contents for validation warnings and errors. ICE validation can be performed with the Platform SDK tools Orca and msival2, or with validation tools that ship with the various authoring environments.

For example, some of the ICE rules are:

- ICE09: Validates that any component destined for the System folder is marked as being permanent.
- ICE24: Validates that the product code, product version, and product language have appropriate formats.
- ICE33: Validates that the Registry table is not used for data better suited for another table (Class, Extension, Verb, and so on).

Addressing ICE validation warnings and errors is an important step in the release process.

Versions

Version	Included with	Also available for
1.0	Office 2000	-
1.1	Windows 2000 RTM, SP1, SP2	Windows 95/98 Windows NT 4.0 SP6
1.2	Windows Me	-
2.0	Windows XP RTM, SP1 Windows 2000 SP3, SP4 Windows Server 2003 RTM	Windows 95/98/Me Windows NT 4.0 SP6 Windows 2000 RTM, SP1, SP2
3.0	Windows XP SP2	Windows 2000 SP3, SP4 Windows XP RTM, SP1 Windows Server 2003 RTM
3.1	Windows XP SP3 Windows Server 2003 SP1, SP2 Windows XP Professional x64 Edition RTM, SP2	Windows 2000 SP3, SP4 Windows XP RTM, SP1, SP2 Windows Server 2003 RTM
4.0	Windows Vista RTM, SP1 Windows Server 2008 RTM	-
4.5	Windows Vista SP2 Windows Server 2008 SP2	Windows XP SP2, SP3 Windows Server 2003 SP1, SP2 Windows XP Professional x64 Edition RTM, SP2 Windows Vista RTM, SP1 Windows Server 2008 RTM
5.0	Windows 7 RTM Windows Server 2008 R2 RTM	-

To check which version is currently installed, type `msiexec /?` into the Windows `Run` box or command prompt.

Tools

Name	Description	License
Microsoft Visual Studio	Microsoft Visual Studio is capable of building Windows Installer Deployment projects that can create installer packages.	Proprietary software
WiX	WiX (Windows Installer XML) is a free software set of tools that helps build a Windows Installer packages from an XML document. It can be either used from command-line or integrated into Microsoft Visual Studio.	Common Public License
7-Zip	7-Zip is essentially an open source file archiver utility, but can also extract the contents of MSI files.	GNU Lesser General Public License