0Part I: Component Object Model Introduction

Part I is an overview and introduction to the Component Object Model. The only chapter in Part I (Chapter 1), explains at a high level the motivations of COM and the problems it addresses. It describes what COM is and its features, and describes the major benefits and advantages of COM.

1Introduction

1.1Challenges Facing The Software Industry

Constant innovation in computing hardware and software have brought a multitude of powerful and sophisticated applications to users' desktops and across their networks. Yet with such sophistication have come commensurate problems for application developers, software vendors, and users:

- ∑ Today's applications are large and complex—they are time-consuming to develop, difficult and costly to maintain, and risky to extend with additional functionality.
- ∑ Applications are monolithic—they come prepackaged with a wide range of features but most features cannot be removed, upgraded independently, or replaced with alternatives.
- ∑ Applications are not easily integrated—data and functionality of one application are not readily available to other applications, even if the applications are written in the same programming language and running on the same machine.
- ∑ Operating systems have a related set of problems. They are not sufficiently modular, and it is difficult to override, upgrade, or replace OS-provided services in a clean and flexible fashion.

This page intentionally left blank.

- ∑ Programming models are inconsistent for no good reason. Even when applications have a facility for cooperating, their services are provided to other applications in a different fashion from the services provided by the operating system or the network. Moreover, programming models vary widely depending on whether the service is coming from a provider in the same address space as the client program (via dynamic linking), from a separate process on the same machine, from the operating system, or from a provider running on a separate machine (or set of cooperating machines) across the network.

In addition, a result of the trends of hardware down-sizing and increasing software complexity is the need for a new style of distributed, client/server, modular and "componentized" computing. This style calls for:

- ∑ A generic set of facilities for finding and using service providers (whether provided by the operating system or by applications, or a combination of both), for negotiating capabilities with service providers, and for extending and evolving service providers in a fashion that does not inadvertently break the consumers of earlier versions of those services.
- ∑ Use of object-oriented concepts in system and application service architectures to better match the new generation of object-oriented development tools, to manage increasing software complexity through

increased modularity, to re-use existing solutions, and to facilitate new designs of more self-sufficient software components.

- ∑ Client/server computing to take advantage of, and communicate between, increasingly powerful desktop devices, network servers, and legacy systems.
- ∑ Distributed computing to provide a single system image to users and applications and to permit use of services in a networked environment regardless of location, machine architecture, or implementation environment.

As an illustration of the issues at hand, consider the problem of creating a system service API (Application Programming Interface) that works with multiple providers of some service in a "polymorphic" fashion. That is, a client of the service can transparently use any particular provider of the service without any special knowledge of which specific provider —or implementation —is in use. In traditional systems, there is a central piece of code—conceptually, the service manager is a sort of "object manager," although traditional systems usually involve function-call programming models with system-provided handles used as the means for "object" selection—that every application calls to access meta-operations such as selecting an object and connecting to it. But once applications have used those "object manager" operations and are connected to a service provider, the "object manager" only gets in the way and forces unnecessary overhead upon all applications as shown in Figure 1-1.

In

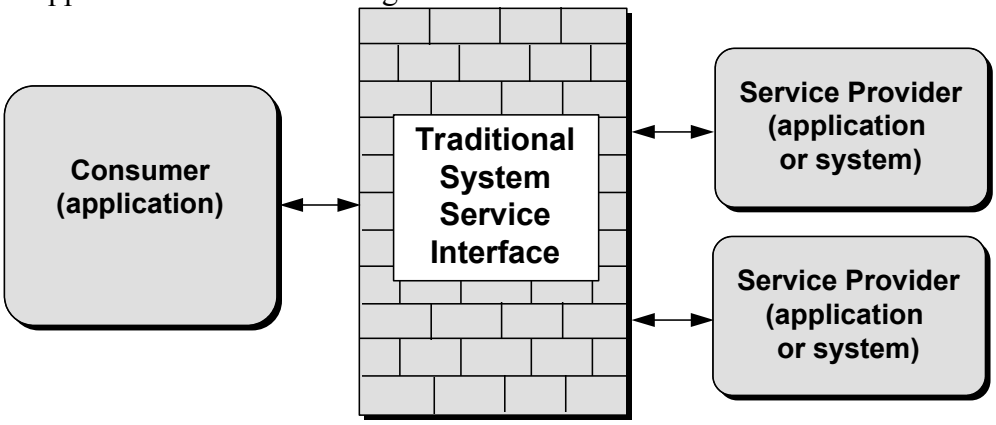

Figure 1-1: Traditional system service APIs require all applications to communicate through a central manager with corresponding overhead.

addition to the overhead of the system-provided layer, another significant problem with traditional service models is that it is impossible for the provider to express new, enhanced, or unique capabilities to potential consumers in a standard fashion. A well-designed traditional service architecture may provide the notion of different levels of service. (Microsoft's Open Database Connectivity (ODBC) API is an example of such an API.) Applications can count on the minimum level of service, and can determine at run-time if the provider supports higher levels of service in certain pre-defined quanta, but the providers are restricted to providing the levels of services defined at the outset by the API; they cannot readily provide a new capability and then evangelize consumers to access it cheaply and in a fashion that fits within the standard model. To take the ODBC example, the vendor of a database provider intent on doing more than the current ODBC standard permits must convince Microsoft to revise the ODBC standard in a way that

exposes that vendor's extra capabilities. Thus, traditional service architectures cannot be readily extended or supplemented in a decentralized fashion.

Traditional service architectures also tend to be limited in their ability to robustly evolve as services are revised and versioned. The problem with versioning is one of representing capabilities (what a piece of code can do) and identity (what a piece of code *is*) in an interrelated, fuzzy way. A later version of some piece of code, such as "Code version 2" indicates that it is *like* "Code version 1" but different in some way. The problem with traditional versioning in this manner is that it's difficult for code to indicate *exactly how* it differs from a previous version and worse yet, for clients of that code to react appropriately to new versions—or to not react at all if they expect only the previous version. The versioning problem can be reasonably managed in a traditional system when (i) there is only a single provider of a certain kind of service, (ii) the version number of the service is checked by the consumer when it binds to the service, (iii) the service is extended only in an upward-compatible manner—*i.e.*, features can only be added and never removed (a significant restriction as software evolves over a long period of time)—so that a version N provider will work with consumers of versions 1 through N-1 as well, and (iv) references to a running instance of the service are not freely passed around by consumers to other consumers, all of which may expect or require different versions. But these kind of restrictions are obviously unacceptable in a multi-vendor, distributed, modular system with polymorphic service providers.

These problems of service management, extensibility, and versioning have fed the problems stated earlier. Application complexity continues to increase as it becomes more and more difficult to extend functionality. Monolithic applications are popular because it is safer and easier to collect all interdependent services and the code that uses those services into one package. Interoperability between applications suffers accordingly, where monolithic applications are loathe to allow independent agents to access their functionality and thus build a dependence upon a certain behavior of the application. Because end users demand interoperability, however, application are compelled to attempt interoperability, but this leads directly back to the problem of application complexity, completing a circle of problems that limit the progress of software development.

1.2The Solution: Component Software

Object-oriented programming has long been advanced as a solution to the problems at hand. However, while object-oriented programming is powerful, it has yet to reach its full potential because no standard framework exists through which software objects created by different vendors can interact with one another within the same address space, much less across address spaces, and across network and machine architecture boundaries. The major result of the object-oriented programming revolution has been the production of "islands of objects" that can't talk to one another across the sea of application boundaries in a meaningful way.

The solution is a system in which application developers create reusable *software components*. A component is a reusable piece of software in binary form that can be plugged into other components from other vendors with relatively little effort. For example, a component might be a spelling checker sold by one vendor that can be plugged into several different word processing applications from multiple vendors. It

might be a math engine optimized for computing fractals. Or it might be a specialized transaction monitor that can control the interaction of a number of other components (including service providers beyond traditional database servers). Software components must adhere to a binary external standard, but their internal implementation is completely unconstrained. They can be built using procedural languages as well as object-oriented languages and frameworks, although the latter provide many advantages in the component software world.

Software component objects are much like integrated circuit (IC) components, and component software is the integrated circuit of tomorrow. The software industry today is very much where the hardware industry was 20 years ago. At that time, vendors learned how to shrink transistors and put them into a package so that no one ever had to figure out how to build a particular discrete function—an NAND gate for example—ever again. Such functions were made into an integrated circuit, a neat package that designers could conveniently buy and design around. As the hardware functions got more complex, the ICs were integrated to make a board of chips to provide more complex functionality and increased capability. As integrated circuits got smaller yet provided more functionality, boards of chips became just bigger chips. So hardware technology now uses chips to build even bigger chips.

The software industry is at a point now where software developers have been busy building the software equivalent of discrete transistors—software routines—for a long time.

The Component Object Model enables software suppliers to package their functions into reusable software components in a fashion similar to the integrated circuit. What COM and its objects do is bring software into the world where an application developer no longer has to write a sorting algorithm, for example. A sorting algorithm can be packaged as a binary object and shipped into a marketplace of component objects. The developer who need a sorting algorithm just uses any sorting object of the required type without worrying about how the sort is implemented. The developer of the sorting object can avoid the hassles and intellectual property concerns of source-code licensing, and devote total energy to providing the best possible binary version of the sorting algorithm. Moreover, the developer can take advantage of COM's ability to provide easy extensibility and innovation beyond standard services as well as robust support for versioning of components, so that a new component works perfectly with software clients expecting to use a previous version.

As with hardware developers and the integrated circuit, applications developers now do not have to worry about *how* to build that function; they can simply purchase that function. The situation is much the same as when you buy an integrated circuit today: You don't buy the sources to the IC and rebuild the IC yourself. COM allows you to simply buy the software component, just as you would buy an integrated circuit. The component is compatible with anything you "plug" it into.

By enabling the development of component software, COM provides a much more productive way to design, build, sell, use, and reuse software. Component software has significant implications for software vendors, users, and corporations:

- **Application developers** are enabled to build and distribute applications more easily than ever before. Component objects provide both scalability from single processes to enterprise networks and modularity for code reuse. In

addition, developers can attain higher productivity because they can learn one object system for many platforms.

- **Vendors** are provided with a single model for interacting with other applications and the distributed computing environment. While component software can readily be added to existing applications without fundamental rewriting, it also provides the opportunity to modularize applications and to incrementally replace system capabilities where appropriate. The advent of component software will help create more diverse market segments and niches for small, medium, and large vendors.
- **End-users** will see a much greater range of software choices, coupled with better productivity. Users will have access to hundreds of objects across client and server platforms—objects that were previously developed by independent software vendors (ISVs) and corporations. In addition, as users see the possibilities of component software, demand is likely to increase for specialized components they can purchase at a local software retail outlet and plug into applications.
- **Corporations** benefit from lower costs for corporate computing, helping IS departments work more efficiently, and enabling corporate computer users to be more productive. IS developers will spend less time developing general purpose software components and more time developing "glue" components to create business-specific solutions. Existing applications do not need to be rewritten to take advantage of a component architecture. Instead, corporate developers can create object-based "wrappers" that encapsulate the legacy application and make its operations and data available as an object to other software components in the network.

1.3The Component Software Solution: OLE's COM

The Component Object Model provides a means to address problems of application complexity and evolution of functionality over time. It is a widely available, powerful mechanism for customers to adopt and adapt to a new style multi-vendor distributed computing, while minimizing new software investment.. COM is an open standard, fully and completely publicly documented from the lowest levels of its protocols to the highest. As a robust, efficient and workable component architecture it has been proven in the marketplace as the foundation of diverse and several application areas including compound documents, programming widgets, 3D engineering graphics, stock market data transfer, high performance transaction processing, and so on.

The Component Object Model is an object-based programming model designed to promote software interoperability; that is, to allow two or more applications or "components" to easily cooperate with one another, even if they were written by different vendors at different times, in different programming languages, or if they are running on different machines running different operating systems. To support its interoperability features, COM defines and implements mechanisms that allow applications to connect to each other as *software objects.* A software object is a collection of related function (or intelligence) and the function's (or intelligence's) associated state.

In other words, COM, like a traditional system service API, provides the operations through which a client of some service can connect to multiple providers of that service in a polymorphic fashion. But once a connection is established, *COM drops out of the picture.* COM serves to connect a client and an object, but once that connection is established, the client and object communicate directly without having to suffer overhead of being forced through a central piece of API code as illustrated in Figure 1-2.

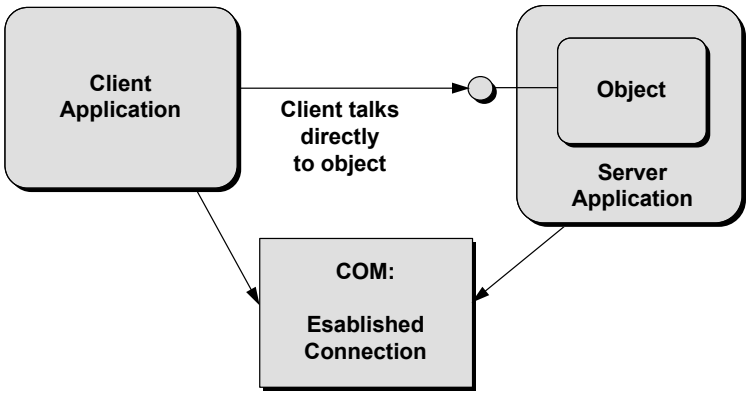

Figure 1-2: Once COM connects client and object, the client and object communicate directly without added overhead.

COM is not a prescribed way to structure an application; rather, it is a set of technologies for building robust groups of services in both systems and applications such that the services and the clients of those services can evolve over time. In this way, COM is a technology that makes the programming, use, and uncoordinated/independent evolution of binary objects *possible*. COM is not a technology designed primarily for making programming necessarily *easy*; indeed, some of the difficult requirements that COM accepts and meets necessarily involve some degree of complexity.[1] However, COM provides a ready base for extensions oriented towards increased ease-of-use, as well as a great basis for powerful, easy development environments, language-specific improvements to provide better language integration, and pre-packaged functionality within the context of application frameworks.

This is a fundamental strength of COM over other proposed object models: COM solves the "deployment problem," the versioning/evolution problem where it is necessary that the functionality of objects can incrementally evolve or change without the need to simultaneously and in lockstep evolve or change all existing the clients of the object. Objects/services can easily continue to support the interfaces through which they communicated with older clients as well as provide new and better interfaces through which they communicate with newer clients.

To solve the versioning problems as well providing connection services without undue overhead, the Component Object Model builds a foundation that:

- ∑ Enables the creation and use of reusable components by making them "component objects."
- ∑ Defines a binary standard for interoperability.
- ∑ Is a true system object model.
- ∑ Provides distributed capabilities.

The following sections describe each of these points in more detail.

[1] "Easy" is a relative term: without COM, some sorts of programming are simply not *possible* and thus the term "easy" is utterly empty.

1.3.1Reusable Component Objects

Object-oriented programming allows programmers to build flexible and powerful software objects that can easily be reused by other programmers. Why is this? What is it about objects that are so flexible and powerful?

The definition of an object is a piece of software that contains the functions that represent what the object can do (its intelligence) and associated state information for those functions (data). An object is, in other words, some data structure and some functions to manipulate that structure.

An important principle of object-oriented programming is *encapsulation*, where the exact implementation of those functions and the exact format and layout of the data is only of concern to the object itself. This information is hidden from the clients of an object. Those clients are interested only in an object's behavior and not the object's internals. For instance, consider an object that represents a stack: a user of the stack cares only that the object supports "push" and "pop" operations, not whether the stack is implemented with an array or a linked list. Put another way, a client of an object is interested only in the "contract"—the promised behavior—that the object supports, not the implementation it uses to fulfill that contract.

COM goes as far as to formalize the notion of a contract between object and client. Such a contract is the basis for interoperability, and for interoperability to work on a large scale requires a strong standard.

1.3.2Binary and Wire-Level Standards for Interoperability

The Component Object Model defines a completely standardized mechanism for creating objects and for clients and objects to communicate. Unlike traditional object-oriented programming environments, these mechanisms are independent of the applications that use object services and of the programming languages used to create the objects. The mechanisms also support object invocations across the network. COM therefore defines a *binary interoperability standard* rather than a language-based interoperability standard on any given operating system and hardware platform. In the domain of network computing, COM defines a standard architecture-independent wire format and protocol for interaction between objects on heterogeneous platforms.

Why Is Providing a Binary and Network Standard Important?

By providing a binary and network standard, COM enables interoperability among applications that different programmers from different companies write. For example, a word processor application from one vendor can connect to a spreadsheet object from another vendor and import cell data from that spreadsheet into a table in the document. The spreadsheet object in turn may have a "hot" link to data provided by a data object residing on a mainframe. As long as the objects support a predefined standard interface for data exchange, the word processor, spreadsheet, and mainframe database don't have to know anything about each other's implementation. The word processor need only know how to connect to the spreadsheet; the spreadsheet need only know how to expose its services to anyone who wishes to connect. The same goes for the network contract between the spreadsheet and the mainframe database. All that either side of a connection needs to know are the standard mechanisms of the Component Object Model.

Without a binary and network standard for communication and a standard set of communication interfaces, programmers face the daunting task of writing a large number of procedures, each of which is specialized for communicating with a different type of object or client, or perhaps recompiling their code depending on the other components or network services with which they need to interact. With a binary and network standard, objects and their clients need no special code and no recompilation for interoperability. But these standards must be efficient for use in both a single address space and a distributed environment; if the mechanism used for object interaction is not extremely efficient, especially in the case of local (same machine) servers and components within a single address space, mass-market software developers pressured by size and performance requirements simply will not use it.

Finally, object communication must be programming language-independent since programmers cannot and should not be forced to use a particular language to interact with the system and other applications. An illustrative problem is that every C++ vendor says, "We've got class libraries and you can use our class libraries." But the interfaces published for that one vendor's C++ object usually differs from the interfaces publishes for another vendor's C++ object. To allow application developers to use the objects' capabilities, each vendor has to ship the source code for the class library for the objects so that application developers can rebuild that code for the vendor's compiler they're using. By providing a binary standard to which objects conform, vendors do not have to send source code to provide compatibility, nor to users have to restrict the language they use to get access to the objects' capabilities. COM objects are compatible by nature.

COM's Standards Enable Object Interoperability

With COM, applications interact with each other and with the system through collections of function calls—also known as methods or member functions or requests—called *interfaces*. An "interface" in the COM sense[2] is a strongly typed *contract* between software components to provide a relatively small but useful set of semantically related operations. An interface is an articulation of an expected behavior and expected responsibilities, and the semantic relation of interfaces gives programmers and designers a concrete entity to use when referring to the contract. Although not a strict requirement of the model, interfaces should be factored in such fashion that they can be re-used in a variety of contexts. For example, a simple interface for generically reading and writing streams of data can be re-used by many different types of objects and clients.

The use of such interfaces in COM provides four major benefits:

1. **The ability for functionality in applications (clients or servers of objects) to evolve over time:** This is accomplished through a request called *QueryInterface* that all COM objects support (or else they are not COM objects). *QueryInterface* allows an object to make more interfaces (that is, new groups of functions) available to new clients while at the same time retaining complete binary compatibility with existing client code. In other words, revising an object by adding new, even unrelated functionality will not require any recompilation on the part of any existing clients. Because COM

[2]The term "interface" is used in a very similar sense in the Component Object Request Broker Architecture (CORBA) design of the Object Management Group. In both cases the idea of an "interface" is a signature of functions and, implicitly, capabilities, entirely abstracted from the implementation. The major difference between COM and CORBA at this high level is that CORBA objects have one and only one interface while COM objects can have many interfaces simultaneously. DCE RPC (from OSF) uses the term "interface" in a similar manner.

allows objects to have multiple interfaces, an object can express any number of "versions" simultaneously, each of which may be in simultaneous use by clients of different vintage. And when its clients pass around a reference to the "object," an occurrence that in principle cannot be known and therefore "guarded against" by the object, they actually pass a reference to a particular *interface* on the object, thus extending the chain of backward compatibility. The use of immutable interfaces and multiple interfaces per object solves the problem of versioning.

2. **Very fast and simple object interaction for same-process objects:** Once a client establishes a connection to an object, calls to that object's services (interface functions) are simply indirect functions calls through two memory pointers. As a result, the performance overhead of interacting with an in-process COM object (an object that is in the same address space) as the calling code is negligible—only a handful of processor instructions slower than a standard direct function call and no slower than a compile-time bound C++ single-inheritance object invocation.[3]
3. **"Location transparency":** The binary standard allows COM to intercept a interface call to an object and make instead a *remote procedure call* (RPC) to the "real" instance of the object that is running in another process or on another machine. A key point is that the caller makes this call exactly as it would for an object in the same process. Its binary and network standards enables COM to perform inter-process and cross-network function calls transparently. While there is, of course, a great deal more overhead in making a remote procedure call, no special code is necessary in the client to differentiate an in-process object from out-of-process objects. All objects are available to clients in a uniform, transparent fashion.[4]

   This is all well and good. But in the real world, it is sometimes necessary for performance reasons that special considerations be taken into account when designing systems for network operation that need not be considered when only local operation is used. What is needed is not pure local / remote transparency, but "local / remote transparency, unless you need to care." COM provides this capability. An object implementor can if he wishes support *custom marshaling* which allows his objects to take special action when they are used from across the network, different action if he would like than is used in the local case. The key point is that this is done completely transparently to the client. Taken as a whole, this architecture allows one to design client / object interfaces at their natural and easy semantic level without regard to network performance issues, then at a later address network performance issues without disrupting the established design.

---

[3] Indeed, in principle the intrinsic method dispatch overhead of COM is in fact *less* than the intrinsic overhead of C++ multiple inheritance method invocations. In a multiple inheritance situation, C++ must on every method invocation adjust the this pointer to be as appropriate for the actual method which is to be executed. In an COM object which supports multiple interfaces, which is directly analogous to the multiple inheritance situation, one must of course also do a similar sort of adjustment, and this is done in the QueryInterface method. However, when using a given interface on the object, one can invoke QueryInterface once and use the returned pointer many times. Thus, the cost of the QueryInterface operation can be amortized over all the subsequent usage, resulting in less overall dispatch overhead. Be aware, however, that this distinction is completely academic. In almost all real word situations, both dispatch mechanisms provide more than adequate performance.

[4] There can be subtle differences in the flow-of-control between calling in-process and out-of-process objects. In particular, an out-of-process object call may result in a call-back prior to the completion of the original call. COM provides standard mechanisms to deal with call-backs and reentrancy, even on single-threaded operating systems. Without such standards, true interoperability between out-of-process objects (of which cross-network objects is just a typical case) is impossible.

4. **Programming language independence:** Because COM is a binary standard, objects can be implemented in a number of different programming languages and used from clients that are written using completely different programming languages. Any programming language that can create structures of pointers and explicitly or implicitly call functions through pointers—languages such as C, C++, Pascal, Ada, Smalltalk, and even BASIC programming environments—can create and use COM objects immediately. Other languages can easily be enhanced to support this requirement.

In sum, only with a binary standard can an object model provide the type of structure necessary for full interoperability, evolution, and re-use between any application or component supplied by any vendor on a single machine architecture. Only with an architecture-independent network wire protocol standard can an object model provide full interoperability, evolution, and re-use between any application or component supplied by any vendor in a network of heterogeneous computers. With its binary and networking standards, COM opens the doors for a revolution in software innovation without a revolution in networking, hardware, or programming and programming tools.

1.3.3A True System Object Model

To be a true system model, an object architecture must allow a distributed, evolving system to support millions of objects without risk of erroneous connections of objects and other problems related to strong typing or definition. COM is such an architecture. In addition to being an object-based service architecture, COM is a true system object model because it:

- Uses "globally unique identifiers" to identify object classes and the interfaces those objects may support.
- Provides methods for code reusability without the problems of traditional language-style implementation inheritance.
- Has a single programming model for in-process, cross-process, and cross-network interaction of software components.
- Encapsulates the life-cycle of objects via reference counting.
- Provides a flexible foundation for security at the object level.

The following sections elaborate on each of these aspects of COM.

Globally Unique Identifiers

Distributed object systems have potentially millions of interfaces and software components that need to be uniquely identified. Any system that uses human-readable names for finding and binding to modules, objects, classes, or requests is at risk because the probability of a collision between human-readable names is nearly 100% in a complex system. The result of name-based identification will inevitably be the accidental connection of two or more software components that were not designed to interact with each other, and a resulting error or crash—even though the components and system had no bugs and worked as designed.

By contrast, COM uses globally unique identifiers (GUIDs)—128-bit integers that are virtually guaranteed to be unique in the world across space and time—to identify every

interface and every object class and type.[5] These globally unique identifiers are the same as UUIDs (Universally Unique IDs) as defined by DCE. Human-readable names are assigned only for convenience and are locally scoped. This helps insure that COM components do not accidentally connect to an object or via an interface or method, even in networks with millions of objects.[6]

Code Reusability and Implementation Inheritance

Implementation inheritance—the ability of one component to "subclass" or "inherit" some of its functionality from another component while "over-riding" other functions—is a very useful technology for building applications. But more and more experts are concluding that it creates serious problems in a loosely coupled, decentralized, evolving object system. The problem is technically known as the lack of type-safety in the specialization interface and is well-documented in the research literature.[7]

The general problem with traditional implementation inheritance is that the "contract" or interface between objects in an implementation hierarchy is not clearly defined; indeed, it is implicit and ambiguous. When the parent or child component changes its implementation, the behavior of related components may become undefined. This tight coupling of implementations is not a problem when the implementation hierarchy is under the control of a defined group of programmers who can, if necessary, make updates to all components simultaneously. But it is precisely this ability to control and change a set of related components simultaneously that differentiates an application, even a complex application, from a true distributed object system. So while traditional implementation inheritance can be a very good thing for building applications and components, it is inappropriate in a system object model.

Today, COM provides two mechanisms for code reuse called *containment/delegation* and *aggregation*. In the first and more common mechanism, one object (the "outer" object) simply becomes the client of another, internally using the second object (the "inner" object) as a provider of services that the outer object finds useful in its own implementation. For example, the outer object may implement only stub functions that merely pass through calls to the inner object, only transforming object reference parameters from the inner object to itself in order to maintain full encapsulation. This is really no different than an application calling functions in an operating system to achieve the same ends—other objects simply extend the functionality of the system. Viewed externally, clients of the outer object only ever see the outer object—the inner "contained" object is completely hidden—encapsulated—from view. And since the outer object is itself a client of the inner object, it always uses that inner object through a clearly defined contracts: the inner object's interfaces. By implementing those interfaces, the inner object signs the contract promising that it will not change its behavior unexpectedly.

---

[5] Although "class" and "type" can often be used interchangeably, in COM a "type" is the total signature of an object, which is the union of the interfaces that the object supports. "Class" is a particular implementation of a type, and can include certain unique implementation-specific attributes such as product name, icon, etc. For example, the "chart" type (identified by a GUID by whomever first defines that particular combination of interfaces) might be supported by Lotus 1-2-3 for Windows and Microsoft Excel for the Macintosh, each of which are separate classes. Normally, types are polymorphic; any consumer of the services provided by interfaces making up the type can use any class that implements the type.

[6] As an illustration of how unique GUIDs are consider that one could generate 10 million GUIDs a second until the year 5770 AD and each one would be unique.

[7] See, for example, Richard Helm (Senior Researcher, IBM Thomas J. Watson Research Center), *Ensuring Semantic Integrity of Reusable Objects (Panel)*, OOPSLA '92 Conference Proceedings, p.300; John Lamping (Xerox PARC), *Typing the Specialization Interface*, OOPSLA '93 Conference Proceedings, p.201.

With *aggregation,* the second and more rare reuse mechanism, COM objects take advantage of the fact that they can support multiple interfaces. An aggregated object is essentially a composite object in which the outer object exposes an interface from the inner object directly to clients as if it were part of the outer object. Again, clients of the outer object are impervious to this fact, but internally, the outer object need not implement the exposed interface at all. The outer object has determined that the implementation of the inner object's interface is exactly what it wants to provide itself, and can reuse that implementation accordingly. But the outer object is still a client of the inner object and there is still a clear contract between the inner object and any client. Aggregation is really nothing more than a special case of containment/delegation to prevent the outer object from having to implement an interface that does nothing more than delegate every function to the same interface in the inner object. Aggregation is really a performance convenience more than the primary method of reuse in COM.
Both these reuse mechanisms allow objects to exploit existing implementation while avoiding the problems of traditional implementation inheritance. However, they lack a powerful, if dangerous, capability of traditional implementation inheritance: the ability of a child object to "hook" calls that a parent object might make on itself and override entirely or supplement partially the parent's behavior. This feature of implementation inheritance is definitely useful, but it is also the key area where imprecision of interface and implicit coupling of *implementation* (as opposed to interface) creeps in to traditional implementation inheritance mechanisms. A future challenge for COM is to define a set of conventions that components can use to provide this "hooking" feature of implementation inheritance while maintaining the strictness of contract between objects and the full encapsulation required by a true system object model, even those in "parent/child" relationships.[8]

Single Programming Model

A problem related to implementation inheritance is the issue of a single programming model for in-process objects and out-of-process/cross-network objects. In the former case, class library technology (or application frameworks) permits only the use of features or objects that are in a single address. Such technology is far from permitting use of code outside the process space let alone code running on another machine altogether. In other words, a programmer can't subclass a remote object to reuse its implementation. Similarly, features like public data items in classes that can be freely manipulated by other objects within a single address space don't work across process or network boundaries. In contrast, COM has a single interface-based binding model and has been carefully designed to minimize differences between the in-process and out-of-process programming model. Any client can work with any object anywhere else on the machine or network, and because the object reusability mechanisms of containment and aggregation maintain a client/server relationship between objects, reusability is also possible across process and network boundaries.

[8]Readers interested in this issue should examine the "connectable object" architecture described in Chapter 11. Connectable objects enable an event model that provides a standard, powerful convention for a COM object to signal to any interested client that is about to do something, that is doing something, and that it is finished doing something. The model also allows clients to cancel the event outright or to cancel it in favor of an "overriding" event supplied by the client. This event model coupled with a few additional conventions could provide COM with all the traditional features of implementation inheritance and more without the traditional risks. For an interesting discussion of the problems of traditional implementation inheritance as well as a description of how an inheritance system might be provide robust type-safety, see Hauck, *Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance*, OOPSLA '93 Conference Proceedings, p.231.

Life-cycle Encapsulation

In traditional object systems, the life-cycle of objects—the issues surrounding the creation and deletion of objects—is handled implicitly by the language (or the language runtime) or explicitly by application programmers. In other words, an object-based application, there is always someone (a programmer or team of programmers) or something (for example, the startup and shutdown code of a language runtime) that has complete knowledge when objects must be created and when they should be deleted. But in an evolving, decentralized *system* made up of objects, it is no longer true that someone or something always "knows" how to deal with object life-cycle. Object creation is still relatively easy; assuming the client has the right security privileges, an object is created whenever a client requests that it be created. But object deletion is another matter entirely. How is it possible to "know" *a priori* when an object is no longer needed and should be deleted? Even when the original client is done with the object, it can't simply shut the object down since it is likely to have passed a reference to the object to some other client in the system, and how can it know if/when that client is done with the object?—or if that second client has passed a reference to a third client of the object, and so on.

At first, it may seem that there are other ways of dealing with this problem. In the case of cross-process and cross-network object usage, it might be possible to rely on the underlying communication channel to inform the system when all *connections* to an object have disappeared. The object can then be safely deleted. There are two drawbacks to this approach, however, one of which is fatal. The first and less significant drawback is that it simply pushes the problem out to the next level of software. The object system will need to rely on a connection-oriented communications model that is capable of tracking object connections and taking action when they disappear. That might, however, be an acceptable trade-off.

But the second drawback is flatly unacceptable: this approach requires a major difference between the cross-process/cross-network programming model, where the communication system can provide the hook necessary for life-cycle management, and the single-process programming model where objects are directly connected together without any intervening communications channel. In the latter case, object life-cycle issues must be handled in some other fashion. This lack of location transparency would mean a difference in the programming model for single-process and cross-process objects. It would also force clients to make a once-for-all compile-time decision about whether objects were going to run in-process or out-of-process instead of allowing that decision to be made by *users* of the binary component on a flexible, ad hoc basis. Finally, it would eliminate the powerful possibility of composite objects or aggregates made up of both in-process and out-of-process objects.

Could the issue simply be ignored? In other words, could we simply ignore garbage collection (deletion of unused objects) and allow the operating system to clean up unneeded resources when the process was eventually torn down? That non-"solution" might be tempting in a system with just a few objects, or in a system (like a laptop computer) that comes up and down frequently. It is totally unacceptable, however, in the case of an environment where a single process might be made up of potentially thousands of objects or in a large server machine that must never stop. In either case, lack of life-

cycle management is essentially an embrace of an inherently unstable system due to memory leaks from objects that never die.

There is only one solution to this set of problems, the solution embraced by COM: clients must tell an object when they are using it and when they are done, and objects must delete themselves when they are no longer needed. This approach, based on reference counting by all objects, is summarized by the phrase "life-cycle encapsulation" since objects are truly encapsulated and self-reliant if and only if they are responsible, with the appropriate help of their clients acting singly and not collectively, for deleting themselves.

Reference counting is admittedly complex for the new COM programmer; arguably, it is the most difficult aspect of the COM programming model to understand and to get right when building complex peer-to-peer COM applications. When viewed in light of the non-alternatives, however, its inevitability for a true system object model with full location transparency is apparent. Moreover, reference counting is precisely the kind of mechanical programming task that can be automated to a large degree or even entirely by well-designed programming tools and application frameworks. Tools and frameworks focused on building COM components exist today and will proliferate increasingly over the next few years. Moreover, the COM model itself may evolve to provide support for optionally delegating life-cycle management to the system. Perhaps most importantly, reference counting in particular and native COM programming in general involves the kind of mind-shift for programmers—as in GUI event-driven programming just a few short years ago—that seems difficult at first, but becomes increasingly easy, then second-nature, then almost trivial as experience grows.

Security

For a distributed object system to be useful in the real world it must provide a means for secure access to objects and the data they encapsulate. The issues surrounding system object models are complex for corporate customers and ISVs making planning decisions in this area, but COM meets the challenges, and is a solid foundation for an enterprise-wide computing environment.

COM provides security along several crucial dimensions. First, COM uses standard operating system permissions to determine whether a client (running in a particular user's security context) has the right to start the code associated with a particular class of object. Second, with respect to persistent objects (class code along with data stored in a persistent store such as file system or database), COM uses operating system or application permissions to determine if a particular client can load the object at all, and if so whether they have read-only or read-write access, etc. Finally, because its security architecture is based the design of the DCE RPC security architecture, an industry-standard communications mechanism that includes fully authenticated sessions, COM provides cross-process and cross-network object servers with standard security information about the client or clients that are using it so that a server can use security in more sophisticated fashion than that of simple OS permissions on code execution and read/write access to persistent data.

1.3.4Distributed Capabilities

COM supports *distributed objects*; that is, it allows application developers to split a single application into a number of different component objects, each of which can run on a different computer. Since COM provides network transparency, these applications do not appear to be located on different machines. The entire network appears to be one large computer with enormous processing power and capacity.

Many single-process object models and programming languages exist today and a few distributed object systems are available. However, none provides an identical, transparent programming model for small, in-process objects, medium out-of-process objects on the same machine, and potentially huge objects running on another machine on the network. The Component Object Model provides just such a transparent model, where a client uses an object in the same process in precisely the same manner as it would use one on a machine thousands of miles away. COM explicitly bars certain kinds of "features"—such as direct access to object data, properties, or variables—that might be convenient in the case of in-process objects but would make it impossible for an out-of-process object to provide the same set of services. This is called *location transparency*.

1.4Objects and Interfaces

What is an object? An object is an instantiation of some *class*. At a generic level, a "class" is the definition of a set of related data and capabilities grouped together for some distinguishable common purpose. The purpose is generally to provide some service to "things" outside the object, namely clients that want to make use of those services.

A object that conforms to COM is a special manifestation of this definition of object. A COM object appears in memory much like a C++ object. Unlike C++ objects, however, a client never has direct access to the COM object in its entirety. Instead, clients always access the object through clearly defined contracts: the interfaces that the object supports, *and only those interfaces*.

What exactly is an interface? As mentioned earlier, an interface is a strongly-typed group of semantically-related functions, also called "interface member functions." The name of an interface is always prefixed with an "I" by convention, as in IUnknown. (The real identity of an interface is given by its GUID; names are a programming convenience, and the COM system itself uses the GUIDs exclusively when operating on interfaces.) In addition, while the interface has a specific name (or type) and names of member functions, it defines only how one would use that interface and what behavior is expected from an object through that interface. Interfaces do not define any implementation. For example, a hypothetical interface called IStack that had member functions of Push and Pop would only define the parameters and return types for those functions and what they are expected to do from a client perspective; the object is free to implement the interface as it sees fit, using an array, linked list, or whatever other programming methods it desires.

When an object "implements an interface" that object implements each member function of the interface and provides pointers to those functions to COM. COM then makes those functions available to any client who asks. This terminology is used in this document to refer to the object as the important element in the discussion. An equivalent term is an "interface on an object" which means the object implements the interface but the main subject of discussion is the interface instead of the object.

1.4.1Attributes of Interfaces

Given that an interface is a contractual way for an object to expose its services, there are four very important points to understand:

An interface is not a class: An interface is not a class in the normal definition of "class." A class can be instantiated to form an object. An interface cannot be instantiated by itself because it carries no implementation. An object must implement that interface and that object must be instantiated for there to be an interface. Furthermore, different object classes may implement an interface differently yet be used interchangeably in binary form, so long as the behavior conforms to the interface definition (such as two objects that implement IStack where one uses an array and the other a linked list).

An interface is not an object: An interface is just a related group of functions and is the binary standard through which clients and objects communicate. The object can be implemented in any language with any internal state representation, so long as it can provide pointers to interface member functions.

Interfaces are strongly typed: Every interface has its own interface identifier (a GUID) thereby eliminating any chance of collision that would occur with human-readable names. Programmers must consciously assign an identifier to each interface and must consciously support that interface and/or the interfaces defined by others: confusion and conflict among interfaces cannot happen by accident, leading to much improved robustness.

Interfaces are immutable: Interfaces are never versioned, thus avoiding versioning problems. A new version of an interface, created by adding or removing functions or changing semantics, is an entirely new interface and is assigned a new unique identifier. Therefore a new interface does not conflict with an old interface even if all that changed is the semantics. Objects can, of course, support multiple interfaces simultaneous; and they can have a single internal implementation of the common capabilities exposed through two or more similar interfaces, such as "versions" (progressive revisions) of an interface. This approach of immutable interfaces and multiple interfaces per object avoids versioning problems.

Two additional points help to further reinforce the second point about the relationship of an object and its interfaces:

Clients only interact with pointers to interfaces: When a client has access to an object, it has nothing more than a pointer through which it can access the functions in the interface, called simply an *interface pointer*. The pointer is opaque, meaning that it hides all aspects of internal implementation. You cannot see any details about the object such as its state information, as opposed to C++ *object pointers* through which a client may directly access the object's data. In COM, the client can only call functions of the interface to which it has a pointer. But instead of being a restriction, this is what allows COM to provide the efficient binary standard that enables location transparency.

Objects can implement multiple interfaces: A object class can—and typically does—implement more than one interface. That is, the class has more than one set of services to provide from each object. For example, a class might support the ability to exchange data with clients as well as the ability to save its persistent state information (the data it would need to reload to return to its current state) into a file at the client's request. Each of these

abilities is expressed through a different interface, so the object must implement two interfaces.

Note that just because a class supports one interface, there is no general requirement that it supports any other. Interfaces are meant to be small contracts that are independent of one another. There are no contractual units smaller than interfaces; if you write a class that implements an interface, your class must implement all the functions defined by that interface (the implementation doesn't always have to *do* anything). Also note that an object may be attempting to conform to a higher specification than COM, such as a compound document standard like Microsoft's OLE Documents architecture. In such cases, the objects in question must implement specific groups of interfaces to conform to the OLE Documents specification for compound documents. It is then true that all compound document objects will always implement the same basic set of interfaces, but those interfaces themselves do not depend on the presence of the others. It is instead the clients of those objects that depend on the presence of all the interfaces.

The encapsulation of functionality into objects accessed through interfaces makes COM an open, extensible system. It is open in the sense that anyone can provide an implementation of a defined interface and anyone can develop an application that uses such interfaces, such as a compound document application. It is extensible in the sense that new or extended interfaces can be defined without changing existing applications and those applications that understand the new interfaces can exploit them while continuing to interoperate with older applications through the old interfaces.

1.4.2Object Pictures

It is convenient to adopt a standard pictorial representation for objects and their interfaces. The adopted convention is to draw each interface on an object as a "plug-in jack." These interfaces are generally drawn out the left or right side of a box representing the object as a whole as illustrated in Figure 1-3. If desired, the names of the interfaces are positioned next to the interface jack itself.

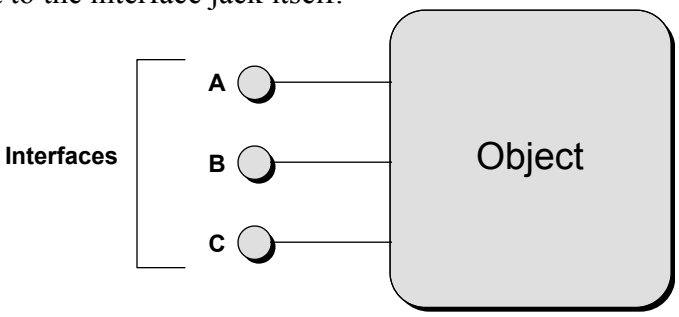

Figure 1-3: A typical picture of an object that supports three interfaces A, B, and C.

The side from which interfaces extend is usually determined by the position of a client in the same picture, if applicable. If there is no client in the picture then the convention is for interfaces to extend to the left as done in Figure 1-3. With a client in the picture, the interfaces extend towards the client, and the client is understood to have a pointer to one or more of the interfaces on that object as illustrated in Figure 1-4.

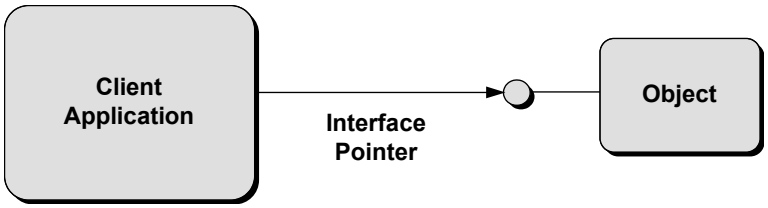

Figure 1-4: Interfaces extend towards the clients connected to them.

In some circumstances a client may itself implement a small object to provide another object with functions to call on various events or to expose services itself. In such cases the client is also an object implementor and the object is also a client. Illustrations for such are similar to that in Figure 1-5.

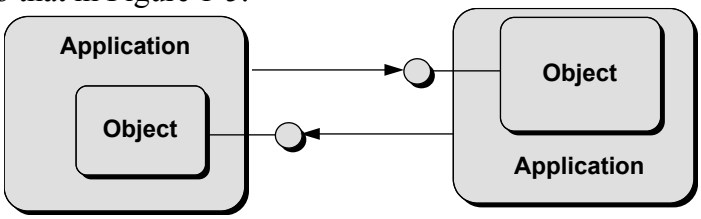

Figure 1-5: Two applications may connect to each other's objects, in which case they extend their interfaces towards each other.

Some objects may be acting as an intermediate between other clients in which case it is reasonable to draw the object with interfaces out both sides with clients on both sides. This is, however, a less frequent case than illustrating an objects connected to one client. There is one interface that demands a little special attention: IUnknown. This is the base interface of all other interfaces in COM that all objects must support. Usually by implementing any interface at all an object also implements a set of IUnknown functions that are contained within that implemented interface. In some cases, however, an object will implement IUnknown by itself, in which case that interface is extended from the top of the object as shown in Figure 1-6.

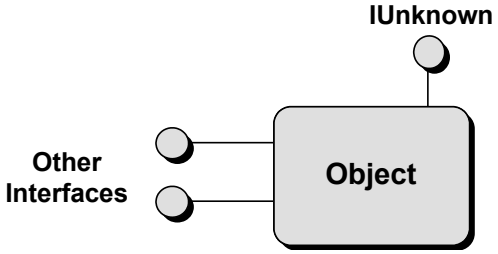

Figure 1-6: The IUnknown interface extends from the top of objects by convention.

In order to use an interface on a object, a client needs to know what it would want to do with that interface—that's what makes it a client of an interface rather than just a client of the object. In the "plug-in jack" concept, a client has to have the right kind of plug to fit into the interface jack in order to do anything with the object through the interface. This is like having a stereo system which has a number of different jacks for inputs and outputs, such as a ¼" stereo jack for headphones, a coax input for an external CD player, and standard RCA connectors for speaker output. Only headphones, CD players, and speakers that have the matching plugs are able to plug into the stereo object and make use of its services. Objects and interfaces in COM work the same way.

1.4.3Objects with Multiple Interfaces and QueryInterface

In COM, an object can support multiple interfaces, that is, provide pointers to more than one grouping of functions. Multiple interfaces is a fundamental innovation of COM as the ability for such avoids versioning problems (interfaces are immutable as described earlier) and any strong association between an interface and an object class. Multiple interfaces is a great improvement over systems in which each object only has one massive interface, and that interface is a collection of everything the object does. Therefore the identity of the object is strongly tied to the exact interface, which introduces the versioning problems once again. Multiple interfaces is the cleanest way around the issue altogether.

The existence of multiple interfaces does, however, bring up a very important question. When a client initially gains access to an object, by whatever means, that client is given *one and only one* interface pointer in return. How, then, does a client access the other interfaces on that same object?

The answer is a member function called QueryInterface that is present in all COM interfaces and can be called on any interface polymorphically. QueryInterface is the basis for a process called *interface negotiation* whereby the client asks the object what services it is capable of providing. The question is asked by calling QueryInterface and passing to that function the unique identifier of the interface representing the services of interest.

Here's how it works: when a client initially gains access to an object, that client will receive at minimum an IUnknown interface pointer (the most fundamental interface) through which it can only control the lifetime of the object—tell the object when it is done using the object—and invoke QueryInterface. The client is programmed to ask each object it manages to perform some operations, but the IUnknown interface has no functions for those operations. Instead, those operations are expressed through other interfaces. The client is thus programmed to negotiate with objects for those interfaces. Specifically, the client will ask each object—by calling QueryInterface—for an interface through which the client may invoke the desired operations.

Now since the object implements QueryInterface, it has the ability to accept or reject the request. If the object accepts the client's request, QueryInterface returns a new pointer to the requested interface to the client. Through that interface pointer the client thus has access to the functions in that interface. If, on the other hand, the object rejects the client's request, QueryInterface returns a null pointer—an error—and the client has no pointer through which to call the desired functions. An illustration of both success and error cases is shown in Figure 1-7 where the client initially has a pointer to interface A and asks for interfaces B and C. While the object supports interface B, it does not support interface C.

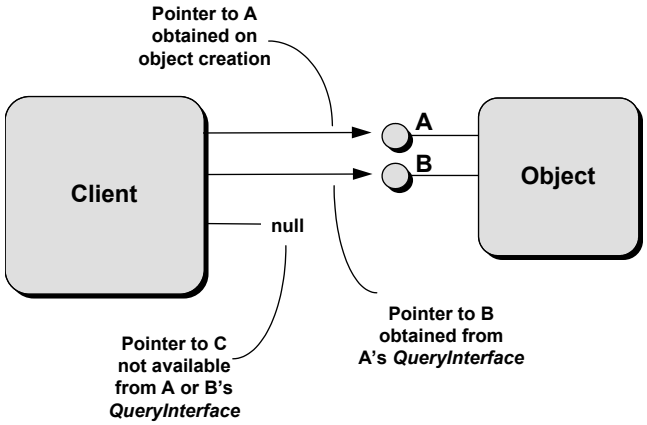

Figure 1-7: Interface negotiation means that a client must ask an object for an interface
pointer that is the only way a client can invoke functions of that interface.

A key point is that when an object rejects a call to QueryInterface, it is impossible for the client to ask the object to perform the operations expressed through the requested interface. A client *must* have an interface pointer to invoke functions in that interface, period. If the object refuses to provide one, a client must be prepared to do without, simply failing whatever it had intended to do with that object. Had the object supported that interface, the client might have done something useful with it. Compare this with other object-oriented systems where you cannot know whether or not a function will work until you call that function, and even then, handling of failure is uncertain. QueryInterface provides a reliable and consistent way to know before attempting to call a function.

Robustly Evolving Functionality Over Time

Recall that an important feature of COM is the ability for functionality to evolve over time. This is not just important for COM, but important for all applications. QueryInterface is the cornerstone of that feature as it allows a client to ask an object "do you support functionality X?" It allows the client to implement code that will use this functionality *if and only if* an object supports it. In this manner, the client easily maintains compatibility with objects written before and after the "X" functionality was available, and does so in a robust manner. An old object can reliably answer the question "do you support X" with a "no" whereas a new object can reliably answer "yes." Because the question is asked by calling QueryInterface and therefore on a contract-by-contract basis instead of an individual function-by-function basis, COM is very efficient in this operation.

To illustrate the QueryInterface cornerstone, imagine a client that wishes to display the contents of a number of text files, and it knows that for each file format (ASCII, RTF, Unicode, etc.) there is some object class associated with that format. Besides a basic interface like IUnknown, which we'll call interface A, there are two others that the client wishes to use to achieve its ends: interface B allows a client to tell an object to load some information from a file (or to save it), and interface C allows a client to request a graphical rendering of whatever data the object loaded from a file and maintains internally.

With these interfaces, the client is then programmed to process each file as follows:

1. Find the object class associated with a the file format.
2. Instantiate an object of that class obtaining a pointer to a basic interface A in return.
3. Check if the object supports loading data from a file by calling interface A's QueryInterface function requesting a pointer to interface B. If successful, ask the object to load the file through interface B.
4. Check if the object supports graphical rendering of its data by calling interface A or B's Querynterface function (doesn't matter which interface, because queries are uniform on the object) requesting a pointer to interface C. If successful, ask the object for a graphic of the file contents that the client then displays on the screen.

If an object class exists for every file format in the client's file list, and all those objects implement interfaces A, B, and C, then the client will be able to display all the contents of all the files. But in an imperfect world, let's say that the object class for the ASCII text formats does not support interface C, that is, the object can load data from a file and save it to another file if necessary, but can't supply graphics. When the client code, written as described above, encounters this object, the QueryInterface for interface C fails, and the client cannot display the file contents. Oh well...

Now the programmers of the object class for ASCII realizes that they are losing market share because they don't support graphics, and so they update the object class such that it now supports interface C. This new object is installed on the machine alone with the client application, but nothing else changes in the entire system. The client code remains exactly the same. What now happens the next time someone runs the client?

The answer is that the client *immediately begins to use interface C on the updated object*. Where before the object failed QueryInterface when asked for interface C, it now succeeds. Because it succeeds, the client can now display the contents of the file that it previously could not.

Here is the raw power of QueryInterface: a client can be written to take advantage of as much functionality as it would *ideally* like to use on every object it manages. When the client encounters an object that lacks the ideal support, the client can use as much functionality as is available on that given object. When the object it later updated to support new interfaces, the same exact code in the client, without any recompilation, redeployment, or changes whatsoever, automatically begins to take advantage of those additional interfaces. This is true component software. This is true evolution of components independently of one another and retaining full compatibility.

Note that this process also works in the other direction. Imagine that since the client application above was shipped, all the objects for rendering text into graphics were each upgraded to support a new interface D through which a client might ask the object to spell-check the text. Each object is upgraded independently of the client, but since the client never queries for interface D, the objects all continue to work perfectly with just interfaces B and C. In this case the objects support more functionality than the client, but still retain full compatibility requiring absolutely no changes to the client. The client, at a later date, might then implement code to use interface D as well as code for yet a newer interface E (that supports, say, language translation). That client begins to immediately use interface D in all existing objects that support it, without requiring any changes to those objects whatsoever.

This process continues, back and forth, ad infinitum, and applies not only to new interfaces with new functionality but also to improvements of existing interfaces. Improved interface are, for all practical purposes, a brand-new interface because any change to any interface requires a new interface identifier. A new identifier isolates an improved interface from its predecessor as much as it isolates unrelated interfaces from each other. There is no concept of "version" because the interfaces are totally different in identity.

So up to this point there has been this problem of versioning, presented at the beginning of this chapter, that made independent evolution of clients and objects practically impossible. But now, for all time, QueryInterface solves that problem and removes the barriers to rapid software innovation without the growing pains.

1.5Clients, Servers, and Object Implementors

The interaction between objects and the users of those objects in COM is based on a client/server model. This chapter has already been using the term 'client' to refer to some piece of code that is using the services of an object. Because an object supplies services, the implementor of that object is usually called the "server," the one who serves those capabilities. A client/server architecture in any computing environment leads to greater robustness: if a server process crashes or is otherwise disconnected from a client, the client can handle that problem gracefully and even restart the server if necessary. As robustness is a primary goal in COM, then a client/server model naturally fits.

However, there is more to COM than just clients and servers. There are also *object implementors*, or some program structure that implements an object of some kind with one or more interfaces on that object. Sometimes a client wishes to provide a mechanism for an object to call back to the client when specific events occur. In such cases, COM specifies that the client itself implements an object and hands that object's first interface pointer to the other object outside the client. In that sense, both sides are clients, both sides are servers in some way. Since this can lead to confusion, the term "server" is applied in a much more specific fashion leading to the following definitions that apply in all of COM:

Object A unit of functionality that implements one or more interfaces to expose that functionality. For convenience, the word is used both to refer to an object class as well as an individual instantiation of a class. Note that an object class does not need a class identifier in the COM sense such that other applications can instantiate objects of that class—the class used to implement the object internally has no bearing on the externally visible COM class identifier.

Object Implementor Any piece of code, such as an application, that has implemented an object with any interfaces for any reason. The object is simply a means to expose functions outside the particular application such that outside agents can call those functions. Use of "object" by itself implies an object found in some "object implementor" unless stated otherwise.

Client There are two definitions of this word. The general definition is any piece of code that is using the services of some object, wherever that object might be implemented. A client of this sort is also called an "object user." The second definition is the active agent (an application) that drives the flow of operation between itself an other objects and uses specific COM "implementation

locator" services to instantiate or create objects through servers of various object classes.

Server A piece of code that structures an object class in a specific fashion and assigns that class a COM class identifier. This enables a client to pass the class identifier to COM and ask for an object of that class. COM is able to load and run the server code, ask the sever to create an object of the class, and connect that new object to the client. A server is specifically the necessary structure around an object that serves the object to the rest of the system and associates the class identifier: a server is not the object itself. The word "server" is used in discussions to emphasize the serving agent more than the object. The phrase "server object" is used specifically to identify an object that is implemented in a server when the context is appropriate.

Putting all of these pieces together, imagine a client application that initially uses COM services to create an object of a particular class. COM will run the server associated with that class and have it create an object, returning an interface pointer to the client. With that interface pointer the client can query for any other interface on the object. If a client wants to be notified of events that happen in the object in the server, such as a data change, the client itself will implement an "event sink" object and pass the interface pointer to that sink to the server's object through an interface function call. The server holds onto that interface pointer and thus itself becomes a client of the sink object. When the server object detects an appropriate event, it calls the sink object's interface function for that even. The overall configuration created in this scenario is much like that shown earlier in Figure 1-5. There are two primary modules of code (the original client and the server) who both implement objects and who both act in some aspects as clients to establish the configuration.

When both sides in a configuration implement objects then the definition of "client" is usually the second one meaning the active agent who drives the flow of operation between all objects, even when there is more than one piece of code that is acting like a client of the first definition. This specification endeavors to provide enough context to make it clear what code is responsible for what services and operations.

1.5.1Server Flavors: In-Process and Out-Of-Process

As defined in the last section, a "server" in general is some piece of code that structures some object in such a way that COM "implementor locator" services can run that code and have it create objects. The section below entitled "The COM Library" expands on the specific responsibilities of COM in this sense.

Any specific server can be implemented in one of a number of flavors depending on the structure of the code module and its relationship to the client process that will be using it. A server is either "in-process" which means it's code executes in the same process space as the client, or "out-of-process" which means it runs in another process on the same machine or in another process on a remote machine. These three types of servers are called "in-process," "local," and "remote" as defined below:

In-Process Server A server that can be loaded into the client's process space and serves "in-process objects." Under Microsoft Windows and Microsoft Windows NT, these are implemented as "dynamic link libraries" or DLLs. This specification uses **DLL** as a generic term to describe any piece of code

that can be loaded in this fashion which will, of course, differ between operating systems.

Local Server A server that runs in a separate process on the same machine as the client and serves "local objects." This type of server is another complete application of its own thus defining the separate **process**. This specification uses the terms "**EXE**" or "**executable**" to describe an application that runs in its own process as opposed to a DLL which must be loaded into an existing process.

Remote Server A server that runs on a separate machine and therefore always runs in another process as well to serve "**remote** objects." Remote servers may be implemented in either DLLs or EXEs; if a remote server is implemented in a DLL, a surrogate process will be created for it on the remote machine.

Note that the same words "in-process," "local," and "remote" are used in this specification as a qualifier for the word "object" where emphasis is on the object more than the server.

Object implementors choose the type of server based on the requirements of implementation and deployment. COM is designed to handle all situations from those that require the deployment of many small, lightweight in-process objects (like controls, but conceivably even smaller) up to those that require deployment of a huge central corporate database server. Furthermore, COM does so in a transparent fashion, with what is called *location transparency,* the topic of the next section.

1.5.2Location Transparency

COM is designed to allow clients to *transparently* communicate with objects regardless of where those objects are running, be it the same process, the same machine, or a different machine. What this means is that there is a *single programming model* for all types of objects for not only clients of those objects but also for the servers of those objects.

From a client's point of view, all objects are access through interface pointers. A pointer must be in-process, and in fact, any call to an interface function always reaches *some* piece of in-process code first. If the object is in-process, the call reaches it directly, with no intervening system-infrastructure code. If the object is out-of-process, then the call first reaches what is called a "proxy" object provided by COM itself which generates the appropriate remote procedure call to the other process or the other machine.

From a server's point of view, all calls to an object's interface functions are made through a pointer to that interface. Again, a pointer only has context in a single process, and so the caller must always be some piece of in-process code. If the object is in-process, the caller is the client itself. Otherwise, the caller is a "stub" object provided by COM that picks up the remote procedure call from the "proxy" in the client process and turns it into an interface call to the server object.

As far as both clients and servers know, they always communicate directly with some other in-process code as illustrated in Figure 1-8.

The bottom line is that *dealing with in-process or remote objects is transparent and identical to dealing with in-process objects*. This location transparency has a number of key benefits:

- **A common solution to problems that are independent of the distance between client and server**: For example, connection, function invocation, interface negotiation, feature evolution, and so forth.
- **Programmers leverage their learning**: New services are simply exposed through new interfaces, and once programmers learn how to deal with interfaces, they already know how to deal with new services that will be created in the future. This is a great improvement over environments where each service is exposed in a completely different fashion.
- **Systems implementation is centralized**: The implementors of COM can focus on making the central process of providing this transparency as efficient and powerful as possible such that every piece of code that uses COM benefits immensely.
  - **Interface designers focus on design**: In designing a suite of interfaces, the designers can spend their time in the essence of the design—the contracts between the parties—without having to think about the underlying communication mechanisms for any interoperability scenario. COM provides those mechanisms for free and transparently.

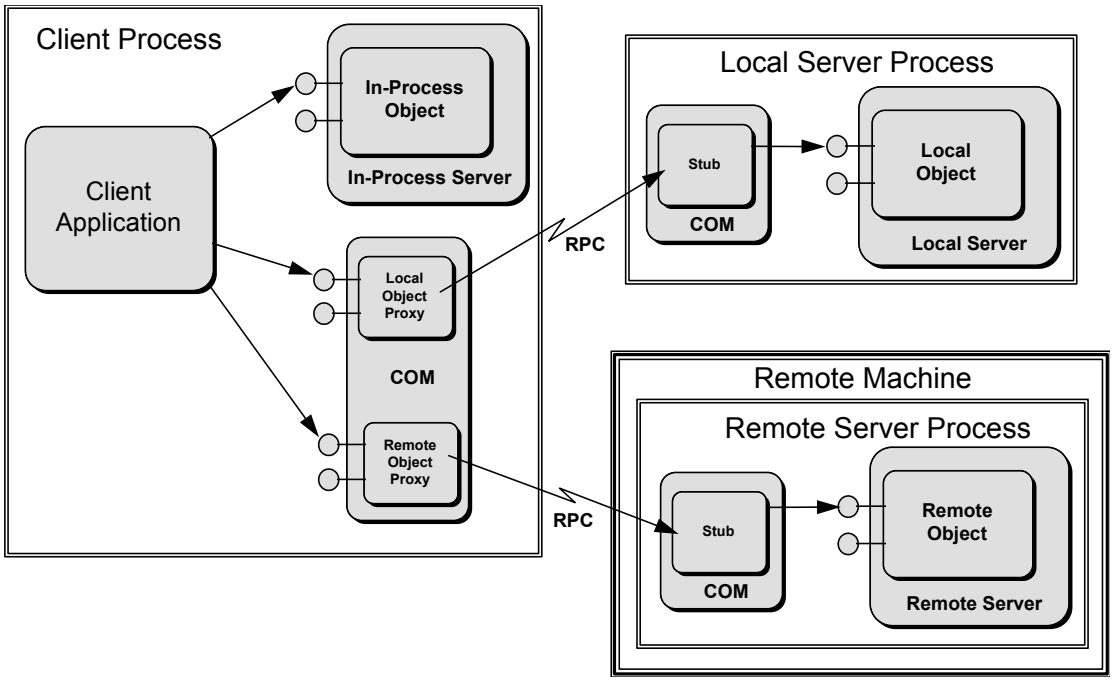

Figure 1-8: Clients always call in-process code; objects are always called by in-process code. COM provides the underlying transparent RPC.

The clear separation of interface from implementation provided by location transparency for some situations gets in the way when performance is of critical concern. When designing an interface while focusing on making it natural and functional from the client's point of view, one is sometimes lead to design decisions that are in tension with allowing for efficient implementation of that interface across a network. What is needed is not pure location transparency, but "location transparency, unless you need to care." COM provides this capability. An object implementor can if he wishes support *custom*

marshaling which allows his objects to take special action when they are used from across the network, different action if he would like than is used in the local case. The key point is that this is done completely transparently to the client. Taken as a whole, this architecture allows one to design client / object interfaces at their natural and easy semantic level without regard to network performance issues, then at a later address network performance issues without disrupting the established design.[9]
Also note again that COM is not a specification for how applications are structured: it is a specification for how applications interoperate. For this reason, COM is not concerned with the internal structure of an application—that is the job of programming languages and development environments. Conversely, programming environments have no set standards for working with objects outside of the immediate application. C++, for example, works extremely well to work with objects inside an application, but has no support for working with objects outside the application. Generally all other programming languages are the same in this regard. Therefore COM, through language-independent interfaces, picks up where programming languages leave off to provide the network-wide interoperability.

1.6The COM Library

It should be clear by this time that COM itself involves some systems-level code, that is, some implementation of its own. However, at the core the Component Object Model by itself is a specification (hence "Model") for how objects and their clients interact through the binary standard of interfaces. As a specification it defines a number of other standards for interoperability:

- The fundamental process of interface negotiation through QueryInterface.
- A *reference counting* mechanism through objects (and their resources) are managed even when connected to multiple clients.
- Rules for memory allocation and responsibility for those allocations when exchanged between independently developed components.
- Consistent and rich error reporting facilities.

In addition to being a specification, COM is also an implementation contained what is called the "COM Library." The implementation is provided through a library (such as a DLL on Microsoft Windows) that includes:

- A small number of fundamental API functions that facilitate the creation of COM applications, both clients and servers. For clients, COM supplies basic object creation functions; for servers the facilities to expose their objects.
- Implementation locator services through which COM determines from a class identifier which server implements that class and where that server is located. This includes support for a level of indirection, usually a system registry, between the identity of an object class and the packaging of the implementation such that clients are independent of the packaging which can change in the future.
- Transparent remote procedure calls when an object is running in a local or remote server, as illustrated in Figure 1-8 in the previous section.
- A standard mechanism to allow an application to control how memory is allocated within its process.

[9] Not only are there situations where there is a need for designs optimized for cross network efficiency, but there are also cases where in-process efficiency is more important. Just as COM provides mechanisms whereby the remote case can be optimized (custom marshaling) it also allows for the design of interfaces that are optimized for the in-process case.

In general, only one vendor needs to, or should, implement a COM Library for any particular operating system. For example, Microsoft has implemented COM on Microsoft Windows 3.1, Microsoft Windows 95, Microsoft Windows NT, and the Apple Macintosh. Part V of this document specifies in detail the internals of the COM Library for those vendors who wish to implement the COM Library on a platform for which it does not already have support.

1.7COM as a Foundation

The binary standard of interfaces is the key to COM's extensible architecture, providing the foundation upon which is built the rest of COM and other systems such as OLE.

1.7.1COM Infrastructure

COM provides more than just the fundamental object creation and management facilities: it also builds an infrastructure of three other core operating system components.

Persistent Storage: A set of interfaces and an implementation of those interfaces that create structured storage, otherwise known as a "file system within a file." Information in a file is structured in a hierarchical fashion which enables sharing storage between processes, incremental access to information, transactioning support, and the ability for any code in the system to browse the elements of information in the file. In addition, COM defines standard "persistent storage" interfaces that objects implement to support the ability to save their persistent state to permanent, or persistent, storage devices such that the state of the object can be restored at a later time.

Persistent, Intelligent Names (Monikers): The ability to give a specific *instantiation* of an object a particular name that would allow a client to reconnect to that *exact same object instance with the same state* (not just another object of the same class) at a later time. This also includes the ability to assign a name to some sort of *operation*, such as a query, that could be repeatedly executed using only that name to refer to the operation. This level of indirection allows changes to happen behind the name without requiring any changes to the client that stores that particular name. This technology is centered around a type of object called a *moniker* and COM defines a set of interfaces that moniker objects implement. COM also defines a standard *composite moniker* that is used to create complex names that are built of simpler monikers. Monikers also implement one of the persistent storage interfaces meaning that they know how to save their name or other information to somewhere permanent. Monikers are "intelligent" because they know how to take the name information and somehow relocate the specific object or perform an operation to which that name refers.[10]

Uniform Data Transfer: Standard interfaces through which data is exchanged between a client and an object and through which a client can ask an object to send notification (call event functions in the client) in case of a data change. The standards include powerful structures used to describe data formats as well as the storage mediums on which the data is exchanged.

The combination of the foundation and the infrastructure COM components reveals a system that describes how to create and communicate with objects, how to store them, how to label to them, and how to exchange data with them. These four aspects of COM form the core of information management. Furthermore, the infrastructure components

[10] Monikers are COM's way of providing support for what other object systems (e.g. CORBA) call persistent interfaces.

not only build on the foundation, but monikers and uniform data transfer also build on storage as shown in Figure 1-9. The result is a system that is not only very rich, but also deep, which means that work done in an application to implement lower level features is leveraged to build higher level features.

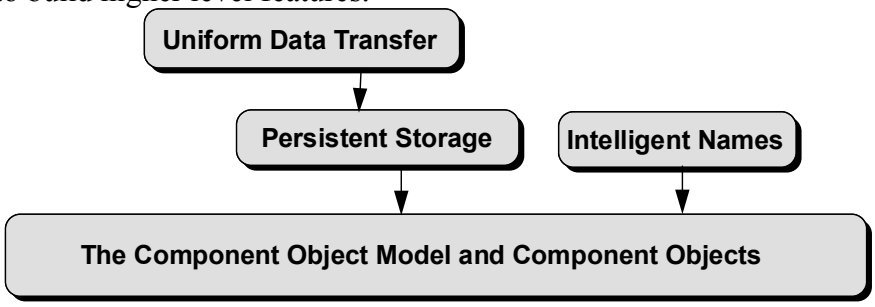

Figure 1-9: COM is built in progressively higher level technologies that depend upon lower level technologies.

1.7.2OLE

Microsoft's OLE technology is really a collection of additional higher-level technologies that build upon COM and its infrastructure. OLE version 2.0 was the first deployment of a subset of this COM specification that included support for in-process and local objects and all the infrastructure technologies but did not support remote objects. OLE 2 includes mostly user-interface oriented features based on usability, application integration, and automation of tasks. All of these features are implemented by means of specific interfaces on different objects and defined sequences of operation in both clients and servers and their relationships and dependencies on the lower level infrastructure of COM is shown in Figure 1-10.

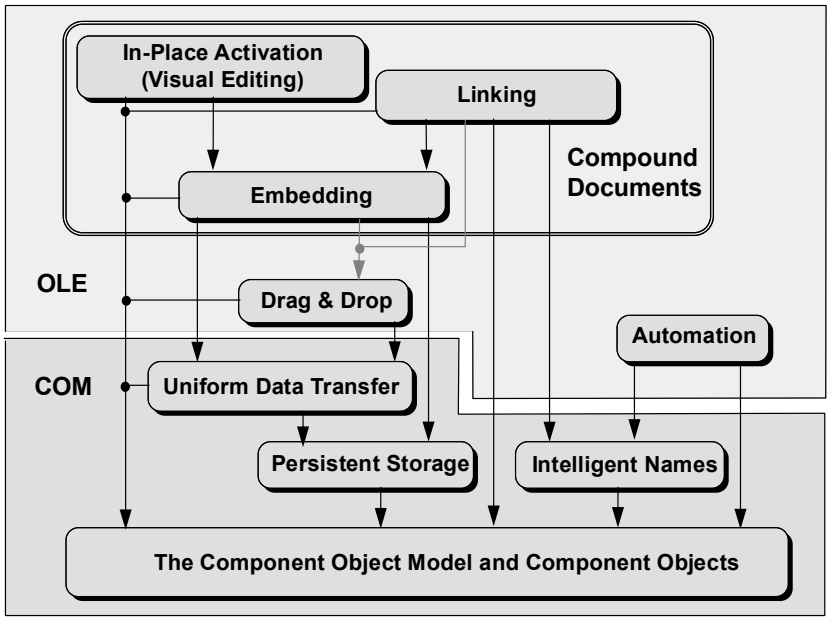

Figure 1-10: OLE builds its features on COM.

Drag & Drop: The ability to exchange data by picking up a selection with the mouse and visibly dropping it onto another window.

Automation: The ability to create "programmable" applications that can be driven externally from a script running in another application to automate common end user tasks. Automation enables cross-application macro programming.
Compound Documents: The ability to embed or link information in a central document encouraging a more document-centric user interface. Also includes In-Place Activation (also called "Visual Editing") as a user interface improvement to embedding where the end user can works on information from different applications in the context of the compound document without having to switch to other windows.
Microsoft in cooperation with other vendors is continuing to enhance OLE with new interfaces to extend compound documents and to define architectures for creating components such as OLE Controls, OLE DB, OLE for Design & Modeling, OLE for Healthcare, and in the future more system-level OLE architectures that build not only on the COM infrastructure but also on the rest of OLE as well. Again, the key is leveraged work: by implementing lower level features in an application you create a strong base of reusable code for higher level features.

This page left intentionally blank.