Project DOE: Future Solaris Technologies

16

Introduction

Project DOE *(Distributed Objects Everywhere*) is a major SunSoft initiative to bring the leverage of open systems to the application arena; greatly increasing the same open systems benefits achieved in Sun's hardware and SVR4 products. With true network transparency across heterogenous machines, fully scalable object sizes and industry-wide standard interfaces, DOE extends the Solaris model with the full range of capabilities, services and partnerships required by an enterprise-ready, distributed object environment. Starting with ToolTalk, DOE is being delivered in stages to insure that Solaris' evolution to a fully distributed object platform preserves the customer's investments of today while insuring them the advantages of tomorrow. The promise of DOE is clear; distributed object computing will provide significant and far reaching advantages over the traditional monolithic computing model of today.

SunSoft's major motivation in developing DOE is to aid in reducing the high cost of ownership associated with enterprise computing. Reducing that cost extends beyond migrating mainframes to lower cost hardware systems (or *rightsizing*). It includes reducing the cost of development, integration, administration and maintenance as well as embracing new and better ways of managing and sharing information.

DOE enables developer innovation, making it possible to build and sustain new, highly functional applications, and to easily construct successive generations of those applications on the foundations that precede them. DOE provides a software framework for a wide spectrum of solutions and supports standards-based, interoperability within heterogeneous computing environments.

Standards Driven Innovation

The Object Management Group (OMG) was founded by SunSoft and 10 other companies in April, 1989, specifically to define the standards required to facilitate distributed object applications development and use in heterogeneous environments. The OMG now includes close to 300 members - including platform vendors, object-oriented database vendors, software tool developers, corporate developers and software application vendors.

The goal of the OMG is the definition of the Object Management Architecture (OMA), comprising four major components: the Object Request Broker (ORB), Object Services, Common Facilities, and Application Objects.

The the Object Request Broker (ORB) provides basic object communications and management services, thus forming the basis of distributed object computing. SunSoft's implementation of the ORB, known as the Distributed Object Management Facility (DOMF), forms the core of DOE.

SunSoft is currently working on systems that will fill the needs of the other areas of the OMA. SunSoft's implementation of Object Services is currently under consideration by the OMG for standards adoption.

The Benefits of Distributed Objects

The DOE environment enables SunSoft to bridge the existing gap between operating system and application; further blurring that line. By supporting standards such as OMG OMA along with the overall framework provided by DOE the commercial customer will realize a great many benefits. Below are a partial list of benefits that the DOE framework provides:

- Greater Functionality

  With standardized interfaces as a core architectural feature, application developers can rely on existing inventories of objects to provide base level application functionality. This rich application framework allows developers to concentrate on the unique aspects of the application and not dilute the development effort with collateral but important additional work. Developers can now make much better use of the efforts of others and truly focus the majority of their development effort on the high value aspects of the application.

- Interoperability

  Interoperability between existing applications is greatly simplified. By using standardized interfaces, applications have one way in which to talk to one another, fully transparent to the network. The implementation structure of an application is shielded from the rest of the system and cannot interfere with the interfacing mechanism.

- Extensibility

  Because the interfaces of objects are separate from their implementation, existing applications can extend their functionality by substituting key application components with either newly developed objects or by drawing from an inventory of existing objects; new and more functional software segments can be substituted in place of less functional components extending the utility of the application. Since the interface structure remains the same, the engineering efforts are limited to substituting new objects for old; greatly reducing integration test requirements so onerously expensive in non-object-based systems.

- Supportability

  Distributed object applications can be structured so that key functional components are readily identifiable and easily modified. By taking advantage of an object's network transparency and its modularity, systems can be architected so that critical application components can be updated easily to reflect the changing structure of the system or to fix a bug.

- Reduced Development Cost

  Assembling new applications out of an inventory of existing or purchased components greatly reduces the cost of new development. Moreover, once an object is developed and tested, it need ever be tested again.

- Transportable

  Distributed object applications span the network easily and transparently. Whether the object is local or on another continent, the application uses that object in the same way. Complex network interfaces are no longer required, due to powerful, transparent mechanisms within the DOMF which manage global naming, location and communication of objects.

- Language Independence

  Distributed object applications are not dependent on any specific computer language. Objects can interoperate with any other object regardless of its language of implementation. Because an object's interface is defined via the OMG Interface Definition Language (IDL) any computer language that has IDL support can be used. SunSoft has made an IDL compiler front end available for language developers to incorporate into their systems. It is currently available, with restrictions, but at no charge, from the OMG server on the Internet[1]. The OMG IDL compiler front end can also be ordered from SunSoft directly for a nominal media charge.

- Higher Quality

  In developing distributed object applications, programmers need only be concerned with the objects they themselves build. As mentioned above, objects that have been purchased or pulled from inventory have already been tested and need not be tested again. The integration testing of non-object-based systems so onerously expensive becomes greatly reduced under DOE. Testing resources for DOE applications can be concentrated on the new objects being developed, not on the overall system being installed.

- Scalability

1.Use anonymous FTP at omg.org

Objects can be as small as a spreadsheet cell or as large as a legacy database. Within the DOE architecture either size is supported with equal ease. Moreover, DOE is architected to efficiently support applications spanning a broad set of platforms from the single user, single processor desktop to large multiprocessor servers.

The Future of Objects

Through Project DOE, SunSoft and Sun Microsystems have defined a path to the future of objects. By providing a framework for distributed object solutions Project DOE will enable greater interoperability between systems and simplify application architectures.

DOE addresses many of the key problems in developing and using application software today - reducing the cost of development and ownership, raising quality, increasing functionality and improving interoperability. DOE enables the enterprise to have a more flexible computing infrastructure with fewer computing and informational bottlenecks; making those business better able to implement the appropriate business policies required to compete in an ever changing marketplace.

With ToolTalk already shipping in volume today, the path to DOE is clear. All ToolTalk APIs will be supported in DOE and through compatible services and capabilities, application developers can begin their journey toward complete object orientation today. Distributed object technology holds the key to a more efficient responsive enterprise. Project DOE will deliver on that promise; providing those benefits to the commercial customer.

Features of DOE

DOE includes a broad set of features providing for the support and integration of object-based applications in a distributed, client-server computing environment. Among those capabilities and services are:

- A distributed, object-oriented computation model

  Providing the flexibility to provide object-oriented facilities across heterogeneous distributed networks

- Location independence

  With complete protocol and network transport transparency enabling complete freedom for object location

- Scalability

  Across computing platforms and networks ranging in scale from stand-alone to large area networks

- Support for full interoperability between applications

  In 100 percent compliance to the OMG standard for complete object and application interoperability

- Standard interfaces which allow interchangeability

  Leveraging the object-oriented benefit of information hiding and extensibility to permit graceful evolution of objects and applications

- Full range of services

  Including security, persistence and naming

- Framework

  A sound architecture based upon the Object Management Group ORB standard for distributed object management

- Tools

  For migration, application development, and system management

Figure 16-1 illustrates the architecture of how DOE services are layered.

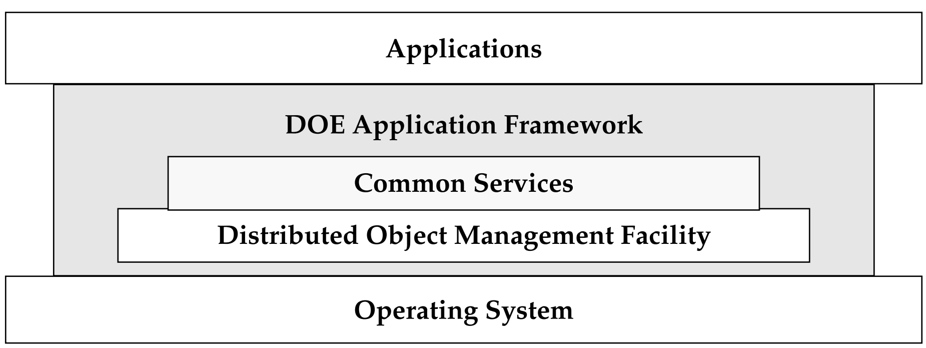

Figure 16-1 DOE Architecture

The DOE Computational Model

Object Reference

From a programmer's viewpoint, *object reference* is one of the key concepts of the DOMF. A convenient way to think of an object reference is as a distributed, typed handle for an object. This concept is analogous to the way a C++ or a C computation works. A typical C++ program creates an object (or a structure in C). The class instances are interconnected through the computation and are placed in tables, linked lists or some other structure. Pointers are passed via the references by function calls. References can be made through pointers into tables. By this method data is remembered and the computation proceeds in a single process C or C++ computation.

While analogous to C++, the method in which object reference works is similar but has additional capabilities and considerations for multiple processes including enabling those processes to be distributed over many machines. For example, instead of using pointers, there are object references. These objects references point at the object and stand as a surrogate for an object that is remote. The user of the reference to an object is the "client" while the implementation of the object is the "server". The references are typed so that they can be used to determine which type of object they are using. Hence, they are not just type `void*`, but are typed objects.

Object references are passed around throughout the computation. They can be stored internally to a program as "pointers" to be used in different parts of a computation. They also can be passed in the DOMF invocations when doing calls between objects. The difference is that this computation is distributed and runs in multiple processes rather than existing inside a single process. Object references are much like the coin of the realm. They are the common dominator that holds the computation together.

As far as the client can tell, the object reference *is* the object. It has methods which can be called and it acts precisely as if it were the object. From the client viewpoint, the fact that the object is actually a remote object is normally not noticed except in cases of failure or security violation.

An object reference is a run-time structure that cannot be written onto a disk file. Object references can be persistified and unpersistified. The object reference can be unpersistified and a new object reference can be reconstituted anywhere, in any process, within the same ORB, at a later time. Only the object reference is changed, and not the object itself, because the object exists elsewhere. And it is possible to persistify this object reference to remember where these objects are in the program. The persistified object reference can be written to a disk file. The persistent state is what the program actually writes out.

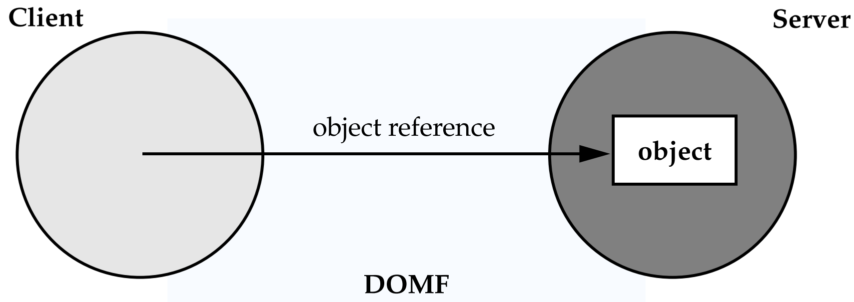

Figure 16-2 Computation in DOE: Object Reference

The Interface Definition Language (IDL)

As discussed earlier, object references are typed. The typing system, known as the *Interface Definition Language* (IDL) is an industry standard produced by the OMG.

Interfaces are a contract between implementor and client, defining what the implementation provides as a service. The interface does not define internal details, but rather specifies what implementor and client agree to regarding what the server is going to provide.

When constructing a piece of software, particularly the functional portion, it is best to consider the computation taking place in abstract terms. In a perfect world, designers would have the capability to clearly and completely state what that computation or abstraction would be. Unfortunately, given the differences between computers and people, and in the way we examine and solve problems, this is not yet possible.

Instead we must rely on a more syntactic way of dealing with the definition of problems using interfaces. These interfaces are analogous to header files in C and C++, but are far more powerful because they express a real type system and do not have the same implementation dependencies.

In defining and using IDL, it is important to think about what the objects actually do and what the interfaces are to the system. In a syntactic manner, an IDL definition specifies how the operations and features of the system work rather than the semantics behind them. Consequently it is important to think abstractly when using IDL.

IDL, as a language, is supported by a run-time system that provides many features, including distributed, strong typing. In a small system, strong typing is often important inside a single process. It allows more robustness in the way the software is built. When these concepts are used in a large distributed system there is significant potential for problems. Robustness can be compromised if there is no mechanism for using typing in a distributed system.

How is it possible to build long-lived systems that are robust, large, and which could have multiple vendors adding software to them? How would such a system allow for change without a strong notion of how the system is held together? The approach chosen by the OMG is the typing system represented by the interfaces generated by IDL.

Interfaces written in IDL provide a method of defining the requirements between two parts of the system in such a way that is able to be made long lived. Different parts of system can be constructed and plugged together much later because the interfaces were well defined and understood from the beginning. Software can be written and put into the system knowing only the interfaces and nothing else. This is a requirement in creating a large distributed system.

The most basic characteristic of IDL is an interface. This is analogous to a class definition in C++. The interface defines specific operations which may be invoked and they behave much like C++ virtual member functions. Interfaces also have attributes which are the data values associated with the particular interface.

The interfaces have multiple inheritance to allow easy construction of new interfaces from their parent interfaces. Operations can also raise exceptions. While exceptions are significant in a single process system, they are extremely important in a distributed system since exceptions can occur from distributed calls. A distributed call may fail due to the distributed nature of the call. The remote server may crash. There has to be a way to model this, and in IDL the way is through exceptions.

IDL is not a complete programming language though the syntax does resemble C++. IDL constructs are easier to understand from C++ perspective, but it is meant to be independent of any programming language. IDL is used to define interfaces and not the implementation.

Language bindings map the features of IDL into a particular programming language. Language bindings provide a means to access IDL from traditional programming languages such as C and C++. The language bindings that are made available from IDL may be simple or difficult to develop depending upon the particular language. Initially only C and C++ language bindings will be available. Components written in two different languages can call each other, for example a FORTRAN client can call a C++ server.

In this sense IDL acts as a boundary. Because IDL is programming language independent each process has to have language bindings that provide for the languages used in that process.

On the client side, these components are called *stubs* and are analogous to those provided by RPCGEN or other RPC protocol generation systems. With these stubs the call appears as a normal function or procedure call in the native language. The stub matches the type system provided by IDL as mapped into the programming language. This permits the application to access IDL from a normal 'call' in the application's native language. The stub then interacts with the support libraries to perform the call. The stub encodes the call and uses run-time support to locate and transmit the call to the server.

The other end of the call, known as the server, provides an analogous piece of software called the *skeleton*. The skeleton performs an inverse mapping to what the stub provides. It takes information that was encoded for the call and reconstituting data structures or object references in the native programming language for the server side of the call.

IDL interfaces are generic and have no implementation details in them. Neither the language nor the operating system is important in defining IDL, thus enabling multiple implementations of the same interface. Performance requirements may dictate different implementations which have different performance characteristics. The interface is the same, but performance may be different for different clients. Other implementations of the interface may offer different security, robustness, availability or other characteristics.

IDL also provides a flexible way to deal with evolution and extension. It assures robustness within the system by providing a mechanism for defining interfaces. When the system evolves, the interfaces may need to evolve as well. By using the inheritance mechanisms of IDL, it is possible to build a server that can support both old and new interfaces. And software can be constructed that would support multiple interfaces. This approach also eases evolution, since a system may support the multiple versions of an interface while simultaneously providing for all of the interfaces within the same process. As with C++, extension through IDL is provided through subclassing. Straight subclassing of an interface allows more specialization as would be done in a normal object-oriented system.

Object references are the glue that holds the system together. An object reference is useful when it is used to "call" an object to do an operation. The OMG defines the object system as working in a particular manner through something called a *call* or *invocation*, which assumes the form of a call made to a target object. The syntax of the call shows a zeroth argument to the call, representing the target object. This approach is similar in concept to the C++ language where the object to the left of the arrow is the one that will be called.

The target object is responsible for performing the semantics of the call. The required argument(s) are obtained from the call to the stub that has the same signature (type and number of arguments) as the IDL specifies for the operation. In the C++ programming language, the call would be similar to:

```
result = objref->operation(arg1, arg2)
```

The IDL specifies an action, called `operation` in this interface. `objref` is an object reference that is a program variable of a type appropriate to this object in this interface. `arg1` and `arg2` are two arguments and the `result` is the single valued return. All of these are type checked in C++ at compile time.

These calls are typically synchronous calls which work in a similar manner to function calls. The caller blocks waiting for the return. There are also other ways of insuring concurrency like deferred synchronous and one-way calls. A one-way call is initiated with no result to indicate it happened. Deferred synchronous calls provide a way of polling to see if the result occurs. Another method of performing calls is dynamic, which doesn't have to be mapped into a programming language but can be constructed during the computation.

Concurrency is generally achieved through the use of threads. Even though synchronous calls are normal, in many cases clients will be multithreaded which effectively provides for the concurrency. The default for a server will be where all incoming calls are dispatched concurrently.

The following figure provides an example of how a call works and illustrates some of the issues about how it takes place to provide a flavor of how the system runs internally.

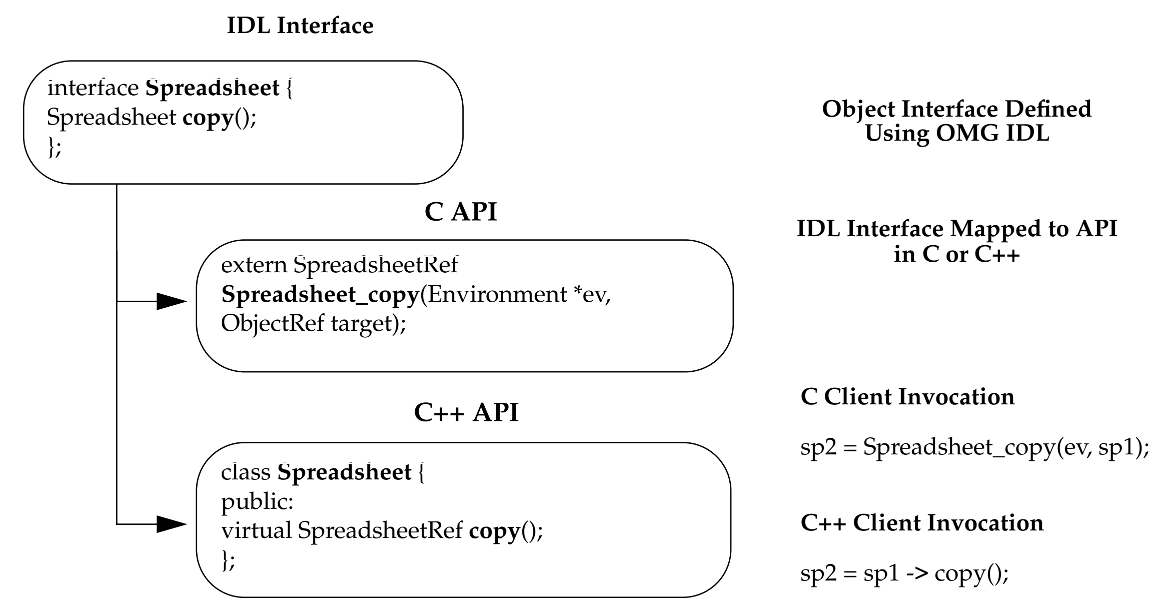

Figure 16-3 The IDL Interface C and C++ Client Invocation Language Mapping

Object Servers

An *object server* is an object that provides services to client objects. Object servers are usually multithreaded processes designed to provide concurrent access. Consider an example illustrating an object server, where a spreadsheet cell has value to many people in a corporation. The content of the spreadsheet cell reflects the profit and loss for a quarter, and it is desirable to incorporate this information into a large number of reports. It may also be desirable to access this cell from remote sites. But most importantly, it would be desirable to access it concurrently.

To achieve this level of functionality, there needs to be thread support and concurrency within the object server. Concurrency improves the responsiveness and availability of resources. Multiple concurrent users can access the spreadsheet cell above, no matter what the server process is doing.

An object server is usually a multithreaded process. It is particularly important to have object servers be multithreaded if they are going to be used recursively. For example, a particular object server implements a particular object and provides all the functionality for that object. It is layered so that this particular object server can clearly make use of other objects. In this fashion an object system works with one object implemented and others used, and those objects may recursively use features of this particular object server.

If only a single thread is utilized, it becomes difficult to have recursion work properly. It is still possible, just much harder to do. Most object servers that are written for DOE will likely be multithreaded processes - new code written for DOE will almost certainly be multithreaded processes. Software that is not originally written for DOE but is ported to it may use threads only in a limited fashion. Multithreading and concurrency are discussed in more detail in *the* section of this chapter called Architecting for Concurrency.

Object servers are Solaris processes and can have implementations of multiple instances of objects. A particular process can supply the execution for many different objects of the same type. A particular process can also have multiple interfaces to different objects within it. Every interface has the capability to have multiple implementations internal to it. There is nothing that's a "one", everything is in multiples. There may be multiple server processes per host from the same executable due to the desire to insulate a particular process from other processes for performance, resource management or security reasons. There can be multiple processes supporting multiple objects on the host and there can be multiple objects within a host. And each object can have different interfaces with different implementations within a process.

Object Adapters

The *object adapter* is the interface from the server to the DOMF core. The object adapter provides an easier way of using the particular DOMF Core or ORB, and it can provide many more features that the raw DOMF would provide. By including additional semantics it may also extend the features of how an object reference actually works.

The object adapter might provide automatic security mechanisms, initiation mechanisms for processes to implement particular objects or other forms of run-time structure creation to ease method invocation. It may also do internal-to-process fine grained method dispatch.

The OMG requires only one object adapter called the Basic Object Adapter (BOA). Additional object adapters may be provided but only the Basic object Adapter is required. The BOA has some extended features beyond normal objects. For example, the BOA includes reference data that goes with a BOA object reference allowing the DOMF to retain some state information about a particular object. This can be utilized to find persistent storage, for example. In the

CORBA (OMG's Common Object Request Broker Architecture standard) model, there is also a security module that can be used that allows implementation of security within the BOA framework. The BOA also provides for process activation - for example, when the BOA discovers that there is a use of an object that does not have a suitable process to run in, it will fork and exec a process. Finally, there is a method to specify how the BOA will dynamically create processes and bind them together in the whole object system. This method can also specify the call that is actually sent to a particular process.

The BOA provides a very fine grained method dispatch for the method inside of the process. Its much finer grained than that used in the *Remote Procedure Call (RPC)* system. The interfaces and the implementations are made visible within the BOA to allow specification of the particular method that is to be called. The BOA then handles the dispatch for that particular method call.

Different object adapters can be built to provide different levels of service. For example, an object adapter may deal with fine granularity or security. The flexibility and capabilities of object adapters make them a very powerful concept within the OMG framework.

The following figure illustrates an invocation. The call starts with the client, as shown on the left. Using a C++, C or language specific binding statement in a programming language, the program calls a stub. The stub then takes the arguments of the call, and in conjunction with whatever transport and marshalling mechanisms are provided, takes the calls and encapsulates it as an RPC or other remote message intended for a server.

The DOMF core provides all the location and transport mechanisms necessary to get the call from the client to wherever the server may be. The client and server may be in the same process, in a different process on the same machine, or on a completely different machine. The DOMF core works in conjunction with an object adapter to find a server process that is suitable for implementing this particular object. Once a server process is found or created, the DOMF core works in conjunction with the object adapter to deliver the encapsulated call into that process.

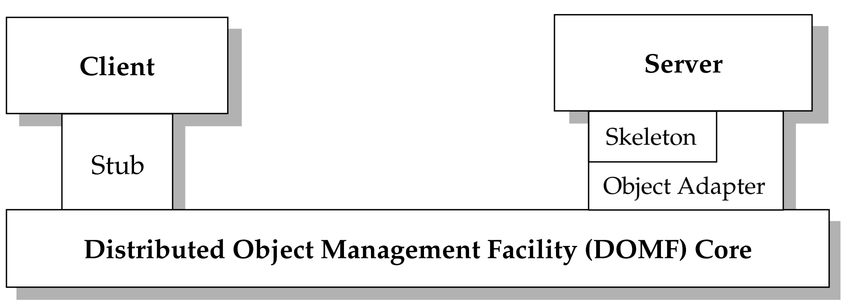

Figure 16-4 Invocation

The object adapter works with the skeleton which has the knowledge of what the particular interface is that is being called. It takes the encapsulated call and de-encapsulates it into a data structure suitable for use in a programming language of a particular implementation within the server. The server may have support for multiple programming languages, and have multiple skeletons that are tuned for those different languages.

The skeleton works in conjunction with the object adapter to do the fine grained dispatch to the method the server programmer has written. The server can then perform the required computation. This is the value-added area most attractive to VEUs and ISVs - providing for the actual computation which is not part of the system infrastructure.

Another viewpoint examines address space and programming language issues. The following figure illustrates the process view of an invocation. The rounded block on the left represents the address space of the process. Inside this process, there is a program variable, `foo` which is an object reference of some IDL type mapped into C++. To use this particular program variable `foo`, the normal C++ language binding is used. For example, this might take the form `foo->op` together with its arguments. As far as the program is concerned a call is made to a different process, possibly the same process, or even to a server process on a different machine.

This server has the code written to implement this operation. In the illustration, the bottom box is the server process, inside of which is code which has an the implementation of `foo` called `fooImpl`, together with its particular operation and arguments.

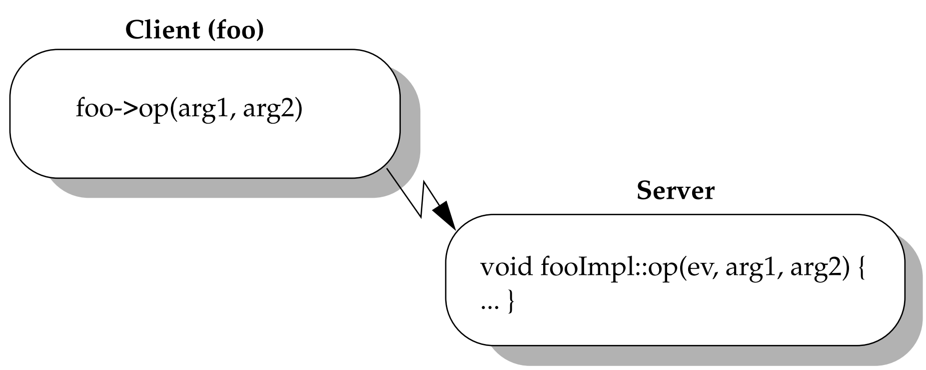

Figure 16-5 Process View of an Invocation

A key point in this example is that the role of the client and server are momentary - and assumed only for the duration of a particular call. The server is a full-fledged DOE process that can immediately change roles to become a client and call other servers.

This is an example of layering. Because client and server are just a momentary concept for a particular call, they are unlike clients and servers under Solaris. For example, a file server is a leaf server and does not require other servers to get its job done.

The Object-Oriented Model

All objects have types which in turn are represented to the client by the object references that also have these types. These types are the interfaces defined by IDL. Some object systems utilize no type or `void*`. These are typeless pointers to objects, which are not utilized in DOE because they can compromise the robustness of the system.

The DOE architecture also provides for type arithmetic. The concept of ‘narrow’ is an important facility in DOE. For example, there may be an object reference of a general type that is desirable to treat as an object of a more specific type. If there is an object reference which points at an object `Printer` and it is convenient to discover if can be used as a `PostScriptPrinter`, it might prove useful to use the operations of `PostScriptPrinter` rather than the more generic operations of the standard `Printer` object. The type arithmetic operation of ‘narrow’ asks the question, “are you really a `PostScriptPrinter`?” or, “are you a `ColorPrinter`?”, or even “are you a `ColorPostScriptPrinter`?”. The DOMF environment, supports narrow typing directly. The following figure illustrates IDL inheritance for narrowing.

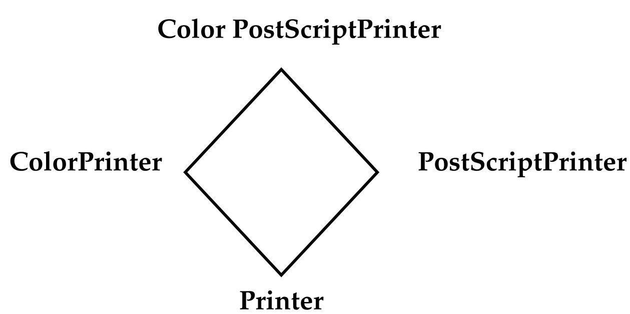

Figure 16-6 IDL Inheritance for Narrowing

The object-oriented type model also supports multiple inheritance for which there are many different uses. Objects can be implemented in multiple ways and it frequently proves a convenient way of constructing objects as a refinement of their functionality.

To be able to perform type arithmetic on a distributed system there must be distributed type checking. Type checking within a single process is challenging but type checking within a distributed system proves to be even more difficult. This feature has been provided in DOE because the robustness of the system requires the availability of type arithmetic functions. This also provides a way of self-describing the object. The interface of the object tells what the object is about. However it does not reveal its implementation language, where it is located, the CPU architecture, or the operating system it runs under. Nonetheless, it does provide the necessary information about the object: its interface.

In DOE, the interface is what the object truly is. And this identity is explored through the distributed typing model. Because objects are self described, it is possible to ask type questions about them.

Reusability

One of the major objectives of DOE is to enable writing common code once. To do a good job it might take slightly longer, but coding, testing, and documenting should take place just once. A developer should consider making commonly used code an object server. That is, make this code be a first class entity within the DOE framework. This then provides a building block for other code to be built upon.

As requirements are refined or if the performance is not acceptable, the programmer should carefully evolve the particular interface as well as the implementation. Reuse of the same code is possible because it can support the same old interfaces too. It may evolve with a new interface but the system maintains the same abstraction.

When the system grows, these building blocks can also be used elsewhere in the system. In this manner software can be constructed much like a pyramid. With more modularity and greater uniformity in the construction, software becomes more universal. Traditionally, a piece of software is written for a specific purpose. Someone else with similar but slightly different requirements comes along, steals the source and edits it, to create a new piece of software.

Slowly, with time, this original piece of software, spreads in mutated forms throughout the entire organization. If there were bugs in the original software, they are usually reproduced in all the subsequent implementations. If requirements change and it is desirable to modify the software, it would be necessary to track down everywhere the software has spread throughout the organization and fix it. This quickly becomes a software maintenance nightmare.

If the system has finer modularity, more uniformity can be designed into it because all of the code for an abstraction exists in one place. So problems can be fixed just once. And they can be tested and documented only once. The ultimate effect is to provide greater uniformity in how the system behaves, and more consistency in how the computation works. There is also more scalability because software does not mutate as easily and it is possible to solve all of the problems of a software component at one time.

Consider the example in the following figure. Here a document editor has been configured to use various other software components. One of these deals with video. The original video component may have only recognized one video encoding. The video software component can be replaced at any time with an improved component that has better real-time properties and deals with more video encoding formats. Software that used the old video component should function just the same, provided the new component supports all the old interfaces. But now, the video is delivered in real-time and more video components can be supported.

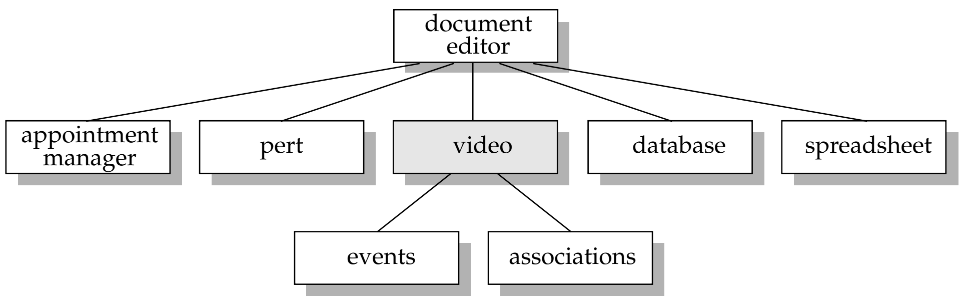

Figure 16-7 Replacing the Video Component Used by a Document Editor

Another problem to consider is performance. Whenever smaller granularity is used, this tends to make the object server smaller, creating the potential for performance problems due to more boundary crossings. DOE has been designed as a system with balanced performance, but trade-offs may be necessary in terms of layering and reusability.

The cost of software development is immense compared with cost in buying hardware. One of DOE's significant benefits is to lower the software costs and permit a change in the business model, thus making it easier for people to build and deploy software.

Interoperability

One of the principal reasons for starting the Object Management Group was to enable object-based applications to interoperate. Prior to the OMG, many companies developed strategic alliances to enable their separate pieces of software to work together. In some cases, industry consortiums in particular market segments were formed as an attempt to provide integration between their respective systems. This approach was crippled from the onset because of the lack of integration between dissimilar systems, such as MIS and CAD.

There has historically not been a broad range of agreement of defining what the methods of working together were to be, what the object model was, and how things were going to work together. Application interoperability with DOE and other ORB environments is achieved through agreement on the IDL interfaces and the CORBA specification. Therefore IDL is a standardization mechanism for interoperability. IDL provides the mechanisms, but not the interfaces. Consequently the OMG, and SunSoft through its DOE implementation of the ORB, provide the language, not the agreement. IDL is realized through the DOMF as infrastructure, so that within a DOMF, applications that use the same interfaces as clients and servers, will be able to call each other.

In addition to application interoperability, standard interfaces enable the interchangeability of services. For example, suppose a particular implementation of an application just isn't fast enough. Performance analysis might assist in the discovery of an alternative algorithm with better response. When that better algorithm is implemented it can be swapped into the system easily. The interface is still the same, just made better. In this way, interfaces work to allow simple interchangeability of services.

Layering

Layering is the concept that one service is built over another. This implies that it is not a true client-server model, since servers can also be clients. Peer-to-peer is a somewhat better term for abstractly describing what is taking place. All services may be thought of as peers in the computation. A service can call back to what would be thought of as clients to find out more information about a call. This proves to be a convenient way to think of how computation works. This approach is also known as a *call back mechanism*.

Layering provides a means to build up the functionality of a system. Rather than starting from scratch each time, applications can be developed which leverage the work of those that preceded it. It is naturally desirable to be able to build a piece of software, test it, debug it, document it, and finally use it as a fundamental building block of the entire system. As the system grows, it becomes increasingly easier to expand and evolve because the functionality underneath is broader and richer.

Consider the example in the following figure. Here a mailtool user interface is being used to display messages. In addition to the window services (not shown) are various software components. Events, associations, properties and the PSM are DOE services covered in the next section. Spreadsheets, document editors, audio attachments, and video attachments are ISV software. The mailtool is part of a hypothetical desktop software suite. A VEU developer has combined all of these components (some were prepackaged as part of the mailtool) and added a notification service. Note that components show up at multiple levels (e.g. spreadsheets and events).

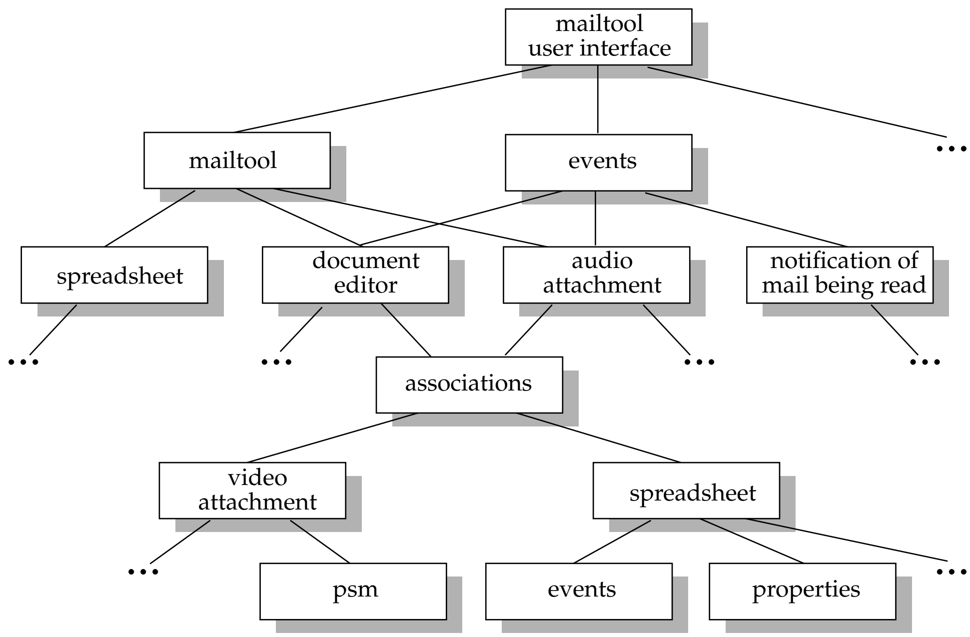

Figure 16-8 Layering of Software: Combining ISV, VEU, and DOE Components

Another approach to layering is referred to as *delegation*. Essentially an object is extended with new functionality by writing an object service that basically encapsulates it. The new object then possesses a new interface to both the old object and the new extension. There are now two objects, but the clients think of them as a single one.

Evolution

Interfaces are the backbone of DOE, and in essence are the fundamental, computational structures that hold the system together. As a representative implementation subscribing to the ORB architecture, DOE is a big multi-vendor, evolving system. To facilitate the evolution, DOE will provide version management for interfaces. When systems change and their interfaces change, version management will enable those interfaces to gracefully evolve. As a result, systems will be able to provide both old and new interfaces so that old software works correctly while new software is not hampered by requirements of dealing with the old interfaces.

To insure effective evolution, it is important to be able to remove and replace components. For example, it might be desirable to clip out one component that has some security or performance problem and replace it with a different component having better characteristics.

Just as importantly, DOMF is committed to internal evolution. It is critical that it be possible to build a system so that new releases of DOE (with the DOMF inside) do not require a reboot of the network or a relink of all of the object servers. The system stays up and users can gradually move to the new system. Compatibility with the old system is never lost and old software continues to work along side of the new.

DOE Services

DOE is a feature-rich environment that incorporates a suite of core services designed to provide facilities which simplify the manipulation and interaction of objects. DOE includes a number of services, including those listed below:

Naming

Naming provides a basic way of taking a compound string name and mapping it to a particular object reference. This name space approach is very similar to file system naming such as with hierarchical compound file names mapped to particular files. However instead of files, in DOE there are object references. Naming is one of the principal ways of locating things in a distributed space.

Persistent Storage Manager

Many applications today create object persistence by simply writing their persistent state out to a file. While this basic capability is part of DOE, additional capabilities are provided by the *Persistent Storage Manager (PSM).*

The persistent storage manager has much greater object-oriented data base flavor to it than regular files. From a program standpoint, it also appears more similar to persistent C++ instances than files. Using the persistent storage manager does not require elaborate coding and decoding to write persistence data out to a file. All the program need do is use the API to the persistent storage manager to perform the mapping and the persistification is carried out automatically.

Associations

Associations define well formed formal relationships between objects. Associations tie objects together, making them easier to manipulate and invoke. Associations tie together independent objects that have some relationship. For example, an author and a document written by the author have a definable relationship. By having formal associations, they become first-class entities in the system and can be standardized.

Properties

Objects have both state and meta-data. The state and meta-data are analogous to a document and its keywords. The document itself is the state but the key words contain information *about* it. The author is another example of meta-data. Hence both keywords and the author represent meta-data about what the object is rather than data itself. Another example of meta-data would be file properties in the UNIX file system, such as who owns it and the creation time of a file. They are properties about the file rather than the contents of the file itself.

Events

Events provide a way of structuring computations. Basically, events provide a mechanism for one object to signal a change in state to other interested objects. For example, consider a file whose contents have changed, and it wishes to update itself on a display screen.

Events are akin to a data flow computation where the changes of data are being sent to objects rather than polling for changes to try to find out if things occurred. The events mechanism is a service because it is important to be able to allow events between a wide number of objects without having the object necessarily have to implement them themselves.

Life-style

Objects are created, moved, copied, and destroyed - this is their *life-style*. Objects that are moved or copied between systems have to be externalized as they are moved from the origin system and internalized in the destination system. This action requires a set of protocols to insure that it is completed uniformly, so that each particular object doesn't have to establish it's own set of conventions. The life-style set of services provide a set of protocols to make full life-style object management easier.

Solaris OpenWindows and DOE

Solaris OpenWindows will also evolve in the DOE environment by keeping the same look and feel, while implementing full distributed object functionality through DOE objects. It is also desirable to interoperate with OLIT-based applications. The OpenWindows system under DOE will initially have some DOE objects and some non-DOE objects (or icons). The OpenWindows icons representing DOE objects will be fully distributed and have all the functionality, and benefits of DOE as other objects.

Migrating to DOE

Preparing for DOE

Migration to DOE requires a conscious planning effort. The suggestions that follow that will help prepare the application developer for DOE. While DOE is not currently a product, there are some things that can be done to position application development to ease the transition to DOE when it becomes available.

Using Threads

Threads are an important structuring concept guiding how software can be built. Although very important for concurrency, threads is also a very natural way to structure a computation. The DOE environment, and particularly the object server, relies upon multiple threads. An understanding of the problems and use of threads, and the thread package, can help prepare conceptually for this part of DOE.

Working with Objects

The object-oriented nature of DOE requires that the developer understand the use and development of objects. One way to do this is to use an object-oriented language such as C++, and to actually begin developing object-based applications. The software should use encapsulation and interfaces which are some of the important features of object-oriented programming and that are heavily used by DOE.

Using ToolTalk

ToolTalk, SunSoft's inter-application communication mechanism will also aid in the transition to DOE. SunSoft is committed to supporting the ToolTalk API's in the DOE environment, enabling applications which use ToolTalk today, to be transparently supported under DOE tomorrow. Its also beneficial to use ToolTalk because many of the concepts in DOE and ToolTalk are similar. In transitioning to ToolTalk programmers and their corporations will be able to understand the conceptual shift that will happen in moving to a system like DOE.

IDL-Compatible Interfaces

Developers can also write their interfaces (which will be similar to `.h` files) today, in an IDL compatible manner. A copy of the IDL Developer's Guide is available from SunSoft.

Plan for Replaceable Components

Developers should also plan for replaceable components. This is a more abstract concept than those already mentioned. Having replaceable components requires the definition of interfaces between them, as well as knowledge of precisely where the interfaces are and what available components exist. With a large system it can be difficult to replace the subcomponents. It is very important to architect the system so the components are visible, have clean interfaces and are well segregated. Thus they have less dependence on one other in the case when a change is necessary.

Segmenting Applications

Often an application has multiple parts to it, such as an internal semantic part, and some kind of interface, or GUI. If it is possible to architect the application so that the semantic part has an interface, then one can expose that engine as an IDL interface when DOE becomes available. This approach serves to provide better layering.

The only way to interact with a spreadsheet today is through the GUI. A user must manipulate the mouse to change the spreadsheet. However, spreadsheets have many powerful computational features inside of them, in essence making them constraint-based systems. There are relatively few constraint systems available today, but spreadsheets are one of them. If a spreadsheet can expose the engine part or reveal the computation performed from an API instead of a GUI, then the engine can become a layerable component within the system. The engine could look at cells and set up the computation, establish the constraints from an API which might be built into a piece of software without having to manipulate the spreadsheet through the GUI. Access is then possible to the computational layer, or more precisely, its semantic part. While this does not apply to all applications, it is important to do wherever possible since it improves layering.

Migrating Existing Applications to DOE

In migrating an existing application to DOE, there are three basic strategies that can be taken:

Re-Architecture

Rearchitecting the program for DOE requires redesigning it as if it were a new program. This approach obviously involves the most work, but potentially provides the most benefit. Re-architecture is the best approach to use if the goal is to support concurrent processing of operations or concurrent editing by multiple users. Typically, this approach involves significant use of multiple threads and may involve splitting the application into several components.

Make Small Modifications

A second strategy for migration is to make small modifications to the program that do not radically alter its structure and allow most of the existing code to be preserved. The goal of this strategy is to obtain the most benefit for the least cost in development. This approach is often the most appropriate for the developers of existing applications.

Minimal Changes

A third approach is to adapt the program using techniques that do not require the source code for the application. This approach is of most interest to end- user developers who want to migrate an application developed by someone else. It is not necessarily easy, but in some circumstances, it may be the only choice.

Developer Issues

In migrating applications to DOE, there are several key issues:

- Should the application be a DOMF client, a DOMF server, or both?
- How is client code written?
- How is server code written?
- How does one architect for concurrency, handle concurrent requests, avoid deadlock, and use multiple threads without performing major surgery to the program?
- How are user interfaces supported? How does DOE programming interact with the UI toolkit? And how can multiple users be coordinated without major changes to the program?

Client, Server, or Both?

A DOMF client is a program that issues requests on objects and a DOMF server is a program that implements objects. This is an important distinction because it has a significant effect on development tasks. It also impacts the approach to handling concurrency. It is easiest to be a pure client, since it has less to do.

In most cases, there is no choice. If the application provides services to other applications, then the application must be a server (which implements objects). If the application invokes other applications, the application must be a client. If the application does both, then it must be both a client and a server.

Sometimes, to use an object, the application must be a DOMF server, as well as a DOMF client. This generally happens when the object wants to call back to the application. The way DOE manages this is to provide an object reference to an object that is implemented by the service consumer. This object is known as a *callback object*, because its role is to allow the service provider to call back to the service consumer. The service provider calls back to the application by issuing a request on the callback object. Therefore, to be a service consumer, the application is both a DOMF client and a DOMF server.

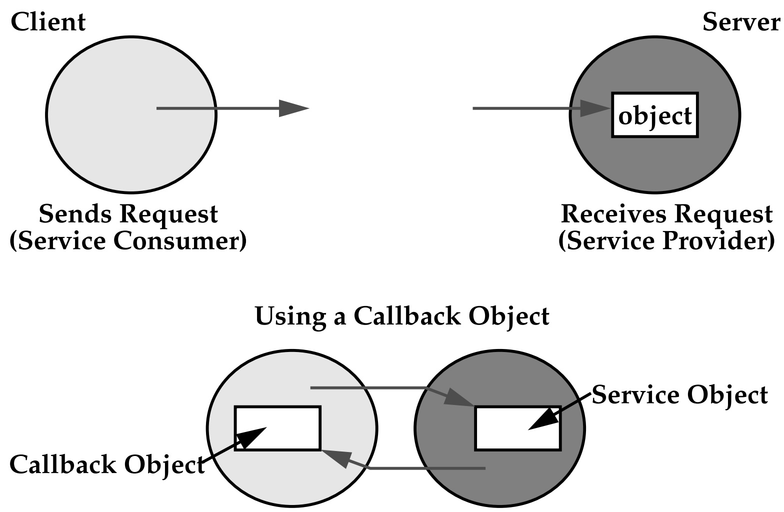

Figure 16-9 Client vs. Server Objects

There are some cases where there is a choice in being a pure client or both a client and a server. For example, the standard service that supports linking can be used by pure clients to send or receive data. A pure client issues requests to send data and issues requests to poll for incoming data. However, it may be preferable to be notified when new data is available, instead of polling for new data. Notification of new data is provided by a callback object.

Writing Client Code

Modifying an application to be a DOMF client is generally very straightforward. Because objects implemented by other programs are being used, it isn't necessary to define new interfaces or write object implementations. All that is necessary is to get hold of an object reference and invoke functions that issue requests to the object. These functions behave like ordinary functions. They block until the result is received. These functions would be determined by reading the IDL interface definition and the generated header file that defines the API in C or C++. Once some experience is gained in using IDL it becomes unnecessary to read the header files.

There are also a couple of housekeeping calls needed to initialize a connection to the DOMF and to shut down the connection when the application is finished.

Obtaining Object References

One thing that may not be clear is how to obtain an object reference in the first place so that requests can be issued to it. There are several possible sources of object references. The two most common sources of object references for a client are naming contexts and user gestures (such as ‘drag and drop’).

A naming context is like a file directory for objects, supporting operations to bind names to objects and to look up objects by name. Applications receive a default naming context when they initialize their connection to the DOMF.

Naming contexts are typically used for static connections to objects, such as connections made at the time an application is installed. For example, the connection between a spreadsheet program and a visualization program is most likely made during application installation using a naming context.

Naming contexts support customization of an application by end user developers. For example, an end user developer can bind a different visualization object into the naming context to cause the spreadsheet program to use a different visualization program.

User gestures are used to make dynamic connections to objects. For example, when a user creates a link (say, by using a drag and drop gesture), the system implements that link by passing object references to the two applications involved.

Of course, once the developer has one object reference, it is possible to get more object references by issuing requests that return object references. For example, if there exists an object reference for a factory object, one can request the factory to create a new object and return its object reference to the application. The object’s reference can also be stored in a file and retrieved later.

If the application is a server, then there are additional ways it If the application is a server, then there are additional ways it can receive object references, such as through input parameters to requests. It may also store object references in the persistent state of an object.

Client Example: Spreadsheet Calls Visualizer

As a simple example of writing client code, consider the modification of a spreadsheet program to use an external program to display data in a visual form. In this example, assume that it has been determined to make the spreadsheet program be a pure client to keep the implementation as simple as possible. (In fact, this example can easily be implemented using the Lotus toolkit.) The spreadsheet program will issue a request to the visualization program to display data. The data is passed to the visualizer via an input parameter to that request. If the user modifies the data, the new data will be returned to the spreadsheet program via an output parameter of that request.

There are four steps:

1. Initialize a connection to the DOMF and obtain an object reference to the default naming context. Among other things, the default naming context includes object references for factory objects.

   Obtain default naming context:

```
NamingContext nc = CORBA::init().default_naming_context;
```

2. Using the default naming context, obtain an object reference to a visualizer factory. This object reference is bound in the naming context as part of the installation of the spreadsheet application.

   Obtain visualizer factory object reference:

```
VisualizerFactory vf = nc->resolve("factories/Visualizer");
```

3. Using the visualizer factory, create a new visualizer object. Pass the name of the display to the visualizer so that it can open a window on the user's screen.

   Create visualizer object:

```
Visualizer v = vf->create(display_name);
```

4. Ask the visualizer object to display the data. When the user closes the visualizer, it returns, passing the modified data (if any).

   Present data:

```
Boolean modified = v->display_data(data, new_data);
```

Client Example: Using Events

While the previous example works, it has the limitation that changes to the data made by the user via the visualizer are not reflected in the spreadsheet until the user closes the visualizer. The example can be extended to allow real time update of the spreadsheet using an *event channel*.

Event channels are an object service provided by DOE. An event channel supports the asynchronous delivery of data from a sender to one or more receivers. An event channel has been chosen in this example to simplify the control structure, so that the visualizer object does not have to wait for the spreadsheet to process an update.

- Create an event channel:

```
EventChannelFactory ecf = nc->resolve("factories/EventChannel");
EventChannel ec = ecf->create();
```

- Create a visualizer object:

```
Visualizer v = vf->create(display_name, ec);
```

- Present the data:

```
v->display_data(data);
```

- Poll for changes (within application main loop):

```
if (ec->poll_for_data(event_data)) { // retrieve new data, if any
update_data(event_data);
}
```

In the new design, an event channel is used to send updates from the visualizer to the spreadsheet. The visualizer sends the data to the spreadsheet by way of the event channel.

The event channel object is created by the spreadsheet, using an event channel factory, which the spreadsheet program obtains through the default naming context. Again, the binding to the factory object is made when the spreadsheet application is installed.

The spreadsheet program passes the event channel object reference to the visualizer factory as a parameter of the request that creates the visualizer. The visualizer then retains the object reference in its persistent state.

Whenever a user updates data in the visualizer, the visualizer sends the updated data through an event channel. The event channel works asynchronously: when the visualizer issues a request to the event channel, that request returns as soon as the event channel receives the data; the visualizer does not have to wait for the spreadsheet to receive the data.

In this new design the spreadsheet program is a pure client. Therefore, in the main loop of the spreadsheet program, there is an added a statement to check for incoming data on the event channel and update the spreadsheet.

Writing Server Code

If an application is being modified to be a server to implement objects, several different tasks are required.

The most important task, is determining what the objects are that the application implements. In most cases, it is necessary to define one or more new interfaces. Writing the implementations of those objects involves defining the methods that implement the operations that the objects support.

In addition, there are bookkeeping chores, such as initializing the object adapter, registering the object implementations, and shutting down the object adapter when the application terminates.

What are the Objects?

The most important task when designing a server is to determine what the objects are that the application implements. DOE supports a wide variety of applications.

Some applications might implement just a single object. For example, there might be an application that provides access to a single database. It could implement one object that provides a set of operations to access that database. Another might be a spelling checker that supports a single spelling checker object. The spelling checker object basically provides a service to check for spelling errors in some text.

Other applications might implement multiple objects that all support the same interface. A spreadsheet application might support multiple spreadsheet objects; one object for each spreadsheet file.

Other applications could implement multiple objects supporting multiple interfaces. For example, the spreadsheet application would support spreadsheet objects and cell objects. Or, the spreadsheet application might support both spreadsheet objects (for its clients) and callback objects (to receive data from an event channel).

It is even possible for a single application to contain multiple implementations of the same interface. For example, one could implement two different kinds of callback objects that support the same interface but are used in different situations. An example could be to have two implementations of callback objects used with two different event channels. The implementations differ by having different methods for responding to new data arriving from the event channel.

Designing Interfaces

The next step involves defining the interfaces to be supported by the objects. In some cases, existing interfaces can be used. For example, if a callback object is being implemented, the existing interface describes the operations invoked on the callback object by the service provider.

If the developer is defining an API so other programs can invoke the application to perform services, one or more new interfaces need to be defined that describe the operations that the application can perform. The IDL language and compiler are used to define the interface and construct the client stubs and server skeleton.

The following example is of an interface that a spreadsheet program might support:

Code Example 16-1 An Interface Example

```
interface Spreadsheet {
    exception Bounds;
    exception WrongType;

    Spreadsheet copy();

    string get_cell_value(in long row, in long column)
        raises(Bounds);

    void set_cell_value(in long row, in long column, in string new_value)
        raises(Bounds, WrongType);

    void establish_link_to_cell(in long row,in long column, in EventChannel ec)
        raises(Bounds);

};
```

In this design, each spreadsheet is an object. A spreadsheet can be copied and the values of individual cells can be read or written. In addition, one can establish a link to a cell. The virtue of this design is that there are no additional objects, like cells.

Although this interface is simple, it is sufficient to allow a client program to use a spreadsheet to perform a computation. The client programs first copies a template spreadsheet, writes the input values into the appropriate cells, and reads the output values from the appropriate cells.

Using Interface Inheritance

In IDL, interface inheritance can be used to define an interface that specializes another interface. For example, interface inheritance allows multiple interfaces to share an operation. And clients can use an object without knowing its most specific interface. Interface inheritance is like defining a C++ abstract class using public base classes, as shown in the following example:

Code Example 16-2 Interface Inheritance

```
interface Spreadsheet {
    exception Bounds;
    Spreadsheet copy();
    Cell get_cell(in long row, in long column)
        raises(Bounds);

};

interface Cell {
    void establish_link(in EventChannel ec);

};

interface NumericCell : Cell {
    double get_value();
    void set_value(in double new_value);

};

interface StringCell : Cell {
    string get_value();
    void set_value(in string new_value);

};
```

There is also another way to define a spreadsheet interface that takes advantage of interface inheritance. This design is very similar to the previous design, with the main difference being that each cell is an object in its own right. The spreadsheet provides an operation to return cell objects.

Inheritance is used to define two different kinds of cells, numeric cells and string cells. All cells support an operation to establish a link. This operation is defined in the common parent interface, cell. It is not necessary to know what kind of cell one has to establish a link to it. However, numeric cells and string cells support different operations to read and write their values. The operations differ in the types of the parameters and results. Unlike the previous example, numeric cells accept and return numeric values, not strings. The advantage of using interface inheritance in this example is that it is possible to define these specialized operations that are meaningful only to certain kinds of cells.

Implementing an Object

Once the interfaces of the objects are defined, the implementations of those objects must be written. There are several things that must be done when an object implementation is defined.

First, a method must be provided for each operation that the object supports. In C, a function is written for each method. In C++, a concrete C++ class is written that defines a member function for each method.

Next, the persistent state for the objects must be defined. For example, it might be decided that each object has its own file, or that a file contains multiple objects. In the spreadsheet examples, each spreadsheet would correspond to a file. In the design that implements cell objects, the persistent state of a cell is a small part of the file that stores the state of the spreadsheet.

The DOMF stores a small amount of private data with each object. This data can be used to store a pointer to the persistent state of an object. For example, for a spreadsheet object, one might store the name of the file in this private data. For a cell object, you might store the row and column of the cell. This allows the program to find the persistent state when a request comes in.

It is also unnecessary to worry about starting the program or calling methods. Once the program is registered with the DOMF, it will start the program as needed and invoke methods whenever a request is received.

The basic tasks of writing an object implementation are illustrated by using the first spreadsheet example, where each spreadsheet is a single object that implements the *Spreadsheet* interface. Code Example 16-3 is written in C++. To implement an object in C++, a C++ concrete class is written called `SpreadsheetImpl`.

Code Example 16-3 SpreadsheetImpl

```
class SpreadsheetImpl : public SpreadsheetBOAImpl {
    public:
        SpreadsheetRef copy();
        char *get_cell_value(const long row, const long column);
        void set_cell_value(const long row, const long column,
            const char *new_value);
        void establish_link_to_cell(const long row, const long column,
            const EventChannelRef ec);

        static SpreadsheetBOAImpl *lookup(const ObjectRef oref);
        SpreadsheetImpl(const ObjectRef oref); // constructor

    private:
        // transient data for spreadsheet
        YourDataStructure *data; // pointer to the internal data structure
};
```

This class inherits from the class `SpreadsheetBOAImpl`, which is defined by the IDL compiler. (It is the C++ skeleton corresponding to the `Spreadsheet` interface.) `SpreadsheetBOAImpl` defines several virtual member functions that are implemented by this implementation class.

The first four member functions in this class correspond to the four operations in the IDL interface. These member functions are invoked to perform an operation on the spreadsheet object. These member functions are defined as virtual member functions in the base class. This class provides an implementation of each of these member functions. (It is unfortunate that C++ requires the definitions of these functions be repeated.)

The next two functions relate to how instances of this class are created. The static member function called `lookup` is invoked by the DOMF to find the C++ object corresponding to a given DOE object. The constructor is used to initialize the state of a new C++ object.

Finally, the class defines the data representation for the C++ object. This is transient data associated with a spreadsheet while it is active in the program. In this example, a pointer to the existing internal data structure is stored that the application uses to represent a spreadsheet.

First look at the implementation of the lookup function and the constructor. The role of the `lookup` function is to return the C++ object that corresponds to a given spreadsheet. The spreadsheet is identified by an object reference. This function may be invoked by the DOMF multiple times in a given activation of the spreadsheet program. The first time it is invoked for a given spreadsheet it must create a new C++ object. On subsequent invocations, it returns the same C++ object.

Code Example 16-4 SpreadsheetBOAImpl

```
SpreadsheetBOAImpl *
SpreadsheetImpl::lookup(const ObjectRef oref) {
    SpreadsheetID id = get_spreadsheet_id(oref);
    SpreadsheetBOAImpl *p = registry->lookup(id);
    if (p == NULL) {
        p = new SpreadsheetImpl(oref); // create C++ object for spreadsheet
        registry->register(id, p);

    }
    return p;

}
SpreadsheetImpl::SpreadsheetImpl(const ObjectRef oref) {
    FileName fn = get_spreadsheet_filename(oref);
    data = read_spreadsheet_file(fn);

}
```

The lookup function is implemented using a lookup table. The lookup is then performed based on an identifier associated with the spreadsheet, which serves to distinguish the various different spreadsheets implemented by this program. The identifier was assigned when the spreadsheet was originally created and the DOMF was requested to store it for the developer as the implementor of the object.

The constructor is called when the C++ object is created. It reads in the spreadsheet data from a file and creates the internal data structure. This part of the original application is unchanged. The file name is stored for the developer by the DOMF, just like the identifier.

```
char *
SpreadsheetImpl::get_cell_value(const long row, const long column) {
    return (data->get_cell_value(row,column));
}
```

The member functions are very simple to write and they work directly on the internal data structure. In this example, it is assumed that the code needed to support the service already exists.

Architecting for Concurrency

So far there has been considerable discussion of the “nuts and bolts” of using DOE technology: how to write code that uses objects or implements objects. There are two interrelated, higher level design issues that bear mention: architecting for concurrency and supporting a user interface.

One of the key issues in migrating a program to DOE is deciding how to handle concurrency. Concurrency is critical because it affects both clients and servers.

Clients are concerned about concurrency because requests can take a long time to complete, especially if the request is going across the network. Therefore, as a client it is desirable to do some computing while the request is being processed. This is usually the case if the program implements a user interface. It is also probably a good idea to use multiple threads so that the user interface will remain alive while a request to an object is outstanding. Even when a request is expected to be handled quickly, one may want to be prepared for the unusual case where something goes wrong and it hangs for a long time.

Servers are concerned about concurrency because servers can receive multiple concurrent requests. When designing a server, it is important to consider how to handle concurrent requests. At a minimum, ensure that concurrent requests do not conflict with each other and corrupt the server’s internal state.

If the application is both a client and a server, then it must be concerned with the possibility of deadlock. Deadlock can easily occur if the application blocks when it makes a request. If that request causes a synchronous request to be sent back to the application, that request will block, and the result is deadlock.

This situation could easily arise if another object is used that talks back to the application using a callback object. Consider the following scenario: A client issues a request to the object. All other requests are blocked while servicing this request. A request is issued to another object, providing it a callback object that the program implements. That other object issues a request to the callback object, blocking on that request. That request is blocked because the application is not servicing new requests. Both programs are now blocked waiting for the other to proceed and hence, deadlock.

When designing a new program, often the best approach is to use multiple threads. Using multiple threads allows a server to process multiple requests concurrently. It allows a client to keep its user interface alive while another request is pending. However, most existing applications are single threaded. It may not be possible to invest the engineering effort to convert all of the code to use multiple threads. Fortunately, there are techniques that can be used to address concurrency issue that minimize the amount of old code that must be changed.

Summary

Project DOE reflects SunSoft's significant commitment to future object-based applications. As an OMG-compliant implementation, DOE will afford complete interoperability of applications while bringing the benefits of objects to distributed, enterprise computing networks.

Developers can begin their transition to objects today, by migrating their development activities to object-oriented languages such as C++, and by utilizing SunSoft's ToolTalk interface. The paradigm shift in developing applications with reusable components, will spawn new applications and opportunities for developers and end-users alike.

SunSoft is committed to leveraging its established expertise into new areas of enterprise computing, including distributed, object-oriented technology. And as the leading provider of distributed client-server based systems, SunSoft understands the needs of developers and users of applications in today's complex computing environments. Building a technology foundation for future object-based environments is a major undertaking, and SunSoft is committed to a leading role in delivering on the promise of object-based computing.

Bibliography

Solaris OpenWindows: OpenWindows V3 Collection: Release Reports and White Papers, Part Number 91021-0, SunSoft Inc., Mountain View, CA

Solaris SunOS 5.0: SunOS 5.0 Multithreading and Real-Time, Part Number 91025-0, SunSoft Inc., Mountain View, CA

Solaris ONC: Design and Implementation of Transport-Independent RPC, Part Number 91028-0, SunSoft Inc., Mountain View, CA

Solaris SunOS: SunOS 5.0 Release Report, Part Number 91023-0, SunSoft Inc., Mountain View, CA

Multithreading and Real-Time in Solaris: Terms and Concepts, Part Number 91024-002, SunSoft Inc., Mountain View, CA

The ToolTalk Service, Part Number 91022-002, SunSoft Inc., Mountain View, CA

Introduction to the ToolTalk Service, Part Number 91031-002, SunSoft Inc., Mountain View, CA

Solaris OpenWindows: ToolTalk in Electronic Design Automation, Part Number 91032-0, SunSoft Inc., Mountain View, CA

Tool Inter-Operability: A Hands On Demonstration: A Simple Demonstration of How the ToolTalk Service Works, Part Number 92205-001, SunSoft Inc., Mountain View, CA

Solaris ONC: Network Information Service Plus (NIS+), Part Number 91027-0, SunSoft Inc., Mountain View, CA

Solaris ONC+: Network Information Service Plus (NIS+): An Enterprise Naming Service, Part Number 92245-001, SunSoft Inc., Mountain View, CA

Project DOE: Distributed Objects Everywhere, Part Number 91035-0, SunSoft Inc., Mountain View, CA

*The ToolTalk Service: An Inter-Operability Solutio*n, Part Number ISBN 013-088717-X. SunSoft Press/Prentice Hall, Englewood Cliffs, NJ

ToolTalk and Open Protocols: Inter-Application Communication, Part Number ISBN 013-031055-7, SunSoft Press/Prentice Hall, Englewood Cliffs, NJ (June 1993)

The Common Object Request Broker: Architecture and Specification, OMG