Portable Distributed Objects (PDO): The NEXTSTEP/OpenStep Object Model

Executive Summary

An object model defines the ªlook and feelº of objects from a developer's point of view. It describes the built-in properties of objects (such as inheritance) and the rogramming language(s) that provide access to the model's facilities. To be effective for enterprise computing, an object model should also describe general-purpose services, such as distribution and interoperation. Examples of object models include Microsoft's OLE/COM, the Object Management Group's CORBA, and NeXT's Portable Distributed Objects (PDO) object model.

Viewed from a business perspective, the PDO object model offers an unmatched combination of power, maturity, and interoperability. The model's power enables software developers to respond more rapidly to the changing information requirements of today's agile and innovative enterprises. Proven and refined over six years and three major releases, the PDO object model can be selected with confidence for ªbet the enterpriseº applications. While competing object models slowly mature, the PDO object model is already supplemented with application-building frameworks and tools that are the industry benchmarks for rapid development. The PDO object model is particularly well-suited to heterogeneous information processing environments. It is available in client and server configurations for most leading computer architectures and platforms. Alone among object models, it supports interoperation with Microsoft OLE/COM objects today; tomorrow it will be extended to work with CORBA objects.

From a technical perspective, the PDO object model combines comprehensiveness with simplicity. The model provides transparent distribution of objects across address spaces, local networks, and even the Internet; it supports object persistence in files and databases; its memory management facility is both fast and easy to use; PDO objects interoperate across different platforms and even different object models; and classes can be extended, modified, or repaired in the field without recompiling. Yet these featuresÐand moreÐare integrated in a simple, coherent design that is easier to learn and use than most other object models.

I. Business Perspective

The software world is becoming inundated with different kinds of objects. There are PDO

objects, Smalltalk objects, OLE/COM objects, C++ objects, CORBA objects, and many others. Each kind of object is governed by a different object model: a description of the intrinsic properties of objects and of the language(s) by which objects can be manipulated. Because they all describe objects, object models have many similarities. But their differences can be substantial. For example, all object models are similar in that they support inheritance, but they are different in the extent to which they support it: some provide single inheritance, some multiple; some support implementation inheritance, others only interface inheritance.

Object Models Are Important

When an organization commits to object-oriented development, it must make a crucial decision: which object model, or models, should it adopt?

The decision is important because an object model is the foundation on which object-oriented software is built. An object model can empower or frustrate developers; can accommodate a range of hardware and operating system platforms or be limited to one or two; can support the distribution of objects flexibly and easily, or narrowly or not at all; and can quickly adjust to changing requirements or stubbornly resist them.

From the standpoints of training, staff communication, and code reuse, adopting a single comprehensive model is best. However, in many enterprises the adoption of a single object model is as unrealistic as the adoption of a single computer architecture and operating system. Many development groups will find themselves working with a mix of object models, as well as hardware and software platforms. In such an environment, interoperation among object models is paramount if the work embodied in objects is to be shared rather than duplicated, and if applications are to cooperate rather than merely coexist.

The best object model is one that has the innate power to surmount the challenges of enterprise computing without cutting developers off from other object models.

Portable Distributed Objects (PDO):
The NEXSTEP/OpenStep Object Model

The PDO object model was designed to fit the business needs of today's heterogeneous and fast-moving enterprise computing environments. The PDO model is notable for:

· Multi-platform availability. The PDO object model runs on major personal computers, workstations, and servers, guaranteeing deployment flexibility and independence from hardware and operating system vendors.

· Dynamic orientation. The PDO object model is based on a run-time architecture whose facilities enable the development of smarter objects; objects that can adapt to changes that occur after their classes were compiled, that can be customized with parameters supplied by other objects at run time, and that can be extended with new or replacement methods without recoding, recompiling, or restarting applications.

· Transparent distribution. The PDO object model supports a wider range of transparent distribution options than any other model, permitting server objects to be located (and moved) where they can be shared and make best use of hardware resources, while always appearing local to clients.

· Interoperation with other object models. The PDO object model interoperates with the Microsoft OLE/COM object model today, providing a gateway to popular desktop applications, and a means of sharing server objects among Windows NT, Windows 95, NEXTSTEP, and OpenStep clients; it will be extended to CORBA tomorrow.

· Extensive testing, refinement, and infrastructure. The PDO object model has been proven and enhanced through three major releases; its head start on the competition has made possible a range of development tools and application frameworks that will take years for the providers of other models to put in place.

One Object Model, Many Platforms
The PDO object model is available on a variety of personal computer, workstation, and server class machines from different vendors. Adopters of the PDO object model can choose a mix of hardware based on performance, price, familiarity, and serviceÐnot on the dictates of the object model. The following configurations will be available in 1995:

· NEXTSTEP for Hewlett-Packard PA-RISC, NeXT, SPARC, and Intel hardware platforms.

· OpenStep (an operating system-independent subset of NEXTSTEP 4.0) implementations for Digital's OSF/1 and SunSoft's Solaris operating systems.

· The PDO product (PDO) Ðthe PDO object model supplemented with server object development toolsÐ for Digital (OSF/1), Hewlett-Packard (HP/UX), Microsoft (Windows NT¸ and Windows½ 95), and SunSoft (Solaris and SunOS) operating systems.

Comprehensive.eps ¬

Figure 1: The PDO Object Model Spans Multiple Heterogeneous Platforms

No matter what mix of supported platforms and configurations an enterprise chooses, or how the mix changes over time, the uniform PDO object model ensures that the core behavior of all objects is identical. The model insulates developers from the heterogeneity of underlying platforms, freeing the enterprise to upgrade, decentralize or add hardware vendors in accordance with changing business needs.

A Dynamic Model Enables Smarter Objects

One of the great divisions in object model design is the degree to which the model makes decisions at compile time versus run time; said another way, the degree to which the model is static versus dynamic. The more dynamic an object model is, the more adaptable applications built on that model can be to changes that occur at run time. An application based on a dynamic object model can more easily accommodate new data types and new executable modules introduced at run time without requiring that the application be recompiled.

The PDO object model is rich in properties that support a dynamic model. These properties are defined and discussed in detail in the ªTechnical Perspectiveº section of this paper, and this section summarizes their business advantages.

Traditionally, the scope of an application is determined at the time it is builtÐat compile time. For example, an application used for financial analysis typically will never have more analytical tools than it had when it was compiled. But consider the advantage of designing applications that can be deployed with a limited feature set and then field-upgraded with new tools as they become available. The PDO object model makes it easy to create applications that can be extended without recoding, recompiling, or even restarting the core application.

The ability of a running application to load additional executable code is called dynamic loading. Dynamic loading is a powerful feature, but one that must be supported by other features of the object model for it to be a reasonable design choice. Some object models permit dynamic loading but at the cost of significant constraints on an application's architecture. For example, some object models require that all dynamically loaded objects inherit from a common superclass and that the methods used for communication be declared in a special way (as is the case with C++ virtual functions). These models may also require that all loadable modules be recompiled whenever a method is added to their common superclass. The PDO object model removes these restrictions on dynamic loading through its support of dynamic typing and dynamic binding (see the ªTechnical Perspectiveº

section for more information).

Extensible applications present additional challenges to the developer, so it's important that an object model provides adequate tools to ensure application integrity at run time. The PDO object model allows objects to reasonÐat run timeÐabout themselves and others, a feature called object introspection. For example, in the financial application, as a class is loaded from the tool module into the core application, it could be asked for its version number or whether it declares specific methods. Thus, the core application can decide whether the module contains, say, an early-style tool that supports a limited command set or a later one with extended features.

Dynamic models are sometimes perceived as being less safe in use since problems that might have been caught at compile time are deferred to run time. Because compile-time checking is valuable, the PDO object model lets the developer choose where to use dynamic facilities and where not. Where a dynamic model is a benefit, features such as object introspection make it safe. Where there's no clear benefit, the PDO object model allows the developer to write code using the static model and gain the advantage of type checking during compilation.

Dynamic models have also been criticized for exacting too great a performance cost for the benefits they offer. But the optimizations developed over the years at NeXT have yielded an order-of-magnitude improvement in performance over naive implementations of this model. For the vast majority of applications, the overhead of the PDO dynamic model is not perceptible.

Transparent Distribution
Object distributionÐacross address spaces, machines and networksÐcan expand the domain of object technology from stand-alone programs to cooperating applications, information sharing, and remote access of enterprise information. Object distribution is also the object-oriented counterpart of remote procedure calls: the foundation for client/server style applications. The PDO object model provides the widest range of distribution options while maintaining transparency to clients. Because PDO objects are location-transparent, clients interact with all objects identically, whether they are local, across the street, or across the country.

The PDO object model supports the following transparent distribution options:

· Objects can be local to a process.

· Objects can be distributed among processes on one machine (the basis for inter-application cooperation, such as embedding).

· Objects can be distributed among machines on the same local area network (to make them sharable, to put them on servers chosen for number crunching or database access).

· Objects can be distributed among machines anywhere on the Internet (to enable worldwide access).

The distribution flexibility provided by the PDO object model enables developers to put objects where they are needed and where they make best use of computing resourcesÐand to change the distribution as business needs change without affecting clients.

Interoperation With OLE/COM Objects

Although the PDO object model is sufficiently powerful to support all enterprise applications, in practice, it is unlikely to be employed exclusively. Just as many enterprises use different kinds of computers, they will use different object models. A world of heterogeneous object models is easier to manage if the models are designed to interoperate as the PDO object model is. Object model interoperation can promote reuse of work already embodied in ªforeignº objects, sharing of objects across disparate applications, and allocation (and reallocation) of server objects to the most effective hardware.

Microsoft's new OLE/COM object model is likely to be increasingly important for both market and technical reasons. As an example of the latter, consider that OLE Automation makes industry-standard applications such as Excel drivable by any application that also supports OLE Automation. Prudent development organizations may base many or most of their applications on a different object model, but rather than ignore OLE/COM objects, will make the most of them through interoperation.

NeXT's PDO product for the Windows NT and Windows 95 platforms supports bidirectional interoperation of PDO and OLE/COM objects over a network. Automatically generated adapter objects, running in a Windows NT or Windows 95 process, make OLE/COM server objects appear to PDO objects as PDO distributed objects. Similarly, adapters make PDO server objects appear to OLE/COM objects as OLE/COM objects.

Figure 2: Adapter Objects Provide Many Ways for PDO and OLE/COM Objects to Interoperate

The PDO adapter technology can be used in several ways:

· PDO objects can drive applications that support OLE Automation. A PDO object might, for example, open a remote Excel spreadsheet, send it values, and receive results computed by the spreadsheet.

· Windows 95 or Windows NT applications can use PDO distributed objects. For example, a heterogeneous environment might include powerful cycle or database servers based on the SPARC, PA-RISC, PowerPC, or Alpha architectures. If the servers run NEXTSTEP, OpenStep, or PDO products, objects residing on them can be shared by Windows, NEXTSTEP, and OpenStep applications.

· OLE/COM objects can be distributed across a networkÐa capability not yet available for OLE/COM itself. Distribution is accomplished by using the PDO adapters on two Windows machines to effect the network transfers, leaving the OLE/COM objects on the two machines with the impression that they are merely communicating across an address space boundary.

PDO and OLE/COM interoperation opens the way for cooperation between PDO and OLE/COM-conforming applications, opens up a new world of high-powered sharable servers to OLE/COM, and extends OLE/COM capabilities by making distribution possible today.

What About CORBA?
The Object Management Group's evolving Common Object Request Broker Architecture (CORBA) is a promising effort to form a vendor-independent standard for distributed object systems. The most beneficial and technically challenging aspect of the standardÐinteroperationÐis the subject of a recent specification. NeXT, along with SunSoft, Hewlett-Packard, and Digital, are currently working on the best interoperability strategy for the PDO object model and CORBA. There is, in fact, considerable similarity between the PDO distributed objects that are deployed today on multiple platforms, and the CORBA specification of distributed objects.

The Advantages Of A Head Start
In contrast to just-released or about-to-be-released object models, including CORBA and OLE/COM, the PDO object model was first deployed in 1989. Since then, thousands of real applications for real enterprises have been built on it; over 100,000 users work with these applications. As a result, the PDO object model is among the most thoroughly tested in

existence. Enterprises that commit to it today do not face the ªnew technologyº risk implicit in other models.

Six years of active deployment have been used to test the model as well as refine it. Anyone who has developed or used a complex system knows it doesn't turn out perfectly in the first release. 1994 saw the PDO object model's third major release.

There is a third advantage to an object model with a long history of successful use. NeXT has had time to build and refine tools and frameworks that drastically reduce the time required to develop applications. Since their introduction, the NEXTSTEP Interface Builder and Project Builder tools have been the industry benchmarks for object-oriented development tools. Frameworks from NeXT supply the structure and much of the code for user interfaces, utility classes (such as sets and dictionaries), database access, and object-oriented renderings of operating facilities such as threads and timers. All are available today. Although similar facilities for some object models may be lashed together from third party products, the resulting assemblages lack the integration, consistency, and trustworthiness of the NEXTSTEP originals.

Business Perspective Conclusion
Object-oriented software is built on an object model. It takes a powerful object model to effectively support the range of diverse and rapidly evolving information needs that characterizes today's enterprises. The PDO object model, with its dynamic orientation, multi-platform support, array of transparent distribution options, and six years of active deployment, is both powerful and proven.

As strong as any object model may be, in many enterprises, it is not helpful to stand alone. That's why the PDO object model supports symmetric interoperation with OLE/COM objects today and will be extended to work with CORBA objects tomorrow.

II. Technical Perspective

This section provides a more detailed and technical description of the PDO object model elements in Figure 3: the object model core; its syntax; and its services for memory management, persistence, distribution, interoperation, and evolution. Technical readers may want to compare the PDO object model facilities described here with those of other object models.

Elements.eps ¬
Figure 3: Elements of an Object Model

Object Model Elements And Terminology
An object model can be thought of as a core of properties and facilities, such as inheritance and service invocation, and one or more languages that give developers access to the core. Such a model is sufficient for writing object-oriented programs. Object-oriented systems require additional services such as distribution and persistence. Although they are usually not part of an object model's core, these services are of such fundamental importance and general utility that they are well considered as elements of an object model.

The core elements of an object model include:

· Objects. Objects are encapsulated abstractions that respond to requests for service. In enterprise computing, an object commonly represents a business entity, such as an employee, portfolio, or vendor. Most objects both respond to requests and make them of other objects.

· Methods. In some models, the services provided by an object are called methods. A request is made by sending an object a message which carries the name of the method and its parameters. In other models, a service is represented by a function; a request is made by invoking the function. Whether requests are modelled as messages or invocations makes little practical difference; this paper uses the message/method terminology.

· Separation of interface and implementation. An object's interface is distinct from its implementation. The interface defines the object's methods to clients of the object; it is the contract between the object client and implementor that describes what the object will do. The implementation is the embodiment of how the methods work, and is encapsulated to prevent clients from depending on implementation details which might be changed.

· Classes. A class defines the methods for the objects of one kind. Objects of the same class are called instances or members of the class. In some models, classes exist only at compile time; in others they are also present at run time in the form of class objects.

· Inheritance. Class definitions may be arranged in hierarchies, with subclasses inheriting the methods of superclasses. Some models support single inheritance (a class has exactly one superclass), others support multiple inheritance
(a class may have multiple superclasses). Some models support interface inheritance, some support interface and implementation inheritance.

· Polymorphism. Methods are polymorphic, that is, different classes may support the

same method, yet each class can implement it differently. The result is a simplification for clients who can treat diverse objects similarly, sending them the same message and leaving it to the objects to ªdo the right thingº when they receive it. (Consider a print method which every object might implement differently.) In some models polymorphism is constrained by inheritance; in others polymorphism can be defined across any set of classes.

· Static or dynamic orientation. Object models vary in the degree to which they are static or dynamic. Purely static models make decisions at compile time; the compiler must be told the class of every object used in a program, and the compiler computes the address of the method that each message will be sent to. A purely dynamic model defers these decisions to run time; object classes are not declared, and method addresses are looked up when messages are sent. A dynamic model may also make resources available to objects at run time so they can, for example, determine if an object supports a particular method before sending a message to it. Some models have both static and dynamic characteristics.

Although programming languages are sometimes thought to be synonymous with object models, in fact they are one element. However, an object model's core facilities may be so tightly bound to one language that they are indistinguishable; C++ is an example. By contrast, the CORBA object model core is independent of any language but is accessible using either C or C++, and additional ªlanguage mappingsº are possible. Notice that a C++ programmer working with the CORBA object model is actually working with two object models: C++ native objects and CORBA objects which are accessed through C++. The developer must decide which objects are to be C++ objects and which are to be CORBA objects, and then implement them differently. The same holds for Microsoft OLE/COM and C++ objects.

Object model services which are important for enterprise computing include:

· Memory management. The more complex an object-oriented system is, the more difficult it is to know when an object's memory can be released. Memory management is even more complicated when objects are distributed. Some models provide no memory management assistance. Others take complete responsibility for it, periodically collecting and recycling the ªgarbageº memory allocated to unused objects. Still other models take a middle approach, providing substantial automation without incurring the performance degradation of garbage collection.

· Distribution. Distributing objects can promote information sharing, inter-application cooperation, scalability, and flexible hardware allocation. Distribution is a natural way to separate applications into client and server components. For distribution to work well, it

must be transparent from the client's perspective, that is, all objects should appear to be in the client's address space. Object models vary greatly in their support for transparent distribution, from none, to distribution across address spaces, to distribution across a local area network, to distribution across the Internet.

· Interoperation. As transparent distribution makes a client independent of an object's location, interoperation can make a client independent of an object's implementation language, platform architecture and operating system, and, ultimately, even its object model. The more an object model interoperates, the better it accommodates the diversity of the typical enterprise computing environment.

· Persistence. Many business objects must persist independently of the applications that create and use them. Customers, vendors, and orders are examples. Object models may support the persistent storage of object data in files, relational databases, or object-oriented databases.

· Evolution. Enterprise information needs are changing faster than ever. The encapsulation of implementation, which is characteristic of all object models, helps confine the effects of change, as does the transparent distribution supported by some models. However, some object modelsÐnotably dynamic onesÐgo farther, permitting methods to be altered at run time without recom-piling, assisting with the management of multiple versions, and supporting the development of generic objects whose behavior can be customized by parameters supplied at run time.

Core Elements
The PDO object model is like many in its fundamentals:

· Objects are self-contained units that group a data structure (variables) with methods, the functions that affect or make use of the data. These methods respond to messages sent by other objects.

· The interface and implementation of objects of one type is defined in a class. The interface to a class is public; its implementation, however, is private, or encapsulated.

Major areas where the PDO object model differs from othersÐinheritance and dynamicsÐare described in the next sections. Some smaller, but important, differences can be described more succinctly:

· In the PDO object model, classes themselves are represented as objects at run-time.

Beyond providing object creation, initialization, and other class-wide services, class objects underpin much of the model's dynamic orientation (as discussed in the ªBusiness Perspectiveº section). For example, a class object can be queried at run-time to determine an object's inheritance lineage or which methods it implements.

· Many object models tie polymorphism to inheritanceÐrequiring that objects be related to one another before they can receive identical messages. The PDO object model removes this restriction, thus removing a design burden from programmers. For example, in NEXTSTEP/OpenStep a programmer can act on an array of objects (telling each to update, display, or do some other action) without first ensuring that they all descend from a common class.

· The PDO object model does not dictate that every entity be an object; for example, loop counters can be ordinary integers, giving the model the ability to support applications with serious number-crunching requirements.

Inheritance

Although inheritance is often considered monolithically, it is better divided into interface and implementation inheritance, since some object models, including CORBA and OLE/COM, provide only the former. The PDO object model supports single implementation inheritance and multiple interface inheritance. The combination can be used to create a subclass (reusing all or part of its superclass's code) and to express the protocol(s) a class conforms to.

Interface inheritance is an intellectual lever that improves developer productivity by making classes easier to comprehend. If Giraffe is a subclass (that is, inherits from) Mammal, a developer who understands mammals but has never worked with a giraffe nevertheless knows a lot about it because it's a kind of mammal. As important, it's equally clear what a giraffe is not: when working with giraffes, it's not necessary to think about trees, for example.

Multiple interface inheritance permits clusters of generic methods, for example, lock and unlock, to be defined in one interface that can be implemented by any class, whether or not it is a subclass. By inheriting the ªlockingº interface, for example, a class advertises to client developers that its objects can be locked in a well-understood wayÐthat the class, in other words, observes the ªlockingº protocol. The NEXTSTEP/OpenStep facility for defining a multiple inheritable interface is in fact called a protocol. A class indicates that it supports one or more protocols by formally adopting them. The class developer implements protocol-defined methods in the way that makes sense for the class, in keeping with the object-

oriented principle of polymorphism.

The PDO object model supports single implementation inheritance. Where interface inheritance saves developers time in understanding class relationships, implementation inheritance saves them effort in constructing subclasses. For example, imagine a View class which provides its subclasses with the structure for drawing and printing. Among other things, such a class must define a coordinate system and standard operations (scaling, rotation, etc.) on that system. Code of this complexity should be written once and inherited by all subclasses that deal in drawing. The benefits of such code reuse are clear.

Although the benefits are evident, implementation inheritance is the subject of some controversy. Some contend that implementation inheritance is not practical since a subclass implementorÐin the absence of superclass source codeÐcan't be sure whether a subclass method that overrides inherited behavior will interact destructively with other superclass methods. Others contend that although implementation inheritance may work for one release of the superclass, as soon as the superclass is modified in a later release all subclasses will break.

However, there is an existence proof that implementation inheritance can work: The NEXTSTEP/OpenStep class libraries have seen three major releases since 1989 and numerous minor ones, all without fundamentally disrupting the subclasses that developers have built from them. In most cases, the class libraries have been updated without requiring the modification or recompilation of client classes. How is this possible?

To support implementation inheritance, classes must be designed and updated with implementation inheritance in mindÐsomething that takes experience on the part of the class designers. In addition, classes have to be well documented so that subclass implementors know how superclass methods operate and interact. Finally, implementation inheritance must be supported by the language tools that express the object model. Some object models (notably the one associated with C++) would require that a subclass be recompiled whenever an inheritable method is added to the superclass.

Dynamics

The PDO object model provides the flexibility of a fully dynamic model and, at the developer's discretion, the safety of static type checking. The dynamic model is embodied in a run-time system that performs some services automatically (such as looking up the address of the method to receive a message) and offers other services that applications can invoke as needed. The run-time system is implemented as a small library whose memory requirements are shared across all applications running

on a host.

In a static object model, the compiler determines the address of the method that is to receive a particular message. Another way of saying this is that a message is statically bound to a method. In the PDO dynamic object model, the binding of message to method is performed dynamically at run time: messages are dynamically bound to methods in the PDO object model. Ignoring optimizations, the run-time system determines the receiving object's class, then matches the name in the message against the class's method names; if there is no match, it searches up the class hierarchy, ultimately generating an error if the method is not found. (Optimizations can't, however, be ignored; caching and other techniques make the NEXTSTEP/OpenStep run-time system fast enough that message dispatch time is rarely noticeable.)

When combined with dynamic typing (described shortly), dynamic binding gives applications a flexibility that is difficult to achieve with a static model. For example, code written today to send a display message to any of three classes, does not have to be changed tomorrow when a fourth class with a display method is added to the application. (The four classes do not have to be descended from a common ancestor, as is required in the late binding facility provided by some object models, notably C++.) In other words, dynamic binding makes the development of extensible applications routine.

The PDO object model supports a range of object typing options that permits the developer to specify how strongly typed a reference to an object should be. Objects are referred to by pointers; declaring a pointer to be of type id allows assignment of any type of object; this is dynamic typing. But declaring a pointer's type to be that of a particular class limits assignments to objects of that class and its subclasses; this is static typing. Alternatively, an object pointer can be typed to a protocol (or multiple protocols) to ensure that the object only receives messages defined in the protocol. Thus, the developer, not the object model, makes the trade-off between stricter type checking and more generic code.

An application based on the PDO object model can be statically linked and loaded as a unit, or can be structured as a statically loaded core that dynamically loads classes asÐand ifÐthey are needed. As with dynamic binding and typing, dynamic loading can be the basis for more easily adapted applications. Class implementations can be updated or extended without relinking applications that use them, and new classes can be added without relinking. Moreover, a dynamically loaded application's memory usage is limited to the classes actually used in a particular execution. It is even possible, in the PDO object model, to dynamically unload a class, permitting updates to be made to running applications.

For the ultimate in flexibility, the PDO object model permits messages themselves to be constructed dynamically. An object building a message can determine the method name, the class of the receiving object, and the message parameters at run time. Again, the advantage is adaptability. Consider a generalized timer object, for example, that can be initialized with a time value, the id of an object, and a message to be sent to the object when the timer expires. One timer class, developed and tested one time, can be used to send any message to any object after a specified interval. Without the ability to construct the time-out message at run time, all timer clients would either have to accept a standard message or write their own custom timers.

The PDO dynamic object model makes it possible for an object to ask questions of itself or other objects. An example is a technique called delegation which permits the development of classes whose objects can be customized at run time. At key points in the execution of its methods, a delegating object sends a message to another object (its delegate) if such an object is present and if the delegate has implemented the method corresponding to the message. Depending on the classes involved, the semantic of the delegate message can range from a simple notification to a request for intervention.

For example, a delegate of an object that represents a window could prevent the window from being closed or from being resized beyond certain minimum or maximum dimensions.

By customizing the behavior of a delegating object, a delegate developer effectively reuses the delegating class without subclassing it. Delegation is possible because a delegating object can ask itself at run time if it has a delegate and can ask the delegate if it implements a method. Without such a delegate, the object behaves in some standard way. With one, it lets the delegate intercede in selected actions.

Syntax

The syntax for the PDO object model is a small superset of ANSI C called Objective C. Because the language base is true C, and because the extensions are minimal, Objective C is much easier for C programmers to learn than C++. The C base also permits legacy algorithms written in C to be easily incorporated into method implementations. In contrast to the CORBA C language mapping, which represents methods as functions which must have globally unique names, Objective C class and method names are easy to write and read.

The NEXTSTEP/OpenStep compiler and debugger also supports C++. Objective C and C++ statements can be mixed in the same source file, and a C++ method can send a message to an Objective C object, and vice versa. Developers do not have to reimplement

C++ code that's already working. Note that although Objective C and C++ objects can coexist in one application, they cannot be intermingled. It is not possible, for example, for a C++ object to be stored in an Objective C collection object (such as an array or set object) or to be directly distributed across address spaces.

Services
To support the development, deployment, and subsequent evolution of enterprise software, an object model must be enriched with a variety of services. The PDO object model has much to offer in the areas of memory management, object persistence and distribution, interoperation with other object models, and extensibility.

Memory Management
A common bug in all programs that allocate memory dynamically is the memory leak: the memory allocated to some no-longer-needed objects or data structures is not released, making the running application grow ever larger. The bug is understandable because it is often hard to tell when an object is no longer needed. It is easy enough for object A to know it no longer needs object B, but it is hard to know globally when no object needs object B and therefore object B's storage can be reclaimed. The PDO object model gives developers consistent, powerful tools to manage this problem.

Traditionally, one approach to relieving developers of the burden of memory management is automatic garbage collection, in which a system process periodically looks for objects to which no other object holds a reference. These objects are of no use since there is no way to access them, and so their storage is freed. Garbage collection is very convenient for programmers because it absolves them of responsibility for determining when an object's memory can be released. It is not as good for users because garbage collection is a processor-intensive task that can periodically make an application sluggish.

Another approach is reference counting, in which each object keeps track of the number of outstanding references to it. Creating an object sets its reference count to 1. To copy an object pointer, a designated method provided by the object must be invoked so the object can increment its reference count. Similarly, an object must be sent a designated message when it is no longer needed, so the object can decrement its reference count. When the object decrements its reference count to zero, it releases its memory. Reference counting is less prone to error than manual deallocation, but it is not foolproof; developers can forget to inform an object when it's no longer needed. It is slow and is overkill if applied to all objectsÐsome objects' lifetimes are known when they are created.

NEXTSTEP/OpenStep memory management approaches garbage collection in ease of use

and resistance to memory leaks without incurring the performance penalty. The single mechanism works identically for local and distributed objects. It is based on the observation that most applications are event-driven and that they create objects in response to an event, most of which are needed only for the duration of the application's responseÐthat is, for one iteration of the event loop.

When a method creates an object, the object's memory comes from a pool that has a lifetime of the current event loop iteration. The memory allocated to the object is reclaimed automatically when the application waits for the next event. Thus, by default, all objects are temporary. A temporary object can be passed out of the creating method (returned as a result) and further passed around freely, but its memory is automatically released when its associated event has been handled.

The minority of objects that must persist longer than an iteration of the event loop are managed by reference counting. An object is kept longer than the current event loop by sending it a retain message. An object that sends a retain message is responsible for sending a matching release message when it no longer needs the retained object. Retained objects delete themselves when their retain messages are balanced by an equal number of release messages.

Persistence
By copying an object's data to a file or some other location, the object can be made to persist between sessions of the same application or to be transported from one application to another. The PDO object model lets developers choose from a range of persistence facilities: the archiver stores object data in files, and NeXT's Enterprise Objects Framework stores object data in either relational or object-oriented databases.

Persistence facilities typically take the form of frameworks built on either files or databases. Storing object data in files is cheapÐthere is no need to buy or learn a databaseÐbut it is not necessarily easy. Files are simple byte streams, whereas objects often contain structured variables, such as arrays. Frequently, one object contains references to others. Such structured object data must be ªflattenedº and tagged with identifiers when it is written to a file so it can be reconstructed when read back into memory. Pointers must similarly be dereferenced and tagged so they can be rebuilt. The ªflatterº the objects, the better file-based persistence works.

The NEXTSTEP/OpenStep archiver directs the transfer of a compound object's data to or from a file or other stream-based medium, including shared memory or another application. To be archivable, a NEXTSTEP/OpenStep class implements methods which the archiver

invokes. (Objects must read and write their variables because they are encapsulated.) The archiver automatically keeps track of class information and ensures that even in a complex graph of objects each object is written only once.

Although file-based persistence requires little investment, files have limitations that are difficult to overcome:

· They do not work well for sharing: it is difficult to coordinate the access of multiple users to a common data store.

· They do not support transactions: crashes or power failures can leave file-based data corrupted.

Surmounting these deficiencies requires either writing something rather like a database, or better, using an actual one. Either relational or object-oriented databases can be used for persistent stores, and NeXT's Enterprise Objects Framework supports both types.

Relational databases, the information backbones of corporations, are fast, robust, and well understood. A persistence facility that is based on a relational database presents an opportunity to encapsulate existing data into objects, thereby incorporating it into new object-oriented applications. Using existing data in new applications means there is only one copy of the data, so there is no need to keep multiple copies synchronized (and deal with the nearly inevitable discrepancies). On the negative side, the relational database data model (tables of scalar values) was not designed to represent object data. Without the proper tools, it can be difficult to cleanly and efficiently map complex compound objects, for example, into tables.

NeXT's Enterprise Objects Framework provides the tools you need to unite object-oriented application technology with relational database storage. In addition to storing object data in relational tables, the Enterprise Objects Framework can create object variables from existing database rows. It can thus provide object-oriented access to the wealth of legacy data that is already stored in relational databases. Existing database applications are not threatened by the ªobjectificationº of rows they use; the framework does not alter table definitions, and runs all transactions through the database server.

In contrast to relational databases, object-oriented databases are very good at representing object relationships. For compound objects organized in trees and networks, object-oriented databases are both easier to use and faster than relational databases. Although object-oriented databases are relatively new and less proven than relational databases,

they are likely to gain more general acceptance in the coming years.

NeXT's Enterprise Objects Framework accommodates various database storage mechanisms through the use of replaceable adaptors, communication and translation links between the database and the object-oriented application. Adapters for Sybase and Oracle are included with the product; others can be readily developed for both relational and object-oriented databases.

Distribution
Objects are natural units of distribution. Because a message does not identify the location of the receiver object, it can theoretically be intercepted and re-routed with no effect on the sender. Similarly, a receiver doesn't know (and therefore doesn't depend on) the location of a sender. Thus, objects provide the potential not just for distribution, but for transparent distribution.

Despite the attractiveness of distributed objects, many object models support only process-local objects: senders and receivers must occupy the same address space. This is not surprising since for a long time the term ªobject-orientedº was usually followed by ªprogram.º Now organizations want to build object-oriented systems.

In contrast, the PDO object model gives developers tremendous flexibility in distributing objects with full transparency. Communicating PDO objects can be:

· Local: the sender and receiver reside in the same process (address space).

· Inter-process: the sender and receiver reside on the same machine but in different processes. This style of distribution supports inter-application cooperation. It can provide the basis for drag-and-drop, embedding with in-place editing, printer spooling, or for application-independent services such as spell checking or dictionary lookup.

Distribution.eps ¬
Figure 4: The PDO Object Model Supports Local to World-wide Transparent Distribution of Objects

· Inter-machine: the sender and receiver reside on different machines connected by a network. The ability to match hardware resources with object requirements (for example, cycle servers, number crunchers, or database servers) can make applications scalable without reprogramming. Hosting widely used objects on fast servers makes them sharable among applications. Not to be overlooked are the administrative advantages of putting

critical shared objects on secure and automatically backed-up machines.

· Inter-network: The sender and receiver reside on different machines in different networks connected by the Internet.

With any kind of object distribution, the key criterion is transparency. The more that remote objects resemble local ones, the less that developers have to learn, and the greater the flexibility for changing the deployment of objects without affecting applications.

The NEXTSTEP/OpenStep Distributed Objects (DO) facility enables a sender to communicate with an inter-process, inter-machine, or inter-network receiver. Distributed objects provide a natural structure for cooperating applications (running on the same machine) or client/server applications (running on different machines).

A distributed object developer describes the object's interface in a protocol (see ªInheritanceº on page8) which senders inherit. From a sender's point of view, there's no difference in sending messages to local or distributed objects. The only thing a sender needs to know about any object is its id and the messages it responds to. A distributed object is represented in the sender's address space by an automatically created proxy object. The sender interacts with the proxy as if it were the ultimate receiver. A message sent to a distributed object's proxy will not match the methods defined by the proxy; therefore, the run-time system does what it always does in such a case: it sends the proxy a forward:: message. The proxy responds to the forward:: message by passing the original message across the address space or network to the distributed object. Notice how dynamic binding and typing make proxies possible: dynamic binding allows a proxy to be substituted at run time for the distributed object; dynamic typing allows a single proxy class to represent distributed objects of any class.

Distributed objects are ultimately identified by name. They can't be identified by ids which are pointers and not valid across address spaces, whereas names are. Distributed object names are maintained by objects called connections. Sending a name to a connection returns a proxy for the corresponding distributed object. Thus a decision to distribute what has been a process-local object has no effect on clients of the object; only the code that initially produces the object's id (the code that creates the object) has to be changed.

Proxies copy parameters and results that are passed by value, and dereference and copy data passed by reference, allocating memory and creating pointers, as necessary. If a sender passes an object to a distributed object, a proxy for it is automatically created in the distributed object's address space; similarly, when a distributed object returns an object, the

message sender automatically gets a proxy for it. (There is also an option for creating sender-local copies of return results.) Because distributed objects can return objects, they can be employed as security guards for sensitive objects, returning them only to senders that supply appropriate credentials.

Interoperation
As transparent distribution can make a sending object independent of a receiver's location, interoperation can make a sender independent of a receiver's implementation. Interoperation extends the object-oriented principle of interface-implementation separation.

The PDO object model supports cross-platform and cross-model interoperation. Cross-platform interoperation extends inter-machine object distribution to include machines of different architectures and operating systems. It is crucial in heterogeneous networks, those whose hosts are the products of different computer vendors. If a sender can only communicate with receivers that run on the same kind of platform, either it will not have access to some objects, or objects will have to be multiply implemented (and kept synchronized as they are updated). Neither option is desirable.

The Portable Distributed Objects (PDO) product extends the Distributed Objects facility to the platforms shown in Figure 1. The PDO product consists of the PDO object model and development tools that can be invoked locally on a PDO host or remotely from a workstation running NEXTSTEP/OpenStep. PDO automatically resolves data representation differences (for example, integer byte order) between sender and receiver host architectures. A server built to run under NEXTSTEP/OpenStep using the Distributed Objects facility can be easily converted to run on the greater number of operating systems supported by PDO; usually a recompile is all that's necessary.

Many enterprises will deploy applications based on different object models just as they deploy computers based on different architectures. At a minimum, cross-model interoperation can eliminate object isolation and duplication. More ambitiously, it can allow the features of one object model to fill the gaps in another model; distribution or support for a particular platform (such as a powerful cycle or database server) are examples. Finally, cross-model interoperation has the potential to enable objects based on different models to be ªwired togetherº into coherent super-applications.

The PDO product for Windows NT and Windows 95 support transparent, cross-model interoperation of PDO and OLE/COM objects. Interoperation is achieved through the use of adaptor objects that intercept PDO distributed objects messages and translate them into standard OLE/COM function calls. Communication between the object models is

symmetrical; when adapters receive function calls from OLE/COM objects, they convert them into distributed object messages and transmit them back to the PDO objects, even if those objects are distributed over a network. As OLE/COM and PDO objects return new objects, their adaptors dynamically create and register them with both object models. Thus, the PDO product allows OLE/COM objects to interoperate with PDO and OLE/COM objects across a network.

Evolution
It is in the nature of complex software to change, usually by refinement and addition of features. The encapsulation of class implementations, which is central to any object model, is a help. A class implementation can be changed without having to modify clients. Unless the model supports dynamic loading, however, applications using an updated class implementation must be relinked. In static models, interface changes of any kind necessitate recompilation of clients.

The PDO object model simplifies class and application evolution. Dynamic typing makes it possible to introduce new classes into applications without changing source code or recompiling. The model's transparent distribution and interoperation facilities make it easy to restructure monolithic applications into distributed ones that operate in today's increasingly heterogeneous environments. Additional facilities not touched on in earlier sections of this paper include:

· Extensible interfaces: NEXTSTEP/OpenStep classes can be extended with new methods without recompiling clients subclasses. Because the model is dynamic, method dispatch tables are constructed as classes are loaded. Adding a method to a class makes the class's dispatch table longer but doesn't require that client classes be recompiled. Adding methods is the low-impact way to make classes more capable.

· Field-replaceable implementations: An Objective C construct called a category supports the addition of new methods and the reimplementation of existing ones. The methods defined in a category are inherited by subclasses, and cannot be distinguished at run time from those they supersede. A category implementation can be installed without having the source code for the associated class, making it a convenient way to distribute upgrades and patches.

Technical Perspective Conclusion
The PDO object model has the features that developers need to build and adapt complex software systems in the increasingly complex environments characteristic of today's enterprises.

· A core that provides the flexibility of run time decision-making and supports multiple interface inheritance and single implementation inheritance.

· A syntax that accommodates legacy C and C++ code and is far simpler than C++.

· Memory management that exhibits much of the simplicity of garbage collection while maintaining good performance.

· A range of persistence facilities based on files, relational, and object-oriented databases.

· Transparent distribution of objects across an address space or across the world.

· Object-to-object interoperation across multiple platforms and Microsoft's OLE/COM object model.

These features have been refined and proven where it countsÐin real applicationsÐsince 1989. The PDO object model is the most comprehensive and trustworthy choice for an enterprise's object foundation.

(C)1995 NeXT Computer, Inc. All Rights Reserved. NeXT, the NeXT logo, NEXTSTEP, OpenStep, and Portable Distributed Objects are trademarks of NeXT Computer, Inc. All other trademarks mentioned belong to their respective owners.