Maximizing leverage from an object database

by C. Alfred

With increasing frequency, object database management systems (ODBMSs) are being used as a persistent storage framework for applications. This paper shows that ODBMS frameworks provide a natural repository for supporting object-oriented systems, because they store and manage objects as their atomic units. In addition, these frameworks can offer a great deal of leverage to the developers of applications with the integration of two distinct paradigm shifts: the object-oriented development model, and the direct-reference storage model. Software developers who understand the implications of both paradigm shifts are more likely to use the technology effectively and realize most or all of the potential leverage. Highlighted is ObjectStore™ from Object Design, Inc., which is available as part of the IBM object database solution.

Over the past five years, developers of complex software systems have turned, with increasing frequency, to object database management systems (ODBMSs) to satisfy their requirements for persistent storage management. Today there are many deployed applications that are based on ODBMS technology. These applications span a wide range of problem domains, including engineering design, geographical information systems, office automation, and telecommunications, to name a few. In addition, a large percentage of commercial software development companies today either have a funded development project that incorporates this technology, or are in the process of evaluating this technology for use in an upcoming project.

This paper discusses each of the underlying paradigm shifts upon which today's ODBMS technology is built and focuses on the properties of applications that this technology serves best. The first section briefly introduces the concepts of object-oriented modeling, database management systems, and paradigm shifts. The section "Object Model Sources of Leverage" explores how the principles of object-oriented modeling supported by an ODBMS provide benefit, and the section "Direct-Reference Storage Model" discusses how the storage model used in ObjectStore** offers extra leverage. The "Comparison of Storage Management Technologies" section examines the synergy between these two sources of leverage, and draws a contrast between relational database management system (RDBMS) and ODBMS technologies. The object-oriented development model and the direct-reference storage model are available in the Object Design, Inc. product, ObjectStore, as part of the IBM object database solution for its customers. Finally, the section "Impact of ODBMS on Software Development Process" helps to examine the impact on the overall development process of using a direct-reference, object-oriented, persistent storage framework.

Paradigm shifts and their leverage

This section describes what an object database management system is, and identifies the paradigm shifts on which the technology is based.

Object databases as a source of leverage. A storage management system is a facility that provides persistent storage for data items used by one or more computer applications. A database management system (DBMS) is a type of storage management system that manages the logical definition and physical structure of that data and provides read and update access to the data. A DBMS helps manage distributed data by making the physical location of the data accessed by an application transparent to that application. Finally, a DBMS protects the referential integrity (the consistent application of changes to relationships) of the data from hardware or software failures, or from conflicting access by multiple concurrent applications.

Objects are program entities that have identity and play one or more roles in some larger system. They have responsibilities that are consistent with those roles, and collaborate with other objects (requesting and providing services) to carry out those responsibilities. Objects have a life cycle; they are created, they are moved through one or more well-defined states, and they are destroyed. Objects also store information about themselves and about the identity of other related objects.

An ODBMS is a database management system that stores and manages objects as its atomic units. An ODBMS framework is an object database management system that is embedded into an application (or set of related applications) by linking in a client library. This library has an application programming interface (API)—that is, a set of methods on related classes, or a set of related free-standing functions—that is called by the application in order to access services from the ODBMS.[1]

ODBMS frameworks such as ObjectStore, Objectivity/DB**, Versant**, and GemStone**, to name a few, provide a very significant source of leverage to developers of complex applications. Frameworks like these provide a persistent storage mechanism that fully supports the object model as a way of representing a computer system. In other words, objects are the unit of persistent storage, and objects that reside in the database are full-fledged entities that can be used in application programs, just like their counterparts that reside in transient storage (i.e., a data segment, stack, or dynamically allocated storage for a process).

ObjectStore, in particular, offers another source of leverage. Almost all other database systems are based on a read/write storage model. The ObjectStore Virtual Memory Mapping Architecture (VMMA)[2] is based on a direct-reference storage model where object layout in memory is mapped directly onto a disk (the concept used with the IBM Application System/400* [AS/400*] architecture). Typically, operating system support for direct-reference storage has been limited to the domain of processes that execute on the same computer. ObjectStore takes this concept and extends the domain to be any computer attached to a network. In other words, client application processes can execute on any node in a network, access one or more databases that reside on any node in the same network, and map regions of those databases directly into the virtual address space of their process.

ObjectStore extends the domain to be any computer attached to a network.

In addition to this, client applications under ObjectStore Release 3.0[3] running on entirely different machine architectures can have shared access to the same objects, as long as the two applications were compiled by compilers that use the same object layout. This means, for instance, that an application running under Operating System/2* (OS/2*) on an Intel 80486**-based computer can transparently share data with another client application running under Advanced Interactive Executive* (AIX*) on a RISC System/6000* system.

Leverage and paradigm shifts. At this point, it is appropriate to point out that ObjectStore (like any other framework) offers its potential leverage to application developers, not end users. This leverage magnifies the strength of the developers and amplifies the power of their efforts. In this way,

Table 1 Paradigms, paradigm shifts, and paradigm mismatches

Paradigm	Paradigm Shift	Paradigm Mismatch
According to Steven Covey, "the word *paradigm* comes from the Greek. It was originally a scientific term, and is more commonly used today to mean a model, theory, perception, assumption or frame of reference. In a more general sense, it's the way we 'see' the world—not in terms of our visual sense of sight, but in terms of perceiving, understanding, interpreting . . . a simple way to understand paradigms is to see them as maps." (*The Seven Habits of Highly Effective People*, Simon & Schuster, NY, 1989, p. 23.)	A *paradigm shift* is triggered by a significant technological or sociological breakthrough that enables a whole new frame of reference. The invention of the computer, the advent of the airplane, and the Civil Rights Movement are all examples of breakthroughs that triggered massive paradigm shifts in our society during the past century.	A *paradigm mismatch* occurs whenever a breakthrough triggers a paradigm shift, but the pre-breakthrough frame of reference is retained. An example would be trying to operate an airplane as if it were an automobile. (Imagine someone headed from New York City to Atlantic City for the weekend, taxiing a twin-engine plane down the high-speed lane of the highway, and approaching a toll plaza.) An equally humorous example would be trying to develop a computer system using an ODBMS as if it were an RDBMS.

it is like the car jack in the trunk of an automobile. When used correctly, a small amount of effort (perhaps the effort needed to lift 30 pounds to a height of six inches above the ground) is transformed into the power to lift 3,000 pounds to the same height.

Technological breakthroughs, such as the direct-reference storage model and the object-oriented development model, produce paradigm shifts. In other words, they change the technical ground rules so significantly that they require a new way of looking at things. Trying to incorporate the technological breakthrough while retaining an old frame-of-reference is called a *paradigm mismatch.* In the best case, a paradigm mismatch results in the loss of most or all of the potential leverage. In the worst case, the paradigm mismatch can result in "reverse leverage" that magnifies undesirable effects into serious problems. Returning to the car jack example, if the car is stopped on a hill (with the front of the car facing uphill) and blocks are not placed behind the downhill tires, the effort needed to lift 30 pounds can be transformed into the power necessary to launch a runaway, unmanned 3,000-pound projectile downhill. Table 1 summarizes the thoughts about paradigm shifts and paradigm mismatches.

Object model sources of leverage

In the previous section, I asserted that full support for the object model in an ODBMS was a major source of its potential leverage. In this section, I offer four reasons to help justify this position.

Complexity. There are several effective ways to model complex, static systems. Modeling complex, dynamic systems is a much more challenging problem. Please note the emphasis on the word *dynamic*, which clearly implies motion and change over time. Without an effective way to understand a system's dynamic behavior, a series of interdependent external events and the resulting impact of these events on subcomponents of a system can be quite confusing.

Consider a team sport like ice hockey as an example. When the teams are lined up for a "face-off," the individual players and their interrelationships are apparent, even to someone watching the game for the first time. Once the puck is dropped and play begins, the resulting motion can seem like a chaotic blur. The difference between a spectator who is a novice and one who is experienced is the ability to understand the roles of the respective players, and how the members of a team collaborate within the context of their roles to forward the purpose of their team (i.e., score more goals than their opponent). The accompanying sidebar on the object model explores this notion in more detail.

In my view, the real strength of object-oriented modeling is that it is a programming paradigm that reflects how people think, as opposed to how

computers work. While this benefit is far more intangible than tangible, its importance should not be underestimated. A development process that is organized around a model that mimics how people think about complexity, will improve the quality of communication among its participants. End users and developers can discover a common language for describing the requirements and overall behavior of a system. Two developers can more easily exchange ideas about how the system might be structured. Also, as developers are replaced over time, the presence of a well-defined object model makes it easier to understand the intent of the original developers.

Stability. Over time, the functions of a system are likely to change significantly, while the high-level objects in the problem domain remain relatively constant. For instance, a favorite word processor might add spelling and grammatical checking, but it still remains organized around documents, paragraphs, and format styles.

A system that has its foundation built on top of stable aspects will be more able to withstand change. Quality improves when developers are able to localize their changes to small, relatively independent parts of a complex system. Another way of stating this is that as the scope of a change widens, the risk of error increases, often at a faster rate. This is because one or more of the areas needing change might be forgotten, or the change might have undesirable side effects.

It is important to keep in mind that use of an object-oriented development process is an important, but not sufficient step toward improving the stability of a system over time. In order to achieve maximum stability, a system must be designed properly and make effective use of two related concepts: *abstraction* and *encapsulation.*

Abstraction supports the definition of classes that represent (or model) concepts in the real world, by focusing on essential properties while ignoring inessential details. Encapsulation exposes the essential behaviors of an abstraction, while hiding the implementation details of those behaviors and the internal structure of the abstraction's data attributes. Encapsulation also helps to reduce coupling between interdependent components of a system.

Abstraction within a system typically occurs at multiple levels. One example of this is the notion of vertical layering. A system can be decomposed into a set of subsystems, each of which is a distinct entity that has specific responsibilities and plays a well-defined role at a high level of abstraction. In turn, each subsystem can be further decomposed into its own distinct modules, at lower levels of abstraction. Each component at each level of abstraction presents an external interface that represents the services it provides to support its responsibilities, and hides the details of how it carries out those services from its collaborators. This tactic serves to reduce unnecessary coupling between components of a system. A second example of multiple levels of abstraction is closely tied to the concept of polymorphism, which literally means "many things." This notion will be discussed in more detail in a later section.

Wirfs-Brock[4] describes a design methodology called "responsibility-driven design" that focuses on the definition of the external interface to a class (or subsystem) as a set of contracts. These contracts, in turn, are composed of a set of related methods (services) offered by that class as part of its external interface. Each contract represents an agreement between the class and some subset of its collaborators, regarding the responsibilities of both parties. Over time, a class may enter into new contracts, or may amend its contracts by agreeing to provide additional services, but should never renege on an existing contract.

In essence, contracts between classes define a stable interface protocol. For example, the ObjectStore collections class library offers contracts for:

- Collection maintenance (insert, remove, replace, etc.)
- Querying (query, query_pick, exists, etc.)
- Status inquiry (cardinality, empty, has_index, etc.)
- Index maintenance (add_index, drop_index, has_index)

Note that the relationship of methods to contracts is many-to-many. A contract usually is composed of several methods, and a method (i.e., has_index) may be part of more than one contract.

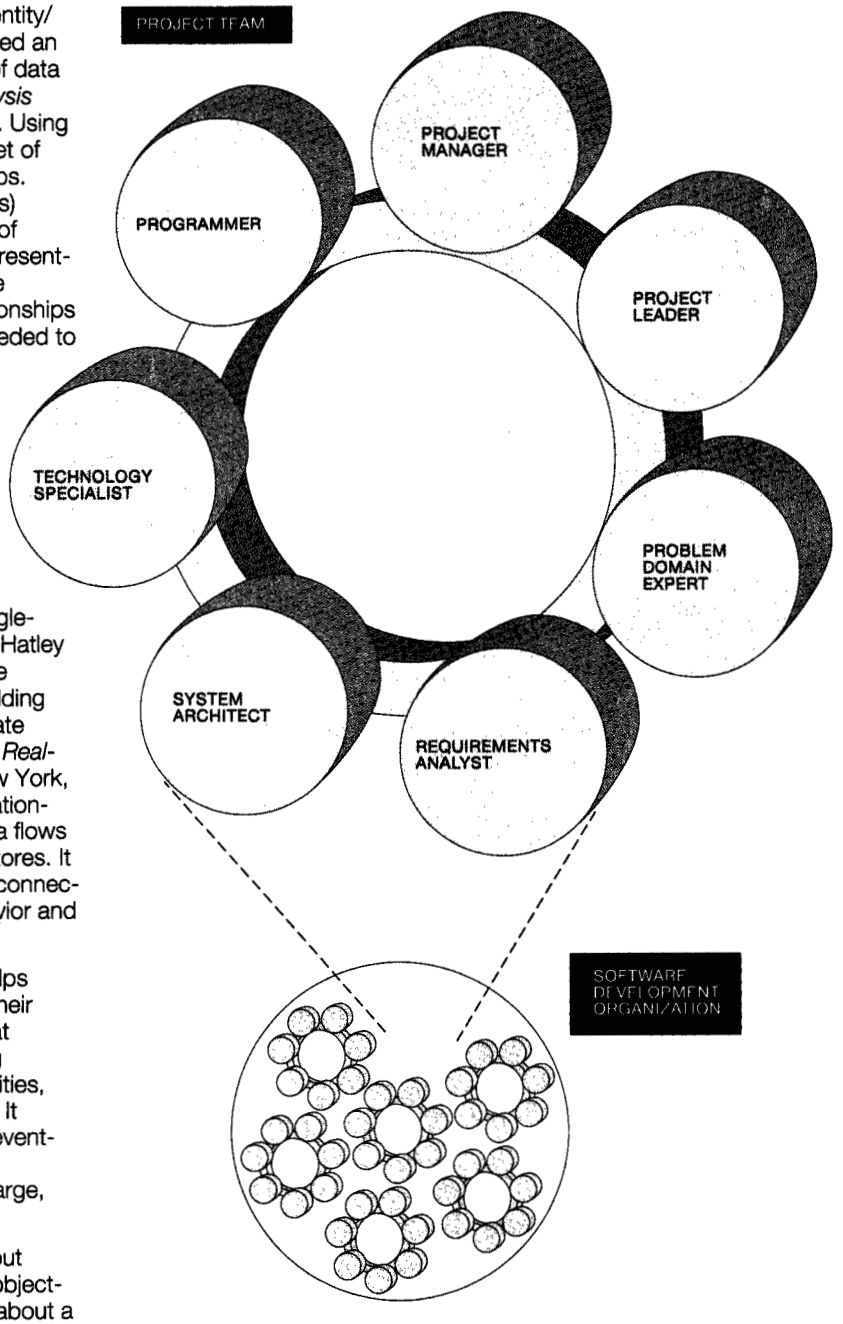
The object model reflects how people think about complex,

In the late 1970s, T. Chen introduced an entity/relationship modeling method that provided an effective way to represent the static structure of data (*Entity Relationship Approach to Systems Analysis and Design,* North Holland, Amsterdam, 1979). Using this method, information analysts identified a set of entities and modeled their semantic relationships. The end products of this method (E/R diagrams) were useful as the logical data model for a set of related application programs. Entities were represented as a row in a relational table, attributes were represented as columns in the table, and relationships highlighted the requirement for foreign keys needed to join related tables.

Also in the late 1970s, E. Yourdon and T. DeMarco introduced structured analysis methods designed to model the dynamic behavior of a system as a network of interconnected processes, where each process transforms one or more input data streams into one or more output data streams (T. DeMarco, *Structured Analysis and System Specification,* Yourdon Press/Prentice Hall, Englewood Cliffs, NJ, 1978). In the mid-1980s, D. J. Hatley and I. A. Pirbhai extended these methods to be suitable for real-time control applications by adding control flows and process state, and linking state transitions to process activation (*Strategies for Real-Time System Specification,* Dorset House, New York, 1987). In both cases, entities and their interrelationships were represented either as labels on data flows or invisible components of one or more data stores. It was quite difficult to represent or visualize the connection between dynamic, process-oriented behavior and static, data-oriented entities and relationships.

More recently, object-oriented modeling helps us to package the concepts of entities, their behaviors, and the chain reaction of events that drives these behaviors. It does this by focusing attention on entities and their roles, responsibilities, behaviors, attributes, associations, and states. It merges data-oriented, function-oriented, and event-oriented decomposition methods into a single, unified approach to modeling the behavior of large, complex systems.

As evidence that people naturally think about the dynamic behavior of a system in an object-oriented way, consider the way that you think about a software development organization. A software development organization is a part of some larger corporation, and it receives requests to design and develop computer software to solve some set of problems for some set of intended users. The software development organization employs a number of people, who collaborate in a (mostly) coordinated way to respond to these requests for service.



dynamic systems

We make sense of the behavior of the system (the work flow of the software development organization) by decomposing it into component parts (i.e., project teams, departments, etc.) and understanding the patterns of behavior of these components. As we study project teams, we notice similarities and differences among them. Each team is organized a bit differently, but most teams seem to be organized around a similar set of *roles*. Project manager, project leader, problem domain expert, requirements analyst, system architect, technology specialist, and programmer are examples of roles that commonly occur in medium- to large-scale projects.

The main benefit of defining roles is that it helps us to understand the dynamic behavior of a large, complex system by breaking it up into smaller, more comprehensible pieces. Roles represent patterns of behavior that occur commonly within similar types of systems or subsystems. Roles also help to identify *collaborations* inside the system. Each role is played by one or more *entities* (objects) and each entity may play multiple roles.

In each interaction between a pair of entities, it usually is possible to identify the role that each participant is playing. For example, project managers ask project leaders for estimated completion dates of project deliverables; project leaders work with system analysts to identify component tasks; and project leaders ask programmers to estimate the time needed to complete one or more tasks.

Entities that play a specific role carry a set of *responsibilities* that are consistent with that role. Each responsibility can be thought of as a set of related *behaviors* that the entity will exhibit in response to a request from an entity playing a related role. For example, a programmer has the responsibility to develop code for some component of the completed system. Interaction must occur with designers to understand exactly what the component must do and how it interacts with other components, source code must be written in the chosen programming language, the source code must be compiled and language syntax errors fixed, and the source code for unit tests must be written, and so on.

In order to carry out these responsibilities, an entity must possess knowledge. This knowledge may be represented by *attributes* (facts remembered by the entity) or *associations* (connections to other entities that can help when needed). In certain cases, an entity must maintain knowledge of its own *state* of being, especially when this state affects its behavioral response to a stimulus.

Component reuse. One of the often cited benefits of object-oriented technology is that it facilitates technology reuse. The principles of *abstraction* and *polymorphism* are important for this. Useful class libraries define interfaces to and implement behavior for abstractions such as strings, dates, collections, vectors, matrices, etc. Since these abstractions are commonly used in a wide range of programs, the functionality of the class library need not be separately implemented in each one. Examples of popular generic class libraries that are in widespread use today include Tools.h++** and Math.h++** (from Rogue Wave), the NIHCL** libraries (from the National Institutes of Health), and the Booch Components**.

Generic class libraries facilitate small-scale integration. A developer picks and chooses individual components that the developer wants to reuse in his or her application. These individual components may be combined to form higher-level constructs. For example, a symbol table may be implemented using a Set class and a String class.

Closely related to the concept of a generic class library is the concept of a framework. Booch[5] describes a framework as a "collection of classes that provide a set of services for a particular domain; a framework thus exports a number of individual classes and mechanisms that a client can use or adapt." Booch goes on to point out that frameworks may either be domain-neutral (they apply to a wide variety of applications) or domain-specific. The Microsoft (Corp.) Foundation Class Libraries** and the ObjectStore class libraries are examples of domain-neutral frameworks. A bond price/yield library and a geographical information library are examples of domain-specific frameworks. Where generic class libraries provide small-scale integration, frameworks facilitate large-scale integration. Frameworks usually are more complex than generic libraries, but they also provide a great deal of additional leverage.

Simplification. This benefit is closely related to the previous three. A system that is more understandable, stable, and loosely coupled is easier to enhance. In addition, the principle of polymorphism makes it easier for developers to add new functionality by specializing or extending existing classes. Taylor[6] describes polymorphism as "the ability to hide different implementations behind a

common interface, simplifying communications among objects."

In an object-oriented model, each object is an instance of some class. Different classes, however, may be related by inheritance. In other words, one class (called a *derived class*) is a specialization of another more general class (called a *base class*). Base classes and derived classes exist at different levels of abstraction. This distinction is useful for any object that collaborates with an instance of the derived class, because it can choose to interact with the object as an instance of whichever class provides the appropriate level of abstraction. This tactic simplifies system enhancement by enabling the addition of a new specialization at some lower level of abstraction, without affecting the components of the system that interact with instances of that class at any of the higher levels of abstraction.

For example, consider a class that represents the job of sales vice president. This job can be viewed at several different levels of abstraction, including that of employee, manager, executive, and vice president. An instance of this class would represent a specific sales vice president for a specific company during a specific interval of time. This object interacts with several other objects during its lifetime. Some of these, like the United States Internal Revenue Service, are interested in interacting with the object as an employee (i.e., by withholding payroll taxes). Others, such as a member of the Human Resources department, may interact with the object as a manager (i.e., through communication of equal employment policies). Still others, like the Corporate Counsel, interact with the object as an executive (i.e., via signature authority on contracts).

Direct-reference storage model

In an earlier section, I asserted that the use of a direct-reference storage model in ObjectStore was a major source of its potential leverage. In this section, I hope to elaborate on and justify this claim.

Read/write versus direct reference. In a read/write storage model, the memory-resident objects (data structures in a non-object-oriented paradigm) have an identity or layout that differs from the persistent representation in the database. Three variations of this read/write storage model are important:

- Nondatabase file systems represent the complex in-memory data structures in a much less general form in persistent storage (than RDBMSs). Sometimes this form is quite similar to the in-memory form (to minimize access time). Other times, it is a highly compressed form (to minimize storage requirements). In both cases, a typical access pattern is to read all or part of a file into memory, update the copies, then write the results back. Spreadsheet models and word-processing documents are two common examples.
- An RDBMS decomposes complex data structures into representations made up of simple two-dimensional tables (a process called *normalization*). Component pieces of the original data structure are scattered across the various tables. On access, the original data structure is reassembled by locating the components by performing *associative lookup* operations on each of the tables, searching for the items that match a particular attribute value, then combining the table subsets into a single result (a process called *joining*). On update, the inverse occurs.
- Many ODBMSs model the complex in-memory data structures in a very similar form in persistent storage, but link related objects using *handles* instead of pointers. Handles are unique object identifiers that must be opened or read before the underlying object can be used.

Direct-reference storage (otherwise known as memory-mapped file I/O) is a technique that allows a contiguous region of a file to be mapped directly into the address space of one or more processes. Updates to a virtual memory address within this region are translated by the operating system into updates to the underlying file. In short, memory-mapped files are used in place of the swap device, as a named backing store for virtual memory. This concept will be quite familiar to AS/400 veterans.

An ODBMS like ObjectStore, which is based on the direct-reference storage model, uses the same object layout in memory as on disk. In addition, the persistent and transient objects have the *same identity* (i.e., one is not a copy of the other). Interobject links are represented with ordinary programming language pointers, although the ODBMS

Table 2 Domains for information sharing in direct-reference storage models

Application Type	Domain for Accessing Shared Information
Single-threaded application	• No sharing
Multi-threaded application	• Single process, single machine • Each thread has its own stack, but shares a common heap and data segment
Two or more single-threaded applications with shared memory or memory-mapped I/O	• Cross-process, single machine • Each process has its own stack, heap, and data segment • Across processes, sharing is limited to the contents of a memory-mapped file or some other form of shared memory
Two or more multi-threaded applications with shared memory or memory-mapped I/O	• Cross-process, single machine • Each thread has its own stack • Threads in the same process share a common heap and data segment • For threads in different processes, sharing is limited to the contents of a memory-mapped file or some other form of shared memory
Single-threaded ObjectStore client applications	• Cross-process, cross-machine • Each process has its own private stack, heap, and data segment • Persistent objects in any database may be shared
Multi-threaded ObjectStore client applications	• Cross-process, cross-machine • Each thread has its own stack • Threads within the same process share their heap and data segment • Threads in different processes may share persistent objects in any database • Threads in the same process may need to take special action to ensure concurrency control

may map the pointers between process-specific and process-neutral form (known as "relocation" or "swizzling").

Shared access to persistent objects. As was mentioned earlier, a direct-reference storage model provides shared access to objects by concurrently running threads of control. The various types of direct-reference storage models differ from each other primarily in the domain over which shared information is accessible. Table 2 shows the different kinds of models along with the domain for accessing shared information.

Another important point to note here is that ObjectStore treats persistence as another kind of storage class, not as a behavior to be inherited from some special base class. This has several important ramifications. First, it means that any type of object can be stored persistently, including all of the types that are built into the programming language. Second, it means that the decision about whether an object is persistent or transient is made at the time the object is allocated. In either case, the storage layout of the object is the same. In almost all cases, exactly the same method code will work on both persistent and transient instances. The three noteworthy exceptions to this rule are:

1. All access to persistent objects must be performed inside a transaction (for multiprocess concurrency control reasons). Transient instances may be accessed at any time (i.e., inside or outside a transaction).
2. Persistent objects may not (by definition) point at transient objects outside of a transaction, and vice versa.
3. Method code for a class that allocates instances of related classes must include a decision about where to locate these instances (typically, the correct choice is to allocate in the same area of the database as the parent).

ObjectStore handles access and locking of persistent objects in a very transparent manner. When ObjectStore initialization occurs, a contiguous region of virtual address space for a process is reserved as the location where persistent storage will be mapped during a transaction. At the start of a transaction, the access mode of all pages in this region is set to no-access. When reference to a pointer into the persistent storage region is eliminated, a page fault occurs. The ObjectStore client library handles the fault, and is able to determine the 96-bit-wide database address that corresponds to the faulting address. It sends a request to the appropriate server, seeking to lock and fetch the desired page. If the page is available, the server grants the request and returns the page. The ObjectStore client makes some adjustments to the contents of the page, sets the protection to read-only, and returns from the fault handler.

Similarly, when the contents of the page are modified, another page fault occurs. This time, ObjectStore merely needs to have the server upgrade the lock. On success, the page protection is changed to read-write and the fault handler returns. The fact that most of the real work performed by the ObjectStore client library is hidden inside a page fault handler is what gives the ObjectStore API its nonintrusive "look and feel."

Database functionality. ObjectStore provides transparent access to objects in persistent storage, and supplies traditional database functionality to accompany it. This includes:

- Locking support to prevent inconsistent updates by concurrently running processes that access the same data
- Transaction commit/abort semantics (i.e., either all changes commit or none commit), including a two-phase commit protocol to coordinate updates affecting databases spanning two or more servers
- Automatic failure recovery on server restart, thereby preserving a transaction-consistent database
- Query-style access using attribute values (i.e., associative lookup)
- Evolution of the physical object layout in a database when the underlying schema is changed

In addition, some functions that are not commonly required of a DBMS are included as well:

- Ability to store and access multiple versions of a group of objects in order to preserve the attribute values and interobject associations as of a particular point in time
- Automatic maintenance of both sides of an interobject relationship whenever one side changes (for example, if John Doe changes from working at ABC Company and starts a new job at XYZ Company, a change to the link from John Doe to his employer will automatically remove John Doe from the ABC set of employees and insert him into the XYZ set of employees)

Minimize run-time performance overhead. In a direct-reference storage model, persistent objects have exactly the same structure as transient objects. As a result, the processing overhead for accessing a page of persistent storage exceeds the raw cost of locking and transferring that page by just a small percentage. This additional overhead consists of some minor adjustments made to the contents of the fetched page by the ObjectStore client library during the page fault handler. These include fixing up pointer values from process-neutral form to correspond to the address space setup of the accessing process, and matching the byte ordering and floating-point representation to the machine architecture requirements of the accessing process.

By contrast, in a read/write storage model such as the one used by commercial RDBMS products, when you access the database, the result is returned as a relational table (or a cursor on a relational table). This is not a form that is useful to an object-oriented programming language such as C++ or Smalltalk. In order to put the data into a useful form, one or more transient objects must be allocated and constructed from each row of the query result. This can be a very expensive transformation, especially if complex interobject associations must be re-derived from the values of joined fields.

Caching of objects. As was mentioned earlier, an ObjectStore application typically accesses a database by taking a page fault in the region of virtual address space reserved for persistent objects. The ObjectStore client library handles the page fault by having the appropriate server lock and fetch the page. Once a page has been ac-

cessed within a transaction, following any other pointer onto that page will be just as fast as resolving an ordinary pointer to transient memory. This is a very crucial point to keep in mind for computationally intensive applications.

In addition to caching pages referenced multiple times within the same transaction, ObjectStore also tries to cache pages that are used in successive transactions. The success rate of this cross-transaction caching depends on the physical structure of the database, and the access patterns of concurrent clients. Read-only pages may remain cached as long as no other client process is trying to update them. Updated pages may remain cached as long as no other client is trying to read them.

Comparison of storage management technologies

In the preceding sections, we have explored the primary sources of leverage available from an ODBMS like ObjectStore. A complete coverage of the sources of leverage available from other storage management technologies, such as relational databases and Compound Document Architectures, is beyond the scope of this paper. However, a reader who is familiar with these technologies is likely to observe that each one is based on a fundamentally different paradigm. As a result, it is reasonable to conclude that there are classes of applications for which different subsets of these storage technologies are well suited. For some classes of applications, one of the technologies will be a significantly better fit than the others. For other classes of applications, more than one of the technologies might be capable of providing an equally acceptable solution.

Based on this conclusion, it follows that the storage technologies complement each other, and a system of related applications may use more than one (if not all) of these technologies in combination. The purpose of this section of the paper is to assess the strengths of the respective storage technologies, with the goal of helping the reader to assess the suitability of each of the storage technologies to support the requirements of a specific kind of application.

How they add value. In order to put the comparison of the different storage technologies into context, I will start with a metaphor. In many ways, the capture and transformation of raw data into useful knowledge is similar to the capture and transformation of raw crude oil into useful petroleum-based products. These similarities exist in spite of the fact that the former represents the processing of intangible goods, while the latter represents the processing of physical commodities. Since it is often easier to understand tangible, physical processes, a metaphor such as this one can add to our understanding of the nature of less tangible, abstract processes.

The use of multiple storage technologies may complement each other.

The "crude oil" metaphor. Crude oil is a raw material that is the primary source of a wide variety of useful end products. These products range from fuels such as gasoline and home heating oil, to lubricants such as grease and motor oil, and to derived materials such as plastics, synthetic fibers, and other polymers.

It is interesting to note that for some of the types of end products, the raw material does not possess the essential characteristics of the derived product. For example, crude oil in its raw form cannot be used as a substitute for plastic or a synthetic fiber. However, for other types of end products, the raw material also possesses the essential characteristics of the end product. For example, fuels are burned to produce energy, and lubricants are used to reduce friction. However, even in these cases, it is useful to note that the raw material is less effective than the end products for which it could be a substitute. Crude oil burns, but not nearly as efficiently as gasoline or home heating oil. Also, crude oil is slippery, but is also too sticky to provide the viscosity of motor oil or grease.

The obvious conclusion here is that while crude oil is a very valuable natural resource, the vast majority of its economic value comes from its

transformation into some other more useful form. This added value comes from one of three activities:

- Processing. This activity represents the steps that actually transform the raw material into an intermediate form, or transform any of the intermediate forms into another intermediate form, or a useful end product.
- Transportation. This activity represents the steps that move the raw crude, or any of its intermediate or end products, to a more useful location.
- Storage. This activity represents the containment of the raw crude, or any of its intermediate or end products, during the time period between when it is produced and when it is used or consumed.

Implications. It is useful to observe, at this point, that the primary source of added value is processing. Transportation and storage each provide secondary value. The value contribution of transportation depends on the extent to which it improves the effectiveness of a related processing or consumption step. The value contribution of storage depends on the extent to which it improves the effectiveness of a related transportation, processing, or consumption step.

This observation also applies to capture and transformation of raw data into useful knowledge. In this set of activities, processing is the primary source of added value. Intra- or inter-process transportation and storage add secondary value. As with crude oil, this value increases to the extent that the transportation or storage activity increases the effectiveness of the processing it supports.

The implication of the crude oil metaphor is that transportation and storage are both means to an end, rather than ends in and of themselves.

For evidence that this is true, consider the fact that pumping home heating oil from a storage tank into an oil truck, transporting it to your home, and pumping it into the fuel tank that supplies your furnace is a service worth paying for. By contrast, transporting raw crude oil from the wellhead to your home and storing it in a tank has virtually no value. This implication also applies to the transformation of raw data into knowledge. Storage management technology should be viewed as a means to an end, and should not be considered as an end in itself, in isolation of the types of processing that it is intended to support.

Limitations. As with any metaphor, the crude oil metaphor has its limitations. While the process of transforming crude oil into useful petroleum-based products is similar to the process of transforming raw data into knowledge, these processes are not identical. In particular, the transformation of raw data into knowledge is not quite as linear a process. Figure 1 is an oversimplified representation of the process of how this transformation occurs. As this diagram illustrates, there is a relationship among six distinct components: raw data, refined data, concepts, information, models, and knowledge.

Raw data are transformed into knowledge using filtering, deduction, organization, synthesis, and verification.

A few points are worth noting at this stage. First, raw data and refined data have similar properties. Both have relatively a simple structure, and typically can be represented by simple, record-oriented structures. By contrast, concepts represent loosely formed ideas, and frequently are in the form of textual outlines, prose, or freehand sketches.

As Figure 1 suggests, information can be seen as the merging of concepts and refined data; human beings classify and structure observations about the world around them, in an initial attempt to understand the interdependencies among these observations. Another way of saying the same

thing is that information is data with semantic structure added.

Information and concepts are then synthesized into hypothetical models that try to represent systems as hierarchies of related components, and explain the behaviors of those systems in terms of cause and effect relationships among the components. Finally, models are tested by using them to predict future events based on known information. Once a model is verified, it becomes part of a larger body of knowledge and is useful in inference and inductive reasoning to develop new concepts and ideas.

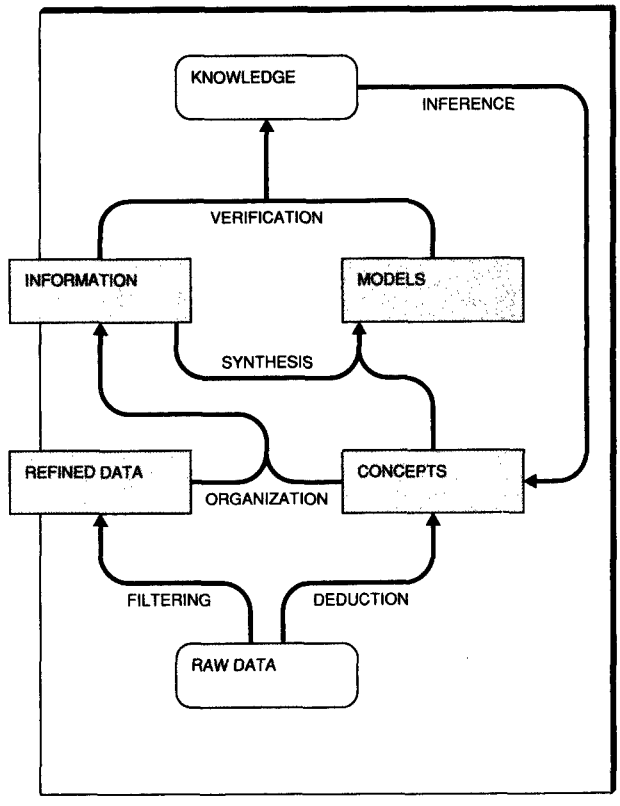
Storage technology comparisons. The discussion in the previous section about how data and concepts are refined into information and models helps us to understand how the different storage technologies support various classes of applications.

RDBMS storage technology. RDBMS technology is organized around relations, which are two-dimensional tables. The heading of each table is comprised of an unordered set of primitive attributes, each of which is a column in the table. Each row in the table is called a *tuple* and consists of an attribute-value pair for each attribute in the heading. A group of one or more attributes is designated as the primary key for the relation. Each tuple is required to have a unique value for the primary key. Relational algebra can be used to create new relations dynamically using a subset of attributes from one or more existing relations. When multiple relations are involved, they are joined by matching the data values of a column from tuples in each of the relations.

RDBMS technology is quite well suited for managing raw and refined data, as well as representing information derived from that data. In particular, RDBMS technology tends to be the most appropriate choice when:

- The structure of the data is relatively simple and can be represented effectively in two-dimensional, rectangular tables
- The structure of the data or the nature of the inter-relationships among entities is subject to frequent change
- Applications that access data perform mostly associative accesses (i.e., query by matching attribute values)

Figure 1 Transformation model for changing raw data into knowledge



- Database access patterns are spread evenly across several different access paths, or there is a high frequency of ad hoc access
- Applications have a tendency to access or update small amounts of data in a transaction
- Within an application, the same data records have a low-to-medium probability of being accessed in a series of successive transactions
- Peak transaction throughput rates are high, and high-availability features (i.e., protection against a single point of failure) are necessary

CDA storage technology. Compound Document Architecture (CDA) technology is most suitable for managing the storage and access of concepts. Concepts typically are expressed as textual outlines, prose, or unstructured diagrams. Also, applications that access these concepts typically provide capabilities to display, edit, print, and

transmit an entire document or specific sections of a document.

The Microsoft Object Linking and Embedding (OLE)** framework is an example of a Compound Document Architecture. OLE 2.0 is becoming a *de facto* standard on desktop personal computers that run Microsoft Windows** 3.1 or Windows NT. This framework provides a standard API that allows a document created by one OLE-compliant application to link to or directly embed a piece of a document created by another OLE-compliant application.

In a limited sense, OLE is an example of an object-oriented storage model. Documents and sections of documents are treated as objects. The application program that creates a document is responsible for providing display, edit, printing, and storage services for other application programs that want to link to or embed all or part of that document. In this way, the application program that manages a document encapsulates the document with a public interface, and hides detailed knowledge of the internals of the document from other applications.

In a wider sense, a compound document architecture (such as OLE 2.0) may not be a fully object-oriented storage model. In particular, the concepts that are represented in a document typically exist as part of the contents of the document as opposed to being represented as objects with attributes, behavior, and links to other related concepts.

ODBMS storage technology. ODBMS technology is well-suited for the storage and access of higher value-added components such as information, models, and knowledge. In essence, object-oriented modeling provides an excellent way to structure and classify data entities, and to represent the semantics of the interconnections between related entities.

In addition, an ODBMS (like ObjectStore) that uses a direct-reference storage paradigm allows the method code for persistent objects to be unified with the underlying storage representation. In other words, the ODBMS framework is tightly integrated with the programming language so that method code typically need not be aware of whether it is operating on a persistent or transient object.

ODBMS technology is likely to be a very suitable choice for applications that have:

- Classes defined at multiple levels of abstraction, and that make significant use of inheritance and polymorphism
- A need to represent relationships between classes that are more complex than simple associative sets (data structures such as lists, trees, hash tables, and networks are awkward and inefficient to represent in a two-dimensional table)
- Object structures that are relatively stable over time (i.e., they are changed in well-defined software release cycles)
- A higher frequency of—or greater importance placed on—repeatable, predictable access patterns than random, ad hoc access patterns
- Complex algorithms that perform optimizations, simulations, or analyses that access or update medium-to-large-size networks of related objects, and have very high-performance requirements for these types of access patterns
- A medium-to-high likelihood of referencing the same set of objects in a series of successive transactions
- Moderate transaction throughput requirements
- A need to manage multiple versions of the same set of objects

Impact on information technology strategies. The previous section identified the relative strengths of three of today's primary storage technologies. Table 3 summarizes some application classes where one of the storage technologies has established a dominant position. For these classes of applications, the refinement category of the data elements and the primary nature of value added by the application helps to explain why each storage technology holds its dominant position.

Information systems application development. Over the next five years, there are several application classes where the relative advantages of one storage technology over another will not be quite as clear. One interesting example is the area of decision support systems for businesses. The term *decision support* means different things to different people, so let me describe the meaning I am using in this paper.

Decision support applications analyze large quantities of business operations data, and help filter, organize, and present them in a way to support

Table 3 Dominant storage technologies for selected application classes

Application Class	Storage	Category	Processing Value Added
Mission-critical OLTP	RDBMS	Raw data	High availability and high transaction throughput rates
Back-office accounting and information systems	RDBMS	Refined data, *information*	Summarize, present, and compare operating results and flexible, ad hoc queries
Office automation	CDA	Concepts	Display, edit, transmit, and print multimedia documents
Engineering design	ODBMS	Information and models	High-performance display, edit, and simulation of complex, large object networks

tactical or strategic decision making. Quite often, the outputs of transaction-processing applications become the main source of raw business operations data for these decision support applications.

At its most basic level, a budgeting system can be classified as a decision support system. A system of this type analyzes investment, revenue, expense, and profit contribution on a product line or organizational unit basis, and compares actual results to plan. It helps managers make tactical and strategic decisions by identifying exception conditions (i.e., large variance of actual results against plan) and highlighting trends in operating results or key ratios. When viewed this way, RDBMS and CDA storage technologies complement each other nicely and are well-suited to managing the level of data refinement, and the nature of the value-added processing. RDBMS technology is useful for capturing the raw data for operating results at its source, and organizing it in a way to make it accessible to fourth-generation language (4GL) tools (report writers for comparisons against plan) and spreadsheet applications (for forecasting and plan production).

At a more advanced level, however, the nature of a decision support application changes significantly. Consider, as an example, the decision support requirements of a manager for a brand of a major soft drink manufacturer. This individual's primary responsibility is to devise a marketing strategy that will keep this brand as the market leader. This strategy must balance several different aspects of the marketing mix such as price, promotional strategies, packaging, product qualities, and channels of distribution.

A much more ambitious example of a decision support system would be an application that tries to model consumer buying behavior. This type of application would require raw sales data, promotion data, and distribution channel data as follows:

- Raw sales data—For each package-type of this brand and competitive brands sold in any distribution outlet, one would need data on the price and quantity sold. Ideally, this information would be available at the granularity of each sale, including the date and time of the sale and some demographic information about the buyer. At a minimum, aggregate totals are needed for each time period (less than or equal to a day).
- Promotion data—For each brand, one would need information about any special promotions, including advertising campaigns. For national advertising campaigns there would be a need to know which ads were run, over what time period, how frequently, and in what time spots. There would also be a need to know what the ratings were, broken down by demographic categories, for each show in which a spot was run. For regional ad campaigns, the same information as used in the national campaign is needed, plus the geographical boundaries of the region reached by the campaign.
- Distribution channel data—One would need to know which outlets carry this brand and each of the competitor's brands. Also needed would be the location of each outlet, how much shelf space is allocated to each brand, what price is charged for each brand, and what hours the store is open.

As one can see, there is an overwhelming number of variables that can have a significant effect on the sales volume of various brands of soft drinks. In addition, one also must consider the interaction effect between various combinations of these variables. For example, what would happen if the price were reduced by $.10 per 12-ounce container in all markets, and an ad campaign were conducted during each football game televised in each major market?

Assume that one could gather all of the data desired, and the long-term suitability of RDBMS and ODBMS technologies were evaluated as the repository for this marketing information. If the expectation existed that most of the filtering and analysis work were going to be performed by people, then the raw data should be stored in a form that is suitable for human use. On the other hand, if the expectation existed that most of the filtering and analysis work were going to be performed by running sophisticated statistical analysis and pattern recognition algorithms against the raw data, the data should be stored in a form that is suitable for programmatical analysis.

Storage technology products. The commercial storage management software market is a multibillion dollar market today. Relational database vendors are very well entrenched and collectively have a large share of this market. Inertia is a very powerful force; many companies have made strategic investments in relational database technology, and would face major investment in technology and training to switch to object-oriented database technology on a wholesale basis.

On the other hand, the rapid advances in computer hardware technology and the widespread availability of powerful, low-cost personal computers and engineering workstations change the rules of the game. Many businesses view computer technology as an opportunity for strategic competitive advantage. This is both a "carrot and a stick" (the reward and the punishment); the incentive to innovate comes from the opportunity to gain a competitive advantage over competitors, as well as from the threat that competitors will harness the technology sooner.

Today's entrenched relational database vendors are, for the most part large, profitable, growing, and highly competitive companies. There are a number of different strategies that are unfolding in response to the emergence of ODBMS technology. Some of the established RDBMS vendors are making strategic investments in companies that specialize in ODBMS technology. The recent investment by IBM in Object Design and the investment in Objectivity by Digital Equipment Corporation are two such examples.

Another strategy is the emergence of extended relational database technology. Simply stated, this strategy is an attempt to preserve the core features of the relational storage model, and augment it to support requirements of object-oriented modeling such as inheritance, complex data types as attributes, and multivalued associations. It is not yet clear how the extended relational and ODBMS technologies compete with each other from the point of view of performance, flexibility, and ease of application development.

Physical and logical data model independence. Many RDBMS supporters today claim that a direct-reference storage model cannot possibly support independence of logical and physical data models. Conventional wisdom says that the absence of this critical feature means that ODBMS technology is a regression rather than an advancement, a step backward to the days of hierarchical and network database technology. As a result, this technology cannot really be given serious consideration to be used as the basis for an enterprise-wide data management strategy. Proponents of this view claim that object-oriented database management systems are at best a niche technology, limited to the small subset of applications that need high-performance persistent storage, and are able to sacrifice physical and logical data model independence.

This argument fails to take into account the synergy that results from integrating the object-oriented development paradigm with the direct-reference storage model. As was mentioned earlier, one of the main principles of the object-oriented development paradigm is the unification of attributes and relationships (data) with responsibilities and functions (behavior). Another main principle of object orientation is encapsulation, the notion of separating the internal implementation of a class from its external interface.

Applying these two principles along with a responsibility-driven design method, one develops a system as a collection of highly cohesive,

loosely coupled classes. Each class offers a set of contracts to its client classes, where each contract is expressed in terms of a group of operations (methods) that are part of the public interface of that class. Because the public interface to a class is limited to functions (organized into contracts), data structure is hidden inside the class as an implementation detail. In other words, in a well-designed object-oriented system, whenever one class depends on another, this dependency is based on services offered through a well-defined contract.

Given this, it is not difficult to construct a system architecture where application programs, which allocate and use persistently stored instances of any number of classes, are completely independent from the physical layout of the classes. Merely obeying the rules listed below is sufficient (Figure 2 illustrates this point):

- Persistent classes are implemented in a sharable library.
- All contracts are guaranteed to be upwardly compatible.
- The application program avoids using the class as a base class, or as the data type for an embedded member (although an embedded pointer to a separately allocated instance of that class is acceptable).
- All accesses to object instances within the application programs are indirect through pointers or references (as a corollary the application program contains no global [data segment-based] or local [stack-based] instances).
- In-line method expansion is not used.

Notice that if these rules are followed, then there is no code in the application program that needs knowledge of the physical storage layout of the class. As a result, application programs need not be rebuilt when a change is made to the object layout. It is sufficient to install a new (upwardly compatible) version of the shared library, update the database schema, and evolve the database to the new schema. All dependencies on physical storage layout are localized in the shared library that implements the class and the database.

There are two additional items about this architecture that are worth mentioning:

- Most current implementations of sharable libraries require that the binding between the executable part of the application and the library be established at the time the application starts up. Frameworks such as the Common Object Request Broker from Object Management Group, a standards organization, allow this binding to be changed during run time.
- Because of the wide variety and type of objects that can be stored in an object-oriented database, it would be feasible for CORBA-compliant frameworks to store the library code that implements a set of related classes directly in an object database. (CORBA is an object-based client/server interface standard that allows object services to be registered with an agent, e.g., the Object Request Broker.) This would also make it possible for the database to maintain several different versions of the same library to choose from.

Impact of ODBMS on the software development process

In previous sections I have suggested that object-oriented database management systems provide two primary sources of leverage, and that these sources have a great deal of synergy with each other. I also mentioned that leverage, by itself, provides no benefit; it all depends on how the leverage is applied.

This section introduces the types of development process changes that are necessary to get the most beneficial leverage out of an object-oriented database management system. Clearly, a complete discussion of this topic would well exceed the scope of this paper. My hope is that a brief discussion will provoke enough thought to still be valuable.

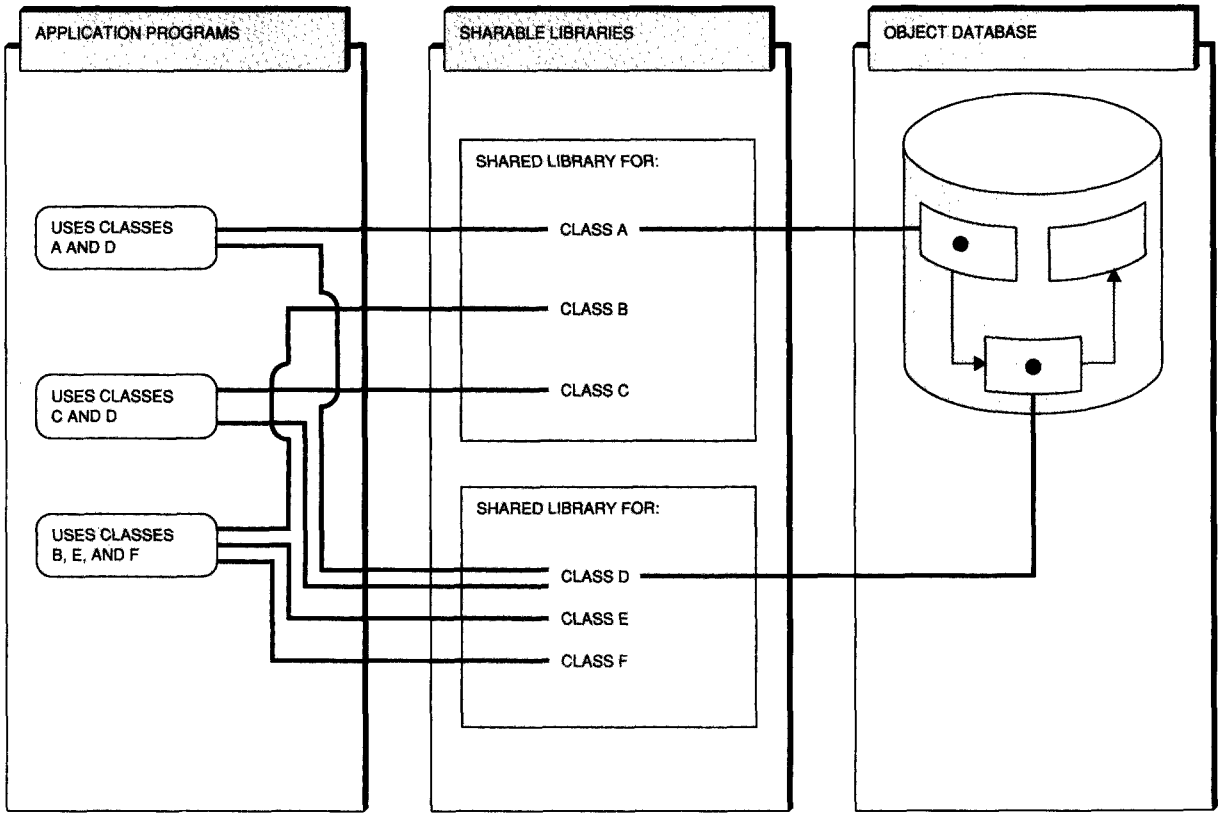
Risk of paradigm mismatch. In my experience, there are two aspects of the software development process that are affected by the paradigm shifts that underlie an ODBMS like ObjectStore:

- Technical processes used in the analysis and design phases of the project
- Management processes used in the planning and organizational phases of the project

Each of these areas is discussed in the following sections.

Analysis and design phases. When using an object-oriented database that is based on the direct-

Figure 2 Object-oriented architecture providing application independence from physical object layout



reference storage model, physical database design is more tightly coupled to overall system design. This is not a requirement of database systems that use the read-write model. The reason is that in the read-write model, the application accesses the database occasionally, and for limited periods of time. In the direct-reference storage model, the application accesses the database every time it uses an object that was allocated in persistent storage.

As a result, extensions to popular object-oriented analysis (OOA) and object-oriented design (OOD) methods are necessary. As Jacobson[7] points out: "Object DBMSs have been developed to store objects as such in the database. Different application areas have different requirements, and many vendors optimize their products for a specific application area."

These extensions are not unlike the set of extensions proposed in the mid-1980s by Ward and Mellor[8] and Hatley and Pirbhai[9] to make structured analysis methods suitable for use on development problems in the real-time domain.

Specifically, in the case where persistent objects are shared by processes distributed across a network, and the storage space occupied by these objects is large, design of an object-oriented system that uses the direct-reference storage model must:

- Specify how to map usage case scenarios into database transactions
- Derive access patterns from object interaction diagrams and object and association representations
- Subdivide the aggregate storage requirements

into meaningful partitions, either by ownership or by class and association, and estimate the space requirements of each partition
- Rank order the access patterns based on one or more criteria that identify their relative importance
- Define an object clustering strategy that is optimized for the highest priority access patterns that are reading or updating the largest amounts of data in their transaction
- Define a new ranking of access patterns, in systems with a large number of concurrent users, based on their level of intrusiveness, where an intrusive access pattern is one that acquires locks on highly shared data and holds those locks for a long duration
- Refine the design to minimize the impact of the most intrusive access patterns on the most important ones (this may involve changing class definitions or association representations, reclustering objects, refining algorithms, or restructuring transaction boundaries)

Figure 3 contains an illustration of an object-oriented analysis and design process that would be suitable for use with an ODBMS. It should be noted that these analysis and design extensions are suitable for any class of problem that allows direct concurrent access to shared objects, and needs to be scalable to large storage requirements (hundreds of megabytes or more) and large numbers of concurrent users, where two or more are performing updates.

Project organization. As I pointed out in the prior section, when using an object-oriented database that is based on the direct-reference storage model, physical database design is more tightly coupled to overall system design. This means that in order to use a framework like ObjectStore effectively, the overall organization of the project and the roles of the developers must match the needs of the work.

First of all, system analysts, system architects, and database designers must collaborate closely throughout the development life cycle. Database designers must be more aware of requirements analysis and system architecture issues. System analysts and system architects must be more aware of ODBMS issues and constraints and how they impact the high-level design of the system. This is very much like the shift that occurred when event-driven general user interface (GUI) frameworks became widely accepted.

Second, for large systems (storage size and number of users), performance analysis work takes on a much more significant role. This work must be started as early in the life cycle as possible. Also, early prototyping becomes a very important strategy for designing scalable systems. If performance tuning is left until the end of the development cycle (as is common in many projects) disastrous consequences can, and often do, result.

Last, but certainly not least, if application programs are to be independent of the internal implementation of the classes they use, then much care must be taken in the object-level design process. As Booch points out,[5] the key decisions about overall system architecture (the macro process) must be made by a small group of experienced people including problem domain specialists and object-oriented design experts. These architectural decisions include:

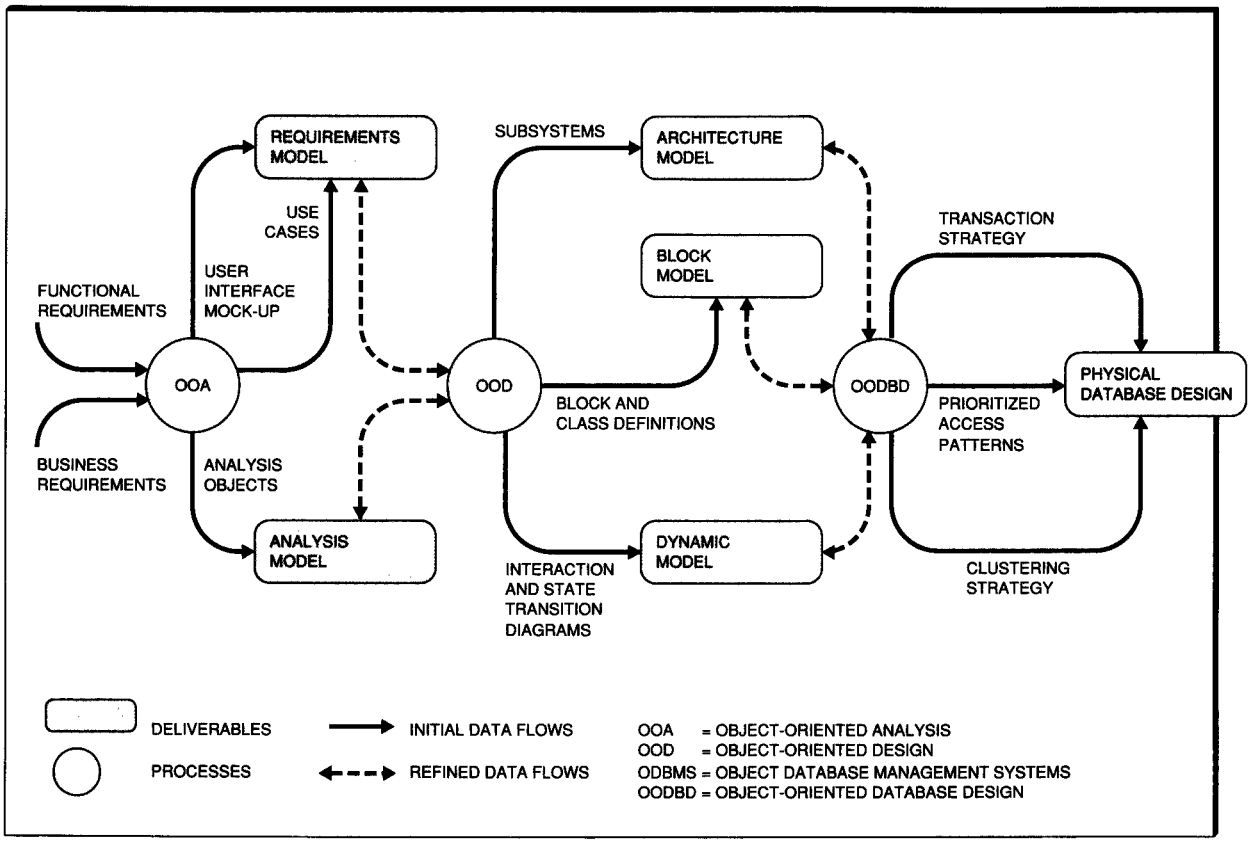
- How the overall system is partitioned into subsystems, and how pieces of each subsystem are visible to other subsystems (i.e., as part of the external interface of the subsystem)
- How the abstractions within a subsystem are organized into layers with well-defined protocols (i.e., contracts) in order to reduce interlayer coupling
- How subsystem layers are packaged in order to increase the modularity of the completed system and increase the potential for component reuse

Summary

The main premise of this paper is that object database management systems provide one source of leverage by fully supporting the object-oriented software development model. Some object database management systems, such as ObjectStore, provide additional leverage by using a direct-reference storage model, rather than a read-write model.

These two sources of leverage are at the heart of the differences between object database and relational database technologies. How important these differences are to a given application area depends largely on how complex is the problem,

Figure 3 OOA/OOD process model suitable for use with ODBMS framework



what is the primary source of value added by applications, how complex, structured, and stable is the underlying data model, how crucial is system performance, and how much of the analysis and filtering work needed to transform raw data into useful information is done by computer algorithms instead of human interaction.

Because of the wide range of answers to these questions, it is highly likely that RDBMS and ODBMS technology will coexist over the foreseeable future, and bridges between these database technologies will become increasingly valuable.

Finally, I point out that proper use of the object-oriented development model allows a system to be constructed that takes advantage of all of the leverage offered by ODBMS technology, without sacrificing either flexibility or configurability. However, in order to accomplish this, project managers and technical project leaders must understand the impact of the underlying paradigm shifts and organize their project teams and development processes to exploit their potential fully.

Acknowledgments

The author would like to thank Greg Baryza, Lorraine Alfred-Cortellessa, Pat O'Brien, Jack Orenstein, Ian Schmidt, Irv Traiger, and Bob Walmsley for their comments on an early draft of this paper.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Object Design, Inc., Objectivity, Inc., Versant Object Technology, Servio Corp., Intel Corp., Rogue Wave Software Inc., National Institutes of Health, Grady Booch, or Microsoft Corp.

Cited references

1. T. Atwood, J. Duhl, G. Ferran, M. Loomis, and D. Wade, *The Object Database Standard: ODMG-93*, R. Cattell, Editor, Morgan Kaufmann Publishers, Inc., San Mateo, CA (1994).
2. C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, "The ObjectStore Database System," *Communications of the ACM* **34**, No. 10, 50–63 (October 1991).
3. K. Nash, "Object Design Boosts Its Database Product," *Computerworld*, pp. 69–72 (December 13, 1993).
4. R. Wirfs-Brock, B. Wilkerson, and L. Weiner, *Designing Object-Oriented Software*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1989).
5. G. Booch, *Object-Oriented Analysis and Design with Applications*, Second Edition, The Benjamin/Cummings Publishing Company, Redwood City, CA (1994).
6. D. Taylor, *Object-Oriented Technology: A Manager's Guide*, Servio Corporation, Alameda, CA; Addison-Wesley Publishing Co., Reading, MA (1990).
7. I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley Publishing Co., Workingham, England (1992).
8. P. Ward and S. Mellor, *Structured Development for Real-Time Systems: Introduction and Tools*, Yourdon Press, Englewood Cliffs, NJ (1985).
9. D. Hatley and I. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House, New York, NY (1987).

Accepted for publication February 15, 1994.

Charles Alfred *Object Design, Inc., 25 Mall Road, Burlington, Massachusetts 01803 (electronic mail: calfred@odi.com).* Mr. Alfred is the Corporate Education Manager at Object Design, Inc., the developer of ObjectStore. His research and professional interests include object-oriented software engineering, software development processes, project organization and management, and team building.

Reprint Order No. G321-5543.