Repositories and Object Oriented Databases[1]

Philip A. Bernstein

Microsoft Corp.
One Microsoft Way
Redmond, WA 98052-6399
philbe@microsoft.com

Abstract

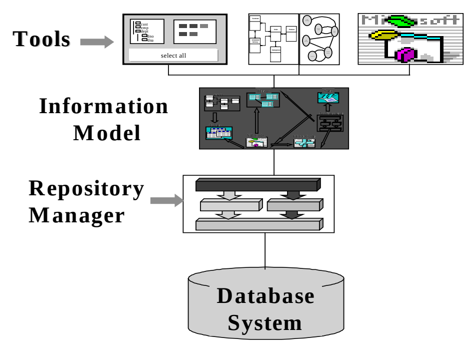
A repository is a shared database of information about engineered artifacts. An object-oriented repository has many of the same features as an object-oriented database: properties, relationships, and versioning. However, the two technologies are different for two reasons. First, a repository system has built-in information models, which are database schemas or object models that cover both generic and tool-specific kinds of information. Second, the features of a repository are often more functional than similar features supported by object-oriented databases. This paper is primarily a survey of the latter features, drawing attention to capabilities that distinguish repositories from object-oriented databases.

1 Introduction

A repository is a shared database of information about engineered artifacts, such as software, documents, maps, information systems, and discrete manufactured components and systems (e.g., electronic circuits, airplanes, automobiles, industrial plants). Designing such engineered artifacts requires using software tools. The goal of a repository is to store models and contents of these artifacts to support these tools. For example, a repository that supports software development and deployment tools could store database descriptions, form definitions, controls, documents, interface definitions, source code, help text, executables, icons, and the contents of makefiles.

Today's software system architectures are object-oriented, to promote better software productivity and reuse. To exploit these architectures and gain their benefits, many existing tools and virtually all new tools are written in an object-oriented way. Since repositories support these tools, new repository systems are nearly all object-oriented too. That is, these repository systems allow tools to store, access, and manipulate objects, rather than records, rows, or entities. In this sense, a repository is an object-oriented database. Indeed, many object-oriented database systems have been marketed primarily as support for software tools.

There *are* differences between an object-oriented database system (OODBMS) and a repository system. One important difference is the repository system's *Information Model*, which specifies a model of the structure and semantics of the artifacts that are stored in the repository. In database terms, the information model is a schema for the repository. In object-oriented terms, an information model is an object model for the objects in the repository. Tools use the information model to interpret the repository's contents.



The information model is implemented on top of a *repository engine*, which offers services to access and manage the repository itself and the information it stores. The repository engine could be an unadorned database system product, such as an OODBMS. But often it's a layer of software on top of the database system, such as a layer that adds object (or entity) and relationship functionality on a relational database system.

[1] This is a slightly updated version of a paper originally published as Bernstein, P.A., "Repositories and Object-Oriented Databases," *Datenbanksysteme in Buro, Technik und Wissenschaft* (*Proceedings of BTW Conference, March 1997, Ulm, Germany)*, Klaus R. Dittrich and Andreas Geppert (editors), Springer-Verlag, Berlin, March 1997, pp. 34-46. It is reprinted with permission from Springer Verlag.

Database system developers normally consider the information model to be part of the application. However, tool developers usually consider it as part of the storage system, because it is generic — across all tools or within a given field.

There are generic object management functions that are common to all tools but are not typically part of a repository engine. For example, a team development model that covers workspaces, checkout/checkin, and versioned configurations is usually beyond the scope of the repository engine, even if that engine supports basic versioning functionality [10]. Another example is an object reuse model that covers functions to group objects into catalogs, assign keywords to objects, and connect objects in libraries to the projects that use them [8]. A third example is a workflow model to track the development of an artifact through an engineering process [9]. These generic object management functions can be captured in information models that are shared by all tool developers.

Within a given area of engineering there are standard types of objects and standard ways to manipulate them. For example, in the software engineering area, there's general agreement on what constitutes a database definition and an interface definition, and there is growing agreement on what constitutes an object model in an object-oriented analysis and design tool. There's no technical reason why each tool developer needs to develop such an information model from scratch. Moreover, if a tool developer produces a customized model instead of sharing a model with others, then the model won't allow his or her tools to share data with other tools produced by other vendors, since they will use different information models. In many fields, such standard information models are being defined, such as the SQL and CDIF standards for database objects and the current OMG's Unified Modeling Language (UML) standard for analysis and design [13]. A repository system can save tool developer effort and promote sharing between tools by including an information model that covers areas where there is some industry-wide agreement.

As in all areas of system software, there is some uncertainty about which functions should be in which layers, so these functions tend to move around over time [1]. For example, when functions in the information model become well accepted with generally agreed-upon semantics and are identified as heavy resource consumers, it's worthwhile to drive them down into lower layers, that is, into the repository engine. Indeed, the desire to move information model and tool function into lower layers helped prompt the development of OODBMSs in the first place.

Although both repository engines and OODBMSs have evolved from the trend to drive tool functions into the underlying storage system, they have not arrived in exactly the same place: Repository engines usually support higher level semantics than OODBMSs. For example, they often use relationships that carry more semantics, type systems that are more extensible, and richer version and configuration models. Justifying this claim would require a careful comparison of features in OODBMSs and repository engines, which is more than we will attempt to do here. Instead, we take just one step down that road by explaining many of the features supported by repository engines. Readers familiar with OODBMS products will recognize that many of these features are higher level than those supported by any OODBMSs. However, we will stop short of making specific claims of difference between OODBMSs and repository engines, since that would suggest a more thorough feature analysis of products than we have done.

2 Repository Engine Functions

The main functions of a repository engine are:

- Object management - Store each object's state (i.e. its *properties* or *attributes*) and provide access to its methods. Every repository object has a type that describes it.

- Dynamic extensibility - The repository's information model consists of type definitions and classes that implement them. It must be easy and efficient to add new type definitions and classes, and modify existing ones.

- Relationship management - Models of engineered artifacts are rich with information about relationships between objects. Tools make heavy use of these relationships. The repository should support powerful relationship semantics, to relieve tools of commonly needed relationship functions.

- Notification - An operation on an object may need to trigger operations on other objects and/or send an event to a tool that registered interest in changes to that object.

- Version management - Engineering design information is usually developed according to some engineering process that relies on versions to manage the evolution of designs. Tools need help in managing versioned design objects.

- Configuration management - A tool user normally works within the narrow context of the objects that are relevant to the problem at hand. Similarly, a component or product model usually contains only some of the objects in the repository. This requires grouping versioned objects into configurations, which are themselves versioned.

We describe each of these functions in turn. Our goal is two-fold: to explain the main facilities offered by repository engines and to show that many of these facilities are beyond what's supported by most OODBMSs.

2.1 Object Management

Objects are in-memory representations of the information held in the repository database. An object offers methods to manipulate the object's state. In particular, it offers operations to read and write the values of its public state variables, called *properties*. Properties can be stored or computed. The latter are much like a relational view.

Properties can be single-valued or multi-valued. A multi-valued property is a *collection*, which is an object that supports methods to insert, delete, scan, and randomly access values in the collection. A collection might also be sequenced, duplicate-free, or scrollable (i.e. can scan in both directions).

It is valuable if objects can be accessed by database queries, in addition to being navigable by object-at-a-time operations. To leverage the repository engine's underlying database system, properties must be stored in a queriable format. For example, a SQL table can be used to store objects, with columns representing properties. For properties that are only occasionally used, a table with columns ObjectID, PropertyName, PropertyValue saves space over the previous design. However, the latter table is not queriable via SQL if PropertyValue can have more than one type (such as integer or string).

For an object to be *in* the repository, the object management layer must be able to load it into memory (including its persistent state), delete it, and store it persistently. The object management layer must also offer methods to create a new object in the repository.

Usually, the programming model for creating, loading, deleting and storing objects and for invoking their methods is defined by the programming language (such as C++ or Java) or by the underlying computing architecture (such as COM, DSOM, or OMG CORBA). Ideally, the repository offers the same, or a straightforward extension of, the programming model offered by the programming language and/or computing architecture used by tools. This is an especially challenging design goal if the repository must support more than one such model.

For some design objects, part or most of their state is stored in a different storage manager than the repository, such as a file or tool-private repository. To use the repository for some of the object's state, a *surrogate object* is stored in the repository. In addition to storing the object's repository state, the surrogate has a property that points to the object's external state. For example, the property might contain a pathname that identifies the file that contains the object's external state.

The above object management facilities are the same in both an object-oriented database system and repository.

2.2 Dynamic Extensibility

Every object in the repository has a type that describes its interface(s), such as its methods and properties. It's what a client can count on about an object before it actually sees the object itself. Type information is also used to drive the repository's implementation of objects. For example, if you define a type "user" that has string-valued properties "name" and "password," the repository can create and manipulate instances of "user" using the type definition as a template. Given the importance of type information, it must be easy to access and manipulate. This is in contrast to many OODBs, whose type information is largely embedded in applications and not easy to find in the OODB itself.

A class is an implementation of a type, that is, a body of code that implements the type's interface. A type may have more than one class that implements it. This capability of multiple classes per type is quite important, because it enables different vendors to implement the same type independently, yet have their tools be able to share each other's objects, since tools only depend on an object's type, not its class.

Repository type definitions must be easy to extend, so that a tool vendor can add state information and behavior to existing type definitions. Such an extension should not break existing tools that are oblivious to the extension, yet tools that are aware of the extension should be able to exploit it. Moreover, tools that exploit the extension should still be able to manipulate old objects that conform to the non-extended definition. This allows extensions to be added piecemeal, rather than having to install a new revision of the information model and all tools that depend on the extended types at the same time. These requirements are not commonly met by the type systems of OODBMSs.

One way to meet these requirements is offered by Microsoft's Component Object Model (COM) [14]. In COM, an object can have multiple *interfaces*, each of which defines a set of methods and their signatures and has a globally unique 16-byte *interface identifier* (IID). An interface specification is immutable, so to extend an interface, you implement a new interface with another IID. By convention, interface names begin with "I".

Every COM object supports the interface `IUnknown`. The method `QueryInterface` on `IUnknown` asks an object if it supports a particular interface, given the interface's IID. The object either returns a pointer to the interface, or null if it doesn't support the interface. Suppose an interface I1 was extended, creating the interface I2. If a tool knows about the extension and is given an object, it can exploit the new behavior by calling `QueryInterface` for I2. If `QueryInterface` returns null, the tool can call `QueryInterface` for I1. Old tools that don't know about I2 just call `QueryInterface` for I1. Similarly, a new object is implemented to support both I1 and I2, so that both old and new tools can use it.

Another benefit of this interface approach is that different vendors can group the same interfaces into different classes (essentially, multiple inheritance of interfaces). For example, one vendor can support the interface ICatalogItem on a class Library, while another vendor supports it on a class DevelopmentProject. If a tool is programmed against interfaces (e.g. ICatalogItem) rather than classes, then it can manipulate objects produced by those different vendors without knowing anything about their classes. If a vendor adds yet another class that supports ICatalogItem, the tool can, without modification, manipulate objects of that class too.

This interface approach to information model definition is used in Microsoft's repository system, which initially shipped in Microsoft Visual Basic 5.0 [4].

2.3 Relationship Management

A relationship is a connection between two objects. It could simply be an object-valued property, as in the ODMG model [6], possibly with a cardinality constraint, such as one-to-one or one-to-many. Or the relationship could be an object that can have its own behavior, such as properties or methods. This provides a richer mechanism for modeling relationships than object-valued properties, which is often important in repositories. For example, a system management tool may need to record properties of the relationship between a communications controller and the computer it's connected to, such as the date it was inserted and the speed at which it operates. If relationships could not have properties, then this would require representing the relationship as another full-blown repository object, which is heavier weight than a relationship object (since it needs to support relationships, be versionable, etc.) and more verbose to manipulate.

Like properties, relationships should have type definitions, should be queriable, can be single- or multi-valued, and can be stored or computed.

Most repository systems restrict relationships to be binary and functional (one-to-many) and exploit these assumptions to simplify and speed up the implementation. Some models allow n-ary relationships, such as the OMG relationship service.

Usually, relationships are bi-directional, meaning that they can be traversed from either object they connect. However, this implies that both objects must know about the relationship (otherwise, how could they provide access to it?). Thus, to add a new relationship type to an object without changing the object's type definition, the relationship cannot be bi-directional. For example, one kind of relationship supported by the OMG relationship service is between objects that are oblivious to the relationship, so the service does not support access to the relationship via the objects it connects.

Often, one wants to define relationships on non-instantiable types, and have them be inherited by classes that inherit that type. For example, one could define a Contains relationship that connects the abstract classes Container and ContainedItem, which are inherited by concrete classes, such as JavaPackage and JavaClass. Using the interface approach described earlier, this can be done by defining relationships on interfaces, such as the interfaces IContainer and IContainedItem; any class that supports one of these interfaces "inherits" the relationship.

An important capability of a relationship is to propagate operations that are applied to an object at one end of the relationship. For example, if a relationship models physical containment (e.g. a table definition contains a column definition), then deleting a container object should cause the deletion of its contained objects (i.e. those connected by a containment relationship). A general approach to this problem was proposed in [15], where one specifies for each operation and relationship type whether the operation applies to relationships of that type (e.g. copying the object implies copying the incident relationships) and/or applies to objects at the other end of the relationship (e.g. copying the object implies creating a relationship to a new copy of the object at the other end). However, repository engines usually only implement propagation for a fixed set of relationship types and operations. For example, the PCTE model allows propagation using the relationship attributes composition, existence, reference, implicit, and designation [16]. Most OODBMSs are much less functional than repository engines in this area.

An interesting special case of operation propagation is *pendant delete*, where deleting the last relationship to an object causes the object to be deleted. This can be complicated by the need to avoid deleting an object that has other references to it. For example, deleting the last project that references a module causes the module to be deleted, unless the module is called by another module. There is also the question of the conditions under which you can create an object that is the destination of a relationship that propagates pendant delete, since it has no instances of that relationship pointing to it at the time it is created.

It's convenient to be able to name objects in a repository using conventional path names, such as A/B/C, even though the relationships in a repository are not limited to being a hierarchy. Suppose each repository database has a unique root object, each object has a name, and each object type has a unique relationship type that is used for name resolution. Then the path "A/B" can be interpreted as "find an object named A connected to the root object by the root's naming relationship type, then find the object named B connected to A by A's naming relationship type."

There are many variations of this naming scheme. For example, you could explicitly identify the relationship used to dereference each name in the path, such as "A/R:B", which says to find the object named B connected to A by relationship type R. Another variation is to attach names to relationships, rather than objects, so that an object can have different names in different relationship contexts. For example, to dereference "A/B", from the root follow the naming relationship labeled by A, and then from "A" follow the naming relationship labeled by B. The same object, A/B, may have other names, such as C/D. For example, this context-dependent naming could be used to give a database different names in different applications that use it.

2.4 Notification

Since tools share data, they need to be notified when certain operations (typically updates) are applied to objects of interest. These notifications are essentially triggers in a SQL database or event-condition-action rules in an active database [7]. Typically, a notification rule is attached to a type definition. If a method executes (i.e. an event occurs) on an object that conforms to the type definition, and that method triggers the notification rule, then the rule is invoked. The rule checks whether its conditions hold and, if so, invokes the action. The rule indicates whether the invoked action executes in the context of the transaction that generated the event (which is useful in workflow systems to log a state transition) or in a separate transaction (usually the best approach for events that propagate to tools that are active).

Notifications usually propagate to tools in the form of events, just like events on display devices, such as mouse button operations. (This use of the term "event" has a different meaning than in event-condition-action rules.) Generally, a tool needs to hear such events only if it is active. Thus, when it becomes active, it registers itself with an event service, which mediates between the notification rules and the tools. For example, ActiveX Connection Points and the OMG Event Service work this way. If a rule wants to notify certain tools that an operation has taken place, then its action is to send an event to the event service, which forwards it to all the tools that are registered to receive that event.

2.5 Version Management

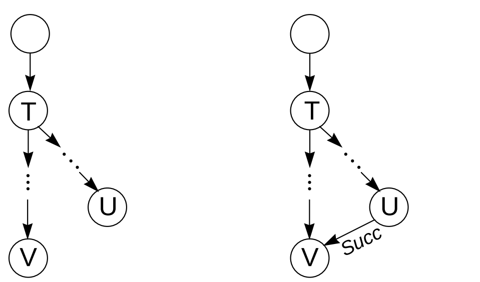
Versions track the lineage of changes to an object over time. Versioning is usually implemented by connecting successive versions of an object by a *Succeeds* relationship. If version B succeeds version A, then B was derived from A. In most version models, A must be *frozen* (i.e. immutable or stabilized) at the time that B is created. Otherwise, the Succeeds relationship would not reflect "is derived from."

We believe that a basis set of methods on a version are freeze, create-version, delete-version, and merge. *Freeze* simply changes a version's state from unfrozen to frozen. Some properties and relationships on an object can be marked as *mutable* (in the object's type definition), meaning they are not frozen when a version of the object is frozen. For example, it may be useful to update a property that tells the state of a version relative to testing (e.g. untested, unit-tested, system-tested), even if the version is frozen. In a system with binary relationships, relationships may have a direction, from origin to destination, where an update to a relationship is regarded as a change to the origin (which must be unfrozen) but not to the destination (which may be frozen). For example, the relationship Program References DatabaseDefinition may be updated even if the DatabaseDefinition is frozen, but not if Program is frozen; that is, the relationship is regarded as part of a Program but not of a DatabaseDefinition.

Create-version on version V creates a new successor version V′ to V with the same state as V. One needs to decide which of V's relationships are copied to V′. A common rule is to copy each relationship for which V is the origin. Create-version may also propagate to destination objects, for example, along relationships that model physical containment. For example, since column definitions are private to table definitions in SQL, creating a new version of a table definition should create a new version of all of its column definitions. Similarly, create-version might propagate to an external object that is supported by a surrogate. However, since databases are shared by applications, creating a new version of an application does not create a new version of the databases it references; it only copies to the new version the previous application version's relationships to databases.

Delete-version deletes the version and re-routes successor relationships around the deleted version.

If V already has a successor when create-version is executed, then V′ begins another *branch*, that is, a new line of descent from V. *Merge* is used to bring two branches back together. Given two versions U and V, neither of which is an ancestor of the other via the Succeeds relationship, then applying Merge to U and V makes V a successor of U. That is, Merge transforms state (a) to state (b) below:



(a) before Merge(U,V)   (b) after Merge(U,V)

Usually, Merge updates the state of V to reflect changes known to U but not to V or V's other predecessor(s). For example, if U and V are text files, Merge may apply to V all changes that were made on U's branch after U and V's latest common ancestor.

The states of versions along a branch are often saved as deltas from a base value — either forward deltas from an initial value, or reverse deltas from a final value. The latter is usually preferable since it enables fast access to the latest version. The mechanism for computing and interpreting deltas, and the related algorithm for merging, is often type- or class-specific (i.e. it's different for text, binary, and structures). So version operations need to be parameterized by these type-specific mechanisms on the type of object being versioned. For example, a set of interfaces to such type-specific mechanisms is documented by `IReconcilableObject` and related ActiveX interfaces [11].

Some relationships are version-independent. For example, it is common to have a relationship to the latest version along a branch, such as a relationship from a C++ makefile to a file it references by a #include, meaning that the latest version of the included file should be used.

It may also be useful to store properties that apply to a *version set*, that is, to the set of all versions of an object. Each version in the version set implicitly has the same value of these properties.

A variety of naming schemes is used for versioning, no one of which seems especially compelling. Version names could simply be assigned sequentially. They may be constructed by concatenating the branch name with a sequential version number within the branch, such as 1.1 and 1.2 for two branches that succeed 1, and 1.1.1.1 and 1.1.1.2 for two branches that succeed 1.1, as in Intersolv's PVCS. The branch name could be a suffix of the version name, such as V3.Unix and V3.Windows, where Unix and Windows are branches of the root version. Major/minor version numbers could be used, even within a single branch, such as 1.1, 1.2, 2.0, and 3.0.

Version semantics could be implemented in the information model, outside the repository engine. But often, versioning is supported by the engine to optimize certain behavior, such as storage efficiency of relationships or speed of computing version-independent relationships. Most repository engines implement their own versioning semantics even if they're implemented on an OODBMS that supports versioning. It would be interesting to understand better why this is true. Do the OODBMSs specify too much policy? Are they insufficiently general? Too slow for certain operations? Etc.

2.6 Configuration Management

A *configuration* represents a set of objects, such as a project's components, a catalog of reusable objects, or a workspace. A configuration contains *configuration items*, which can be versioned or non-versioned objects. A configuration itself can be a configuration item, that is, configurations can be nested. And configurations can be versioned, as is common for a development project.

The main operations on a configuration are to *attach* and *detach* configuration items. Like other elementary repository operations, the attach method may propagate across relationships. For example, attaching a table definition to a configuration implicitly attaches the column definitions and index definitions that it contains. For example, this is called the "expansion" of the object (i.e. table definition) in Sterling Software's (formerly Texas Instruments) Composer tool for application development.

There is usually a get-contents method to get the contents of a configuration, which takes a transitive closure. Again, some relationships propagate get-contents, such as project attaches database definition, database definition contains table definition, table definition contains column definition, and column definition has index. This transitive containment implies that detach doesn't always remove an object from a configuration. For example, if a project attaches a database definition and attaches a table definition, and the database definition contains the table definition, then detaching the table definition from the project does not remove the table definition from the contents of the project.

Some tools require that they work in a version-free manner. That is, they want to use a configuration whose contents include only one version of each object and they want to access those objects without referring to version names. This can be done by assigning a precedence to each relationship of a given type emanating from an object. Get-contents can therefore select the version of each object whose path has higher precedence than any other path to the object.

For the most part, basic object and version operations applied to configurations can use their standard semantics, though there are some special cases to consider.

- Applying the create-version and copy operations to a configuration copies attached relationships but not the objects on the other end of those relationships.

- Since the configuration is the origin of its attaches relationships, you cannot attach an object to a frozen configuration.

- Freezing a configuration should propagate to its attached objects. An alternative semantics is that all of a configuration's attached objects must be frozen before the configuration can be successfully frozen.

- Deleting a configuration does not propagate to its attached objects (since they may be shared by other configurations).

- Merging a configuration C1 into another configuration C2 implies merging the configuration items that appear in both. Care is needed with a configuration item that appears in only one of the configurations, since it may have been explicitly detached from the other and should not be reintroduced.

Some models use a relationship type to capture configuration semantics, rather than using configuration and configuration item as object types. For example, in PCTE, *composite* links ("link" = "relationship") serve this function. Composite links have pendant delete semantics, and you can't add a composite link to a stable (i.e. frozen) origin object. Embedding configuration semantics in a relationship type rather than an object type has some advantages: it can avoid the need for a special attaches relationship type and can simplify the specification of the get-contents method. Instead, each relationship type is simply tagged as composite, meaning that it has containment semantics and propagates get-contents.

In some systems, configurations are implicitly versioned. For example, in Microsoft's Visual SourceSafe, a new version of a configuration (called a "project") is implicitly created whenever a file (i.e. object) is added to or deleted from the configuration, and whenever a new version is created of a file in the configuration. Ordinarily, users reference the latest state of a configuration. However, since every operation has a timestamp, an older version of a configuration can be selected based on timestamp.

Configurations are often used to scope the objects a tool is working on. For example, a configuration could be used as a workspace, in the same way that a file directory is used for file-based tools. Therefore, it is often desirable that, when following a relationship, only relationships to other objects *in the configuration* are returned. This requires some performance optimization, since a popular object may have a huge number of relationships to objects outside the configuration. Other operations might also be scoped by the configuration. For example, create-version could replace the previous version of the object in the configuration. Create-object could attach the new object to the configuration. A query could be applied only to the content of the configuration. Etc. Some of these operations are challenging to implement efficiently.

A configuration that is used as a workspace usually gets its content from larger shared configurations, such as ones containing all the objects in an application or in a database definition. When the workspace is initialized and every time it is opened, you want to get the latest versions of the relevant objects. A configuration definition can be useful for this purpose. The definition is essentially a query. When executed, it identifies objects to be added to the configuration.

2.7 Other Database System Amenities

A repository engine is a persistent storage manager, so it should have the usual amenities that come with any database system. For example, it should support the following features:

- a query language, preferably a dialect of SQL that is a superset of the standard.
- transactions — if not nested transactions, then at least it should count nested invocations of the begin-transaction operation and treat them as null operations, so that a method that brackets a transaction does not actually start a new transaction if it is called from within a transaction.
- a robust fine-grained security model, such as associating privileges with users and access control lists with objects or configurations.
- distribution, such as relationships that connect objects in different repository databases.
- replication, with similar functionality to the primary site and multi-master methods offered by today's relational database systems [5].

Most repository engines and OODBMSs have incomplete support in this area when compared to relational database systems. This has driven many vendors to layer their repository engines on relational DBMSs, to immediately gain many of these features.

3 Information Models

As we discussed in the introduction, an important distinguishing feature of repository systems compared to OODBMSs is the information models they offer. This has always been a controversial issue for tool vendors. On the one hand, tool vendors do not want to needlessly repeat the design of information models that are already well understood. And tool vendors' customers want tools purchased from different vendors to interoperate. On the other hand, tool functionality is intimately linked to the content of the underlying information model, so including certain information in a standard model reduces some vendors' competitive advantage. This cuts both ways — if information is left out of a standard information model, then a vendor's favorite feature may not work for customers that insist on limiting themselves to the standard.

It is a fact of life that information models need to change in response to changing tool requirements. It therefore appears that the best approach is to have the standard model include core functions that most tool vendors and customers can agree to, and to have tool vendors rely on information model extensibility to add the key features that allow them to differentiate their tools. Over time, the best vendor-specific concepts will migrate into the standard model.

To work, this approach requires that the repository be easy to extend independently by different vendors. In practice, this is a hard requirement to satisfy. For example, suppose two vendors want to add independent properties and methods to the same standard class. They can each create a subclass of the standard class. But if a customer wants to use both vendors' tools at the same time, then one of the vendors needs to subclass (i.e. extend) the other vendors' class. Since the vendors may be unaware of each other's extensions, the extension mechanism must allow each vendor to install an extension to a class without knowing about extensions made by other vendors to the same class. Most repository engines are unable to satisfy this requirement.

Whatever their extensibility story, all repository products ship with information models, and over time, the size of the information models exceeds that of the engine.

4 Summary

In this paper, we explained how a repository system is different than an object-oriented database system:

- it includes a repository engine that typically has more function than an OODBMS, and
- it includes information models that extend the repository with the shared semantics of specific kinds of tools.

This casts some doubt on whether the database community has found the right abstraction for OODBMSs. It also points to an opportunity for further research into database system features and interfaces.

That is, looking at repository engine functions that are not in OODBMSs may suggest areas where further innovation in OODBMS functionality would be worthwhile.

Acknowledgments

This paper is based in part on my repository tutorial at the 1995 International Conference on Very Large Data Bases, in Zurich. Many people helped uncover and sort out the issues described here, especially Thomas Bergstraesser, John Cheesman, Bill Dawson, David Maier, Paul Sanders, and David Shutt. I thank them all for their help.

References

1. Bernstein, P.A., "Middleware," *Communications of the ACM*, Vol. 39, No. 2 (Feb. 1996), pp. 86-98

2. Bernstein, P.A., "The Repository: A Modern Vision," *Database Programming and Design*, Miller Freeman, December 1996, pp. 28-35.

3. Bernstein, P.A., U. Dayal, "An Overview of Repository Technology," *International Conference on Very Large Data Bases*, Morgan Kaufmann Publishers, San Francisco, 1994, pp. 705-713.

4. Bernstein, P.A., B. Harry, P.J. Sanders, D. Shutt, J. Zander, "The Microsoft Repository," Invited Keynote address, *Proc. of 23rd International Conference on Very Large Data Bases* (Athens, Greece), Morgan Kaufmann Publishers, San Francisco, 1997, pp. 3-12.

5. Bernstein, P.A., E. Newcomer, *Principles of Transaction Processing*, Chapter 10, Morgan Kaufmann Publishers, San Francisco, 1996.

6. Cattell, R.G.G., D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, D. Wade, "The Object Database Standard: ODMG 2.0, Morgan Kaufmann Publishers, San Francisco, CA, 1997.

7. Ceri, S. and J. Widom (editors), *Active Database Systems: Triggers and Rules for Advanced Database Processing*, Morgan Kaufmann Publishers, San Francisco, CA, 1995.

8. Constantopoulos, P., M. Jarke, J. Mylopoulos, Y. Vassiliou, "The Software Information Base: A Server for Reuse," *VLDB Journal*, 4 (1995), Boxwood Press, Pacific Grove, CA, pp. 1- 43.

9. Jarke, M., C. Maltzahn, T. Rose, "Sharing Processes: Team Coordination in Design Repositories,", *Int'l Journal of Intelligent and Cooperative Information Systems 1,1 (Mar 92)*, World Scientific Publishing, Singapore, pp. 145-168.

10. Katz, R.H. "Toward a Unified Framework for Version Modeling in Engineering Databases," *ACM Computing Surveys 22, 4* (Dec. '90).

11. Microsoft Corp., *Reconciliation*, Oct. 30, 1996, http://www.microsoft.com/msdn/sdk/platforms/doc/sdk/win32/95guide/src/briefcas_2.htm

12. Object Management Group, "Relationship Service Specification," Chapter 9 of *CORBA Services: Common Object Services Specification*, Framingham, MA, March 1996.

13. Object Management Group, "Unified Modeling Language, Version 1.1," http://www.rational.com, Sept, 1997. (Joint submission by Rational Software, Microsoft, Hewlett-Packard, Oracle, Sterling Software, MCI Systemhouse, Unisys, ICON Computing, IntelliCorp, i-Logix, IBM, ObjecTime, Platinum Technology, Ptech, Taskon, Reich Technologies, Softeam)

14. Rogerson, Dale, *Inside COM*, Microsoft Press, Redmond, Washington, 1997.

15. Rumbaugh, J. "Controlling Propagation of Operation using Attributes on Relations," Conf. on Object-Oriented Programming Systems and Languages (OOPSLA), 1988, pp. 285-296.

16. Wakeman, L. and J. Jowett, "PCTE - The Standard for Open Repositories," Prentice-Hall, '93.