The Microsoft Repository

Philip A. Bernstein [1]
Microsoft Corporation

Brian Harry
Microsoft Corporation

Paul Sanders
Texas Instruments, Inc.

David Shutt
Microsoft Corporation

Jason Zander
Microsoft Corporation

Abstract

The Microsoft Repository is an object-oriented repository that ships as a component of Visual Basic (Version 5.0). It includes a set of ActiveX interfaces that a developer can use to define information models, and a repository engine that is the underlying storage mechanism for these information models. The repository engine sits on top of a SQL database system.

The repository is designed to meet the persistent storage needs of software tools. Its two main technical goals are:

- compatibility with Microsoft's existing ActiveX object architecture consisting of the Component Object Model (COM) and Automation and
- extensibility by customers and independent software vendors who need to tailor the repository by adding functionality to objects stored by the repository engine and extending information models provided by Microsoft and others.

This paper explains how the Repository attains these goals by providing an object-oriented database (OODB) architecture based on Microsoft's binary object model (COM) and type system of Visual Basic (Automation).

1. Introduction

Microsoft Repository is composed of two major components: a set of object-oriented ActiveX interfaces that a developer can use to define information models, and a repository engine that is the underlying storage mechanism for these information models. (*Information model* is repository terminology for database schema [3].) The repository engine sits on top of either Microsoft SQL Server[1] or Microsoft Jet (the database system in Microsoft Access) and supports both navigational access via the object-oriented interfaces and direct SQL access to the underlying store. In addition, the Repository includes a set of information models that cover the data sharing needs of software tools.

The two main technical goals of Microsoft Repository are

1. COM/ActiveX Compatibility - It should fit naturally into Microsoft's existing object architecture, consisting of COM and Automation (now subsumed under ActiveX). Thus, the repository should use existing ActiveX interfaces and implementation technology wherever possible and minimize the number of new concepts that the large community of ActiveX users needs to learn.
2. Extensibility - Given the size and diversity of Microsoft's market, it's important that customers and third-party vendors be able to tailor the repository to their needs, both by providing methods on objects stored by the repository engine and by extending persistent state. The latter is done declaratively, with no code.

This paper explains how Microsoft Repository attains these goals by providing an OODB architecture that fits into Microsoft's existing object infrastructure. In contrast to C++ or Smalltalk based OODBs, its object model is a binary standard, not a language API, and is very strongly interface-based, rather than class-based. Fitting an OODB or repository into an existing object model is a delicate activity, which we explain in detail. The reward is a repository that offers the powerful extensibility of COM/ActiveX, without requiring many new extensibility features of its own.

---

[1] Authors' address: Microsoft Corp., One Microsoft Way, Redmond, WA 98052-6399. Email: {philbe, bharry, v-paulsa, dshutt, jasonz}@microsoft.com

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

[1] ActiveX, Microsoft, Microsoft SQL Server, Visual Basic, and Windows are trademarks of Microsoft Corporation.

The Microsoft Repository is also interesting because of its pervasiveness. It ships in Visual Basic 5.0 Professional and Enterprise editions and will therefore have several hundred thousand copies deployed within a year — perhaps more, as it is added to other Microsoft products.

There is very little in the research literature about complete repository systems. A general introduction can be found in [3]. The PCTE standard is described in [10]. Many OODBs are used as repositories, though they are not the same thing, as explained in [2]. Still, the large literature on OODB system descriptions is relevant to the present work, such as [4, 11]. A comparison of our design with other systems is beyond the scope of this paper.

2. COM and Automation

2.1 The Component Object Model

Microsoft's Component Object Model (COM) is the foundation of Microsoft's object architecture. It is a binary standard that describes component-to-component early-bound calling conventions in a language-neutral fashion, so that components written in one language can seamlessly call components written in another language.

In COM, a *class* is an executable program image. It can have multiple *interfaces*, where each interface is a collection of methods, called *members*. All of the Repository's interfaces are COM interfaces.

Each class has a class id, which is a 128-bit globally unique identifier (GUID). Given a class id, the function `CoCreateInstance` creates an *object*, which is an instance of the class. `CoCreateInstance` finds the class's executable by looking up the class id in the registry, which is a small hierarchical persistent store managed by Windows operating systems. Entering the class id in the registry is part of the class's installation procedure.

An interface's specification includes an ordered list of its method names, each method's parameters, and an *interface identifier* (IID), which is a GUID. An interface's specification is immutable. Therefore, to enhance an interface, one must implement a new interface with a new IID.

By convention, interface names begin with *I*. Figure 1 gives a graphical representation of a class with multiple interfaces; interfaces are depicted as lollipops attached to the class or instance of the class. Methods are ordinarily not shown in this representation; for example, the `IForm` interface could support the `Resize` and `AddControl` methods, which are not shown in the figure.

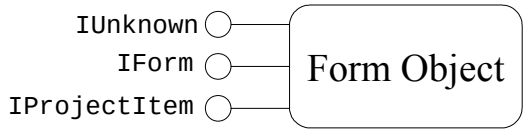

Figure 1 Representation of a COM Component

A COM interface can directly inherit from at most one other interface, in which case it supports the members of the interface from which it inherits. Every COM object inherits, directly or indirectly, from `IUnknown`. COM supports a form of multiple inheritance, in that a class can support many interfaces. It also supports polymorphism, in that an interface can be supported by many classes.

Every COM class, and hence instances of that class, support the interface `IUnknown`. `QueryInterface` is a method on `IUnknown` that allows a client to ask a COM object if it supports a particular interface, given the interface's unique IID. If the object supports the IID, it returns a pointer to that interface on that object. In this case, the client knows exactly what behavior the object will provide, because interface definitions are immutable. If it doesn't support the IID, it returns null. Thus, in Figure 1, an instance of `Form` would respond positively to a call on `QueryInterface` given `IForm's` IID and therefore supports all the methods specified for `IForm`.

The `QueryInterface` mechanism helps cope with type evolution, as follows:

- Since interface definitions are immutable, to change the behavior of an interface, one must define a new interface. Over time, one may have several different interfaces that are effectively versions of each other. We use "version" loosely here; each "version" is an independent interface insofar as COM is concerned.
- A class can support several different interfaces, which may be different versions of an interface that has evolved over time.
- A client can cope with multiple versions of a class's interface as follows: The client queries for the IID of the interface version it prefers. The class's instance replies yes or no. If it answers no, the client tries its second favorite interface, and so on. The client and object can interoperate if the client finds an interface that it knows how to use and that the object supports.

This mechanism allows classes and their clients to be independently upgraded.

Every class has a *class factory*, which can create instances of the class. The class factory returns a pointer to an interface on the object. After receiving this pointer, a client can call methods on that object (locally, or remotely using Distributed COM (DCOM)), or can use `QueryInterface` to find other interfaces on the object and call methods on them.

A COM class, C, can be extended by wrapping it. This technique is called *aggregation* if the wrapper passes through any of C's interfaces, else it is called *containment*. The technique involves writing a class, C′, that supports the extended behavior, which may consist of new interfaces and/or wrapped implementations of C's interfaces. C′ only needs access to C's executable (not its

source code). To make this work, C′ replies to `QueryInterface` only on interfaces C′ implements (i.e. those it wrapped or its new interfaces). It delegates calls on `QueryInterface` for other interfaces to C. Since C′ includes all of C's behavior, C′ uses C's class id, c, so `CoCreateInstance`(c) produces instances of C′. COM aggregation is explained thoroughly in COM documentation and is well known to COM developers [9].

A class can aggregate many classes and be aggregated by many classes; in this sense, COM supports multiple inheritance of implementations. As we will see, the Repository's support of user-defined methods and much of its extensibility is a direct application of COM aggregation.

A COM interface on an object is implemented in memory as a vtable (i.e. virtual table) plus some object-local data structures. A vtable has an entry for each of its members, which points to the in-memory executable for that member. The pointer returned by `QueryInterface` points to that interface's vtable in that object. In this sense COM is a binary calling standard. Since the caller is assumed to know the order (and meaning) of member entries in the vtable, COM is best suited for early-bound access.

2.2 Automation

Automation is a mechanism for late-bound calling of objects, originally developed for Visual Basic and later integrated with COM. Automation functionality is captured by the COM interface *IDispatch*. `IDispatch` supports a method, `Invoke(M, parm1, parm2, …)`[2], which implements a late-bound call to method M with parameters parm1, parm2, …, on the object (i.e., the one that implements `IDispatch`). For an interface on an object to be invoked in this way, it must inherit from `IDispatch`, in which case it is called a *dispatch interface*.

A dispatch interface can have many members, each identified by a *dispatch id*, which is the value used for parameter M in `Invoke`. `IDispatch` also includes a method `GetIDsOfNames`, which maps member names to dispatch ids (for efficient late-bound access), using information contained in a type library (described below).

A member of a dispatch interface can either be an ordinary method or a *property*, which is a shorthand for saying it has methods `get_Foo` and `put_Foo` for the property Foo. A property can either be single-valued or collection-valued. In the latter case, it returns a collection object, which in turns supports the following methods:

- `Add` – inserts an element
- `Count` – returns the cardinality of the collection
- `Item` – retrieves an element by index or key
- `Remove` – deletes an element identified by index or key, and
- `_enum` – returns an enumerator (i.e. cursor) on the collection, which can be traversed by calls to the Next method.

An interface can be both a COM interface and a dispatch interface, called a *dual* interface. This is an optimization that allows some members of a dispatch interface to be called through the early-bound COM mechanism. A caller who knows the definition of the interface at compile time can use this information to make an early-bound call and therefore avoid the overhead of interpretation by `IDispatch`. All interfaces to the Repository engine are dual interfaces.

The "standard" implementation of `IDispatch` (i.e. for Visual Basic) uses a *type library* object to look up the definition of external interfaces it is asked to invoke. To produce a type library, a class developer writes an interface definition in Microsoft's interface definition language (MIDL) and compiles it into a type library, which can either be stored as part of the class's executable or in a separate file. Type libraries can be directly accessed via their own interfaces, such as `ITypeLib` and `ITypeInfo`. Often, they are accessed indirectly via `IDispatch::GetIDsOfNames`.

Visual Basic syntax translates directly into calls on IDispatch. For example, consider this program fragment:

```
DIM X as Object
X.Foo = 7
```

The Visual Basic implementation (called an *Automation Controller*) uses `GetIDsOfNames` to look up Foo, and then uses `Invoke` to call `put_Foo`(7). If X supports multiple interfaces, then the above program accesses property Foo on its *default* interface. Another interface, IBar, could be accessed like this:

```
DIM Y as IBar
Set Y = X
Z = Y.MyFunction()
```

The statement "Set `Y = X`" calls `QueryInterface` on `X` for `IBar` and assigns that value to `Y`.

3 The Repository Engine

3.1 The Repository's Object Model

COM and Automation are used as the native object model by the vast majority of programming tools for Microsoft operating systems. It was therefore a requirement that the repository engine's functionality be exposed as a set of COM and Automation objects. These objects are in-memory representations of the information held in the repository database. Every object supports a set of repository-specific dual interfaces. That is, an object is a repository object if it supports a certain set of repository-specific interfaces.

[2] `Invoke`'s signature is more complex, but the details are unimportant for this discussion.

The repository supports four main kinds of objects:

- **Repository Session** - represents the repository database itself. It behaves much like a database session.
- **Repository Object** - represents the persistent state of an object in a repository. That state consists of the object's properties and collections.
- **Collection Object** - represents a set of relationship objects. A collection of relationships is accessed and updated using the standard collection methods: `Add`, `Count`, `Remove`, `Item`, and `_Enum`.
- **Relationship Object** - represents a connection between two repository objects. A relationship can have properties (unlike the ODMG standard [4]). The relationship's connection and properties are stored in the repository database.

The repository engine is a type-driven interpreter. A user defines classes, interfaces, properties, methods and relationships. The repository engine then provides methods for creating objects that are instances of these classes, and for storing and retrieving these objects' properties and relationships to and from the repository database. One good way to understand the repository's capabilities is to understand what can be expressed in type definitions.

3.2 The Type Model

Repository type definitions are ordinary repository objects that have certain properties and relationships that are interpreted by the repository engine. For example, a class definition is an object that has a property containing its unique identifier and a relationship to the interfaces it implements. This usage of its own storage mechanism for type definitions is analogous to SQL engines, which store type definitions as rows of tables.

Type definitions are grouped into repository type libraries. These have the same logical structure and namespace behavior as Automation type libraries. Having the same namespace behavior is important for the repository to match Automation semantics for name-based access to properties and collections.* Specifically, class and interface names must be distinct (i.e., in `DIM X as ABC`, `ABC` could be a class or interface), and member names must be unique relative to an interface (i.e., in `Object.MyMember`, `MyMember` could be a method, collection, or property). Classes and interfaces also have unique class ids, as in COM.*

A type library contains definitions of the following kinds of objects:

i. Class – defines which interfaces it supports, one of which is its default interface (for Automation).*

ii. Relationship class – defines which collections (on which interfaces) are connected by instances of the relationship class.

iii. Interface definition – defines which properties, collections, and methods are members of this interface, and which interface it inherits from.*

iv. Property definition – defines properties of the property, such as its data type and its mapping to an underlying SQL column.

v. Collection definition – defines properties of the collection, such as min and max cardinality. These are properties of endpoints of a relationship type, called *roles* in some object models.

vi. Method definition – defines properties of the method, such as its dispatch id.*

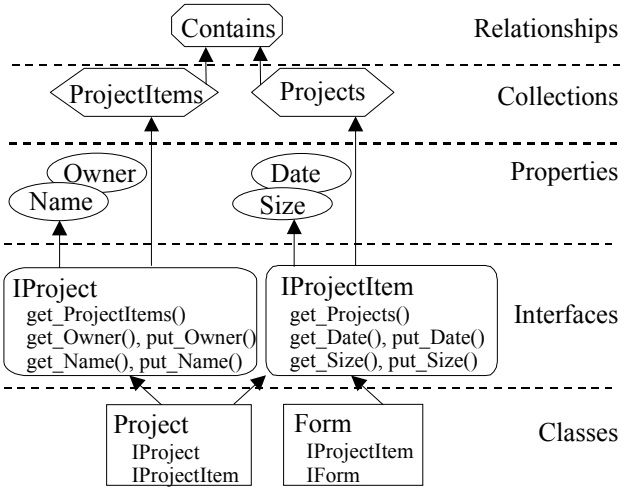

Figure 2 An Information Model

As shown in Figure 2, interfaces are defined on classes, and properties and collections (i.e. relationships) are defined on interfaces.* Interface `IProject` describes project containers and `IProjectItem` describes objects that can be put into project containers. The Project class supports both `IProject` and `IProjectItem` (since a project can have subprojects), while the Form class supports `IProjectItem` but not `IProject`. Properties and relationships that are specific to forms are captured by `IForm` (shown in the Form class but not defined in Fig. 2). The relationship Contains is accessible via the `ProjectItems` collection on `IProject` and the Projects collection on `IProjectItem`.

Like all repository objects, type definitions are instances of classes, which in turn support interfaces that have properties and relationships stored in the repository. For example, the definitions of `IProject` and `IProjectItem` are instances of the class InterfaceDef, which supports the interfaces `IInterfaceDef` (which

* To highlight the effect of COM and Automation on the repository design, we tag each sentence that describes such an effect by an asterisk.

provides the behavior unique to interface definitions) and `IReposTypeInfo` (which allows interfaces to be the target of `DIM` statements in Visual Basic*). Thus, the information summarized in (i) – (vi) above is captured by the interfaces and relationships summarized in Figure 3.

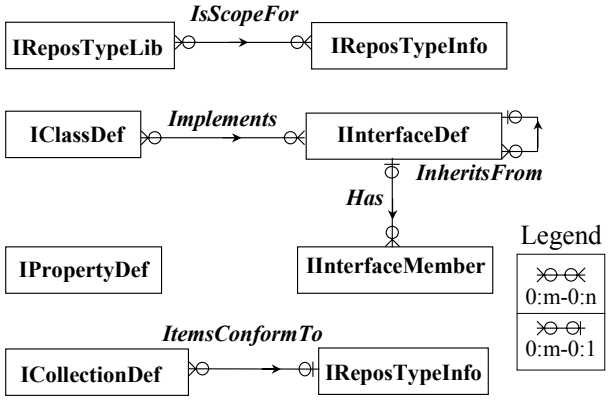

Figure 3 Repository Type Model

The classes that use these interfaces are as follows:

i. ClassDef – supports `IReposTypeInfo` and `IClassDef`

ii. RelationshipDef – supports `IReposTypeInfo` and `IClassDef` (relationship-specific information is in `ICollectionDef`, so no "IRelationshipDef" interface needed)

iii. InterfaceDef – supports `IReposTypeInfo` and `IInterfaceDef`

iv. PropertyDef – supports `IInterfaceMember` and `IPropertyDef`

v. CollectionDef – supports `IInterfaceMember` and `ICollectionDef`

vi. MethodDef – supports `IInterfaceMember`

The COM objects that represent type definitions are instances of the above classes.

The classes are described as instances of themselves. That is, there is an instance of ClassDef for each of the above classes: ClassDef, RelationshipDef, etc. And there is an instance of RelationshipDef for each of the relationships in Fig. 3: IsScopeFor, Implements, Has, etc. In this sense, the repository is self-describing. This characteristic is useful for model-driven tools, such as generic browsers and scripting languages, which need to discover the information model at run-time and which should be applicable to the repository's type model as well as models customized for applications. It also positions the repository to exploit its own new features that appear in future releases. For example, when the repository supports version and configuration management of repository objects, type definitions will automatically be able to be versioned and grouped into configurations too.

Several aspects of interface definitions are worth noting:

- The repository engine supports interface inheritance with the same semantics as COM.* That is, if an interface I2 InheritsFrom an interface I1, then all of the properties and collections that are defined on I1 are also available on I2.
- Not all of the properties of an interface need to be persisted in the repository.
- An interface can include custom methods, whose existence can be documented in the interface definition stored in the repository. The information model developer is responsible for implementing such methods (see Section 4).

3.3 Object Manipulation

Repository Objects

Each repository object has a unique 20-byte opaque "external" id. It can be created by the repository or supplied by the caller when creating the repository object. The latter is useful to give an object and its replica the same identity (e.g. a type definition that's stored in many repositories). Objects also have an internal identifier that is an 8-byte compressed representation of the global identifier, an important storage optimization. The local identifier is always assigned by the repository and can be different in every repository. Object identity can be determined by comparing object IDs (internal or external).

To use the repository, one starts by creating a *repository session*, which is an instance of the class Repository. One then uses the repository session's `Create` method to create a new repository database or its `Open` method to open an existing repository database. Now, one can access repository objects by following relationships from well-known repository objects or by executing queries.

One well-known repository object is the repository's unique root object, which is accessible from the repository session and connected directly or indirectly to all other repository objects in the database. Usually, class and interface definitions are well known, since their object id's are the same in every repository. From a class definition, there is a computed relationship to all repository objects that are instances of that class. Similarly, there is a computed relationship from each interface definition to all repository objects that support that interface.

To link up the result of a SQL query with the object-oriented API, the repository session supports a method `get_Object`, which loads an object given its object id. It also supports the `CreateObject` method, which creates a repository object of a given class.

Repository Objects can have single-valued scalar-valued properties, which are accessible using `IDispatch` (for Automation) and generic `get_value` and `put_value` methods (for COM). The former allows properties to be accessed using ordinary Visual Basic syntax*, such as

```
DIM X as RepositoryObject
X.Foo = 7
```

where Foo is a property of X's default interface.

Relationship Objects

A relationship is bi-directional. That is, it can be followed from either of the repository objects it connects. Like a repository object, it can have properties. Unlike a repository object, it can't have relationships or methods, though this restriction is likely to disappear in a future release. Customers drove us to support attributed relationships, which we accepted since it added no storage or run-time expense to information models that don't use the feature.

Each relationship is an instance of a *relationship class*. A relationship class definition connects two collection definitions (on the same or different interfaces), called the *origin* and *destination*. Although a relationship instance can be traversed in either direction, some semantics of the relationship is sensitive to the relationship's polarity indicated by origin and destination. More on this later.

Starting from a repository object, you can get to a relationship by accessing a relationship collection, and then accessing the relationship within the relationship collection. The repository object you start from is called the *source* and the one you traverse to is called the *target*. That is, the concepts of source and target are relative to the traversal direction. So, the source could be on the origin or destination side of the relationship's relationship class. Notice that a relationship is actually a member of two collections, one on its source and one on its target. In the common case where you don't need access to a relationship's properties, you can skip over the relationship object and go directly from source to target, by using methods on `ITargetObjectCol`(lection) instead of `IRelationshipCol`. This ability to skip over relationship objects avoids one disadvantage of attributed relation-ships — that it makes programs that don't need such attributes more verbose.

For example, consider the Contains relationship between `IProject` and `IProjectItem` in Fig. 2. Contains relationships would be accessed via the relationship collection ProjectItems (on interface `IProject`) on Project objects and the relationship collection Projects (on interface `IProjectItem`) on Form objects. The `GetProjectItems` method on `IProject` returns a collection of relationship objects, each of which points to a repository object supporting `IProjectItem`. (Or it may skip over the relationship objects and return a collection of Form objects.) Figure 4 is an instance-level view of this model, showing COM objects. The ProjectItems collection for the instance of the Project labeled MyProject has three relationships, one of which, labeled x, points to the instance of Form labeled MyForm. MyForm, in turn, supports `IProjectItem` and therefore has the collection Projects, which contains two relationships pointing to instances of Project, one of which is x pointing to MyProject.

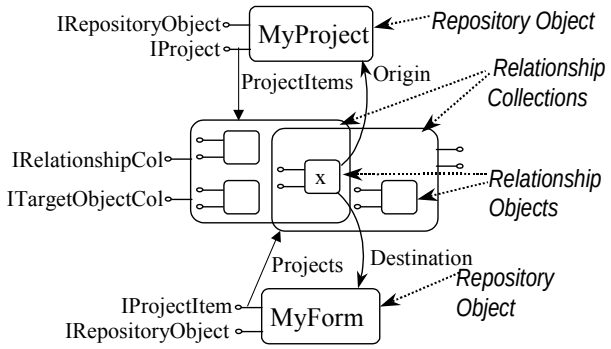

Figure 4 Relationships and Relationship Collections

Much of the interesting semantics of a repository is captured in the behavior of relationships. In ours, a relationship class can have three kinds of semantics: naming, sequencing, and delete propagation.

A relationship can have a name, which identifies the destination object relative to its origin. The origin collection definition of the relationship class specifies whether it's a naming relationship and, if so, whether names are case sensitive and/or unique (i.e. whether two instances of the relationship from the same origin must have different names). Since there can be more than one naming relationship to a repository object, a repository object can have different names in different contexts. For example, if the contains relationship type in fig. 2 is a naming relationship, and `IProject` is the origin, then there could be two relationships from different projects to the same form. That is, a form could have a different name in different projects, as shown in fig. 5.

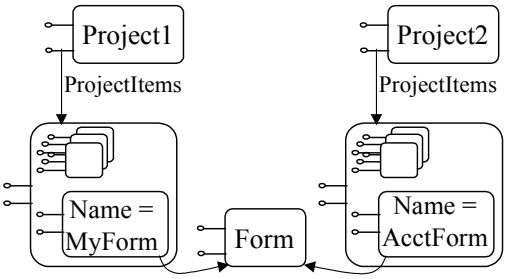

Figure 5 Named relationships

As a convenience, an object can have a name that's the same in all contexts by using a special interface `INamedObject`, which has one property called `Name`. If a repository object supports `INamedObject`, then the put_`Name` method on `IRepositoryObject` assigns the same name to that `Name` property and to all naming relationships to that object. A direct update to the `Name` property of `INamedObject` updates that property only. This avoids the extra API complexity of assigning names to every relationship to an object for programs that don't need context-dependent naming.

Within a relationship collection, the destination objects can be sequenced within the context of a particular origin object and relationship type. This is indicated by setting a

flag on the origin collection definition. One can use the `Insert` and `Move` methods on relationship collections to control the sequencing. (If it's a naming relationship and is not sequenced, the collection is ordered by name.) Sequencing is useful in many design scenarios, such as ordering column definitions in a table definition and ordering member definitions in an interface definition.

`Delete` methods can propagate to objects beyond the one being deleted. Deleting a relationship usually affects only the relationship being deleted. However, if the delete propagation flag is set on the collection definition of the relationship's origin, and the relationship is the last relationship of its type that points to the destination object, then the destination object is deleted too. This is useful for containment hierarchies, where an object that has no container should be deleted. Deleting a repository object causes the deletion of all incident relationships, some of which may propagate as just described.

Support for IUnknown

Some of a repository object's interfaces are generic interfaces supported by the repository engine on every repository object (e.g., `IRepositoryObject, IDispatch`). Others are custom interfaces defined in the information model. The properties and relationships on these interfaces are implemented by the generic repository engine by interpreting these interfaces' type definitions. Making these properties and relationships available through Automation involves making them accessible via interfaces that inherit from `IDispatch`.*

Making these interfaces available through COM involves supporting COM methods to access them. This follows immediately from the Automation implementation, since `IDispatch` is a COM interface,* with one exception. The engine's generic implementation of repository object would not respond positively to a `QueryInterface` call on `IUnknown` for custom interfaces. That is, the generic implementation of repository object would only know about interfaces that existed when its implementation was compiled. It would not know about interfaces that are defined later — custom interfaces, such as `IProject`. To ensure these custom interfaces are bona fide COM interfaces, the repository engine synthesizes such interfaces.* For each custom interface, such as `IProject`, it dynamically constructs a vtable for `IDispatch` that knows about the properties and collections of the custom interface. An object that supports this custom interface has a pointer to that vtable, and that pointer can be returned by a call to `QueryInterface` with the custom interface's interface id as parameter.

Support for Model-Driven Tools

Model-driven tools need to discover information models at run-time. This can be done by traversing type information stored in the repository. As a convenience, the repository offers a more direct way to get this information. It supports an interface `IRepositoryDispatch`, which inherits from `IDispatch` and supports one method, `Properties`. This method returns a collection of the properties and relationship collections defined on this interface, including those that are inherited from ancestor interfaces. To use this feature, interfaces defined in the information model should inherit from `IRepositoryDispatch`, rather than `IDispatch`.

The repository also supports the COM equivalent of `QueryInterface` for Visual Basic programmers.* Recall from Section 2.2 that one can force an execution of `QueryInterface` in Automation, by declaring an object variable to be of a particular interface, as in "`DIM Y as IBar`." But this only works for interfaces known to an application at compile time. To give the same capability to model-driven tools, which discover the information model at runtime, repository objects support a method called `Interface`, which takes an interface as a parameter and casts the object to the requested interface. Thus, if `IBar` were discovered at run-time, one could access property `Foo` on `IBar` as follows:

```
DIM X as RepositoryObject
Set Y=X.Interface("IBar").Properties("Foo")
```

3.4 Storage Model

The repository engine stores its data in a SQL database. This database contains the properties and relationships of objects stored in the repository. Some of the tables in this database are generic — they are present in every repository. The main generic tables are the object table and relationship table, which contain the basic information the engine needs to know about every repository object and relationship. Figure 6 shows how rows of these tables are related. Other tables are specific to the information model, such as the ProjectItem table in Figure 6. They

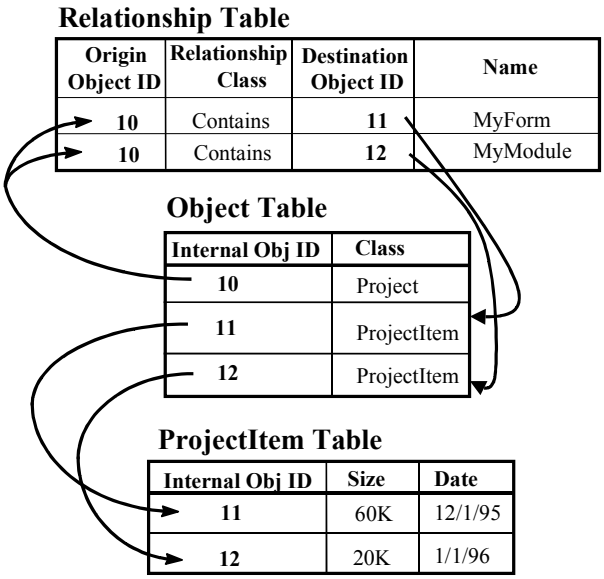

Figure 6 Repository Table Layout

contain the properties that appear in custom interface definitions (cf. IProjectItem definition in Fig. 2).

For most purposes, the user of the repository (a tool programmer) calls methods on ActiveX objects. However, users will sometimes want to issue SQL queries to the repository database for faster or more complex retrievals. They can do this using the `ExecuteQuery` method on `IRepositoryODBC`, which is supported by the Repository class. `ExecuteQuery` takes a SQL query that includes object id and class id in the SELECT clause, so that it can cast the returned rows as repository objects, which it returns in a collection. Updating the tables directly is *not* recommended, since the repository engine's update methods maintain the integrity of the database in subtle ways that a repository user might miss

When repository type definitions are created or modified, the repository automatically generates and modifies the layout of SQL tables that persist interface-specific properties. Each table having interface-specific properties is keyed on internal object id (the 8-byte compressed form) and has all of the properties of each interface that is stored in that table. Thus, the unit of mapping from information model properties to a database schema is the interface.

By default, the engine maps each interface on an object to a separate table. However, users can control this mapping by storing several interfaces in the same table. E.g., a user can have the properties of `IProjectItem` stored in the same table as those of `IForm`. Also, users can add and remove indexes on these tables, in addition to the index on internal object id, which the engine defines by default.

For fast traversal of relationships, the relationship table has a clustered index on [origin object id, relationship type, name] and a secondary index on the primary key [destination object id, relationship type, origin object id] (in SQL Server, a table's clustered index needn't be on its primary key). The most common queries on this table are to retrieve a relationship collection for a given object (given the origin id and relationship type, find the destination objects, or vice versa). For named collections, the third column in the index allows us to find a named relationship within a collection or retrieve the relationships in the collection in name sequence.

The type definition classes ClassDef, RelationshipDef, InterfaceDef, etc. are mapped to tables in the same way as other classes, by mapping the interfaces they support (see Fig.3) into tables. For example, IPropertyDef has the properties: APIType, SQLType, SQLSize, SQLScale, ColumnName, and Flags. This interface is mapped to a table whose columns include these properties and an internal object id (its key). Given a property definition (a row in this table) and its associated interface definition (which identifies the interface's table), the repository engine can find and interpret instances of this property.

Since type information is frequently accessed, it is cached in an optimized main memory structure that's persisted, to avoid recomputing it every time the repository is opened. The cost is updating this structure whenever a type definition is updated, a non-trivial but infrequently-incurred expense.

3.5 Transactions

Advanced transaction capability was not a goal of our version one product. Rather, we wanted to minimize the implementation effort by passing through the transaction behavior of the underlying SQL DBMS. Still, even this modest goal required that we include some transaction functions in the repository engine itself.

Like most database access models (e.g. ODBC [6]), we attach transaction behavior to the user's connection to the database, which in our case is a repository session. Thus, each repository session offers the `Begin`, `Commit` and `Abort` methods. Every repository object is loaded in the context of a repository session and retains that context as long as it's loaded. So its transaction context is implicit and need not be passed as a parameter to any calls.

Transactions are flat, i.e. not nested. All methods on a repository object execute within the transaction of its corresponding repository session. Methods within a transaction read committed data, so its updates are isolated from other transactions until it commits, when the updates are permanently installed in the database. That is, degree 2 (read committed) consistency is the default [1,7].

Like other DBMS designers before us, we found that degree 3 consistency was fairly low on our customers' priority list, so we swallowed our pride and deferred serializability for a later release. However, we do offer a lock primitive that allows users to explicitly synchronize access to shared data and thereby get the effect of two-phase locking, albeit with some application programming.

Each repository session only allows one transaction to execute at a time. To have two concurrent transactions on the same repository database, one can create two repository sessions connected to that database. If the repository sessions execute in the same process, then they share the database cache. Therefore, updates by a transaction T in one repository session are visible to transactions in the other repository session as soon as T commits. Repository sessions in other processes will not see T's updates until that process's repository engine refreshes its cache, which it does periodically. Methods are offered to tell the repository engine to refresh its cache immediately, so an up-to-date view of the repository database can be obtained if needed. This explicit refresh seemed rather crude to us, but actually reflects the behavior of most of the tools that would use the repository. Most Windows-based tools, beginning with the Explorer, offer an explicit refresh.

If an application has two repository sessions connected to the same database in the same process and loads the same

repository object through both repository instances, it will get two COM objects representing the same persisted repository object. This is required because each repository (COM) object retains the context of the repository session that loaded it, where it gets its transaction context. To avoid a cache coherency problem, we ensure that both COM objects share the same cached copy of the repository object's persistent state. We ensure this in all situations where two COM objects representing the same persisted repository object are concurrently active.

4. Extensibility

Defining a class that has only properties and relationships involves only providing type definitions for the class and all of its interfaces. This is akin to writing data definitions in SQL. Moreover, one can extend classes in this way dynamically. For example, one can add an interface to an existing class, and the repository engine will create and alter table definitions as necessary.

One can extend the behavior of the repository engine by providing custom code. Useful extensions could include validating special kinds of integrity constraints (which are not supported by the repository), adding custom methods to interfaces (such as supporting a `Build` method on `IProject`), or storing some properties of an object outside the repository (e.g. in a file). This is done by writing a wrapper for the repository object, re-implementing interfaces that you want to extend, and calling the repository engine's base implementation of those interfaces to read and write properties and relationships. (See Fig.7.) Interfaces that you do not want to extend are simply passed through. The mechanics of this wrapping is defin-ed by COM aggregation, which was mentioned in Section 2.1.* When the repository creates or loads an object, it calls CoCreateInstance, thereby invoking the user's customized class.

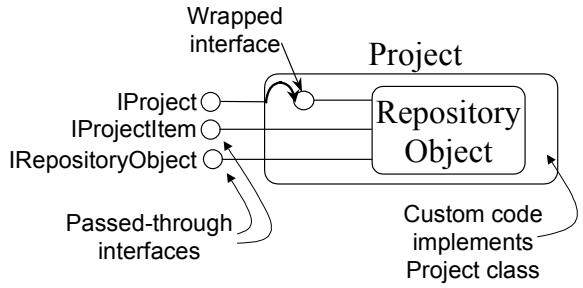

Figure 7 Using COM Aggregation to Extend a Repository Object Class

This is another example of the repository using a standard COM mechanism, in this case one for extending objects.

Another form of extensibility is the ability to create new versions of interfaces — that is, new versions of information models. This is a major problem in many repository systems. We support it using the standard COM approach explained in Section 2.1:* Every COM interface is immutable. Its interface id identifies a contract that, once published, cannot be changed. So, to change an interface, you define a new interface. Newly written clients are built to prefer the new interface but cope with the old one; newly written classes are built to support both interfaces, so that old clients can use them.

One should then write an aggregation of the class that supports the new interface, and supports the old interface too by mapping old interface members to new interface members. The amounts to a view. One could automate this by a model-driven tool that creates the aggregated class from the interface definitions.

5. Interface-Oriented Information Models

COM is highly interface-centric. One can write programs that access objects by navigating interfaces and never know the class of which those objects are instances. The only reason to know an object's class is to create the object in the first place.

COM's interface-centric view has a profound effect on tools that share objects in the repository. To share data, tools only need to agree on interface definitions, not on class definitions.* For example, suppose we define an interface `IComponentDescription` that includes properties `Owner`, `TechnologyType` (e.g. ActiveX Control, Stored Procedure, Java applet), and `Status` (e.g., draft, unit-tested, system-tested) and a collection `Keywords`. A component reuse tool that understands `IComponentDescription` could display useful information about the component and offer keyword-based search of components. Many different tools could create reusable components that support `IComponent-Description`. For example, development tools for ActiveX controls, stored procedures, and Java applets could create objects of different classes, but all those classes could support `IComponentDescription` and therefore be visible to the component reuse tool.

Thus, to support sharing between tools, the important part of an information model is the interface definitions, not the class definitions. The interface definitions define the properties and relationships that tools of a certain category need to depend on. Such categories are called *subject areas* in information modeling terminology. Example subject areas are component-based design, databases, data warehouses, and project configurations.

To share information, class definitions from different vendors support the same interfaces. However, similar classes from different vendors (such as the table definition class supported by database design tool vendors) don't need to support the same combination of interfaces and typically have different implementations of those interfaces.* This flexibility has high payoff to a large vendor like Microsoft, which expects many other vendors to use its repository.

Still, vendors often need to make some assumptions about which sets of interfaces are used in combination. Therefore, an information model should specify which sets of interfaces should be implemented together. For example, it might say that if a class support `IForm`, then it must also support `IProjectItem`. In ActiveX, such a combination of interfaces is called a *cotype*.*

At the level of mechanism, an information model consists of a set of interface definitions, each uniquely identified by its repository object id and COM interface id, along with the property, collection, relationship, and method definitions that it references. Generally, an information model is packaged in its own repository type library, so it's easy to tell if a repository has that information model loaded and so that independently developed information models need not worry about name conflicts and the like.

Microsoft is collaborating with other vendors to publish open information models in areas that are relevant to its tool groups. This will enable independent tool vendors to share objects with Microsoft tools and each other. Given how easy it is to extend the repository, vendors will be able to specialize those information models to their needs without sacrificing interoperability with other vendors that conform to those information models.

6. Conclusion

The repository is used in Visual Basic 5.0 as the storage for a component reuse tool. It also supports an information model for Rational Software's Unified Modeling Language (UML), which is under consideration as an OMG standard [8]. It is used in to support the exchange of object models with Visual Basic's Visual Modeler tool.

Since the Microsoft Repository is a new product, it's too soon to draw strong conclusions about whether tool vendors find it a useful place to store and share persistent objects. Likewise, it's too soon to tell whether the trade-offs that were made to meet the product release schedule were the optimal ones.

However, we do feel quite confident that the primary goal of fitting hand-in-glove with COM and Automation has been well met, yielding several major benefits:

- Extensibility and evolvability of types and classes without breaking applications
- Class-independent sharing of type information using interfaces
- Easy prototyping of information models without writing any code
- Use of Visual Basic as a persistent programming language with no impedance mismatch

The first two benefits we attained by supporting COM, with its interface-oriented type system, globally unique interface ids, and self-describing objects via the method QueryInterface. The third benefit is attained by the type-driven interpreter. And the last benefit was attained by supporting IDispatch, with a type system that is compatible with type libraries.

Acknowledgments

The design of the Microsoft Repository interfaces was a joint effort between Microsoft Corp. and Texas Instruments, Inc., originally conceived by David Vaskevitch of Microsoft and Keith Short of Texas Instruments. We're grateful for their sponsorship of the effort. In addition to the authors, early contributors to the design effort included John Cheesman and Bill Dawson (TI), and Melissa Waldie and Laura Yedwab (Microsoft). We learned much from their initial investigations. We also thank Thomas Bergstraesser, Murat Ersan, and David Maier for their help with many aspects of the design and implementation.

References

1. Berenson, H., P. A. Bernstein, J. Gray, J. Melton, E. O'Neil, P. O'Neil, "A Critique of ANSI SQL Isolation Level," *Proc. ACM SIGMOD 1995,* ACM, N.Y
2. Bernstein, P.A., "Repositories and Object-Oriented Databases," *Proceedings of BTW '97*, Springer, March 1997, pp. 34-46.
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. Cattell, R.G.G., T. Atwood, D. Barry, J, Duhl, J. Eastman, G. Ferran, D. Jordan, M. Loomis, D. Wade, *The Object Database Standard: ODMG-93*, Morgan Kaufmann Publishers, San Francisco, CA, 1995.
5. 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.
6. Geiger, K., *Inside ODBC*, Microsoft Press, Redmond, WA, 1995.
7. Gray, J., R. Lorie, G. Putzolu and, I. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," in *Readings in Database Sys*, 2nd Edition, Chapter 3, Michael Stonebraker, Ed., Morgan Kaufmann 1994 (originally published 1977).
8. Rational Corp., "Unified Modeling Language Resource Center," http://www.rational.com/uml.
9. Rogerson, D., *Inside COM*, Microsoft Press, Redmond, WA, 1997
10. Wakeman, L. and J. Jowett, *PCTE - The Standard for Open Repositories*, Prentice-Hall, '93.
11. Zdonik, S.B., and D. Maier, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, San Francisco, 1990.