Managing Meta-Information with Microsoft Repository

OMNI/01/00

Egil P.Andersen, NR

Oslo
August 2000

Norsk Regnesentral
ANVENDT DATAFORSKNING

NR-notat/NR Note

Tittel/Title: Managing Meta-Information with Microsoft Repository

Dato/Date: 31.August
År/Year: 2000
Notat nr: OMNI/01/00
Note no:

Forfatter/Author: Egil P.Andersen (NR)

Sammendrag/Abstract:
A *"meta-information repository"* is one or more databases that contain meta-information, and applications that provide access to this information. Meta-information ("information about information") can be e.g. schema information on UML models for a particular project, information on documents that belong to a particular project (system documentation, user documentation, QA plans, etc), and typically all kinds of information that is relevant to a software development process.

The purpose of using a meta-information repository is both to ease access to all relevant project meta-information, and also to ease the maintenance of this information, which can possibly be very large. The information may well be distributed, but e.g. a developer or a project manager should be able to access all project information from a single entry point.

The rest of this document provides an introduction to *Microsoft Repository*; a meta-information repository available on the NT/Windows platform. In summary, the following seems to be the main advantages and disadvantages of using MS Repository v.2.0.

advantages:

- It is "mainstream" and may become a widespread standard for managing meta-information.
- It is huge and covers meta-information from many subject areas. It can also be extended to include meta-information from new subject areas.

disadvantages:

- Some tool support already exists, and more can be expected, but the latest version of Rose, Rose 98i, does not support Repository as much as one could desire. Manual adaptions and extensions must be implemented.
- It takes some time to become familiar with the basic, generic model used by every Repository (the Repository Type Information Model); particularly with respect to deciding how to record new kinds of meta-information.

Emneord/Keywords: Meta-information repositories, UML, XML, COM, databases

Tilgjengelighet/Availability: Open

Prosjektnr./Project no.: 636011, SINAI

Satsningsfelt/Research field: Software architectures for distributed information systems

Antall sider/No. of pages: 32

Norsk Regnesentral / Norwegian Computing Center
Gaustadalléen 23, Postboks 114 Blindern, 0314 Oslo, Norway
Telefon 22 85 25 00, telefax 22 69 76 60

Contents

1. Background and Motivation

Meta-Information Repositories

A *"meta-information repository"* is one or more databases that contain meta-information, and applications that provide access to this information. Meta-information ("information about information") can be e.g. schema information on UML models for a particular project, information on documents that belong to a particular project (system documentation, user documentation, QA plans, etc), and typically all kinds of information that is relevant to a software development process.

The purpose of using a meta-information repository is both to ease access to all relevant project meta-information, and also to ease the maintenance of this information, which can possibly be very large. The information may well be distributed, but e.g. a developer or a project manager should be able to access all project information from a single entry point.

The rest of this document provides an introduction to *Microsoft Repository*; a meta-information repository available on the NT/Windows platform. In summary, the following seems to be the main advantages and disadvantages of using MS Repository v.2.0.

advantages:

- It is "mainstream" and may become a widespread standard for managing meta-information.
- It is huge and covers meta-information from many subject areas. It can also be extended to include meta-information from new subject areas.

disadvantages:

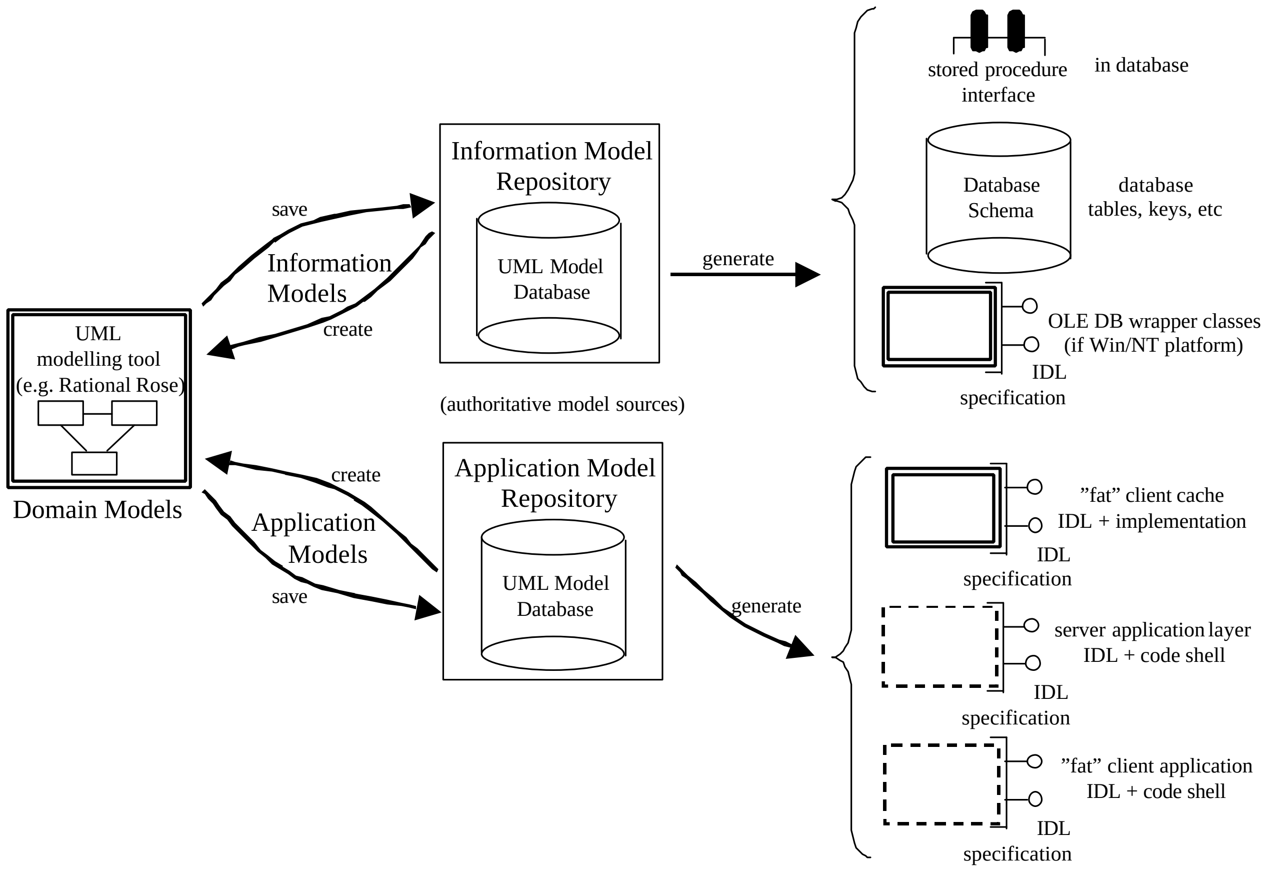
- Some tool support already exists, and more can be expected, but the latest version of Rose, Rose 98i, does not support Repository as much as one could desire. Manual adaptions and extensions must be implemented.
- It takes some time to become familiar with the basic, generic model used by every Repository (the Repository Type Information Model); particularly with respect to deciding how to record new kinds of meta-information.

SINAI Code Generation from UML Models

This introduction to MS Repository particularly emphasises aspects relevant to managing meta-information about UML models. This is a key issue in the SINAI project ("Seamless Integration of Non-homogenous Applications and their Information") at OMNI.

UML models, e.g. made in Rational Rose, play a key role in SINAI. They will, amongst others, be used to generate parts of the application specific code in server- and ("fat") client side components. Figure 1 illustrates which kind of code it is relevant to generate. SINAI distinguishes between *information models* versus *application models*. Both these models, for a particular domain or application, can be defined in e.g. Rational Rose. They are each given a globally unique identifier (GUID) and kept in a model repository *as the authoritative source for model meta-data*. Authorised clients will have access to this information the same way they will have access to objects implementing these models, and repositories can be distributed similar to how object instance databases are distributed.

From the model repository, an information model can be used to generate a database schema for object instances, a stored procedure interface for this schema, and possibly OLE DB wrapper classes (for Win/NT platforms). An application model cannot, in general, be used to generate actual application code. That is, except for "fat" client cache components (both their IDL and also an implementation of this IDL). It can be used to generate IDL specifications, and possibly code shells for their implementation, however.



stored procedure interface
in database
Database Schema
database tables, keys, etc
OLE DB wrapper classes (if Win/NT platform)
IDL specification
Information Model Repository
UML Model Database
save
Information Models
create
generate
UML modelling tool (e.g. Rational Rose)
Domain Models
(authoritative model sources)
Application Model Repository
UML Model Database
create
Application Models
save
generate
"fat" client cache IDL + implementation
IDL specification
server application layer IDL + code shell
IDL specification
"fat" client application IDL + code shell
IDL specification

Figure 1. *UML models and model information repositories in SINAI.*

2. Microsoft Repository v.2.0

Unfortunately, and at least to my knowledge, there are currently (autumn '99) no good books describing how to work with the *Microsoft Repository*. Microsoft Repository is not easy to get into at first, and it is huge. It is much more than just a database schema, more like an architecture for the management of meta-information in continuously evolving domains. According to the MSDN Library, Microsoft Repository "provides a common place to persist information about objects and relationships between objects, and in doing so it provides a standard way to describe object-oriented information used by software tools".

Microsoft Repository v.2.0 ships with Visual Basic v.6.0, Visual Studio v.6.0 and SQL Server v.7.0. It uses UML for modelling domain-dependent meta-data, relational databases for data storage, SQL for data retrieval, and XML for meta-data interchange. It consists of two major components; a set of interfaces for defining *Open Information Models* and to operate on the information they describe, and a *Repository engine* that is the underlying storage medium for these models and their data.

The Open Information Model (OIM) is a model that describes the objects that represent meta-information in the Repository, and their relationships. The OIM is defined in UML and organized into several domains (subject areas). That is, the OIM consists of an abstract core model that is extended with new domains; e.g. to store information on UML models, relational database schemas, COM components, datatypes, and more. There are mechanisms for extending the OIM to enable further evolution of the models, and the inclusion of new domains.

The OIM is developed by Microsoft together with over 20 industry-leading companies, and based on industry standards such as SQL, COM, Java, and UML, and has been reviewed by over 300 companies as part of Microsoft's Open Process. In December, 1998, Microsoft transferred rights to maintain and evolve the Open Information Model to the *Meta Data Coalition (MDC)*, an industry consortium comprising dozens of vendors of enterprise tools, meta data management tools, and data warehousing products. The Meta Data Coalition will maintain and evolve the Open Information Model as a technology-independent and vendor-neutral meta data standard. In July 1999, the Meta Data Coalition announced the acceptance of version 1.0 of the MDC Open Information Model as a standard.

The Repository engine sits on top of a Repository database, which can be either *SQL Server* or a *Microsoft Jet* database (Access). It manages data in the Repository, and it provides basic functions to store and retrieve Repository objects and relationships between them.

The repository engine exposes repository information as COM objects with interfaces for manipulating data stored in a Repository, and thereby making the database schema and its storage format transparent to a user. It allows for both COM and Automation programming, thus making it relatively easy to integrate Repositories and different programming tools on the Microsoft platform.

In addition to basic object management, Microsoft Repository v.2.0 offers version and configuration management facilities. To support evolving meta-information shared by different development teams, the repository engine allows objects to be versioned, grouped into configurations, and managed using check-out/check-in.

MS Repository SDK 2.1b

There exists a *Repository SDK v.2.1b* that can be downloaded from http://msdn.microsoft.com/repository. The SDK works with Microsoft Repository v.2.0+, and using it requires Visual Studio v.6.0 Enterprise Edition, or Visual Basic v.6.0 Enterprise Edition.

The SDK contains a number of useful files in the following catalogs:

- **Readme.htm** -- general information
- **Setup folder** -- files needed to install (or remove) the SDK executable programs on your hard disk
- **Bin folder** -- files necessary for the SDK to run
- **Doc folder** -- the help file *RepSDK.chm* and three example models discussed in the documentation
- **Include folder** -- header and constants files
- **OIM folder** -- all files and subfolders of the Open Information Model (OIM)
- **Samples folder** -- development sample programs.
- **SQLTools folder** -- add-on tools for SQL Server 7.0.

The SDK also includes a *Model Development Kit (MDK)*, which is a development environment for information models, used to define new information models extending the current OIM. The MDK tool takes a Visual Modeler, or Rational Rose, model as input, performs basic validations, and generates all necessary deliverables to develop and deploy the custom made information model extending the OIM.

Finally, the SDK includes sample code and the following Repository add-on utilities:

- Repository Browser
- Information Model Installer
- XML Import/Export
- OLE DB Scanner

Sources of Repository Information

The main sources of information on the Repository are:

- On-line help for MS Repository SDK 2.1 (*RepSDK.chm*)
- MSDN library

- http://msdn.microsoft.com/repository (lots of information, but also very much useless information)

Newsgroups

The *microsoft.public.repository* newsgroup features peer-to-peer discussion of the Microsoft Repository. In order to access this newsgroup, you must go through the *msnews.microsoft.com* server. If you are using Outlook Express as your news reader, you can add this server to your list through the "Tools|Accounts" menu item.

3. The Repository Browser

The repository browser is a good starting point to get an overview of how repository information is organized, and which services are available for working with repositories. However, at first it is quite difficult to understand how the various information elements, e.g. models, objects, interfaces, etc, are related to each other.

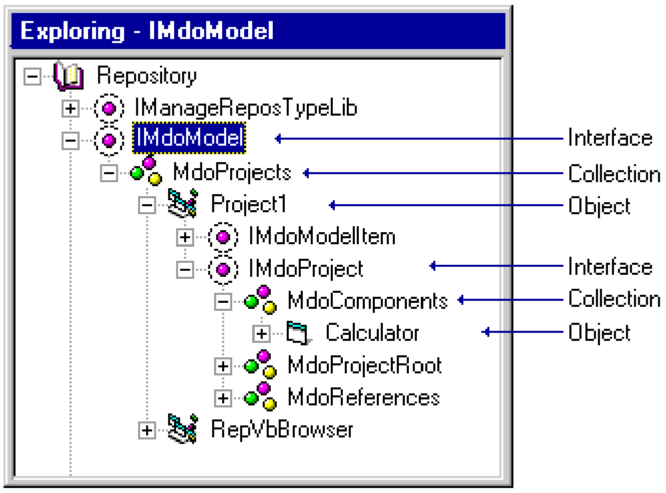


Figure 2. *Browsing Repository content*

The Repository Browser visualizes Repository data in a hierarchical fashion, and every Repository has a unique root object, which is where the Repository Browser starts its browsing. All objects are normally related to the Repository root, either directly, or indirectly through other objects and relationships, which thus makes it possible to navigate the Repository hierarchically. Figure 2 illustrates an example, where the book icon named "Repository" is the root object.

A Repository object is an instance of a class that has been defined in a Repository information model (within the OIM). A class implements one or more interfaces, and a Repository object thus provides one or more interfaces as defined by the class to which it conforms. A Repository object is encapsulated and is only accessible through its interfaces. For example, the "Repository" root object in figure 2 has two interfaces "IManageReposTypeLib" and "IMdoModel".

Repository data may be thought of as having a set of repeating patterns (see section 6 for further details):

- Repository objects have interfaces
- Interfaces are composed of properties, methods, and collections
- Collections contain relationships relating Repository objects (or actually object interfaces)

The Repository Browser displays Repository data by iterating through all of the interfaces attached to the root object, then iterating through the collections attached to those interfaces, and so on. For example, in figure 2 then the interface IMdoModel has a collection MdoProjects that contains the objects Project1 and RepVbBrowser (which are both instances of a class MdoProject, but this class and these object-class relationships are not seen here). The object Project1 has two interfaces IMdoModelItem and IMdoProject, that each may have yet other collections, and so on.

A *Repository collection* is itself a Repository object containing a set of *Repository relationship objects* (i.e., a collection is a collection of relationship objects). A relationship (object) is a logical connection between two Repository objects. For each relationship (object) between two Repository objects, one is assigned as the *Origin* and the other as the *Destination* for this relationship (this independent of how they are navigated - in which case the "from" object is called the *"source"* and the "to" object the *"target"*, but this independent of which object is the origin and which is the destination). In general, an object above in the tree-view control (the left window in the browser) is the origin in a relationship with the object(s) below as destination objects in this relationship. For example, the "Repository" root object is the origin object in a relationship with the Project1 object, as the destination object, via a relationship object in the MdoProjects collection. The root object is similarly related to the RepVbBrowser object via another relationship object in the same MdoProjects collection.

The Repository Browser can view two kinds of properties (and methods); properties that are common to all Repository objects, and properties that are specific to a particular class of Repository objects. The properties that are common to all Repository objects are displayed in a special tabbed dialog box. Properties that are specific to a particular class of objects are represented as members of a particular Members collection of the interfaces that are implemented by the object.

The definition of a Repository object may be extended by adding interfaces to its corresponding class, or by adding properties, methods, or collections to an existing interface.

Presentation Filter

The information that is presented by the browser can be customised by selecting the *"View|Filters"* menu item. In the *"Visual"* tab select *"Objects, Interfaces and Collections"* as these are the key elements when browsing the content of a Repository.

In the *"Type"* tab check *"Interfaces"* to view them, and uncheck *"Empty Relationship Collections"* to reduce the amount of information presented.

By default the browser illustrates relationships such that the Origin object is above the Destination object in the tree-view control. Check *"Destination->Origin Relationships"* if you want to navigate along object relationships both ways be expanding tree-view nodes; i.e., the tree-view is a tree structure, but if object A as the origin is related to object B as the destination, then B is below A in the tree-view control, but then A will also exist below B further down.

Browser Services

Beside browsing the content of a Repository, the browser also offers services for

- Creating a new Repository (i.e., database schemas for recording Repository information)
- Installing different model subsets of the overall OIM
- Exporting or importing Repository information formatted in XML (XIF - see section 7)

4. The Object Information Model - OIM

OIM Domain Models

An *information model (IM)* specifies the structure and semantics of information. In database terminology, an IM is a schema. For example, it may consist of the tables, columns, keys, etc., in a relational database. In logical database design, an IM is often expressed as an entity-relationship model, consisting of entities, attributes of the entities, and relationships between the entities. In object-oriented systems, an IM is an object model, consisting of classes, interfaces, members, properties, etc.

An IM is a *meta model*, and a language in which IMs are expressed is called a *meta meta model*. Examples of meta meta models are SQL data definition language, entity-relationship diagrams, interface definition language (IDL), and COM type libraries. Each persistent store and data exchange protocol has its own meta meta model for describing data that it stores or transfers.

An IM can be used to describe data in a persistent store. It tells applications how to interpret data that conforms to the model, that is, how the data is structured, what constraints must be honored, and how the data can be used. This enables applications to share data, by telling applications what they can count on from data supplied by other applications. An IM can also be used to describe data that is exchanged between applications.

The organization of Repository information is based on the *Open Information Model (OIM)*, which is a set of meta data models that facilitate sharing and reuse. OIM v.1.0 is huge, containing about 250 interfaces and about 100 relationship types. It is described in UML and grouped into subject areas addressing domains of varying complexity and generality. Based on an abstract core model, additional subject areas are added over time. The current OIM v.1.0 targets three main areas:

- Object-oriented Analysis and Design
- Component Description and Specification
- Database Schema and Data Warehousing

and it is composed of the following subject areas:

Table 1. Open Information Model subject areas

RTIM	Repository Type Information Model. This is the meta meta model – the language in which the other information models (or meta models) are defined.
Uml	Unified Modeling Language Model. This information model represents the conceptual UML v.1.0 model.
Umx	Uml Extension Model. A set of generic extensions to the Uml model, and a realization of some of the predefined stereotypes in UML v.1.0.
Dtm	Data Type Model. A set of Uml model extensions that supply numerous common data types.
Cde	Component Description Model. A set of generic component-related Uml model extensions describing run-time (executable) components and their specifications.
Com	COM (Component Object Model) Model. A set of COM-specific extensions to the Cde and Dtm models.
Gen	Generic Model. A set of general-purpose interfaces, relevant across diverse information models.
Dbm	Database Model. A set of extensions to the Uml model covering generic database concepts.
Sql	SQL Server Model. A set of SQL Server-specific extensions to the Dbm model.
Ocl	Oracle Model. A set of Oracle-specific extensions to the Dbm model.
Db2	DB2 Model. A set of DB2-specific extensions to the Dbm model.
Ifx	Informix Model. A set of Informix-specific extensions to the Dbm model.
Tfm	Database Transformation Model. Describes transformations for moving data between databases.
Olp	OLAP Model. An extension of the Dbm model describing multidimensional databases.
Sim	Semantic Model. An extension of the Dbm model that enables users to interact

	with database data without learning data manipulation languages.

Each subject area includes a Visual Modeler/Rational Rose model file, specifications, an installation file, a type library, a database, header files, test scripts, and each subject area has its own documentation. You must install the subject areas of the OIM in the sequence defined by the following dependencies between the individual submodels.

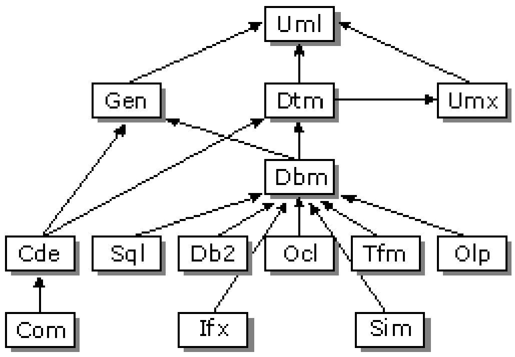


Figure 3. *Subject area dependencies within the OIM*

Figure 3 summarizes the dependencies among the OIM submodels; e.g. the *Dbm* model depends on the *Uml*, *Umx*, *Gen*, and *Dtm* models, and is depended upon by the *Sql*, *Ocl*, *Db2*, *Tfm*, and *Olp* models. These dependencies are derived from dependencies between the interfaces and classes within those models. That is, if there is a dependency between a pair of models, then the interfaces in one model may inherit (or imply) interfaces in the other, or the classes in one model may support interfaces in the other. Figure 3 provides a simplified view of these dependencies by depicting the inheritance between models. Each model can depend in several ways on a parent model. However, only the main dependency for each model is shown. The actual fine-grained dependencies can be determined from the full specification details of each model.

RTIM - The Repository Type Information Model

The *Repository Type Information Model (RTIM)* is an information model that the Repository uses to understand all other information models, and all OIM information models are defined by RTIM; i.e., methods and properties exposed by RTIM are also used to define new information models in the Repository. RTIM itself is automatically stored in the Repository database when a Repository is created (see section 5 for information on Repository creation).

Section 6 describes RTIM in further detail.

The Uml Model in OIM

There are differences between the Repository OIM Uml Information Model and the UML v.1.0 conceptual model, and in this section Uml (lowercase) refers to the Uml information model, whereas UML (uppercase) refers to the UML v.1.0 conceptual model.

The Uml model is an implementation of the conceptual UML v.1.0 model in a specific modeling language, the RTIM, and its role in OIM is twofold. Uml serves first as model for meta data of the analysis and design domain and second as the OIM's core model from which all the other sub-models inherit. Generic modeling features, such as type, containment, dependency, and naming, are all defined by Uml and are subsequently inherited by other subject areas. Each subject area either reuses or refines these core concepts, and thereby avoids redundancy and enables generic access.

The following are some differences between Uml and UML:

- Uml is a *subset* of UML. The Uml model includes the core, structural, and view aspects of UML; it does not include the behavioral aspects, namely the packages for collaborations, state machines, and interactions.
- *Extensibility*. UML supports extensibility mechanisms in the form of stereotypes, tagged values, and constraints. Predefined instances of these elements are provided as Standard Elements in each UML package. Microsoft Repository uses an extensibility scheme based on interface inheritance, as defined in the RTIM model. Although the Uml model includes the definition of stereotype, tagged value, and constraint, the predefined instances of these (as defined in the UML Standard Elements package) are realized with specific repository interfaces. For example, the stereotype <<interface>> of UML Type is realized as the interface ICdeInterface (in the Cde model). This interface, which derives from IUmlType, and the stereotype <<library>> of UML Component, is realized as IUmxLibrary (in the Umx model). IUmxLibrary inherits from IUmlComponent.
- *Modeling Variations*. Since Uml defines the same concepts as UML, yet uses a different modeling language (RTIM), there are some areas where slight modeling variations from UML occur. However, the semantics remain the same. For the most part, the Uml model faithfully follows UML.
  The most important areas of variation are:
- *Association Types* (UML calls these Association Classes.) These are usually realized in Uml as explicit interfaces. For example, UML Generalization is realized as the interface IUmlGeneralization. In cases where the association has a role with cardinality 1 or 0..1, any characteristics of the association are moved onto the interface definition referenced by the inverse role. For example, in UML Package Owns Element, Owns has a property Visibility that is moved onto IUmlElement, and there is no interface definition for Owns.
- *Composition Relationships*. UML Composition (filled-diamond symbol) is a strong form of aggregation. This means the associated items have a related lifetime — they can be created independently, but they are deleted together. This is represented in Microsoft Repository by delete propagation semantics on the appropriate relationship collection.
- *Derived Semantics*. These are not currently supported by Microsoft Repository. In particular, UML-derived associations are not modeled as Repository relationships.

The Dbm Model in OIM

The *Database Information Model (Dbm)* describes meta-data about data sources like e.g. relational database schema information.

Dbm covers the basic elements of SQL data provider, such as tables, columns and relationships. It does not address physical or implementation details. A goal for the Dbm is to provide enough descriptive information to allow any schema stored in a Repository to be re-generated in such a manner that the re-generated schema is indistinguishable from the original schema.

The Dtm Model in OIM

The *Data Type Information Model (Dtm)* is defined as a set of extensions to the *UML Information Model (Uml)*. It provides a set of interfaces for describing data types, and the intent is that this model can be extended and specialized by information models for particular domains.

Custom Made Extensions to the OIM

The extendability of the OIM is essential since meta information models evolve over time, both with respect to new objects and relationships, and also to include entire new domains. The Repository SDK has tools for doing this, and it also includes some examples, but I have not tested this myself.

First, define an information model in Rational Rose/Visual Modeler, save it in a Repository, and then use the SDK tools *Model Checker, Model Generator* and *Documentation Generator*. Model Checker identifies structural errors in the model relative to being a Repository OIM extension. The Model Generator produces a model installation script. The Document Generator produces a document in RTF which defines all interfaces, relationships, and classes in the information model.

The repository engine automatically creates the database tables required to store instances of the information model. After an information model is installed, the repository offers operations for creating objects that are instances of the model's types, and for storing and retrieving these objects' properties and relationships to and from the repository database.

Each information model normally adds an interface either to the Repository root object, or to some other object that is ultimately connected to the Repository root object. The added interface is then defined to contain one or more collections that relate to other objects in that information model.

A new option has been added to Visual Modeler/Rational Rose that enables you to create XML DTD documents from an information model. To do this, select the XML DTD check box in the Repository SDK-Distributable Files dialog box.

5. The Repository Database

Repository Database Schemas

Repository information is stored in a dedicated Repository database, e.g. an SQL Server or Access RDBMS. The Repository database is created and maintained as any other application specific database. Its schema consists of a *standard schema*, and possibly a set of *extended schemas*. The standard schema consists of tables that contain the core information needed to manage all Repository objects, relationships, collections, etc. The standard schema also contains tables that are used by the Repository to store the definition of its information models themselves.

An extended schema exists for each of the OIM subject areas described above. The Repository SDK can be used to generate schema extensions when an information model in the OIM is created or extended. Normally, one table is created for each interface that is defined in the information model. The table contains the instance data for persistent properties that are attached to the interface. One column in the table is created for each property. If an interface is defined without any member properties, then no table is created.

The set of SQL tables that comprise the standard schema are named *"RTbl*"*, while the set of SQL tables that comprise extended schemas are named *"Uml*"*, *"Dbm*"*, *"Dtm*"*, and so on.

Creating a Repository = Installing the Standard Schema

In order for a Repository engine to use a particular database as a Repository database, the Repository standard schema must be installed in the database. That is, to create an SQL Server Repository requires first to create an SQL Server database, as any other database, and then to install the Repository standard schema. To do this for an SQL Server database, first create an ODBC connection for the database by using *"ODBC Data Sources"* in the Control Panel (remember to change the "default database" to this SQL Server database). Then select *"File | New Repository | ODBC Database"* in the Browser menu, and enter the ODBC DSN for this database.

After installation the database will contain a number of tables (currently 15) named *"RTbl*"*, and it will be ready to serve as a Repository for information organized according to the RTIM (Repository Type Information Model); see section 6 below.

Installing Extended Schemas

An export to Repository operation fails if the Repository database does not have the required extended database schemas installed. These schemas can be installed from the Repository Browser, by selecting the *"File | Install Model File"* menu item, and then specify a particular *"*.rdm"* file. There exists a particular *"*.rdm"* file for each extended schema, e.g. *"Uml.rdm"*, and these files can be found in the Repository SDK catalog *".../oim/install/*.rdm"*.

It is also possible to install a particular schema into a Repository database programmatically from e.g. Visual Basic or Visual C++. In the folder *"\Program Files\Common Files\Microsoft Shared\Repostry"* there is a dll *"Insrepim.dll"* with a component offering a dual interface *IMInstall* with a single function *installRDM* for installing extended schemas:

```
installRDM(<DSN:string>,<RDMFile:string>,<UserName:string>,<Password:string>)
```

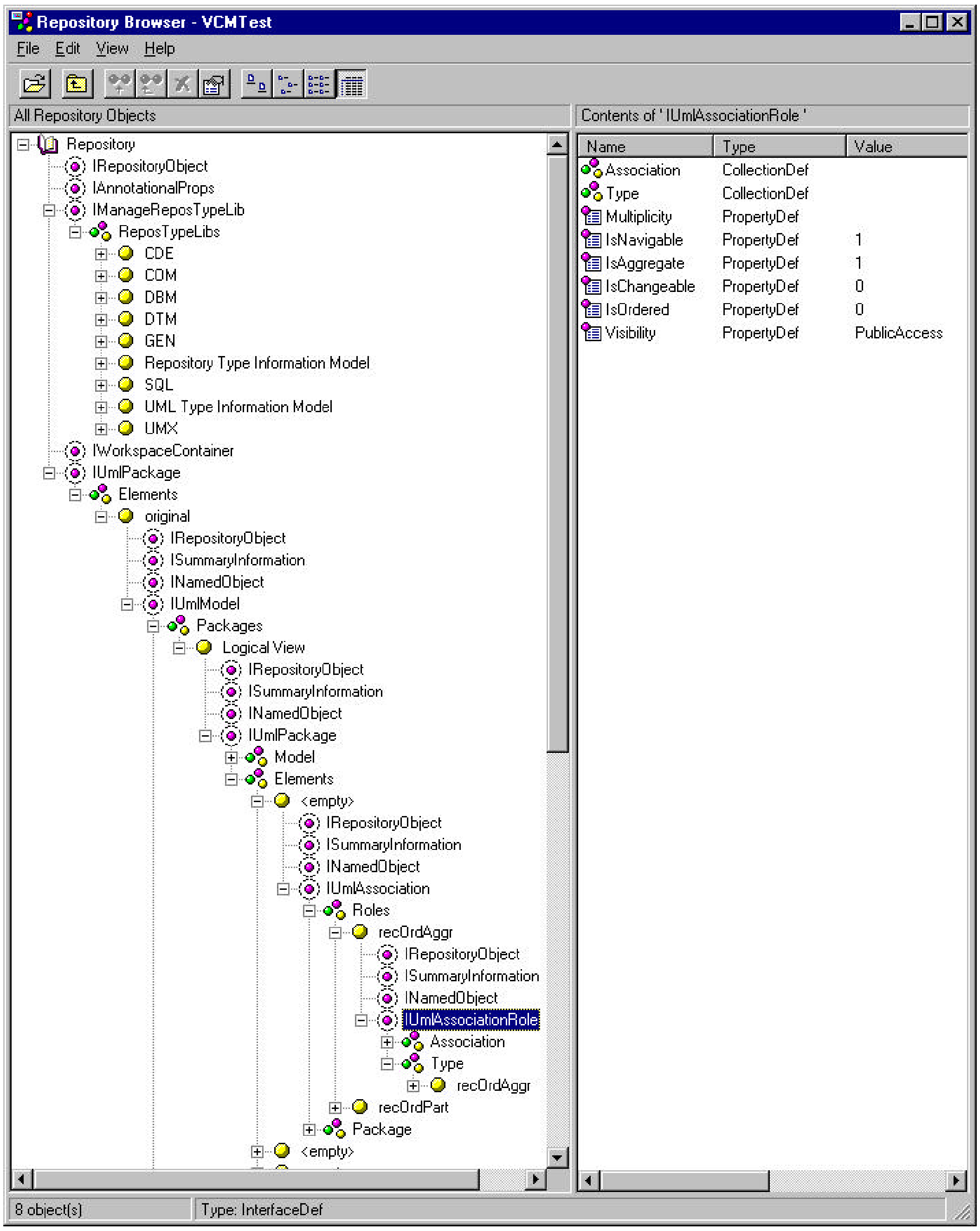
Read but no Write to a Repository Database

You can construct an SQL query to extract specific information from a Repository database. That is, the database can be queried directly with SQL queries, but an application should never update the Repository's database tables directly with SQL statements, since the Repository engine's update methods maintain the integrity of the database in subtle ways that a user could easily miss.

Transaction Support

When programming against the Repository (see section 6 for more information on this), transactions are supported. The following is an example in Visual Basic:

```
Dim refRepository As Repository
.....
refRepository.Transaction.Begin
.....<transaction body>.....
refRepository.Transaction.Commit (or .Abort)
```



Repository Browser - VCMTest

File Edit View Help

All Repository Objects

```
Repository
    IRepositoryObject
    IAnnotationalProps
    IManageReposTypeLib
        ReposTypeLibs
            CDE
            COM
            DBM
            DTM
            GEN
            Repository Type Information Model
            SQL
            UML Type Information Model
            UMX
    IWorkspaceContainer
    IUmlPackage
        Elements
            original
                IRepositoryObject
                ISummaryInformation
                INamedObject
                IUmlModel
                    Packages
                        Logical View
                            IRepositoryObject
                            ISummaryInformation
                            INamedObject
                            IUmlPackage
                                Model
                                Elements
                                    <empty>
                                        IRepositoryObject
                                        ISummaryInformation
                                        INamedObject
                                        IUmlAssociation
                                            Roles
                                                recOrdAggr
                                                    IRepositoryObject
                                                    ISummaryInformation
                                                    INamedObject
                                                    IUmlAssociationRole
                                                        Association
                                                        Type
                                                            recOrdAggr
                                                recOrdPart
                                            Package
                                    <empty>
```

Contents of ' IUmlAssociationRole '

Name	Type	Value
Association	CollectionDef	
Type	CollectionDef	
Multiplicity	PropertyDef	
IsNavigable	PropertyDef	1
IsAggregate	PropertyDef	1
IsChangeable	PropertyDef	0
IsOrdered	PropertyDef	0
Visibility	PropertyDef	PublicAccess

8 object(s) Type: InterfaceDef

Figure 4. *Objects, interfaces, properties, collections and relationships in a Repository*

Repository Performance

A Repository database can be tuned to optimize performance and data retrieval, and new indexes can be added to tables. The implementation of Repository database operations are also optimized. For example, the engine generates and later calls information model specific stored procedures to avoid run-time compilation of SQL. It also maintains a cache of objects in the application's process, so that most retrieval operations can be serviced without accessing the database.

The article *"Performance Hints for Using Microsoft Repository"*, available at *"http://msdn.microsoft.com/repository/technical"*, should be considered before extending the Repository with new information models and database objects.

6. The Repository Type Information Model (RTIM)

RTIM - The basic foundation for the Repository Information Models

The *RTIM (Repository Type Information Model)* is the basic information model in any Repository. Every Repository information model consists of a set of type definitions, namely classes, relationship types, interfaces, property and methods definitions, and collection types. The Repository engine provides methods for creating objects that are instances of these classes, and for storing and retrieving these objects' properties and relationships to and from a Repository database.

Repository type definitions are grouped into *Repository Type Libraries* (which have the same logical structure and namespace behaviour as *Automation Type Libraries*). This type library abstraction provides a mechanism for partitioning large information models, and different type libraries are used for the type definitions of particular OIM subject areas. The Browser snapshot in figure 4 illustrates this. The Repository root object (the book icon) has an interface *IManageReposTypeLib,* which has a collection *ReposTypeLibs* containing one type library for each of the OIM subject areas that are currently installed in this Repository database (in this case Cde, Com, Dbm, Dtm, Gen, RTIM, Sql, Uml and Umx).

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

- Class - defines the class ID, name, and which interfaces it supports.
- Relationship class - defines which collections (on which interfaces) are connected by instances of the relationship class.
- Interface - defines the interface ID, which properties, methods and collections are members of this interface, and which interface it inherits from (properties, methods and collections are inherited). Interfaces are supported by classes, and they contain properties, methods and collections.
- Property - defines properties of the property, amongst others its data type.
- Method - defines properties of the method, such as its dispatch ID.
- Collection - defines properties of the collection, such as min and max cardinality. These are properties of endpoints(roles) of a relationship type.

Repository type definitions are themselves ordinary Repository objects (see below) that have certain type-specific 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 that it implements. Like all Repository objects they are instances of classes, which in turn support interfaces that have properties and relationships stored in the Repository. Hence classes, relationship classes, interfaces, properties, methods and collections are instances of the classes *ClassDef*, *RelationshipDef*, *InterfaceDef*, *PropertyDef*, *MethodDef* and *CollectionDef,* respectively (and these classes are described as instances of themselves, i.e., all these classes are instances of *ClassDef,* and their relationships are instances of *RelationshipDef*, and so on).

Repository Objects

When programming against the RTIM you will be working with the following kinds of Repository objects (each offering different kinds of interfaces) supported by the Repository engine.

Repository

A Repository is accessed via, and programmatically represented by, a *Repository object*; "Dim refRepository As Repository" in Visual Basic.

Transactions are supported via this object since a *RepositoryTransaction object* cannot be instantiated directly. When you connect to a Repository, a RepositoryTransaction object is created for you, and made accessible via the *Repository.Transaction* property.

RepositoryObject

A *RepositoryObject object* is a Repository object that represents an instance of a Repository object defined as a class in a Repository information model.

Notice that a "RepositoryObject object" is a "Repository object" in line with Relationship objects, Collections, and so on.

Relationship

A Relationship object represents a connection between two RepositoryObjects. That is, a Relationship has an *origin* RepositoryObject, a *destination* RepositoryObject, and a set of properties. Each Relationship conforms to a particular Relationship type, i.e., similar to how a RepositoryObject is considered an instance of a particular class. Relationship objects are used to manipulate the properties of a relationship, to delete a relationship, or to refresh the cached image of a relationship.

Notice that a Relationship between two objects is actually a relationship between interfaces of these two objects. This is a key point in working with Repository data: *the Repository contains collections of relationships between interfaces of objects*. The example below may make this more clear.

Property (ReposProperty)

A *Property object* (called *ReposProperty*) is a persistent single-valued property or collection that is attached to a Repository object. It is used to retrieve the name, type or value of a property, or to set the value of a property.

Collection (ObjectCol/RelationshipCol)

These are some different kinds of collections in a Repository:

- A *RelationshipCol object* (called a *relationship collection*) represents a collection of Relationship objects for the relationships from a single particular RepositoryObject, called the *source*, to a set of one or more related RepositoryObjects, called the *targets*.
  Notice that all of the Relationships in a relationship collection must conform to the same relationship type.
- An *ObjectCol object* represents a collection of RepositoryObjects that conforms to a particular class (the instances of this class) or that supports a particular interface in the Repository.
- A *Properties collection object* represents a collection of ReposProperties attached to a RepositoryObject or Relationship via a particular interface.

Workspace

A *workspace* is a repository object that can provide a context for your work separate from other work occurring in the Repository

Object Identification

Every Repository object has an *internal ID* which uniquely identifies this object within a particular Repository database, but it is not necessarily unique to all Repositories. Every Repository object has also an *external object ID* which uniquely identifies this object across all Repositories.

Versioning

In Repository v.2.0, RepositoryObject and Relationship objects can be versioned. A RepositoryObject version is a particular rendition of a RepositoryObject object. Each version of an object can differ from other versions of that object in its property values and collections. When you manipulate a RepositoryObject programmaticaly you are actually manipulating a particular version of that object, i.e., you manipulate a *RepositoryObjectVersion* object.

Similarly, every Relationship object is a *VersionedRelationship* object that can connect a particular version of a RepositoryObject to one or more specific versions of its related RepositoryObjects.

Example

Figure 5 illustrates how Repository meta-information, e.g. in a new OIM subject area concerning files and directories, can be defined in RTIM and its data represented by the above Repository objects.

In the RTIM information model there are two classes, *File* and *Directory*, and the following three interfaces:

- *IFile* exposes behavior unique to files. Thus, only the *File* class implements it.
- *IDirectory* exposes behavior unique to directories. Thus, only the *Directory* class implements it.
- *IDirectoryItem* exposes behavior appropriate to any object that can appear as an item within a directory. Since this is the case for both *File* and *Directory* objects, both the *File* and the *Directory* class implements this interface.

The *IFile* interface exposes one property; the *Size* property. The *IDirectoryItem* interface exposes one property; the *ModificationDate* property. The *IDirectory* interface exposes one property; the *ChildCount* property. None of the interfaces exposes any methods.

There is one relationship type, *Containment*, and there are two collection types in association with this containment relationship:

- Collections that conform to the *ItemsOfDirectory* collection type are origin collections for *Containment* relationships. The *IDirectory* interface exposes this collection.
- Collections that conform to the *DirectoryOfItem* collection type are destination collections for *Containment* relationships. The *IDirectoryItem* interface exposes this collection.

Actual data from this model will be represented as follows in a Repository. *Directory* and *File* objects will be represented by *RepositoryObjects*, supporting the interfaces *IDirectory* and *IDirectoryItem*, and *IFile* and *IDirectoryItem*, respectively. *ChildCount*, *ModificationDate* and *Size* are represented by *ReposProperties*. *ItemsOfDirectory* and *DirectoryOfItems* are relationship collections represented by *RelationshipCol* objects.
Finally, *Containment* relationships are represented by *Relationship* objects.

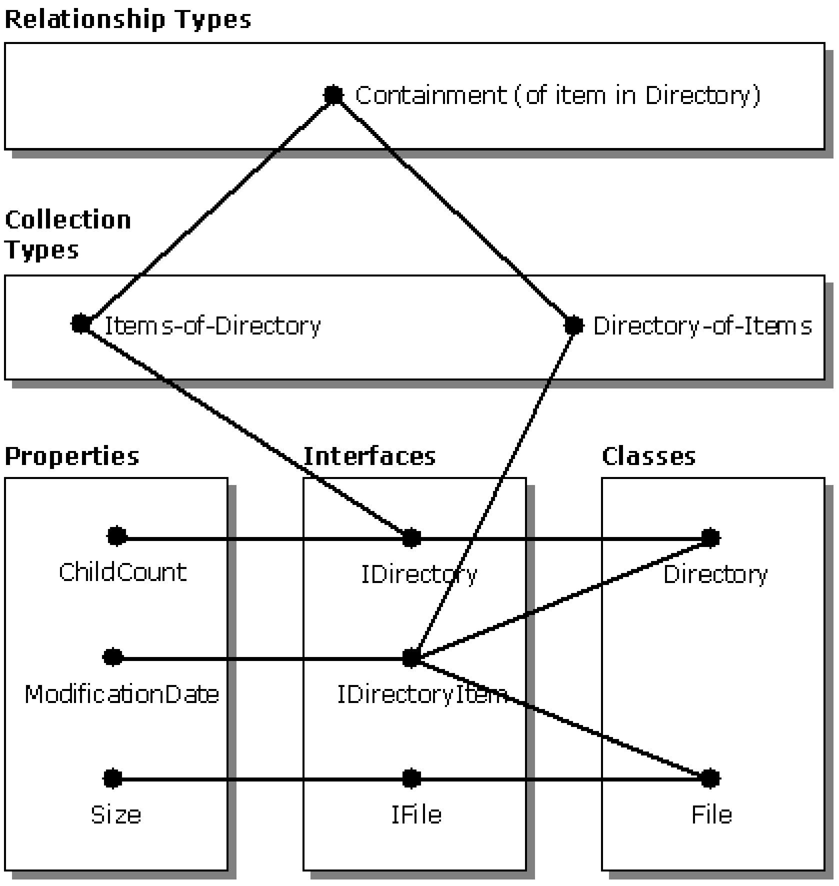


Figure 5. *How information is organized according to RTIM*

Basic RTIM Programming against the Repository

To program with the basic RTIM interfaces of a Repository use the *"repodbc.dll"* located in the *"\Program Files\Common Files\Microsoft Shared\Repostry"* catalog. In Visual Basic this corresponds to making a reference to *"Microsoft Repository"*. To run the example below include also the file *"...\Microsoft Repository SDK\include\RepEng.bas"* from the Repository SDK into the Visual Basic project. *"RepEng.bas"* contains object ID's for a number of predefined RTIM objects.

The following Visual Basic example is an adaption of the *"TreeWalk"* sample accompanying the Repository SDK. Starting from a particular Repository object (objectID "{{AFEA9392-9B2C-11D3-AA33-0060973166C2}, 00000001}" - copied from the Repository Browser), its interfaces with their properties and collections are traversed, and the traversal continues with the objects that are related to this object via relationships in its interface collections. Notice that the *IRepositoryItem* interface is common to both RepositoryObject objects and Relationship objects.

```
' This example uses the Repository Type Information Model (TIM) to navigate
' relationships between objects in the repository. Note that while
' only Origin->Destination relationships are navigated, this does not ensure
' that the cyclic relationships will not be found.

Option Explicit

Private Repos As Repository

Sub Main()
  Dim Root As RepositoryObject

  Set Repos = New Repository
  ' Initialize some Repository constants for VB - a method in RepEng.bas
  Call InitRepository

  ' Open a repository
  Set Root = Repos.Open("SERVER=OSLPCM55;DATABASE=VCMTest", "sa", "")

  ' Starting point - the Repository.Object function returns a an object
  ' with a particular object ID. The objectID can be copied from the Browser
  ' (right mouse button in right browser window.
  Set Root = Repos.object("{{AFEA9392-9B2C-11D3-AA33-0060973166C2},00000001}")

  ' Navigate the repository from this object
  Debug.Print "Beginning Repository Walk"
  Call ReposTree(Root, "")
  Debug.Print "Finished Repository Walk"
End Sub

' Navigate the specified repository object, recursively traversing
' origin->destination relationships.
Sub ReposTree(Obj As IRepositoryItem, indent As String)
  Dim Relship As Relationship
  Dim ObjType As RepositoryObject
  Dim ObjProp As ReposProperty
  Dim ObjPropType As RepositoryObject
  Dim ObjTarget As RepositoryObject
  Dim Iface As RepositoryObject

  ' Print the object's name
  Call PrintName(Obj, indent)

  ' Get the object's type definition (Obj.Type = class ID) and output the name.
  Set ObjType = Repos.object(Obj.Type)
  Debug.Print indent & "Class: " & ObjType.Name

  ' Find the interfaces supported by Obj.
  ' ObjType is the class of Obj, and thus supports "IClassDef" from which the
  ' Interfaces function can be used to find the interfaces supported by this
  ' class and thus its instances.
  For Each Iface In ObjType.Interface("IClassDef").Interfaces
      ' Print the interface name.
      Debug.Print indent & "Interface: " & Iface.Name

      ' Disregard annotational properties
      If Iface.Name = "IAnnotationalProps" Then GoTo NextIface

      ' Disregard Repository Type Libraries
      If Iface.Name = "IManageReposTypeLib" Then GoTo NextIface

      ' Get the list of properties and collections of the interface Iface
      ' supported by Obj (ObjProp is a ReposProperty which can be a
      ' single valued property or a collection
      For Each ObjProp In Obj.Interface(Iface.Name).Properties

          ' Get the type definition of ObjProp
          Set ObjPropType = Repos.object(ObjProp.Type)

          ' Check if ObjProp is a property or collection
          If SameOBJID(ObjPropType.Type, OBJID_PropertyDef) Then
              ' Print the property name
              Debug.Print indent & "  Property: " & ObjProp.Name & " = ";

              ' Print property value
              If IsNull(ObjProp.Value) Then
                Debug.Print "Null"
              ElseIf IsArray(ObjProp.Value) Then
                Debug.Print "(Array)"
              ElseIf IsEmpty(ObjProp.Value) Then
```

```
                  Debug.Print "(Empty)"
                Else
                  Debug.Print ObjProp.Value
            End If

            ' Check if ObjProp is a collection
            ElseIf SameOBJID(ObjPropType.Type, OBJID_CollectionDef) Then
                ' If this object is the origin of this relationship
                If ObjPropType.Interface("ICollectionDef").IsOrigin <> 0 Then

                   ' Print the collection name
                   Debug.Print indent & "  Collection: "; ObjProp.Name

                   ' Enumerate the relationships in this collection for the object
                   For Each ObjTarget In ObjProp.Value
                      ' Traverse target objects.
                      Call ReposTree(ObjTarget, indent & "  ")
                   Next
                End If
            End If
        Next
NextIface:
  Next
End Sub

' Compare the two object IDs. ObjIDs are defined as an array of 5 longs,
' which represents the 16 single byte values in a contiguous GUID variant
' plus a 4 byte unique value.
Private Function SameOBJID(ID1 As Variant, ID2 As Variant) As Boolean
    Dim i As Integer

    For i = LBound(ID1) To UBound(ID1)
        If ID1(i) <> ID2(i) Then
            SameOBJID = False
            Exit Function
        End If
    Next
    SameOBJID = True
End Function

Private Sub PrintName(Obj As IRepositoryItem, indent As String)
    On Error GoTo NoName
    Debug.Print indent & "Object: "; Obj.Name
    Exit Sub
NoName:
    Debug.Print indent & "Object: Unnamed"
End Sub
```

Programming against specific Subject Areas in the Repository

Each OIM subject area information model contains an interface model and a class model. All interface models define a set of interfaces through which clients can manipulate Repository objects. All class models define a set of classes that *may* be used to create Repository objects. These are not a standard, and many clients using a Repository will typically define their own class models (and additional interface models). Normally, only the client that creates an object needs to know its class. All other clients manipulating the object will only be concerned with the interfaces it supports. That is, clients should not make assumptions about the classes used by other clients to create objects. The classes provided are intended as a basic starting point and as an example of how to combine interfaces in useful ways.

Repository files for programming in Visual Basic or Visual C++

Automation type libraries for programming with various subject area interfaces are located in the Repository SDK catalog *"\oim\TypeLib"*; e.g. *Uml.tlb, Dbm.tlb, Sql.tlb,* etc.

When programming with Visual Basic the catalog *"\oim\VBConst"* contains object ID's for a number of predefined Repository objects relevant to a particular subject area; e.g. *Umlconst.bas, Dbmconst.bas, Sqlconst.bas*, etc.

When programming with VC++ the catalog *"/oim/CHeader"* similarly contains .h files, and the catalog *"/oim/idl"* contains IDL files for the various subject areas.

An example of programming with the Uml Interfaces

The following Visual Basic example requires the following references to be made:

- Microsoft Repository (\Program Files\Common Files\Microsoft Shared\Repostry\repodbc.dll)
- Unified Modeling Language Information Model (\Microsoft Repository SDK\oim\TypeLib\Uml.tlb)

and the following files must be included into the project:

- RepEng.bas (\Microsoft Repository SDK\include catalog)
- Umlconst.bas (\Microsoft Repository SDK\oim\VBConst catalog)

Notice that all Repository objects for Rose model elements support the *INamedObject* interface, and this is the only place to find the name of model elements. It can be used as follows from Visual Basic:

```
Dim refName As INamedObject
Set refName = ....some Repository object....
refName.Properties.Item("name")
```

Example code (it does nothing meaningful - just for demonstration purposes):

```
Dim Repos As Repository
Dim root As RepositoryObject
Dim repObj As RepositoryObject
Dim repObj2 As RepositoryObject
Dim refName As INamedObject
Dim UMLTypeLib As RepositoryObject

Dim refModel As UML.IUmlModel
Dim refPackage As UML.IUmlPackage
Dim refClass As UML.IUmlClass
Dim refAssos As UML.IUmlAssociation
Dim refRole As UML.IUmlAssociationRole
Dim refAttribute As UML.IUmlAttribute
Dim refOperation As UML.IUmlOperation
Dim refPar As UML.IUmlParameter

Dim i As Integer

' Initialize some Repository constants for VB - method in RepEng.bas
Call InitRepository

' Open a repository.
Set Repos = New Repository
Set root = Repos.Open("SERVER=OSLPCM55;DATABASE=VCMTest", "sa", "")

Set UMLTypeLib = GetTypeLib(root, "UML Type Information Model")

Debug.Print "------------------- NEW SESSION ------------------------------------"

' Find the Repository object of a particular Rose model and print its name
```

```
Set refModel = Repos.object("{{9B6271E3-A0C0-11D3-AA34-0060973166C2},00000001}")
Call PrintName(refModel, "")

' Find its first package and print its name (it is the Logical View in this case)
Set refPackage = refModel.Packages.Item(1)
Call PrintName(refPackage, "")

' Start a Repository transaction
Repos.Transaction.Begin

' Traverse all elements in the Package (they are either Associations or Classes)
For i = 1 To refPackage.Elements.Count
    Set repObj = refPackage.Elements.Item(i)
    Call PrintName(repObj, "")

    If (InterfaceSupported(repObj, "IUmlClass")) Then
        ' It is a class
        Debug.Print "Class"

        ' Make the class abstract
        Set refClass = repObj
        refClass.IsAbstract = True

        ' Add an extra attribute
        Set refAttribute = CreateReposObject(UMLTypeLib, "UmlAttribute")
        Set refName = refAttribute
        refName.Properties.Item("name") = "newAttribute"  ' Set its name
        refAttribute.TypeExpression = "long"              ' Set its type
        Call refClass.Members.Add(refAttribute)           ' Add it to the class

        ' Print the members, as attributes and/or operations of this class
        Debug.Print "     Members:"
        For Each repObj In refClass.Members
            Call PrintName(repObj, "")

            If (Repos.object(repObj.Type).Name = "UmlAttribute") Then
                Debug.Print "        Attribute"
                Set refAttribute = repObj
                Debug.Print "            Type=" & refAttribute.TypeExpression
            ElseIf (Repos.object(repObj.Type).Name = "UmlOperation") Then
                Debug.Print "        Operation"
                Set refOperation = repObj
                Debug.Print "            Operation parameters:"

                For Each repObj2 In refOperation.Parameters
                    Call PrintName(repObj2, "            ")
                    Set refPar = repObj2
                    Debug.Print "                Type: " & refPar.TypeExpression
                Next
            End If
        Next
    ElseIf (InterfaceSupported(repObj, "IUmlAssociation")) Then
        ' It is an association
        Debug.Print "Association"

        ' Traverse the two roles of this association
        Set refAssos = repObj
        For j = 1 To refAssos.Roles.Count
            Set refRole = refAssos.Roles.Item(j)
            Call PrintName(refRole, "     ")

            Set refName = refRole
            If (refName.Properties.Item("name") = "ra") Then
                ' Set the multiplicity of role "ra" and make it ordered
                refRole.Multiplicity = "*"
                refRole.IsOrdered = True
                Debug.Print "Role ra changed into ordered!"
            Else
```

```
                    ' Set the multiplicity of other roles beside "ra" to "2..7"
                    refRole.Multiplicity = "2..7"
                    Debug.Print "Multiplicity changed!"
                End If
            Next
        End If
    Next

    ' Commit the transaction
    Repos.Transaction.Commit

    ' Terminate repository
    Set Repos = Nothing

    ' Close file
    Debug.Print "------------------- END SESSION ------------------------------------"

' Find the specified repository type library in the repository. it will
' be located in a collection located off the repostiory root.
'
Private Function GetTypeLib(RootObject As RepositoryObject, _
                            TypeLibName As String) As RepositoryObject
    Dim ManageLib As RepositoryObject
    Set ManageLib = RootObject.Interface("IManageReposTypeLib")
    Set GetTypeLib = ManageLib("IManageReposTypeLib").ReposTypeLibs.Item(TypeLibName)
End Function

' Create the specified object instance in the repository. First, verify
' that the class exists by attempting to find the class definition in the
' type library, then create a new object accordingly.
'
Private Function CreateReposObject(TypeLib As RepositoryObject, _
                                   ClassName As String, _
                                   Optional ObjectId As Variant) As RepositoryObject
    Dim ClassDef As RepositoryObject
    Dim Repos As Repository

    Set ClassDef = TypeLib("IReposTypeLib").ReposTypeInfos.Item(ClassName)
    Set Repos = TypeLib.Repository

    If IsMissing(ObjectId) Then
      ObjectId = OBJID_NULL
    End If
    Set CreateReposObject = Repos.CreateObject(ClassDef.ObjectId, ObjectId)
End Function

' Returns "true" if the object supports the named interface
Private Function InterfaceSupported(objRep As RepositoryObject, _
                                    iface As String) As Boolean
    On Error GoTo Exception

    TypeName objRep.Interface(iface)
    InterfaceSupported = True
NormalExit:
    Exit Function
Exception:
    InterfaceSupported = False
End Function

' Compare the two object IDs. ObjIDs are defined as an array of 5 longs,
' which represents the 16 single byte values in a contiguous GUID variant
' plus a 4 byte unique value.
'
Private Function SameOBJID(ID1 As Variant, ID2 As Variant) As Boolean
    Dim i As Integer

    For i = LBound(ID1) To UBound(ID1)
        If ID1(i) <> ID2(i) Then
```

```
            SameOBJID = False
            Exit Function
        End If
    Next
    SameOBJID = True
End Function

' Print the name of a property
Private Sub PrintProperty(Obj As ReposProperty, indent As String)
  On Error GoTo NoName

  Debug.Print indent & "Property: " & Obj.Name
  Exit Sub
NoName:
    Debug.Print indent & "Property: Unnamed"
End Sub

' Print the name of an object
Private Sub PrintName(Obj As IRepositoryItem, indent As String)
  On Error GoTo NoName

  Debug.Print indent & "Object: " & Obj.Name
  Exit Sub
NoName:
    Debug.Print indent & "Object: Unnamed"
End Sub
```

7. XML for Exchanging Meta-Information

XIF - XML Interchange Format

The *XML Interchange Format (XIF)* is used as the exchange format for the Open Information Model (OIM). XIF is based on the *Extensible Makeup Language (XML) v.1.0*, as defined by the W3C, and it consists of a set of rules that govern the encoding of metadata objects described by OIM in XML format. XML interchange facilities described below can be used to move meta data between a Repository and other OIM-compliant systems.

Special care has been taken in XIF such that all instances of an information model are represented as content, references between objects as HREFs, and all model and support information as begin/end tags and attributes of tags. This ensures that XML browsers and applications are able to process the XML documents even if they do not understand the semantics expressed by the information model.

XML Namespaces provide a simple method to qualify names in XML documents. Namespaces are still in a draft state in the W3C. XIF makes use of namespaces, since without namespace modifications, an information model would have to be modified to have globally unique names, which is considered unacceptable.

XML Links provides concepts to describe links between objects in XML documents. Links are still in a draft state in the W3C, and XML Links are currently not used in XIF since their addition would not add more expressiveness or simplify the concepts. However, care has been taken to ensure that the specified format is compatible with current proposals.

The Meta Data Coalition (MDC) will support multiple XML vocabularies for the OIM in order to make it easy for different vendors and end-user to integrate the standard into their environments. By standardizing on the MDC OIM as semantic model for a metadata interchange, it is straightforward to convert between different XML vocabularies using technologies like the *Extensible Style Sheet Language (XSL)*, a standard mapping technology. The MDC will use XSL to provide mappings between the alternative XML vocabularies for the MDC OIM.

XML Export

The *XIF Exporter* is a utility that exports objects from a Repository by using the XIF encoding; i.e., it generates an XML representation of repository object instances based on XIF rules. Notice that using XIF to export objects from a Repository is a two-step process:

- Marking objects for export
- Generating the XML file

The export is handled by a COM component with the ProgID *MSRXML.XIFExport*, that has a single dual interface *IXIFExport*. First the client marks objects for export by using the *IXIFExport::Add* method to create a list of objects to be exported. The Add method eliminates all duplicate objects that may exist. The order in which objects are added determines the order in which the objects will appear in the XML document. After the collection has been created, a client can enumerate through this collection and get information, such as the number of objects, just like any normal collection.

Afterwards the client starts the export by invoking the *IXIFExport::ExportXML* method. The name of the file into which the XML document should be exported is passed as a parameter. The method will overwrite the file if it already exists. The client can specify flags that control the way in which objects are handled in the output.

The following Visual Basic code illustrates how the XIF Exporter can be used programmatically. Notice that the Visual Basic project has references to:

- Microsoft Repository (\Program Files\Common Files\Microsoft Shared\Repostry\repodbc.dll)
- MSRXML 1.0 Type Library (\Program Files\Common Files\Microsoft Shared\Repostry\msrxml.dll)

In this case just a single Repository object is exported, namely the object with the objectID "{{BD8C9483-8DEF-11D3-AA2B-0060973166C2},00000001}"

```
Dim Exporter As MSRXMLLib.XIFExport
Dim Repos As Repository
Dim repObj As RepositoryObject

Set Repos = New Repository
Set repObj = Repos.Open("SERVER=OSLPCM55;DATABASE=VCMTest", "sa", "")
Set repObj = Repos.object("{{BD8C9483-8DEF-11D3-AA2B-0060973166C2},00000001}")

Set Exporter = New MSRXMLLib.XIFExport
Call Exporter.Add(repObj)  ' Mark this object for export

Call Exporter.ExportXML("test1.xml",                           ' XML file name
                        "Repository XML genererт for Test1.mdl",   ' Description
                        "...more info...",                     ' Publisher
                        0)                                     ' Flags
```

Notice that when generating XML from the Repository Browser, as opposed to when doing it programmatically, then it is not necessary to explicitly mark every object for export. Instead it is possible to specify that every child node below a particular node in the Browser tree-view control shall be marked for export.

XML Import

The *XIF Importer* is a utility that can be used to import objects encoded in XIF format into a Repository OIM. The import process takes an XML document and converts it into OIM object instances in a Repository according to encoding rules defined by XIF. The XIF Importer uses the Microsoft XML parser to read the XML document. It makes use of a plug-in mechanism that the XML parser defines, which allows the parser client to plug in its own handler for the entities to be parsed (the custom NodeFactory mechanism).

The import is handled by a COM component with the ProgID *MSRXML.XIFImport*, that has a single dual interface *IXIFImport*.

PS: When testing the XIF Importer I have **not** actually been able to import XML into a Repository!

An Example of Generated XML

The XIF XML produced is quite voluminous, but it is outside the scope of this document to run performance tests to see how time-consuming it is to parse these XML files.

The following is an excerption of XML produced based on a small Rose model with a few classes and associations.

File outline:

```
<?xml version="1.0" encoding="windows-1252"?>
<xif:Transfer
       version="1.1"
       xmlns:xif="http://msdn.microsoft.com/repository/oim/xif.dtd"
       xmlns:UML="URN:REPOS:UML Type Information Model"
       xmlns:xif.SummaryInformation="URN:REPOS:Repository Type Information
Model/ISummaryInformation"
       xmlns:UML.Model="URN:REPOS:UML Type Information Model/IUmlModel"
       xmlns:UML.Package="URN:REPOS:UML Type Information Model/IUmlPackage"
       xmlns:UML.ModelElement="URN:REPOS:UML Type Information Model/IUmlModelElement"
       xmlns:UML.Element="URN:REPOS:UML Type Information Model/IUmlElement"
       xmlns:UML.AssociationRole="URN:REPOS:UML Type Information
Model/IUmlAssociationRole"
       xmlns:UML.Association="URN:REPOS:UML Type Information Model/IUmlAssociation"
       xmlns:UML.Relationship="URN:REPOS:UML Type Information Model/IUmlRelationship"
       xmlns:UML.Generalization="URN:REPOS:UML Type Information Model/IUmlGeneralization"
       xmlns:UML.GeneralizableElement="URN:REPOS:UML Type Information
Model/IUmlGeneralizableElement"
       xmlns:UML.Type="URN:REPOS:UML Type Information Model/IUmlType"
       xmlns:UML.Attribute="URN:REPOS:UML Type Information Model/IUmlAttribute"
       xmlns:UML.Operation="URN:REPOS:UML Type Information Model/IUmlOperation"
       xmlns:UML.Member="URN:REPOS:UML Type Information Model/IUmlMember"
       xmlns:UML.Parameter="URN:REPOS:UML Type Information Model/IUmlParameter"
       xmlns:UML.Class="URN:REPOS:UML Type Information Model/IUmlClass"
       xmlns:UML.ObjectType="URN:REPOS:UML Type Information Model/IUmlObjectType">
       <xif:TransferHeader>
              <xif:Exporter>MSRXML</xif:Exporter>
              <xif:ExporterVersion>1.0</xif:ExporterVersion>
              <xif:TransferDateTime>1999-10-23T11:34:11</xif:TransferDateTime>
       </xif:TransferHeader>

<UML:Model xif:id="_0" xif:objid="{{0DB06784-A18C-11D3-AA35-0060973166C2},00000001}">
<xif:Name>test6rep</xif:Name>
<UML.Model:Packages>
<UML:Package xif:id="_1" xif:objid="{{0DB06784-A18C-11D3-AA35-0060973166C2},00000003}">
<xif:Comments></xif:Comments>
<xif:Name>Logical View</xif:Name>
<UML.Package:Elements>
.....associations, classes and generalizations/specializations....
</UML.Package:Elements>
</UML:Package>
</UML.Model:Packages>
</UML:Model>
</xif:Transfer>
```

Association and its roles:

```
<UML:Association xif:id="_2" xif:objid="{{0DB06784-A18C-11D3-AA35-
0060973166C2},00000031}">
<xif:Comments></xif:Comments>
<xif:Name>assosAC</xif:Name>
<UML.Association:Roles>
<UML:AssociationRole xif:id="_3" xif:objid="{{0DB06784-A18C-11D3-AA35-
0060973166C2},00000033}">
<xif:Comments></xif:Comments>
<xif:Name>ra</xif:Name>
<UML.AssociationRole:Multiplicity>0..*</UML.AssociationRole:Multiplicity>
<UML.AssociationRole:IsNavigable>1</UML.AssociationRole:IsNavigable>
<UML.AssociationRole:IsAggregate>0</UML.AssociationRole:IsAggregate>
<UML.AssociationRole:IsChangeable>0</UML.AssociationRole:IsChangeable>
<UML.AssociationRole:IsOrdered>1</UML.AssociationRole:IsOrdered>
<UML.Element:Visibility>PublicAccess</UML.Element:Visibility>
</UML:AssociationRole>
<UML:AssociationRole xif:id="_4" xif:objid="{{0DB06784-A18C-11D3-AA35-
0060973166C2},00000035}">
<xif:Comments></xif:Comments>
<xif:Name>rc</xif:Name>
<UML.AssociationRole:Multiplicity>1..1</UML.AssociationRole:Multiplicity>
<UML.AssociationRole:IsNavigable>1</UML.AssociationRole:IsNavigable>
<UML.AssociationRole:IsAggregate>0</UML.AssociationRole:IsAggregate>
<UML.AssociationRole:IsChangeable>0</UML.AssociationRole:IsChangeable>
<UML.AssociationRole:IsOrdered>0</UML.AssociationRole:IsOrdered>
<UML.Element:Visibility>PublicAccess</UML.Element:Visibility>
</UML:AssociationRole>
</UML.Association:Roles>
</UML:Association>
```

Class, its roles and elements (attributes and operations/methods):

```
<UML:Class xif:id="_11" xif:objid="{{0DB06784-A18C-11D3-AA35-0060973166C2},00000005}">
<xif:Comments></xif:Comments>
<xif:Name>Class A</xif:Name>
<UML.Type:Roles>
<UML:AssociationRole xif:id="_28" href="#_6" xif:objid="{{0DB06784-A18C-11D3-AA35-
0060973166C2},00000039}"></UML:AssociationRole>
<UML:AssociationRole xif:id="_29" href="#_3" xif:objid="{{0DB06784-A18C-11D3-AA35-
0060973166C2},00000033}"></UML:AssociationRole>
</UML.Type:Roles>
<UML.Type:Members>
<UML:Attribute xif:id="_12" xif:objid="{{0DB06784-A18C-11D3-AA35-
0060973166C2},00000009}">
<xif:Comments></xif:Comments>
<xif:Name>att1</xif:Name>
<UML.Attribute:InitialValue></UML.Attribute:InitialValue>
<UML.Attribute:TypeExpression>integer</UML.Attribute:TypeExpression>
<UML.Element:Visibility>PrivateAccess</UML.Element:Visibility>
</UML:Attribute>
....more attributes.....
<UML:Operation xif:id="_21" xif:objid="{{0DB06784-A18C-11D3-AA35-
0060973166C2},0000001D}">
<xif:Comments></xif:Comments>
<xif:Name>opC</xif:Name>
<UML.Operation:Concurrency>Sequential</UML.Operation:Concurrency>
<UML.Operation:IsPolymorphic>0</UML.Operation:IsPolymorphic>
<UML.Operation:IsQuery>0</UML.Operation:IsQuery>
<UML.Operation:IsAbstract>0</UML.Operation:IsAbstract>
<UML.Operation:Parameters>
<UML:Parameter xif:id="_22" xif:objid="{{0DB06784-A18C-11D3-AA35-
0060973166C2},00000021}">
<xif:Name>inpArg</xif:Name>
<UML.Parameter:DefaultValue></UML.Parameter:DefaultValue>
<UML.Parameter:Kind></UML.Parameter:Kind>
<UML.Parameter:TypeExpression>Boolean</UML.Parameter:TypeExpression>
</UML:Parameter>
```

```
<UML:Parameter xif:id="_23" xif:objid="{{0DB06784-A18C-11D3-AA35-
0060973166C2},00000023}">
<xif:Name>opC</xif:Name>
<UML.Parameter:DefaultValue></UML.Parameter:DefaultValue>
<UML.Parameter:Kind>return</UML.Parameter:Kind>
<UML.Parameter:TypeExpression>Class A</UML.Parameter:TypeExpression>
</UML:Parameter>
</UML.Operation:Parameters>
<UML.Element:Visibility>PublicAccess</UML.Element:Visibility>
</UML:Operation>
....more operations.....
</UML.Type:Members>
<UML.GeneralizableElement:Specializations>
<UML:Generalization xif:id="_33" href="#_19" xif:objid="{{0DB06784-A18C-11D3-AA35-
0060973166C2},00000045}"></UML:Generalization>
</UML.GeneralizableElement:Specializations>
<UML.GeneralizableElement:Generalizations>
<UML:Generalization xif:id="_16" xif:objid="{{0DB06784-A18C-11D3-AA35-
0060973166C2},00000043}">
<xif:Name>Class C-&gt;Class A</xif:Name>
</UML:Generalization>
</UML.GeneralizableElement:Generalizations>
</UML:Class>
```

Specialization/Generalization Relationships

```
<UML:Generalization xif:id="_34" href="#_16" xif:objid="{{0DB06784-A18C-11D3-AA35-
0060973166C2},00000043}">
</UML:Generalization>
```

8. The Visual Component Manager (VCM) Repository

Visual Component Manager (VCM) is a utility for publishing different kinds of application items, e.g. COM components or Rose/VM models, to a Repository based catalog, where they can be located, inspected, retrieved, and reused. The items are stored along with attributes and search keywords to enable others to find and reuse them. VCM provides a keyword and a full text search mechanism to help organize and cross-reference items. VCM items can be cataloged and searched for by their name, type, description, keywords, and annotations.

In addition to single components and models, VCM can also store component libraries, templates, and complete application frameworks. For example, a tool developer might create a set of forms and basic modules that are the starting point for new forms and modules of the same type. By creating a template stored in VCM, any developer can get a copy of the forms or modules and be able to add to or change it without affecting the original.

Since VCM supports multiple databases, i.e., like Repository, it is possible to selectively browse through items in a local repository database, switch to a departmental repository database (on a shared server), and then finally to enterprise scope on another shared database. VCM can be used to drag items of interest into a local database, or insert them directly into a project.

A particular VCM item may require the presence of additional support files such as .dll files, help files, or documentation, and multiple files can be associated with any VCM item. This association becomes a part of the item's properties in VCM, and the associated files can be loaded along with the item. When used this way, VCM can be the central location for approved project programming conventions, functional specifications, architectural models, and diagrams.

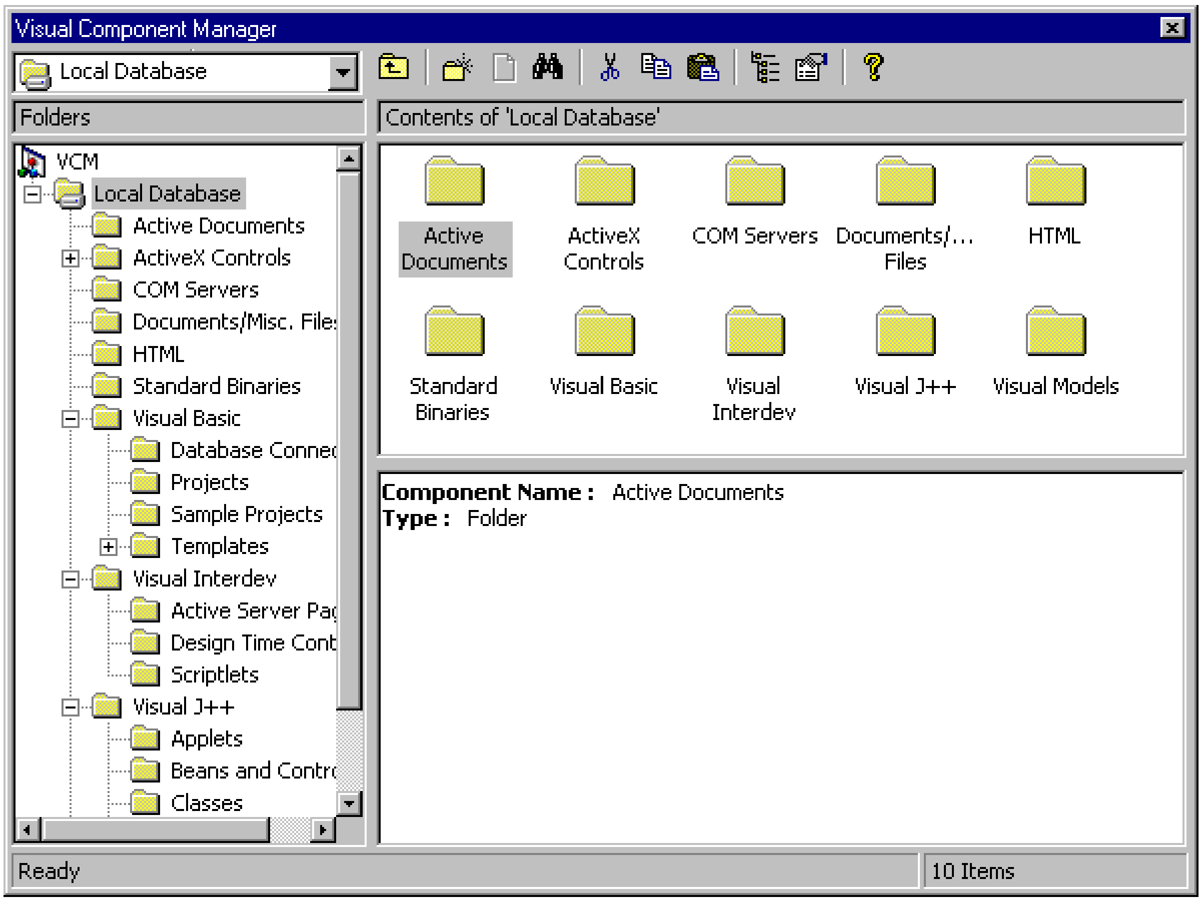
VCM supports the sharing and reuse of code components, documentation, and other pieces of a project, as well as complete projects. After having located a component, reusing it in a project may just require clicking Add to the project on the component's shortcut menu. When reusing components that require COM registration, VCM automatically registers the component when adding it to a project.

PS: The above description is a cut-and-paste from MSDN - I have not tested all these capabilities to see if they can deliver what they promise...

VCM can be started by e.g. selecting it in the View menu in Visual Basic. It provides a simple user interface to navigate through its contents. As illustrated in figure 6 it presents the following three panes:

- The explorer pane
- The contents pane
- The properties pane

To browse a published Rose/VM model, open the folder where the published model is located; e.g. Visual Models in figure 6. Then select the model you want to browse. A brief description of the model is displayed in the lower frame. To view the contents of the selected model, press the right mouse button and click *Browse Item.* Then a hierarchical view of the logical packages, associations, and classes is displayed in the lower right frame.



Component Name : Active Documents
Type : Folder

Figure 6. *The Visual Component Manager user interface.*

The VCM versus the Repository

Technically the VCM and the Repository are similar in many respects, but there are some differences with respect to their information models, and thus the corresponding database tables used to store their information. Repository uses the OIM with its extendable set of submodels for various subject areas. VCM uses a VCM information model that overlaps with the OIM. For example, the RTIM and the Uml models seems to be the same (as far as I can see...), but for other subject areas then VCM extend the OIM submodel (e.g. the Gen model), while others are not present in VCM. In addition VCM has its own core VCM model (as another subject area, prefixed Vcm* for e.g. database tables).

9. Rational Rose 98i

Support for Repository

Rational Rose 98i is integrated with both Repository and VCM. The Repository Add-In adds three commands to the Tools menu in Rational Rose:

- Publish to VCM
- Export to Repository
- Import from Repository

These commands apply to entire models, *not* to single model elements. However, from within VCM a single class or logical package of a published model can be imported into Rose (according to documentation by using drag and drop - but I did not get this to work...).

To export the logical view of a current open Rose model to the Repository:

1. Save the model
2. Choose Export to Repository from the Tools menu
3. In the displayed dialog box, select the folder into which you want to publish the current open model
4. Give the exported model a name to be used in the repository. Click Save

To import a model from Microsoft Repository into an empty model:

1. In Rational Rose, open a new empty model
2. Choose Import from Repository from the Tools menu. A dialog box is displayed
3. Select the model you want to import and click Open

There is an important difference between Repository and VCM with respect to the information exported.

According to Rose documentation:

> "The difference between the Publish to VCM and Export to Repository commands is the type of the published/exported model. Export to Repository uses the UML tool information model [Uml in OIM] to store the model, whereas Publish to VCM uses the VCM tool information model. The type of the published/exported model defines how much information that can be published/exported. The Publish to VCM command covers the entire model file, including all views and diagrams. The Export to Repository command applies only to classes, attributes, operations, and a ***subset*** of the relationships of the logical view of the model. Also, the Export to Repository command does not export any diagrams."

According to Visual Modeler documentation:

> "....The Export to Repository command covers only the logical view of the model, and no diagrams. The Publish to VCM command covers the entire model file, including all views and diagrams."

At first when reading and testing this I believed that VCM was more powerful than Repository (OIM) with respect to the amount of model information recorded. This is not entirely correct, however. It seems clear that VCM both contains a reference to the files of e.g. a published Rose model, as well as some meta information on this model stored separately. That is, some meta-information on a Rose model published to the VCM is available via interfaces and database tables similar to as in

Repository. However, meta information available as in the Repository does not affect the content of a subsequently imported model. That is, if e.g. the name of a class in a published Rose model is changed in the VCM database, then this change is not reflected when importing the model into Rose afterwards (actually, much of the meta information can be deleted without affecting a subsequent import). I have not been able to find out (by tests and by browsing the VCM documentation) exactly how these file references are organized, however (they seem to be independent of file names, amongst others).

There does not seem to be much differences in the meta information explicitly recorded in the database for both Repository and VCM. That is, the difference is mainly that VCM also references the model file - and if a Rose model consists of several files (.cat files), then VCM have troubles in handling this (see below).

Notice that there are some important deficiencies with respect to the meta information recorded, namely:

- That a class is abstract, root or leaf (in its inheritance hierarchy)
- That an association-end is ordered
- That an association-end is qualified

However, while the Rose export/publish programs does not include this information, both the OIM and the VCM information models are able to record this kind of information. Thus this information can be added "manually", i.e., by programmatically accessing the Rose model via its COM/Automation interface. Hence, to be able to export the required model information from Rose into Repository (or similarly VCM) we must replace the built-in export utility in the menu of Rose with a program (or script) that uses this utility programmatically, but in addition collects information on the missing parts above, and then adds it to the Repository itself. The following subsections explains how this can be done.

Retrieving Meta-Information from Rose via its COM/Automation Interfaces

The following Visual Basic code illustrates this. In this case an existing model "test6rep.mdl" is opened, the names of all its classes printed, with a notification for those that are abstract, and if there is a class named "C" then for each of its opposite roles (at the other end of its associations), the names of these roles and any qualifying keys are printed.

To run this program remember to include a reference to *"Rational Rose"* in the references section of VB; i.e., to use *"\Program Files\Common Files\Microsoft Shared\Repostry\repodbc.dll"*.

```
Dim refApp As RationalRose.RoseApplication
Dim refMod As RationalRose.RoseModel
Dim refClassColl As RationalRose.RoseClassCollection
Dim refClass As RationalRose.RoseClass
Dim refRoleColl As RationalRose.RoseRoleCollection
Dim refRole As RationalRose.RoseRole
Dim refAtt As RationalRose.RoseAttribute
Dim refAttColl As RationalRose.RoseAttributeCollection
Dim i As Integer
Dim j As Integer
Dim k As Integer

Set refApp = New RoseApplication
Set refMod = refApp.OpenModel("c:\home epa\vmod projects\test6rep.mdl")

Set refClassColl = refMod.GetAllClasses
For i = 1 To refClassColl.Count
    Set refClass = refClassColl.GetAt(i)

    If (refClass.Abstract) Then
        Debug.Print "Abstract Class: " & refClass.Name
    Else
```

```
            Debug.Print "Class: " & refClass.Name
        End If

        If refClass.Name = "Class C" Then
            Set refRoleColl = refClass.GetAssociateRoles
            For j = 1 To refRoleColl.Count
                Set refRole = refRoleColl.GetAt(j)
                Debug.Print refRole.Name

                Set refAttColl = refRole.Keys
                For k = 1 To refAttColl.Count
                    Debug.Print refAttColl.GetAt(k).Name
                Next
            Next
        End If
    Next

    Call refApp.Exit  ' NB! To avoid muuuuuch trouble - Do not forget!
```

Customizing Rose for Repository

Beside programmatically retrieving missing meta information from a Rose model and recording it in a Repository, the export utility of Rose (invoked when making the Menu selections described above) can also be invoked programmatically. The Rose scripts for this are in the catalog *"\Program Files\Rational\Rose 98i\repos\scripts"*:

- *RVSReposPu.ebx* - publish to VCM
- *RVSReposEx.ebx* - export to Repository
- *RVSReposIm.ebx* - import from Repository

and they can be executed as follows via the Rose COM/Automation interfaces (remember to include a reference to *"Rational Rose"* in the references section of VB; i.e., to use *"\Program Files\Common Files\Microsoft Shared\Repostry\repodbc.dll"*).

```
    Dim roseApp As RationalRose.RoseApplication
    Dim roseMod As RationalRose.RoseModel
    Dim addIn As RationalRose.RoseAddIn
    Dim addInColl As RationalRose.RoseAddInCollection
    Dim j As Integer

    Set roseApp = New RationalRose.RoseApplication
    Set roseMod = roseApp.OpenModel("C:\Home EPA\VMod Projects\dummy\original.mdl")

    ' Traverse add-ins until find the Repository add-in
    Set addInColl = roseApp.AddInManager.AddIns
    j = 1
    While (addInColl.GetAt(j).Name <> "Repository")
        j = j + 1
    Wend
    Set addIn = addInColl.GetAt(j)

    ' Display some information on this add-in
    Debug.Print "---------- ADD-IN"
    Debug.Print "Name: " & addIn.Name
    Debug.Print "Version: " & addIn.Version
    Debug.Print "MenuFilePath: " & addIn.MenuFilePath
    Debug.Print "InstallDirectory: " & addIn.InstallDirectory
    Debug.Print "PropertyFilePath: " & addIn.PropertyFilePath
    Debug.Print "ToolNames: "
    For i = 1 To addIn.ToolNames.Count
        Debug.Print "          " & addIn.ToolNames.GetAt(i)
    Next

    ' Execute publish to VCM
    Call addIn.ExecuteScript("scripts\rvsrepospu.ebx")
    Call roseApp.Exit
```

Support for VCM - Control/Uncontrol

A Rose model can be split into several files; a single .mdl file and a set of .cat files for different logical packages. I am able to publish a Rose model containing several .cat files to VCM, but I receive error messages when I try to import the model back from VCM (the import from Repository command). It seems as if VCM is not able to handle Rose models split over several files. Hence, if we intend to use VCM with Rose then we may have to handle control/uncontrol of .cat files "manually", i.e., by programmatically uncontrol all packages before publishing them to VCM, and then control the corresponding packages into their .cat files after importing the model from the VCM repository.

The following Visual Basic code illustrates how to uncontrol .cat files programmatically from the COM/Automation interface of Rose (remember to include a reference to *"Rational Rose"* in the references section of VB; i.e., to use *"\Program Files\Common Files\Microsoft Shared\Repostry\repodbc.dll"*). Notice that this program does not seem to work if the Rose 98i editor is in use on the model when the program is executed.

```
    Dim roseApp As RationalRose.RoseApplication
    Dim roseMod As RationalRose.RoseModel
    Dim cUnitColl As RationalRose.RoseControllableUnitCollection
    Dim cUnit As RationalRose.RoseControllableUnit

    Set roseApp = New RationalRose.RoseApplication
    Set roseMod = roseApp.OpenModel("C:\Home EPA\VMod Projects\dummy\original.mdl")

    Set cUnitColl = roseMod.GetAllSubUnitItems
    For i = 1 To cUnitColl.Count
        Set cUnit = cUnitColl.GetAt(i)

        ' "Basic Types" and "Work Process" are the packages to uncontrol
        If (cUnit.Name = "Basic Types" Or cUnit.Name = "Work Process") Then
            Debug.Print "------------ Controlled Unit"
            Debug.Print "Name = " & cUnit.Name
            Debug.Print "IsControlled = " & CStr(cUnit.IsControlled)
            Debug.Print "FileName = " & cUnit.GetFileName

            ' Uncontrol the package
            Call cUnit.Uncontrol
        End If
    Next

    Call roseMod.Save
    Call roseApp.Exit
```

10. Repository Support in SQL Server v.7.0

Support for Repository

As far as I have been able to find out, SQL Server does not support Repository except for its Data Transformation Services (DTS) (and via this support it is also possible (but as a detour) to export a specific database schema into a local Repository called MSDB). However, the OLE DB Scanner utility described below seems well suited for exporting a particular SQL Server database schema into a particular Repository.

SQL Server COM Interfaces

SQL Server offers COM/Automation interfaces that can be used to retrieve information on a particular database schema (I have not tried these interfaces myself, however). For SQL Server v.7.0 these interfaces are available from *"MSSQL7\Binn\Resources\1033\sqldmo.dll"*.

It may be relevant to use these interfaces e.g. to compare the schema of a particular database against database schema information in the Repository.

OLE DB Scanner - Importing a Database Schema to Repository

OLE DB Scanner is a utility with a function ScanDB which provided a database available as an OLE DB provider, imports the schema of this database into a Repository. It can be used from Visual Basic by including references to:

- Microsoft Repository (\Program Files\Common Files\Microsoft Shared\Repostry\repodbc.dll)
- DBScanner 1.0 Type Library (\Program Files\Common Files\Microsoft Shared\Repostry\dbscan.dll)

and including the following files into the Visual Basic project:

- RepEng.bas (Repository SDK "...\include" catalog)
- UMLConst.bas (Repository SDK "...\oim\VBConst" catalog)
- DBMConst.bas (Repository SDK "...\oim\VBConst" catalog)

It can then be used as follows:

```
<object>.ScanDB <pIRepository:Repository>, <pDbmDataSource:RepositoryObject>,
                <szProviderName:String>, [<szProviderString:String>],
                [<szDataSource:String>], [<szCatalog:String>], [<szUserName:String>],
                [<szPassword:String>]
```

where <object> is of type *RepOLEDBScanner* (i.e., *"Dim <object> As RepOLEDBScanner"* in Visual Basic), and square brackets indicate optional arguments.

The Repository SDK includes an example of its use, but I have not tested this utility myself. However, in order to compare a particular database schema with information in the Repository, then the SQL Server COM interfaces may be better suited for this.