TEKNOWLEDGE CORPORATION

# THE API OF THE UML INTERFACE

## JAVA & COM

By

ALEXANDER EGYED

SVEN JOHANN

THOMAS PELKEY

Created: 11 December 2003
Updated: 1 June 2004

# THE PROGRAMMATIC API TO THE UML INTERFACE

JAVA & COM

## OVERVIEW

The Unified Modeling Language (UML) is a widely accepted, implementation-independent standard for modeling object-oriented systems. Applications requiring the use of UML require access to some implementation of UML. The UML Interface is such an implementation for the UML version 1.3. It provides a programmatic interface for applications to store and retrieve UML model elements. It also provides a programmatic interface for applications to receive change notifications. The latter is of importance if the same UML model elements are accessed independently by multiple parties at the same time. Changes made by one party are then propagated to all other parties.

The UML Interface also provides a connector to the design tools Rational Rose and Matlab/Stateflow. This connector is useful for externalizing modeling information within those design tools to client applications (your application) that use the UML Interface. A primary benefit of using this connector is to make Rational Rose and Matlab/Stateflow the graphical front-ends to client applications where modeling information is created and maintained within those design tools but accessed through the client applications. The UML Interface also enables change notification for Rational Rose and Matlab/Stateflow. This implies that changes to UML model elements done by those design tools result in change notifications to be forwarded to client applications. Since the change notification occurs in real time, client applications can be made to respond to them.

This document will discuss how to build client tools that use the UML interface in full or in part. The document will use examples for illustration purposes.

## THE UNIFIED MODELING LANGUAGE (UML)

The Unified Modeling Language (UML) is currently the leading object-oriented analysis and design model. It supports a variety of design views, some of which object-oriented in nature (e.g., class diagrams [Booch 1994] [Rumbaugh et al. 1991]) and others more functional (e.g., statecharts [Harel 1987]). For the most part, views in UML are graphical; however, there are also textual descriptions, mostly in the form of add-ons to the graphical notation (e.g., Object Constraint Language [Warmer and Kleppe 1999]). UML is the result of a collaboration between numerous companies and OO modeling experts and it borrows heavily from Booch [Booch 1994], OMT [Rumbaugh et al. 1991], and other OO models such as [Coad and Yourdon 1991a], [Coad and Yourdon 1991b], and [Jacobson et al. 1992].

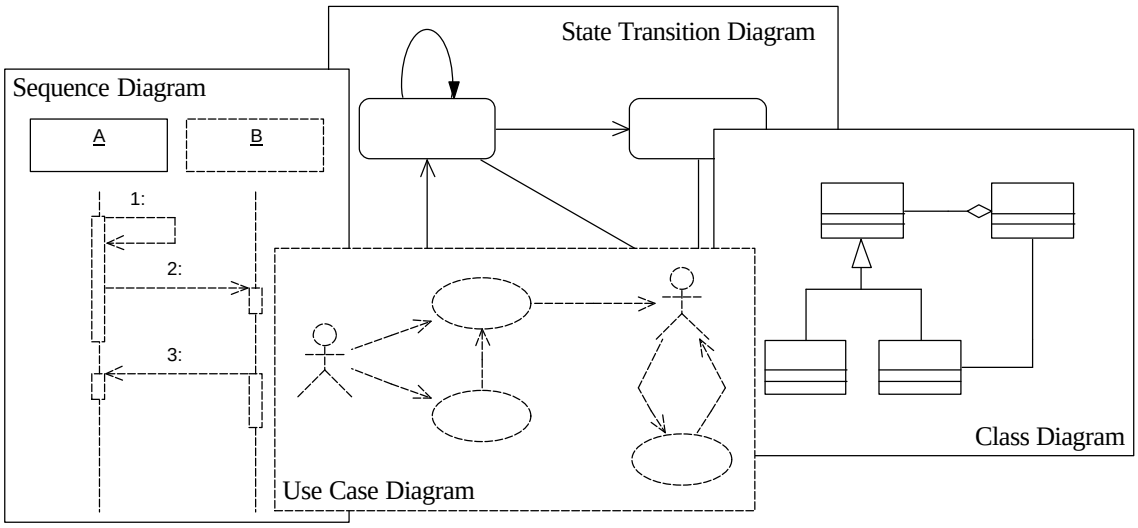


*Figure 1: Some of Diagrammatic Views support by UML*

"UML is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems" [Booch et al. 1999]. These different but overlapping uses of the model can only be achieved by supporting a variety of graphical representations. UML thus defines over 150 types of elements to support eight types of diagrams. The eight diagrams are class diagrams, sequence diagrams, statechart diagrams, use case diagrams, package diagrams, deployment diagrams, collaboration diagrams, and activity diagrams. Some of those diagrams are schematically depicted in Figure 1 showing a sequence diagram (left), a class diagram (right), a use-case diagram (middle-bottom), and statechart diagram (middle-top).

The diagrams are discussed below briefly. A more detailed discussion of UML diagrams is outside the scope of this thesis. We assume the reader to be familiar with the basic UML design concepts. Please refer to [Rumbaugh et al. 1999] or the "OMG Notation and Semantics Guide for UML" [OMG 1999] for more detailed descriptions (this work uses UML version 1.3). Additionally, [Fowler 1997] provides a brief overview of UML.

- **Use Case**: Depict the interaction between users and components or between components. In doing so, use cases provide a high-level view of the usage of a system and frequently shows the interaction of multiple functions of that system. For instance, the task of editing a document involves the functions *open document*, *edit document*, and *save document*.

- **Interaction** (e.g., Sequence and Collaboration diagrams): Sometimes also referred to as Mini-Uses. Interaction diagrams show concrete examples of how components communicate. They can often be seen as test cases and depict sequences of interactions (e.g., calls). A call can refer to user interface invocations (e.g., open file) or to component interactions.

- **Objects and Classes** (e.g., Class diagrams): Classes are the most central view in UML. Class diagrams depict the relationships between classes and objects, which are the smallest stand-alone components in OO. Class relationships further depict their generic interactions (e.g., aggregations, dependencies, etc.).

- **Packages** (e.g., Package diagram): Packages are used to group classes into layers and partitions. As such they show system decompositions.

- **State Transition** (e.g., State and Activity diagrams): Are used in UML to describe the states that classes can go through. In UML, state diagrams are bounded to individual classes. Activity Diagrams are a generalization of state diagrams in that they can also be used to depict events or other 'transitional' elements across class boundaries.

- **Deployment** (e.g., Deployment diagrams): Shows the physical components of the system during deployment. It presents a physical view of the system and is, therefore, frequently used to depict the component dependency of the actual implementation.

## DATA MANAGER

### OVERVIEW

The UML Interface provides classes and methods that reflect the 150+ types of model elements in UML. The most basic use of the UML Interface is to create and maintain instances of those types (Figure 2). In particular, the UML Interface provides a class for every type of UML model element (one-to-one dependency) and it provides class methods for the attributes and relationships of those elements. Client tools may use the provided classes and methods (collectively referred to as *Data Manager* in Figure 2) to create and maintain UML model elements.

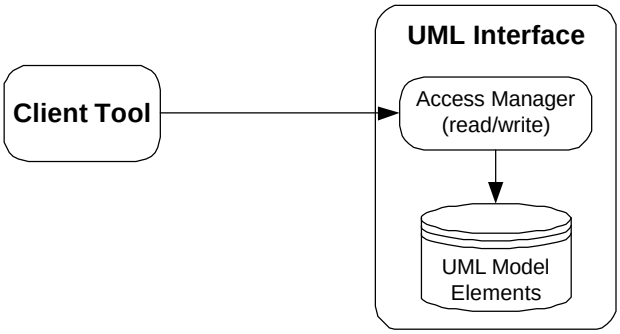


*Figure 2. A Client Tool can create and maintain UML model elements through the Data Manager .*

The Data Manager is an (almost) exact, complete implementation of UML 1.3 and it consists of roughly 150 classes---one class for every type of UML model element. Figure 3 depicts an excerpt of the UML meta model and shows 15 types of UML model elements and some of their attributes and relationships. UML elements are defined hierarchically with the topmost element being *Element*. Every other UML element inherits either directly or indirectly from *Element*. The UML element *Model Element* inherits from *Element* and it is the topmost element for all graphical elements of UML. For instance, a UML class diagram consists of classes (UML element *Class*), methods (UML element *Method*), attributes (UML element *Attribute*), and relationships (UML elements *Association*, *Dependency*, *Generalization*, etc.)---all of which inherit from *ModelElement* either directly or indirectly. As such, the UML element *Class* inherits from *Classifier*, which, in turn, inherits from both *Namespace* and *GeneralizeableElement* (multiple inheritance), which, finally, inherit from *ModelElement*.

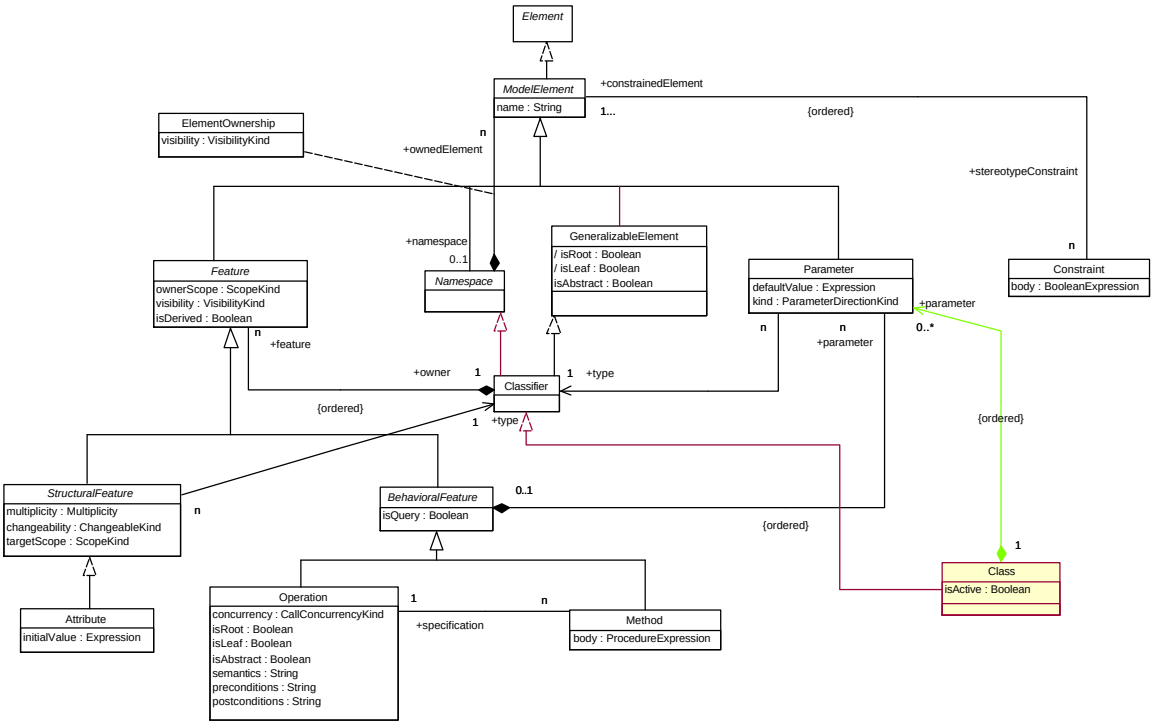


*Figure 3. Excerpt of the UML Meta Model defining types of UML model elements, their attributes, and their relationships*

## UML CLASSES AND METHODS

The UML Interface provides implementations of *Element*, *ModelElement, Namespace*, *GeneralizeableElement*, *Classifier*, *Class* and all other types of UML elements. Each UML element is implemented as a separate Java class. Get and set methods are provided for attributes of UML elements. For instance, the diagram in Figure 3 shows that a *ModelElement* may have a name of type *String*. The implementation of *ModelElement* thus defines an (private) attribute called *name* of type *String* and it provides two (public) access methods for setting and getting the name (called *getName* and *setName*). Get and set methods are also provided for relationships among UML elements. For instance, there is an aggregation relationship between *ModelElement* and *Namespace* indicating that every model element may belong to some kind of 'container' (e.g., classes may belong to packages, states may belong to composite states). The cardinality information on the aggregation indicates that a namespace may contain *n* model elements (*n* indicates a variable number ranging from zero to many) and that a model element must belong to zero or one namespaces (0..1). The aggregation relationship even indicates the role names where model elements part of a name spaces are referred to as *ownedElements* and the name space of a model element is referred to as *namespace*. The implementation of *ModelElement* thus defines an (private) attribute called *namespace* of type *Namespace* (*Namespace* is another Java class implementing the same-named UML element) and it provides two (public) access methods for setting and getting the name space (called *setNamespace* and *getNamespace*). In case where the cardinality of a relationship allows many elements then collection variables are used. For instance, a name space may have zero, one, or more model elements. Therefore, the implementation of *Namespace* defines an (private) attribute called *ownedElements* of type *Set* (set is a collection variable where the order of elements is not important) and it defines four (public) access methods to get and set the whole set or add and remove individual elements (called *getOwnedElements*, *setOwnedElements*, *addOwnedElement*, and *removeOwnedElement*).

Although the implementation of relationships among UML elements and attributes of UML elements may appear similar, there are two key differences (see also Table 1):

1) Only one set of access methods is required for attributes but two sets of access methods are required for relationships: For instance, implementing the aggregation relationship between *ModelElement* and *Namespace* implies one set of access methods in *ModelElement* to refer to *Namespace* and one set of access methods in *Namespace* to refer to *ModelElement*.

2) Access methods of relationships have to maintain consistency: For instance, if a model element defines a namespace then that namespace must have at least that model element as one of its owned elements.

*Table 1. Implementation Skeleton Code for the UML Elements* ModelElement *and* Namespace.

| LOCATION | METHODS |
|---|---|
| **Class *ModelElement***<br><br>**in**<br>**UML**<br>**Interface** | **Java Interface *IModelElement***<br>`public interface IModelElement extends IElement`<br>`{`<br>`  public String getName();`<br>`  public void setName(String name);`<br><br>`  public INamespace getNamespace();`<br>`  public void setNamespace(INamespace namespace);`<br><br>`  ...`<br>`}`<br>**Java Implementation Class *MModelElement***<br>`public class MModelElement extends MElement implements IModelElement`<br>`{`<br>`  public MModelElement() { … }`<br>`  public MModelElement(String name) { … }`<br><br>`  public String getName() { … }`<br>`  public void setName(String name) { … }`<br><br>`  public INamespace getNamespace()  { … }`<br>`  public void setNamespace(INamespace namespace) { … }`<br><br>`  public String _name = …;`<br>`  public INamespace _namespace = …;`<br><br>`  ...`<br>`}` |
| **Class *Namespace***<br><br>**in**<br>**UML**<br>**Interface** | **Java Interface *INamespace***<br>`public interface INamespace extends IModelElement`<br>`{`<br>`  public ISet getOwnedElements();`<br>`  public void setOwnedElements(ISet ownedElements);`<br>`  public void addOwnedElement(IModelElement ownedElement);`<br>`  public void removeOwnedElement(IModelElement ownedElement);`<br>`}`<br>**Java Implementation Class *MNamespace***<br> |

```
public class MNamespace extends MModelElement implements INamespace
{
    public ISet getOwnedElements() { … }
    public void setOwnedElements(ISet ownedElements) { … }
    public void addOwnedElement(IModelElement ownedElement) { … }
    public void removeOwnedElement(IModelElement ownedElement) { … }

    public ISet _ownedElements = …;
}
```

The naming of classes, attributes, and access methods is derived from the UML meta model. Every UML element type has a name and this name is used to create an interface and implementation class where the name for the interface is prefixed with the letter 'I' and the name for implementation class is prefixed with the letter 'M.' For instance, the UML Interface defines for the UML element *ModelElement* a Java interface with the name *IModelElement* and a Java implementation class with the name *MModelElement* where the latter implements the former. Also, every UML attribute part of an UML element is used to create (1) an attribute in the Java implementation class of that UML element, (2) several interfaces of access methods for that attribute in the Java interface, and (3) implementations of those access methods in the Java implementation class. For instance, the UML Interface defines for the attribute *name* (type *String*) of the UML type *ModelElement* an attribute called *_name* in the Java implementation class *MModelElement*, two interfaces for access methods called *getName* and *setName* in the Java interface *IModelElement*, and two implementations for those access methods in the Java implementation class *MModelElement*.

Since some UML elements have multiple parents (e.g., UML element *Classifier* inherits from both *Namespace* and *GeneralizableElement*) the UML Interface separates Java interfaces (which support multiple inheritance) from Java implementation classes (which do not support multiple inheritance). This solution allows Java interfaces to be consistent with the UML definition. It is thus recommended to define UML element always through their interface definitions but never through their implementation classes. As such:

```
public IClass someClass = null;      //CORRECT!
public MClass someClass = null;      //WRONG! although a compiler would not complain
```

For a complete listing of classes and methods please read the users manual and reference manual of the Unified Modeling Language (UML).

### ADMINISTRATIVE CLASSES AND METHODS

There is a necessary administrative overhead in handling UML elements, their attributes and relationships. Most administrative overhead of using the UML Interface is hidden within the *Base* class. Some additional administrative overhead is hidden in the class *Element* which is inherited by all UML elements and thus available to all of them. The *Base* class is the key to working with the UML Interface. Creating a *Base* class is equivalent to creating an instance of the UML Interface (e.g., *new MBase()*). The base class serves as the origin for the creation of UML elements (using methods such as *newClass* or *newState*) and it serves as the origin for the navigating among them (using methods such as *getClasses* or *find*). The *Base* class is very large and contains several hundred methods. Initially, it is not vital to understand most of those methods; Table 2 (top) summarizes important ones.

*Table 2. Administrative Access Methods in* Element *and* Base.

| LOCATION | METHODS |
|---|---|
| **Class *Base***<br><br>**in**<br>**UML**<br>**Interface** | **public IClass findClassByName(String name)**<br>finds an UML class in the UMLInterface<br>**public ISet findModelElementByName(String name)**<br>returns ISet of all model elements<br>**public IElement findByQualifiedName(String qualifiedName)**<br>returns the UML element with the qualified name<br>**public IElement find(String id);**<br>searches for an UML element with the given id. Returns null if no such element is found.<br>**public ISet getAllElements();**<br>returns a list of all UML elements.<br>**public ISet getTopPackages();**<br>returns the topmost package of an UML model which are all packages that themselves are not owned by other packages (namespace == null).<br>**public ISet getClasses();**<br>**public void setClasses(ISet classes);**<br>**public void addClass(IClass aClass);**<br>**public void removeClass(IClass aClass);**<br>retrieving and maintaining classes**.**<br>**public ISet getPackages();**<br>**public void setPackages(ISet packages);**<br>**public void addPackage(IPackage aPackage);**<br>**public void removePackages(IPackage aPackage);**<br>retrieving and maintaining packages.<br>**public ISet getRelationships();**<br>**public void setRelationships(ISet relationships);**<br>**public void addRelationship(IRelationship relationship);**<br>**public void removeRelationship(IRelationship relationship);**<br>retrieving and maintaining relationships.<br>**public ISet getInstances();**<br>**public void setInstances(ISet instances);**<br>**public void addInstance(IInstance instance);**<br>**public void removeInstance(IInstance instance);**<br>retrieving and maintaining instances of classes (objects).<br>**public ISet getStateMachines();**<br>**public void setStateMachines(ISet stateMachines);**<br>**public void addStateMachine(IStateMachine stateMachine);**<br>**public void removeStateMachine(IStateMachine stateMachine);**<br>retrieving and maintaining state machines.<br>**public ISet getCollaborations();**<br>**public void setCollaborations(ISet collaborations);**<br>**public void addCollaboration(ICollaboration collaboration);**<br>**public void removeCollaboration(ICollaboration collaboration);**<br>retrieving and maintaining collaborations (objects).<br><br>**public IClass newClass();**<br>**public IClass newClass(String name);**<br>**public INamespace newNamespace();**<br>**public INamespace newNamespace(String name);**<br>**public IModel newModel();**<br>**public IModel newModel(String name);**<br>**…** |

| | |
|---|---|
| | constructors for all 150+ UML elements. Note: above list is a short excerpt. Two constructors are provided for all UML elements that inherit from *ModelElement* (newXXX() and newXXX(String name)) and one constructor is provided for all remaining UML elements.<br>**`public String getModelName();`**<br>**`public void setModelName(String name);`**<br>retrieving and maintaining model names. Can be an arbitrary textual string.<br>**`public void dispose();`**<br>disposes all model elements (calls *dispose* of all elements individually).<br>**`public void shutdown();`**<br>destroys the UML Interface with all its contents. Also notifies all client tools (if registered) of the destruction and severs the connection to design tool (if any was used). |
| **Class *Element***<br><br>**in UML Interface** | **`public String getID();`**<br>returns a unique id in form a string.<br>**`public String getUMLType();`**<br>returns a string describing the type name of the UML element (e.g., 'Class' for *IClass*).<br>**`public boolean getIsDeleted();`**<br>returns true if the element was deleted; otherwise returns false.<br>**`public String toString();`**<br>returns a string summarizing the contents of the UML element (i.e., useful for debugging).<br>**`public void dispose();`**<br>deletes a model element by resetting its attributes and all references to other elements (i.e., relationships) |

The code below shows an example on how to create an instance of the UML interface in order to add a class. The first statement creates an object of type *IBase*. This object serves as a basis for all further interactions with the UML Interface (tip: store the *Base* object in a global variable). The code then shows how to create a new class by calling the method *mbase.newClass* (name string is optional). This method call create a new instance of type *Class*, places the new instance into the list of all classes in *mbase*, and returns a handle to that class for future reference. A call to *mbase.getClasses* is shown next in the code to verify whether the new class was really added.

```
import UMLInterface.*;
…
try {
    IBase mbase = new MBase();
    if (mbase.getClasses().size()==0)
        new WinMsgBoxDlg("SUCCESS", "SIZE==0").showDialog();
    IClass class1 = mbase.newClass("Class 1");
    if (mbase.getClasses().contains(class1))
        new WinMsgBoxDlg("SUCCESS", "CLASS 1 IS DEFINED").showDialog();
} catch (Exception ex3) { /*ERROR MESSAGE*/ }
```

Three observations must be made about the code: (1) In order to use the interface, it is necessary to import the UML Interface library. This is done through the statement *import UMLInterface.**. (2) A lock must be obtained before accessing any classes and released when finished. (3) The Interface has a notion of internal consistency. Thus, if a class is created using *mbase.newClass* then this method informs the *Base* class of that creation so that its method *mbase.getClasses* knows about the new class.

## GRAPHICAL INFORMATION

In addition to the data repository, Rational Rose also makes UML model elements available in form graphical diagrams. The UML Interface also makes this information available . For example, uploading a model element to a Rosediagram, changing the color, or just writing a message to the Rational Rose log window are possible through the following methods.

*Table 3. Access Methods for Graphical Information of Model Elements.*

| LOCATION | METHODS |
|---|---|
| **Class *Base***<br><br>**in**<br>**UML**<br>**Interface** | **public ISet findElementInClassDiagrams( IModelElement element )**<br>Finds all class and use case diagrams that contain the input model element.<br>@param model element to search for<br>@return Set of all WinClassDiagram's that contain the input model element.<br><br>**public ISet findElementInDiagrams( IModelElement element )**<br>Finds diagrams that contain the input model element.<br>@param model element to search for<br>@return Set of all WinClassDiagram's that contain the input model element.<br><br>**public IPackage findPackage( String pkg )**<br>Finds the named package.<br>@param pkgName input<br>@return the IPackage found if found, null otherwise.<br><br>**public WinDiagram findWinDiagram( String diagram )**<br>Locates a diagram in the WinBase class.<br>@param qualified name of the diagram to find.<br>@return instance of the WinDiagram found or null if not found.<br><br>**public ISet getClassDiagrams( IModelElement element )**<br>@return all class diagrams of the Rose model<br><br>**public boolean highlightElement( IModelElement element )**<br>Highlights the model element in the first diagram found that contains it.<br>If the element is and operation or attribute the parent is highlighted. If<br>the element is not in a diagram, the element's specification is opened.<br><br>**public WinActivityDiagram newActivityDiagram()**<br><br>**public WinActivityDiagram newActivityDiagram( String diagramName, IPackage packageInstance )**<br><br>**public WinActivityDiagram newActivityDiagram( String diagramName, IPackage packageInstance, IStateMachine stateMachine )**<br><br>**public WinClassDiagram newClassDiagram()**<br><br>**public WinClassDiagram newClassDiagram( String diagramName, IPackage packageInstance )** |

| | |
|---|---|
| | **public WinCollaborationDiagram newCollaborationDiagram()**<br><br>**public WinCollaborationDiagram newCollaborationDiagram( String diagramName, IPackage packageInstance )**<br><br>**public WinSequenceDiagram newSequenceDiagram()**<br><br>**public WinSequenceDiagram newSequenceDiagram( String diagramName, IPackage packageInstance )**<br><br>**public WinStateChartDiagram newStateChartDiagram( IModelElement element )**<br><br>**public WinStateChartDiagram newStateChartDiagram( IModelElement element, String diagramName )**<br><br>**public WinUseCaseDiagram newUseCaseDiagram**<br><br>**public WinUseCaseDiagram newUseCaseDiagram( String diagramName, IPackage packageInstance )**<br><br>**public boolean openCustomSpecification( IModelElement element )**<br>Opens the specification of the given element<br><br>**public boolean openSpecification( IModelElement element )**<br>Opens the custom specification of the given element<br><br>**public boolean removeClassInRose( IClass clazz )**<br>Removes the class from rose.<br><br>**public void selectElement( IModelElement element )**<br>Select the model element in the first diagram found that contains it.<br>If the element is an operation or attribute the parent is selected.<br><br>**public boolean uploadActor( IActor actor )**<br><br>**public boolean uploadAssociation( IAssociation assoc )**<br><br>**public boolean uploadAssociationEnd( IAssociationEnd assocEnd )**<br><br>**public boolean uploadAttribute( IAttribute attr )**<br><br>**public boolean uploadClass( IClass clazz )**<br><br>**public boolean uploadDependency( IDependency dependency )**<br><br>**public boolean uploadGeneralization( IGeneralization gen )**<br><br>**public boolean uploadOperation( IOperation op )**<br><br>**public boolean uploadPackage( IPackage pkg )**<br><br>**public boolean uploadParameter( IParameter parameter )**<br><br>**public boolean uploadSet( ISet elements )** |

| | |
|---|---|
| | **public boolean uploadStateMachine( IStateMachine statemachine )**<br><br>**public boolean uploadStereotype( IStereotype stereotype )**<br><br>**public boolean uploadUseCase( IUseCase usecase )** |
| **Class *WinDiagram***<br><br>**in**<br>**UML**<br>**Interface** | **public void show()**<br>Displays the diagram.<br><br>**public void hide()**<br>Hides the diagram.<br><br>**public void layout()**<br>Auto arranges the elements of the diagram<br><br>**public void autoSizeAll()**<br>Auto sizes the elements of the diagram<br><br>**public void update()**<br>Updates the diagram<br><br>**public boolean addElement( IModelElement e )**<br>Adds an element to this diagram<br><br>**public boolean removeElement( IModelElement e )**<br>Removes an element from this diagram<br><br>**public boolean containsElement( IModelElement element )**<br>Determines if the input element exists in the diagram.<br><br>**public static IPackage findPackage( String pkgName )**<br>return the IPackage found if found, null otherwise.<br><br>**public void copyToClipboard()**<br>Copies the diagram to the clipboard as a Windows metafile.<br><br>**public int getZoom()**<br>Retrieves the zoom factor for the current drawing.<br><br>**public void setZoom( int zoom )**<br>Sets the zoom factor for the current drawing.<br><br>**public String addTextBox( String text, int X, int Y, boolean border )**<br>Adds a text box with or without a border.<br>@param text to be written<br>@param border true draws a box around the text, false draws free floating text label.<br>@return unique identifier<br><br>**public void moveTextBox( String textId, int X, int Y )**<br>Adds a text box with or without a border. |

@param text to be written
@param border true draws a box around the text, false draws free floating text label.
@return unique identifier

**public boolean removeTextBox( String id )**
Removes the text box identified by the id passed.
@param unique idenitfication for the text box.
@return true if successful, false otherwise.

**public Hashtable getTextBoxes()**

**public void selectElement( IModelElement element )**
Selects the given model element in this diagram.

**public void unselectElement( IModelElement element )**
UnSelects the given model element in this diagram.

**public void unselectAllElements()**
Unselects all selected items in this diagram.

**public boolean highlightElement( IModelElement element )**
Selects and flashes the class in the rose class diagram.

**public void setElementLineColor( IModelElement element, IColor color )**
Sets the line color of the given model element in this diagram.

**public IColor getElementFillColor( IModelElement element )**
Gets the fill color given model element in this diagram.

**public IColor getElementLineColor( IModelElement element )**
Gets the line color of the given model element in this diagram.

**public void setElementFillColor( IModelElement element, IColor color )**
Sets the fill color given model element in this diagram.

**public int getElementXPosition( IModelElement element )**
Retrieves the X position value of the element passed.

**public void setElementXPosition( IModelElement element, int x )**
Sets the X position value of the element passed.

**public int getElementYPosition( IModelElement element )**
Retrieves the Y position value of the element passed.

**public void setElementYPosition( IModelElement element, int y )**
Sets the Y position value of the element passed.

**public int getElementWidth( IModelElement element )**
Retrieves the width value of the element passed.

**public void setElementWidth( IModelElement element, int width )**
Sets the width value of the element passed.

**public int getElementHeight( IModelElement element )**

| | |
|---|---|
| | Retrieves the height value of the element passed.<br><br>**public void setElementHeight( IModelElement element, int height )**<br>Sets the height value of the element passed.<br><br>**public IFont getElementFont( IModelElement element )**<br>Get the font name of the element passed.<br><br>**public void setElementFont( IModelElement element, IFont font )**<br>Sets the font name of the element passed. |

## CONCURRENT ACCESS

The security manager provides an optional locking mechanism for concurrent access of the interface and design tool (i.e., mouse clicks and keystrokes are nullified). The security mechanism is disabled by default. It should be used if there could be potential race condition or deadlock problems with multiple client tools accessing the UML Interface concurrently. If the use of the security mechanism is desired, the client tools must enable security and use lock before any model elements are retrieved. The application specifies how long to wait for the lock (see table 3 for timeouts). If the interface is accessed before a lock is granted, then a WinSecurityException is thrown. getIsLocked and getIsAccessible methods report the status of the lock.

Each successful lock must be balanced by a call to unlock. Lock increments a counter for each successful lock. Likewise, unlock decrements the counter. The interface and the design tool are unlocked (freed) only when the counter reaches zero. Use Ctrl-Break from the keyboard to unlock the design tool even if the counter is not zero.

*Table 3. Concurrent Access Methods in* Base.

| LOCATION | METHODS |
|---|---|
| **Class** ***Base***<br><br>**in**<br>**UML**<br>**Interface** | **`public boolean enableSecurity();`**<br>enables the locking mechanism. Returns true if successful, false otherwise.<br>**`public boolean enableSecurity(String securityId);`**<br>enables the locking mechanism with a custom identification. Other applications could use this Id for coordinated, concurrent access. Returns true if successful, false otherwise.<br>**`public void disableSecurity();`**<br>disables the locking mechanism. All methods return true.<br>**`public boolean lock(int time_out);`**<br>locks the interface and design tool for accessing and modifying classes. time_out specifies the time to wait for the lock in milliseconds: <1 is infinite wait, 0 is no wait, >1 is specified wait. Returns true if the lock was received in the specified time. Otherwise, false is returned. See specific wait constants.<br>**`public int DEFAULT_TIMEOUT();`**<br>returns the default, constant value used for lock wait times (i.e., the time the lock request waits for a lock).<br>**`public int INFINITE_TIMEOUT();`**<br>returns the constant valued use for infinite waits.<br>**`public boolean unlock();`**<br>releases the interface. Returns true if successful, false otherwise.<br>**`public boolean getIsLocked();`**<br>returns true if the interface is locked by any thread or process. |

| | |
|---|---|
| | **public boolean getIsAccessible();**<br>returns true if the interface is locked by this thread or process. |

It is important to keep in mind that locks must be balanced with unlocks. Dead-lock situations can occur if you leave a lock. It is good practice to put unlocks in error handlers and termination methods. Here is an example in Java that demonstrates this:

```
import UMLInterface.*;
…
try {

    IBase mbase = new MBase();
    if (mbase.lock( 5000 ) ) {

        if (mbase.getClasses().size()==0)
            new WinMsgBoxDlg("SUCCESS", "SIZE==0").showDialog();
        IClass class1 = mbase.newClass("Class 1");
        if (mbase.getClasses().contains(class1))
            new WinMsgBoxDlg("SUCCESS", "CLASS 1 IS DEFINED").showDialog();
    else {
        new WinMsgBoxDlg("LOCK TIMED OUT", "NO LOCK OBTAINED!").showDialog();
    }
} catch (Exception ex3) { /*ERROR MESSAGE*/ }
finally {
    mbase.unlock();
}
```

Note the unlock in the finally statement executes regardless of whether or not an error occurred in the try block. This insures that the lock is freed when it is no longer needed. Also, the lock could fail because another application kept the lock longer than the 5 second wait specified or because another application failed to free the lock.

## COLLECTIONS

The UML Interface uses collections to handle groups of elements. For instance, the method *getClasses* in *Base* returns a set of classes. This set may be empty (*mbase.getClasses().size==0*) if no class exists, or this set may contain one or more classes. The basic collection class in the UML Interface, called *Collection*, is an abstract class which has three implementations: *Set*, *Sequence*, and *Bag*. Sets are collections that may contain at most one instance of any element (e.g., *Class 1* in above example cannot be added twice). For sets it is not important in what order elements are added. Sequences are collections conceptually like arrays. Many instances of the same element may be added and the order of elements is relevant. Bags also allow many instances of elements to be added (like sequences) but the order of elements is not relevant (like sets). Almost all collections used in the UML Interface are sets (i.e., it is not important in what order classes are added; e.g., *mbase.getClasses()* returns a set); only a few cases use sequences (e.g., sequence is used when returning the list of parameters using the method *getParameters()*).

Collections, whether they be sets, sequences, or bags, have many common methods although their semantics may vary. For instance, all sets, sequences, and bags have an *insert* method to add

elements to the collection but in the case of a set, the *insert* method only adds the element if it is not already there; this is not done in the cases of sequences and bag.

*Table 4. Administrative Access Methods in* Element *and* Base.

| LOCATION | METHODS |
|---|---|
| **Class *Collection***<br><br>**in**<br>**UML**<br>**Interface** | **`public int size();`**<br>returns the number of elements in the collection.<br>**`public boolean isEmpty();`**<br>returns true if there is no element in the collection; otherwise returns false.<br>**`public boolean contains(Object o);`**<br>returns true if the given element is in the collection; otherwise returns false.<br>**`public void insert(Object o);`**<br>inserts the given element into the collection.<br>**`public void remove(Object o);`**<br>removes the given element from the collection.<br>**`public boolean containsAll(ICollection c);`**<br>returns true if all given elements are in the collection; otherwise returns false.<br>**`public void insertAll(ICollection c);`**<br>inserts all given elements into the collection.<br>**`public void removeAll(ICollection c);`**<br>removes all given elements from the collection.<br>**`public void retainAll(ICollection c);`**<br>removes all elements but the given ones from the collection.<br>**`public void clear();`**<br>removes all elements from the collection.<br>**`public boolean equals(Object equalsList);`**<br>compares two collections; returns true if collections have identical elements.<br>**`public ISet toSet();`**<br>transforms any collection into a set.<br>**`public IBag toBag();`**<br>transforms any collection into a bag.<br>**`public ISequence toSequence();`**<br>transforms any collection into a sequence.<br>**`public IIterator iterator();`**<br>returns a iterator for the elements in the collection. The iterator can be used to traverse the elements of the collection. Use *hasNext()* to detect whether the collection has still elements and *next()* to get the next element.<br>**`public String toString();`**<br>**`public String toString(String begin, String middle, String end);`**<br>returns a string listing all the elements in the collection (uses '[', ',', and ']' symbols to precede the list (begin), to separate the elements in the list (middle), and to terminate the list (end). |
| **Class *Set***<br><br>**in**<br>**UML**<br>**Interface** | **`public ISet including(Object literal);`**<br>returns a new set containing the same elements plus the literal.<br>**`public ISet excluding(Object literal);`**<br>returns a new set containing the same elements minus the literal.<br>**`public ISet doUnion(ICollection collection);`**<br>returns a new set containing the union of both sets.<br>**`public ISet doMinus(ICollection collection);`**<br>returns a new set containing all elements of the set minus the given ones.<br>**`public ISet doSymmetricDifference(ICollection collection);`**<br>returns a new set containing only non-overlapping elements of both sets.<br>**`public ISet doIntersection(ICollection collection);`** |

| | |
|---|---|
| | returns a new set containing only overlapping elements of both sets. |
| **Class *Sequence*** | **`public boolean insertAll(int index, ICollection c);`**<br>inserts all elements at index; the indices of subsequent elements are increased.<br>**`public Object get(int index);`**<br>returns the element at index.<br>**`public Object set(int index, Object element);`**<br>replaces the element at index.<br>**`public void insert(int index, Object element);`**<br>inserts the element at index; the indices of subsequent elements are increased.<br>**`public Object remove(int index);`**<br>removes the element at index and returns the element.<br>**`public int indexOf(Object o);`**<br>returns the index of the first occurrence of element o.<br>**`public int lastIndexOf(Object o);`**<br>returns the index of the last occurrence of element o.<br>**`public ISequence subSequence(int fromIndex, int toIndex);`**<br>returns a sub sequence of elements (between indices).<br>**`public ISequence doUnion(ICollection collection);`**<br>returns a new sequence containing the union of both sequences.<br>**`public ISequence doMinus(ICollection collection);`**<br>returns a new sequence containing all elements of the set minus the given ones.<br>**`public ISequence doIntersection(ICollection collection);`**<br>returns a new sequence containing only overlapping elements of both sequences. |
| **Class *Bag***<br><br>**in UML Interface** | **`public int count(Object o);`**<br>returns the number of times element o is contained..<br>**`public IBag doUnion(ICollection collection);`**<br>returns a new bag containing the union of both bags.<br>**`public IBag doMinus(ICollection collection);`**<br>returns a new bag containing all elements of the bag minus the given ones.<br>**`public IBag doIntersection(ICollection collection);`**<br>returns a new bag containing only overlapping elements of both bags. |

The code below is an extension of the previous code. It additionally introduces a use case (name: *Use Case 1*) and a dependency (no name). It then adds the previously created class *class1* as a client to the dependency and it adds *useCase1* as a supplier. To demonstrate that this set of operations resulted in a navigable set of model elements, all client dependencies of the class are requested (*class1.getClientDependencies()*). Those client dependencies are then iterated through. There must be at least one dependency among the client dependencies of *class1* that has a supplier equal to *useCase1*.

```
IBase mbase = new MBase();
try {
    if (mbase.getClasses().size()==0)
        new WinMsgBoxDlg("SUCCESS", "SIZE==0").showDialog();
    IClass class1 = mbase.newClass("Class 1");
    if (mbase.getClasses().contains(class1))
        new WinMsgBoxDlg("SUCCESS", "CLASS 1 IS DEFINED").showDialog();
    IUseCase useCase1 = mbase.newUseCase("Use Case 1");
    IDependency dependency1 = mbase.newDependency();
    dependency1.addClient(class1);
    dependency1.addSupplier(useCase1);
    IIterator dependenciesOfClass1 = class1.getClientDependencies().iterator();
    while (dependenciesOfClass1.hasNext())
    {
```

```
            IDependency dependencyOfClass1 = (IDependency) dependenciesOfClass1.next();
            if (dependencyOfClass1.getSuppliers().contains(useCase1))
                new WinMsgBoxDlg("SUCCESS", "LITTLE DIAGRAM IS COMPLETE").showDialog();
        }
    } catch (WinSecurityException wse) {
        new WinMsgBoxDlg("ERROR", "Lock not obtained. " + wse.getMessage()).showDialog();
    } catch (Exception ex) {
        new WinMsgBoxDlg("ERROR", ex.getMessage()).showDialog();
    }
```

The interface always uses the same objects for same elements. Thus any given UML element has exactly one instance. Comparison of instances could be done using the equal symbol (e.g., *class1==dependency1.getClient().toSequence().get(0)*) however it is good programming practice to use the equal operator (e.g., *class1.equals(dependency1.getClient().toSequence().get(0))*). Furthermore, always use the constructors in the *Base* to create new UML elements. This avoids potential problems when using the interface remotely or when connecting it to a design tool (see later sections).

```
public IClass someClass = new MClass();      //OK
public IClass someClass = mbase.newClass();  //BETTER!!
```

The collection class also supports the collective calling of methods. This can reduce deep hierarchies of loops. The method *invoke* takes a method name as a parameter, calls a method with that name for all elements in the collection, and returns all results in form of another collection. For instance, the statement *class1.getClientDependencies().invoke("getSuppliers")* returns a set of all suppliers for all client dependencies of *class1*. The while loop in above sample code can be replaced through the following simple statement:

```
    if (class1.getClientDependencies().invoke("getSuppliers").contains(useCase1))
        new WinMsgBoxDlg("SUCCESS", "LITTLE DIAGRAM IS COMPLETE").showDialog();
```

## JAVA & COM

The UML Interface is a Java library and is best used by client tools that are themselves implemented in Java. If it is not feasible to program your tools in Java then alternatively the Common Object Model (COM) may be used to connect to the UML Interface. The COM option is generally slower but has added benefits such as the ability of separating the processes of client tools and UML Interface. Three forms of integrating a client tool with the UML Interface are supported:

- importing it as a Java library (best) into a client tool,
- importing it as a COM/DLL (slower, less type checking) into a client tool,
- remotely accessing it through COM using the UMLInterfaceServer (slowest due to inter-process communication).

### OPTION 1: IMPORTING THE UML INTERFACE AS A JAVA LIBRARY

This option was used in the previous section. The Java library has to be listed in the CLASSPATH; the UML Interface library has to be imported into the Java program. See the *Instructions* manual and the Java documentation for details on how to register and use the UML Interface as a Java library.

```
import UMLInterface.*;
…
IBase mbase = new MBase();
mbase.newClass("Class 1");
…
```

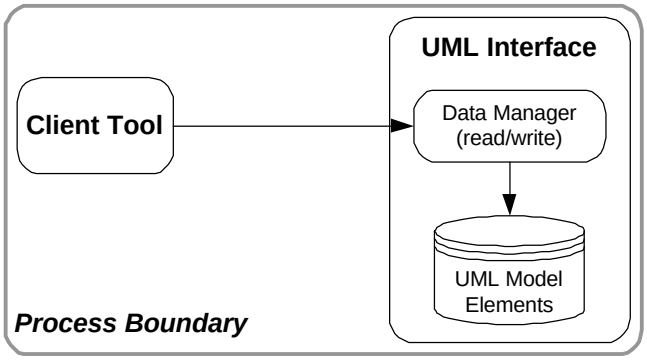


*Figure 4. Importing the UML Interface as a Java Library or COM/DLL Library results in both, the Interface and Client Tool to reside in the same process.*

### OPTION 2: IMPORTING THE UML INTERFACE AS A COM/DLL

This option is useful if the UML Interface should run in the same process as a client tool but the client tool is not able to directly access its Java library (see Figure 4). The Common Object Model (COM) is a middleware technology and supports the standardized interaction between computational components. The complexity of using COM varies and depends on the programming language. The following code shows how to create an instance of the UML Interface and how to add a class to it using the programming language Visual Basic (note: *UMLInterface* has to be selected in the list of available COM services).

```
set mbase = new UMLInterface.MBase
mbase.newClass("Class 1")
?mbase.getClasses
[Class 1]
```

Visual Basic hides most of the complexity of handling COM objects and its code is very simple. Using COM in other languages, such as C++, is less simple. Please consult respective language specifications on how to use COM.

### OPTION 3: REMOTELY ACCESSING THE UML INTERFACE USING COM

This option is useful when multiple client tools require access to the same UML interface (i.e., same UML elements) but it is infeasible to have all of them reside within the same process (see Figure 5). In this option, one client tool creates an instance of the UML Interface, using either one of the two previous options, and then registers the UML Interface with the UML Interface Server to

announce its public availability. Other client tools can then request registered UML Interfaces from the UML Interface Server.

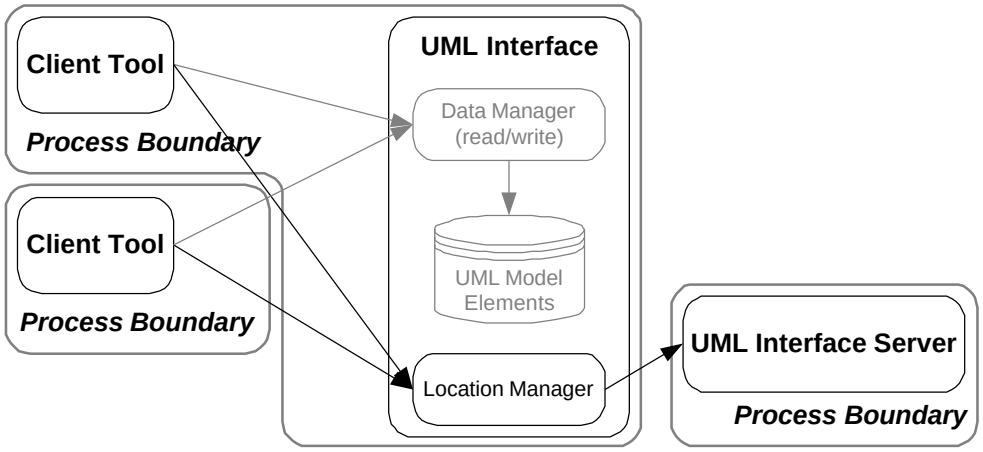


*Figure 5. If the UML Interface is registered with the UML Interface Server then Client Tools can locate and access it remotely.*

The WinLocationService class provides all the support necessary for client applications to communicate with the UML Interface Server. Methods are provided to register UML interfaces and retrieve lists of all registered interfaces and design tools both in-process (local) and out-of-process (remote). Additionally, the Base class provides simplified methods for in-process registration. Table 5 contains the complete list of support methods relevant to registration and access of local and remote interfaces and tools.

*Table 5. Administrative Access Methods in* Base *and* UML Interface Server *to support Local and Remote Accessing*

| LOCATION | METHODS |
|---|---|
| **Class** *LocationService* **in UML Interface Server** | **`public int registerInterface(MBase mb, Object design_tool);`**<br>returns the number of registered interfaces (mbases).<br>**`public int unregisterInterface(int id);`**<br>removes the interface identified by id from the UML Interface Server.<br>**`public Mbase getLocalInterface();`**<br>returns the instance of the local MBase.<br>**`public ISet getRegisteredInterfaces ();`**<br>returns the set of all registered interfaces.<br>**`public ISet getAllInterfaces();`**<br>returns the set of all interfaces. This is the union of local and registered interfaces.<br>**`public Object getLocalDesignTool();`**<br>returns the instance of the local design tool.<br>**`public ISet getRegisteredDesignTools();`**<br>returns the set of design tool objects registered with the UML Interface Server.<br>**`public ISet getAllDesignTools();`**<br>returns the set of all design tool objects. This is the union of local and all design tools registered in the UML Interface Server.<br>**`public MBase getInterfaceOfLocalDesignTool();`**<br>returns a handle to the MBase interface for the local design tool.<br>**`public ISet getInterfacesOfRegisteredDesignTools();`** |

| | |
|---|---|
| | returns the set of interfaces (mbases) for all design tool objects registered in the UML Interface Server.<br>**`public ISet getInterfacesOfAllDesignTools();`**<br>returns the set of all interfaces (mbases) for all design tool objects. This is the union of local and all design tools registered in the UML Interface Server.<br>**`public Object getDesignToolByID(int id);`**<br>returns the design tool instance identified by id registered in the UML Interface Server. Null is returned if the id is not found.<br>**public Object getInterfaceByID( int id );**<br>returns the interface from the server corresponding to the given id.<br>**`public Object getInterfaceByName(String name);`**<br>returns a handle to the mbase object (instance) of a given name. |
| **Class *Base***<br><br>**in UML Interface** | **`public void registerInterface();`**<br>registers itself with the UML Interface Server. This action makes the MBase visible to other client tools that may wish to access it.<br>**`public void unregisterInterface();`**<br>unregisters itself from the UML Interface Server. Usually done before shutdown.<br>**`public boolean getIsInterfaceRegistered();`**<br>returns true if the interface is registered with the UML Interface Server; otherwise returns false. |

UML Interfaces are registered with the UML Interface Server through the method *registerInterface* in the *Base* object. The following Visual Basic code illustrates:

```
set mbase = new UMLInterface.MBase
mbase.newClass("Class 1")
?mbase.getClasses
[Class 1]
mbase.registerInterface()
```

The above code creates a local instance of the UML Interface, obtains a lock, adds a class, registers the interface with the UML Interface Server and unlocks the interface.

The WinLocationService class is used to access interfaces registered with the UMLInterfaceServer. The following Visual Basic code sample illustrates one way to access the UML Interfaces from two separate, remote design tools (Rose applications):

```
Set ls = New WinLocationService
Set mb1 = ls.getRegisteredInterfaces().toSequence().get(0)
?mb1.getDesignToolName()
Rational Rose
Set mb2 = ls.getRegisteredInterfaces().toSequence().get(1)
?mb2.getDesignToolName()
Rational Rose
?mb2.getDesignToolDateTime()
1045848856002
?mb1.getDesignToolDateTime()
1045847663197
```

Notice that no local MBase is created. All that was needed was an instance of the WinLocationService class. From there, getRegisteredInterfaces returned an ISet that was converted to a sequence and the first and second items were indexed. Both design tools had the same name but their time and date stamps differed. Also, note that it is permissible to read simple identification

attributes without obtaining a lock. A lock is required to access the remote interfaces. The following code continues the above sample by adding a class to each remote UML Interface (mbase):

```
Mb1.newClass("Class 1")
Mb2.newClass("Class 2")
Mb1.getClasses()
[Class 1]
Mb2.getClasses()
[Class 2]
```

## DESIGN TOOLS: RATIONAL ROSE/MATLAB STATEFLOW

One of the prime benefits of the UML Interface is its ability to connect to design tools such as Rational Rose and Matlab/Stateflow. This connection allows client tools to be integrated with commercial design tools---for instance, to use the commercial design tools as a development environment and perform additional activities such as model checking, simulation, or transformation through connected client tools. While connected to those design tools, client tools can use the Data Manager to navigate among UML elements defined in them. This is accomplished by downloading UML elements from the design tools to the interface followed by accessing those elements as was discussed before. The UML Interface can be connected to a design tool in two different ways:

- connect the UML Interface as an add-in to the design tool
- connect a running instance of the UML Interface to a design tool

### OPTION 1: CONNECT UML INTERFACE AS AN ADD-IN TO DESIGN TOOL

Design tools such as Rational Rose and Matlab/Stateflow can use the UML Interface as an add-in. This option, depicted in Figure 6, has the interface reside in the same process as the design tool. If Java is used, then the client tool must also be located in the same process as the UML Interface and the design tool. If COM is used then the client tool may also reside outside the process of the design tool. Note: UML Interfaces that are add-ins to design tools register themselves automatically with the UML Interface Server.

Rational Rose:

Only Rational Rose has a build-in support for add-ins. Please consult the users manual for how to build such add-ins in Rational Rose (note: sample add-ins, such as the UMLModelBrowserJava, are distributed with the UML Interface). The UML Interface must be selected in the Rose Add-In Manager (read the *Instructions* manual for information on how to do so) as must client tools if they wish to reside in the same process.

Matlab/Stateflow:

Matlab/Stateflow does not have a build-in support for add-ins. To start up Matlab/Stateflow with the UML Interface in the same process, please use the start icon for Matlab/Stateflow distributed with this package. In order to have client tools reside in the same process, *MBase* provides a method called *instantiateThirdPartyTool*. To use this method, compile the client tool as a COM/DLL

(not COM EXE) and use its unique COM Name (not GUID) to instantiate an instance in the same process where the UML Interface is running. The following code could be used to start up a client tool:

```
set s = new UMLInterfaceServer.SupplierTools
set remote_mbase = s.getInterfaceInstanceObject("A Simple MBase 1")
set remote_tool = remote_mbase.instantiateThirdPartyTool("MyPackageName.MyToolName")
remote_tool.doSomething()
…
remote_mbase.disposeThirdPartyTool(remote_tool)
```

TIP: You should not use the remote mbase within the client tool except for instantiating the tool, calling its pseudo-startup method(s), and disposing it. Once the tool is instantiated within the process of the UML Interface it can then proceed to creating a local (non-remote) reference to mbase using the standard Java or COM constructors (e.g., *new MBase()*). Thereafter, the local mbase object may be used analogous the discussions in the previous sections.

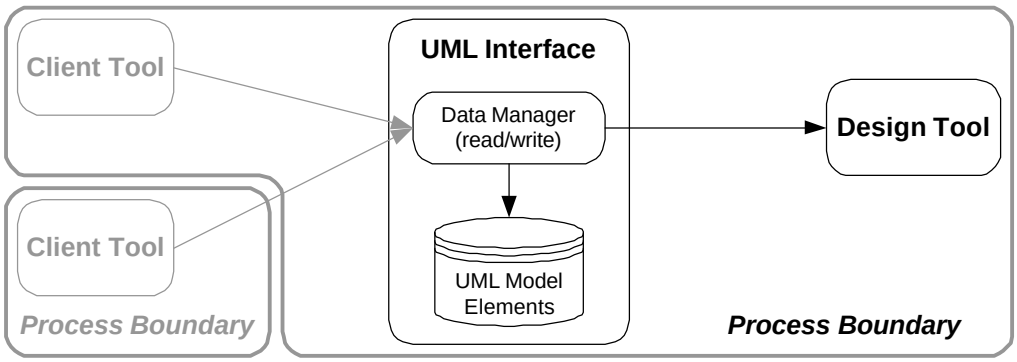


*Figure 6. Using Client Tool in-process with Design Tool*

A sample client, add-in tool for Rational Rose is the UML Model Browser for Java Application (UMLModelBrowserJava). It is distributed with this package (including source code) and the *Instructions* manual describes how to load and use it.

#### OPTION 2: CONNECT RUNNING INSTANCE OF UML INTERFACE TO DESIGN TOOL

If the UML Interface is created by a client tool (new MBase()) then both reside in the same process. A design tool can still be connected to the UML Interface with one restriction. Given that it is not possible to load a design tool into the process of an already running client tool it follows that both tools have to reside in separate processes (see Figure 7). Although this does not impact any interfaces (i.e., no code changes/recompilation is necessary), it does affect performance somewhat since inter-process COM calls are significantly slower than in-process COM calls.

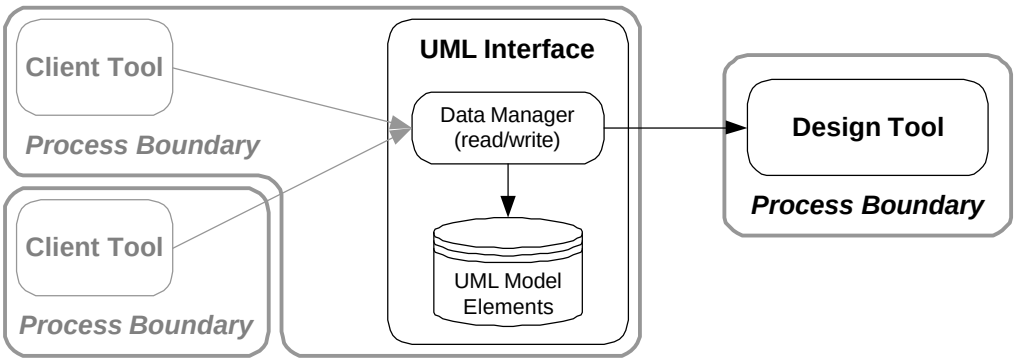


*Figure 7. Using Client Tool in-process with Design Tool*

To implement this option, create a Base object from within the client tool (new MBase()) followed by calling one of the following two methods: *createDesignTool* or *findDesignTool*. The method *createDesignTool* takes the design tool name as a parameter and then creates a new instance of that design tool. For instance, the method call *mbase.createDesignTool("Rational Rose")* creates a new instance of Rational Rose, connects that instance to the UML Interface, and returns true if successful. If the design tool name is null then the method displays a list of design tool choices plus existing design tools registered in the UML Interface Server. The method *findDesignTool* with its parameter being null attempts to locate running instances of any design tool (either Rational Rose or Matlab/Stateflow). If it finds only one such instance then it connects it to the UML Interface and returns true; if it finds two or more such instances then it displays a dialog box to let the user choose the instance to connect to. If no running instance can be found then the method returns false. If the parameter of *findDesignTool* is not null then it will select/display those design tools whose names match. For instance, *mbase.findDesignTool("Rational Rose")* looks for only those registered UML Interfaces that are connected to an instance of Rational Rose (note: remember that design tools always register with the UML Interface Server).

Alternatively, a client tool may choose to create or locate a design tool independently in which case the client tool may use the method *setDesignTool* to connect the interface with the design tool manually.

Regarding UML Interface Server:

Design tools using the UML Interface as add-ins (default) always register themselves with the UML Interface Server using their proper names as references ('Rational Rose' and 'Matlab/Stateflow'). The method *findDesignTool* thus always finds all running instances of Rational Rose and Matlab/Stateflow even though they were not told to register explicitly. If a running interface is connected with a design tool (using methods *findDesignTool* or *setDesignTool*) then the interface will not register itself with the server automatically. Likewise, interfaces that are not connected with any design tools will also not register themselves. In both cases, however, the interfaces may be registered manually using the *registerInterface* method discussed earlier.

*Table 6. Administrative Access Methods in* Element *and* Base *to support Connectors to Design Tools*

| LOCATION | METHODS |
|---|---|
| Class *Element* | **`public void reset();`**<br>resets the attributes and relationships of the element to 'unknown' values. |

| | |
|---|---|
| **in UML Interface** | **`public void cache();`**<br>downloads the current attribute and/or relationships of elements from design tool. |
| **Class *Base***<br><br>**in UML Interface** | **`public String getDesignToolName();`**<br>returns the name of the connected design tool (e.g., *Rational Rose*).<br>**`public String getDesignToolDateTime();`**<br>returns the creation date and time of the connected design tool.<br>**`public String getDesignToolObject();`**<br>returns the COM object of the connected design tool.<br>**`public String getModelName();`**<br>returns file name(s) of currently open model files.<br>**`public void setDesignTool(Object tool);`**<br>connects the interface with a design tool.<br>**`public boolean createDesignTool(String designToolName);`**<br>creates a new instance of a design tool. Parameter should be either 'Rational Rose' or 'Matlab Stateflow'. Returns true if successful.<br>**`public boolean findDesignTool(String designToolName);`**<br>finds a currently running instance of design tool that matches the given tool name. Displays a list of running design tools in a dialog box if multiple tools are found. Otherwise does not display a dialog box. Returns true if a design tool is found.<br>**`public boolean getIsDesignToolAlive();`**<br>probes the design tool to see whether it is still alive and responding. Return true if design tool is alive.<br>**`public ISet getSelectedElementsFromDesignTool();`**<br>returns a collection of all model elements that are currently selected in the design tool.<br>**`public void lockDesignTool();`**<br>**`public void unlockDesignTool();`**<br>locks and unlocks the design tool to block/allow user input (i.e., mouse clicks and keyboard strokes are nullified of the design tool is locked). NOTE: USE THE METHODS *lock* AND *unlock* since they are safe for concurrent use. The methods *lockDesignTool* and *unlockDesignTool* should only be used to overwrite *lock* and *unlock*.<br>**`public boolean getCaching();`**<br>**`public void setCaching(boolean caching);`**<br>fully downloads data from a given design tool if true; otherwise only downloads critical parts and then incrementally downloads additional data if needed. |

Once the interface is connected to a design tool, its methods may be used to navigate among the elements of the design tool. NOTE: CURRENTLY ONLY CLASS, USE CASE, SEQUENCE, AND STATECHART DIAGRAMS IN RATIONAL ROSE CAN BE CACHED AND NAVIGATED; AND ONLY STATECHART DIAGRAMS IN MATLAB/STATEFLOW. FURTHERMORE, THE CONNECTION WITH RATIONAL ROSE AND MATLAB/STATEFLOW ARE PRIMARILY REAL ONLY. ALTHOUGH NEW MODEL ELEMENT CAN BE CREATED IN THE INTERFACE, THOSE CHANGES WILL NOT BE UPLOADED INTO THE CONNECTED DESIGN TOOL. THERE ARE A FEW EXCEPTIONS DISCUSSED LATER.

Since downloading design data can be expensive computationally (i.e., requires many inter-process COM calls), the UML interface supports various options on how to download data from design tools. The simple option is to download the complete model from the design tool to the interface. The method *setCaching(true)* must be called to switch on full caching (only necessary if *getCaching()* is not true). Then the method *cache()* has to be called to completely download all design data. If *getCaching()* is not true then incremental caching is performed where design data is only downloaded once requested. For instance, if *mbase.getClasses* is called then classes are downloaded without downloading (most of) their attributes or inter-relationships. If next the client dependencies

of that class are required (*mbase.getClasses().toSequence().get(0).getClientDependencies()*) then only the client dependencies of that class are downloaded. Incremental caching is the default installation option and it is generally preferable over full caching. The downside of incremental caching will be discussed in the section about the Change Notification Manager.

getIsStandalone

getIsConnectedToDesignTool

getIsInProcessWithDesignTool

## CHANGE NOTIFICATION MANAGER

*As various events occur in the UML Interface, the Change Notification Manger notifies registered client tools by broadcasting event messages. Individual client tool can enable or disable change notification. Change notification, when enabled, is accomplished by two techniques, polling and automatic notification. The polling method requires a client tool to register with the Change Notification Manager and request messages as the registered client determines. The automatic notification technique requires the client tool to have and register an observer object that extends MObserver and implements the notify method. When messages for the registered client tool are received, the Change Notification Manager automatically notifies the target client tool using the notify method from the observer object registered. Table 6 lists all the Base class methods required to register client tools for automatic notification or individual message requests. What follows is a description of how to implement these two notification techniques. Table 7. Administrative Access Methods in* Element *and* Base *to support Change Notification*

| LOCATION | METHODS |
|---|---|
| **Class *Element*** **in UML Interface** | **`public void detectChange();`** detects changes in attributes and/or relationships by comparing cached UML elements in the UML interface with UML element in the design tool. differences indicate changes. |
| **Class *Base*** **in UML Interface** | **`public int registerClientTool( String tool_name )`** registers the tool_name with the Change Notification Manager. It returns the registration identification number if successful or –1 if registration was unsuccessful. **`public int registerClientTool( String tool_name, Object tool_obj )`** registers the tool_name and tool's observer object with the Change Notification Manager. It returns the registration identification number if successful or –1 if registration was unsuccessful. **`public void unregisterClientTool( int tool_id )`** un-registers the tool from the Change Notification Manager. **`public void sendMessage( int origin_id, int type, String desc, Object ref )`** sends a message to all registered tools. **`public void sendMessage( int origin_id, int dest_id, int type, String desc, Object ref )`** sends a message to the specified registered tool. **`public boolean getIsMessageAvailable( int tool_id )`** returns true if a message is available for the tool identified by the tool_id. |

| | |
|---|---|
| | **public WinMessage receiveMessage( int tool_id )**<br>retrieves the next available message for the tool identified by the tool_id. If no message is found, an empty message is returned.<br>**public void catchUp( int tool_id )**<br>brings the client tool messages up to date.<br>**public int getClientToolID( String name )**<br>returns the tool identification number for the name provided. It returns –1 if the name was not found.<br>**public Vector getClientTools()**<br>returns the list (vector) of registered client tools.<br>**public boolean getChangeNotification()**<br>returns the status of change notification for this instance of MBase.<br>**public void setChangeNotification(boolean changeNotification)**<br>sets the status of change notification for this instance of MBase. When set true, the cached data is compared with the latest data in the design tool. Differences are downloaded to update the cache and then reported to interested client tools if requested. |
| **Class** ***WinMessage*** **in UML Interface** | **public int getID() / public void setID(int id)**<br>get/set the message identification number.<br>**public long getIime() / public void setTime(long time)**<br>get/set the system time in milliseconds that the message was created.<br>**public int getOrigin() / public void setOrigin(int origin)**<br>get/set the identification number of the message sender, -1 for unknown.<br>**public int getDestination() / public void setDestination(int destination)**<br>get/set the identification number of the message receiver, -1 for unknown.<br>**public int getType() / public void setType(int type)**<br>get/set the message type, -1 for unknown.<br>**public String getDescription() / public void setDescription(String desc)**<br>get/set the message description.<br>**public Object getReference() / public void setReference(Object o)**<br>get/set the instance of the changed object. It can be null. |

The following Java code illustrates how to implement automatic notification:

```
import UMLInterface.*;

public class Example2
{
    public Example2()
    {
        initExampleObserver();
    }

    private void initExampleObserver()
    {
        IBase mbase = new MBase();
        mbase.findDesignTool( "Example2" );
        IObserver exampleObserver = new ExampleObserver( );
        mbase.registerClientToolNameAndObject( "Example2", exampleObserver );
    }
……
```

An Observer just looks like this:

```
public class ExampleObserver implements IObserver
{
```

```
    public void notify(String toolName, int notificationCode, String notificationMessage, Object
notificationObject)
    {
        // just some examples what information you can get
        String whatHappened = "nothing interesting";
        String affectedElementId = "no interesting id";
        if ( notificationCode == MBase._mbase.MSG_NEW_MODEL_ELEMENT() )
        {
            whatHappened = "new model element";
        }
        else if ( notificationCode == MBase._mbase.MSG_MODIFIED_MODEL_ELEMENT() )
        {
            whatHappened = "modified model element";
        }
        if ( notificationObject instanceof UMLInterface.IClass )
        {
            affectedElementId = ( ( UMLInterface.IElement ) notificationObject ).getID();
        }
        else if ( notificationObject instanceof UMLInterface.IState )
        {
            affectedElementId = ( ( UMLInterface.IElement ) notificationObject ).getID();
        }
    }
```

There is a second possibility to receive change notification from the UMLInterface; to start a thread, which regularly asks the UMLInterface, if a change has been occurred:

```
    import UMLInterface.*;

    public class Example
    {
        public Example()
        {
            new ExampleThread().start();
        }

        public static void main (String[] args)
        {
            new Example();
        }
    }

    class ExampleThread extends Thread
    {
        private int observerID = -1;

        public void run()
        {
```

```
        UMLInterface.IBase mbase = new UMLInterface.MBase();
        boolean isRoseAlive = mbase.findDesignTool( "Rational Rose" );
        if ( isRoseAlive )
        {
            MBase._mbase.writeToLog("Alive");
            // we have to register our tool at MBase to receive notification messages
            // we receive an ID; we use this ID to ask MBase for messages
            observerID = MBase._mbase.registerClientToolName("your tool name");
        }

        while ( isRoseAlive )
        {
            if ( MBase._mbase != null )
            {
                observeRose( );
            }
            try {
                Thread.sleep( 150 );
            } catch ( Exception ex ) {
                // some exception handling
            }
        }
}

/**
 *  Checks if MBase provides new change notification messages for us
 *  If yes, we read the message and write the description, e.g. "123XYZ-MClass"
 *  to the Rose log window.
 *  What happened, e.g. new or deleted model element, tells us getType(). It returns
 *  an int code, the meaning of each value can be found in MBase,
 *  e.g. "public int MSG_NEW_MODEL_ELEMENT() { return 30; };", etc
 */
private void observeRose()
{
    try
    {
        // check if change notification is enabled
        if ( MBase._mbase.getChangeNotification() )
        {
            // ask, if Rose has a message for us
            if ( MBase._mbase.getIsMessageAvailable( observerID ) )
            {
                WinMessage msg = MBase._mbase.receiveMessage( observerID );
                Object o = msg.getReference(); // affected element
                if ( o instanceof UMLInterface.IClass )
                {
                    // doSomething()
                }
                else if ( o instanceof UMLInterface.IState )
                {
                    // doSomethingElse()
                }
                // ... etc
                int type = msg.getType(); // type of change
```

```
                    if ( type == MBase._mbase.MSG_NEW_MODEL_ELEMENT() )
                    {
                        // doSomething()
                    }
                    else if ( type == MBase._mbase.MSG_MODIFIED_MODEL_ELEMENT() )
                    {
                        // doSomethingElse()
                    }
                    //... etc
                    String message = msg.getDescription();
                    MBase._mbase.writeToLog(message);
                }
            }
        }
        catch ( Exception ex )
        {
            // some exception handling
        }
    }
}
```