Interface Definition Modeling Language

User's Manual

Jeff Parsons

Introduction

Interface Definition Modeling Language (or IDML, as it will be called hereafter) is a graphical modeling language for the building blocks of component applications, that is, applications that are built on top of component middleware. It isn't intended to represent, or conform to the specification of, any particular kind of middleware or any particular vendor's product. Rather, it's an environment in which to design and modify component applications in a way that's technology-neutral, in a way that can be translated into the middleware flavor of your choice. One such translator has already been written, and is described in detail at the end of this document. More are in the works.

IDML itself was designed using the Generic Modeling Environment (GME), a powerful modeling tool developed at the Institute for Software Integrated Systems (ISIS) at Vanderbilt University. Models in IDML are also constructed using GME (I told you it was powerful), since IDML itself can be registered with an installed copy of GME and then selected from a list of modeling languages when starting a new model project. Documentation, download and other information can be found on the GME web page at:

http://www.isis.vanderbilt.edu/Projects/gme/

Although many GME-specific details will be explained as they come up in this manual, it is not intended to be both an IDML manual and a GME manual, so we'll refer to the GME manual whenever appropriate or necessary. Familiarity with GME is not a requirement for using IDML, but it couldn't hurt, at least the part of GME where models are created with an existing modeling language (of which IDML is one). The part of GME where you create a modeling language from scratch is not so relevant to the use of IDML, but interesting for those who are curious about how IDML was created.

But, you may point out, modeling interfaces, components, and associated data types is not all there is to an application. What about connections, interactions, configuration and runtime issues? These things are admittedly not handled by IDML, but in addition to its standalone use, IDML is also integrated with another modeling language called Platform Independent Component Modeling Language, or PICML, to use the inevitable acronym. PICML in turn is part of a suite of modeling tools that cover the entire spectrum of component

- definition – IDML, described in this document
- packaging – getting actual software artifacts ready to install

- configuration – tweaking the “knobs”, the various options available
- assembly – creating the desired connections between components
- deployment – getting component packages to the right place and starting them up

The name of the tool suite that does all this is Component Synthesis with Model Integrated Computing or (you knew this was coming) CoSMIC. CoSMIC is an ongoing project, and lots of information about it and the great folks who are developing it (including yours truly) can be found at:

http://www.dre.vanderbilt.edu/cosmic/

The key word in the last sentence is “ongoing” - all the pieces of the CoSMIC tool suite, including IDML are constantly being improved, extended, maintained, and integrated with other tools. With IDML in particular, the “platform independent” claim can be justified by providing translators for IDML models into various middleware platforms. So far there is only one, a CORBA IDL generator, described, as mentioned above, in the last section of this document. Planned future translators include:

- a generator for SLICE. Unraveling the acronym, that’s Specification Language for ICE (Internet Communications Engine), developed by ZeroC, info at http://www.zeroc.com/
- a generator for Enterprise Java Beans (EJB) classes
- an importer for CORBA IDL (okay, just for the sake of completeness, that’s Common Object Request Broker Architecture Interface Definition Language)

Model Building

Getting Started

GME Features

IDML was developed using GME 4, specifically version 4.5.18, so references to the GME User’s Manual will relate to the GME 4 version of the manual. Most, if not all, of these references will be general enough not to become inaccurate or obsolete, at least not any time soon.

When you start up GME, you’ll see its window divided into four panels. They are called, starting from the top left and moving clockwise:

- the Model Editor
- the Model Browser
- the Attribute Browser
- the Part Browser

We’ll use these terms throughout this manual. For more information, see the *Using GME 4* section of the GME manual.

First-Time Steps

Obviously the first thing to be done is to install GME. GME uses InstallShield and the process is a no-brainer. If you are updating to a newer version of GME, be sure to remove the old version first.

Next, IDML has to be "registered" with the new installation of GME, that is, IDML must be added to the list of modeling languages stored in GME's database, so it will be on the list of possible modeling languages to select when starting a new model. To register IDML, it first has to be loaded into GME. This can be done from a binary file having the extension .mga, but this is version-dependent. In other words, .mgs files saved with one version of GME cannot be loaded by another version. A slower, but safer, way to load IDML is from its corresponding XML file, having the extension .xme. This file for IDML is distributed under the name IDML-Meta.xme.

1. Go to the File menu in GME and select Import XML.
2. In the window that pops up, find IDML-Meta.xme on disk and select it.
3. If GME then prompts you with a message box about "metaGME" (the language used in GME to write other modeling languages), just click OK.
4. Then a window will pop up entitled "Import to new project". Select "Create new project" and click Next.
5. Then you'll see any .mga files that are present in the directory you loaded IDML-Meta.xme from. You can select one of these, if there are any (be careful, if the .mga file you select is storing another project, it will get replaced), or type in a name of your own choosing.
6. After the project is loaded and you zap the "loaded successfully" message box, look at the toolbar along the top of the Model Editor. Second from the right is an icon that looks like a flower. Left click on it.
7. In the "Save as" dialog box that pops up you should see a file called IDML.xmp (yet another form of XML that GME calls an XML Paradigm File). Select it, or if it is not there, type the name in yourself. Click on Save.
8. If IDML.xmp was already there, you'll be asked if you want to replace it. Click Yes.
9. Then you be asked if you would like to register your new paradigm. Click Yes again, and zap the "completed" message box when it appears.
10. Finally, go to the File menu and select Close Project.

Unless and until you reinstall GME, you won't have to do this again.

Three Ways to Start a New Model

Which way you choose depends entirely on how you want to deal with predefined types. More on those later, in the Model Elements section, but suffice it to say for now that they are the most simple and basic of types and they already "exist" as opposed to the types you will be creating in your model. They are already a part of IDML, but an optional part, as you will see.

The most likely way you'll want to deal with predefined types (I'm assuming, but I've allowed for alternatives since I'm not 100% sure) would be having them already present in the model you start up, ready for use, and not modifiable (not possible to add more or delete any). If this is what you want, I have just the thing for you. Instead of creating a new model from scratch, you can start with a provided model that has the predefined types already loaded. Again, to avoid GME version concerns, it’s best (at least the first time) to import the model from XML.

1. From GME’s File menu, select Import XML.
2. In the same directory where you found IDML-Meta.xme and IDML.xmp, you’ll see a file called IDMLTemplate.xme. Select it.
3. If GME then prompts you with a message box about “metaGME” (the language used in GME to write other modeling languages), just click OK.
4. Then a window will pop up entitled “Import to new project”. Select “Create new project” and click Next.
5. You’ll then be prompted to name your new project as an .mga file. I suggest using the same name, but it’s up to you.
6. You’ll probably get a “Could not locate paradigm…..” message. Just click OK, and zap the “successfully loaded” message box that appears.
7. Go to File->Save Project As” and save in the directory and under the name of your choice.
8. You may want to change the name of the top-level element in the Model Browser as well. Left click on it and edit it in the Attribute Browser (this is where you should edit the names of all model elements). No need to save again. If you select File->Close Project before closing GME (another thing you should always do), your model will be saved automatically.

Expand the top-level model element in the Model Browser. Now, just underneath it, you’ll see a little purple book-like icon labeled “MGA=<some path + filename>.mga. Expand this, and you see a blue folder-looking icon (representing a GME Folder, see the GME manual if you’re curious) called PredefinedTypes. Expand this, and you’ll see all the predefined types in IDML, their little GME Atom icons preceded by an “L” to indicate that this is a read-only library that has been imported into the model. These predefined types can be dragged (or copied and pasted) from here into the model you create. The details on this action are in the Model Elements section of this manual, as is an explanation of the NewInterfaceDefinitions folder you are also seeing there.

If you’re the do-it-yourself type, or are just curious about how the IDMLTemplate model was created in the first place, you can create a model from scratch and import the predefined types yourself. Here are the steps:

1. Select File->Import XML
2. In the dialog box that appears, go to the directory where you loaded IDML-Meta.xme from.
3. Select PredefinedTypes.xme and follow the same steps for importing an .xme file and saving it as an .mga file that you did for IDML.
4. Select File->Close Project
5. Select File->New Project

6. From the Select Paradigm dialog box that appears, select IDML, then Create New.
7. From the New dialog that appears, select “Create project file” and Next.
8. In the Open dialog box that appears, select a project name and directory, and click Open.
9. Right click on the top-level element in the Model Browser, and select Attach Library.
10. In the Attach Library dialog box that appears, click on the “…” button to the right of the text box.
11. Browse to the .mga file you saved in step 3.
12. Select it, click Open, then click OK in the Attach Library dialog box that reappears.

Adding a NewInterfaceDefinitions folder will make this model exactly like the one you adapt from IDMLTemplate. We’ll cover this step in the Model Elements section,.

If, for whatever reason, you decide you want to use some of the IDML predefined types, but not all, there is a third way to go.

1. Follow steps 5 through 8 in the list immediately above.
2. Right click the top-level element in the Model Browser of the new project.
3. Move the cursor down to Insert Folder in the popup menu and selection PredefinedTypes
4. Right click on the NewPredefinedTypes folder that appears in the Model Browser.
5. Move the cursor down to Select Atom in the popup menu and select a predefined type.
6. Repeat steps 4 and 5 until done.

Later on, if you change your mind about what predefined types you need or want in this project, you can delete them. You may also insert a predefined type more than once, or insert any number of NewPredefinedTypes folders (each one can be renamed as well). Doing any of these things will cause no harm, but seems useless (or maybe I just lack imagination). Preventing these actions in the design of IDML would have been a lot of work, too much, it was decided, for an unlikely action in an uncommon use case. So if you want to add a subset of IDML’s predefined types, or leave yourself the option of deleting some of them later, the responsibility is on you to do sensible things with this third way to create an IDML model.

Building and Editing a Model

In the Model Elements section that follows this one, we’ll cover what each model elements represents and its intended use in IDML models, but in this section, we’ll cover more general issues in building and editing IDML models that don’t apply to any one model element.

Icons and References

The icons used in IDML are hopefully self-evident enough to help you understand what the types are and what they can be used for. Many icons have two versions. When a type is declared/created, you see its basic icon, and when it is used as a reference, there will be a curved arrow in the lower left corner of the icon, similar to what Windows uses to indicate a Shortcut. Icons for predefined types don't have two versions, since predefined types are a special case because they aren't "declared' in the usual sense. In almost all cases, the correct version of the icon will be used automatically – you will rarely have to think about it, and when you do, the right action will be obvious.

Adding Items from the Part Browser

New types are declared by dragging the appropriate icon from the Part Browser into the Model Editor. Some model elements in GME can contain others, and in such cases, you get to the "scope" by double clicking on the model element's icon in the Model Editor. Once "inside", only the icons for the model elements that can legally appear in this "scope" are seen in the Part Browser.

Some types in IDML can contain one or more "members", that is, elements which have a name but are not a declaration of a new type. Rather they are references to a predefined type or to a type declared elsewhere in the model. Members can be dragged in from the Part Browser as well; a generic Member icon appears in the Part Browser when appropriate. However a member added in this way does not yet refer to anything, and is not legal until we drag an existing type on top of it from the Model Browser (more details about this in the next subsection).

When an item is dragged in from the Part Browser, its name in the Model Editor is initially the same as its name in the Part Browser. This name also appears at the top of the Attribute Browser and can be edited there. In fact, names should always be edited in the Attribute Browser, even when the item is selected in the Model Browser. Slowly double clicking on a name in the Model Browser will highlight it for editing, the same as with any Windows icon. However, in GME, editing a name in this way can have unpredictable results – best to avoid editing names this way and always do it in the Attribute Browser.

There are other types of elements which can be dragged in from the Part Browser, and which are neither new types nor members. These items will be described in detail in the next section, but one thing about them is worth mentioning here. These items will have a name in the Part Browser, so we can speak and write about them, but will have no name displayed in the Model Editor, because doing so would be either (1) redundant or (2) downright problematic. Leaving the icon in the Model Editor nameless is the way IDML informs you that you need not edit the name of this item. It will still appear in the Attribute Browser when the item is selected, but editing it will have no effect. A name for such an item will not be imported from any middleware platform by any future importer, and it will not be output anywhere by any generator. This may seem overly complicated at this point, but it was motivated by the goal of making IDML as self-evident and easy to use as possible. Each case where the name is not displayed will be explained in the Model Elements section, and the motivation for having this feature will become clearer.

Using the Model Browser

As mentioned in the previous subsection, a generic member (one that does not yet refer to an existing type) dragged in from the Part Browser has its reference assigned by dragging the reference in from the Model Browser. Since it's a reference we are dragging, it is done a little differently. To paste an item as a reference in GEM, you can either drag while holding down Ctrl+Shift, or

1. right click on the item in the Model Browser
2. move the cursor down to Copy in the popup menu and left click
3. right click in the Model Editor where you want the item pasted
4. move the cursor down to Paste Special in the popup menu and select As Reference from the list that pops up

When you drop a reference on a generic member icon, it will be replaced by the reference version of the icon corresponding to the type being dropped (or just the icon if you are dropping a predefined type).

A reference may be dropped in this way on other kinds of model elements besides members. In such cases, the icon displayed will not change, since it is not generic and conveys information to you. It is always possible, however, to tell what is being referred to (or if the reference has not yet been assigned) by checking the bottom line of the Properties tab in the Attribute Browser.

There is an alternate way to accomplish the actions described in this subsection, one that is quicker but not as intuitive. You can drag or paste in something as a reference from the Model Browser without first dragging anything in from the Part Browser. If there is more than one choice for the type of model element that will be doing the referring, GME will pop up a window and let you select. You will also get an error message from GME if you try to bring in something illegal from the Model Browser.

Constraint Checking

As much as possible, rules of modeling middleware (selected because they seem to be common to many if not all kinds of middleware and programming languages) have been built in to the "grammar" of IDML so that, at best, wrong choices don't even appear, at worst you get an error message from GME as soon as you attempt something illegal. However, it's almost impossible, certainly very impractical, to catch every error in the grammar of any language, let along a graphical one. Therefore, IDML has many constraints built in, so that you will still find out as soon as possible if something is illegal in your model. A constraint, wherever possible, is checked automatically when an action is taken that could violate it. In such cases, you'll find out immediately if something illegal is being attempted – you'll see GME's informative constraint violation window and the action will not be allowed. In some cases, such automatic constraint checking is not possible or would slow down GME too much for some apparently simple action, and so these constraints much be checked manually. It's a good idea to just run the Constraint Manager every so often when building a model. Clicking on the check mark icon at the left of the toolbar above the Model Editor will check constraints for all items in the active Model Editor window, or for a single item if it is selected. Selecting

File->Check->Check All will evaluate all applicable constraints for every item in the project.

General Constraints in IDML

Many of IDML's constraints are specific to one or to a few types, and will be covered in the next section each with its associated type, but below is a list of constraints that are generally applicable.

- **Legal names –** names (the character strings you edit in the Attribute Browser) must begin with an alphanumeric character and contain no other characters except an underscore.
- **Unique names –** Type declarations, member names and operation parameter names must be unique in a given scope. All declarations at the top level inside Files are considered to be at a single, global scope.
- **Non-null references –** If an item is a reference, it must refer to something.

Model Elements

In this section, each model element is described, in alphabetical order, to give you an idea of why it is part of IDML and what it represents. Some of the IDML model elements have GME attributes. Most of these GME attributes apply to a single model element, and will be explained as they come up. However, there are three GME attributes that apply to several model elements. They act as tags for the model elements where they are set, and may be used by a GME interpreter when translating the model to help deal with unique id and versioning issues. Use of these tag attributes in your mode is completely optional, and they all default to empty strings.

Tag Attributes

PrefixTag – This string can be prepended to an id string to help ensure uniqueness, avoid name clashes, or just provide extra information. It could contain something specific to a particular vendor, for example. It applies to the following model elements:

- File
- Package
- Object
- ValueObject
- Event

VersionTag – This type of tag might be used as part of an id string or by itself to help with versioning issues. Although it is processed by IDML as a string, it must have the form of a base 10 number with a single decimal point. It applies to the following model elements:

- Alias
- Aggregate

- Attribute
- Boxed
- Collection
- Component
- ComponentFactory
- Constant
- Enum
- Event
- Exception
- FactoryOperation
- InEventPort
- LookupOperation
- Object
- OnewayOperation
- OutEventPort
- Package
- ProvidedRequestPort
- ReadonlyAttribute
- RequiredRequestPort
- TwowayOperation
- SwitchedAggregate
- ValueObject

SpecifyIdTag – Setting this GME attribute will, if an IDML model interpreter deals with id strings, completely specify the string to be generated. Setting this attribute will override PrefixTag and VersionTag (if the latter is generated as part of an id string). It applies to the same model elements as VersionTag.

Aggregate

Represents a data type that can store dissimilar things. In IDML it contains Members and cannot be empty. Member types may appear repeatedly in the same Aggregate, but Member names must be unique.

Alias

Aliases the name of one type with another name. Thereafter both type names may be used interchangeably. Alias is a reference type in IDML, so assigning the type it refers to is done as described in *Using the Model Browser*. Its icon is not replaced as it is for Members, however, since the original icon tells us that this is an Alias declaration and not something else.

Attribute

Can be declared only inside an Object, ValueObject, Event, Component, or ComponentFactory. An Attribute has a type (a predefined type or one declared elsewhere in the model) but it is not a simple reference as Alias, Boxed and Collection are. Here's why: an Attribute, when translated through however many steps, to a programming language, end up as a pair of set/get operations. So in the model, we can associate references to Exceptions with these operations, even though they don't exist as such in the model (see SetException and GetException). Because of this extra baggage, Attribute is a container, containing exactly one AttributeMember and zero or more SetExceptions and GetExceptions.

AttributeMember

Same as Member except that, since there is only one in any given Attribute or ReadonlyAttribute and needs no name of its own, there is no name displayed when it is dragged into the Model Editor. AttributeMember has the same icon as Member.

Boxed

Like Alias, Boxed is a reference in IDML, and its icon is not replaced when the reference assignment is made. However, Boxed does something different with the type it refers to – it gives the type its own little custom "namespace" with the name of the Boxed declaration. For instance, you could make several Boxed declarations of a String, and each of these declarations, unlike an Alias, is a distinct type when translated to some programming language. Since most mappings use a type corresponding to the IDML ValueObject (see below) for the "namespace", it would be redundant to "box" a ValueObject. In IDML it is illegal.

Collection

Represents a data type that can store an unbounded number (including 0) of similar things. In IDML it is a reference, referring to the single type of thing it can contain. Its icon is not replaced when the reference assignment is made.

Component

These are (as you might have guessed from the name) the centerpieces of component applications. A Component contains ports (which are all references) but no types can be declared inside it. A Component's main function is to tie together and organize the features of the Objects and other types it uses.

A Component may optionally inherit from a single Component. It may also optionally support any number of Objects, but a Component may not both inherit and support. Since a Component does not have the GME attributes 'local' or 'abstract', it cannot support any Objects that have either of these attributes set to TRUE.

A Component may have Attributes and/or ReadonlyAttributes. It may also optionally be 'managed' by a ComponentFactory (see ComponentFactory), represented by a green connection in IDML.

The main feature of a Component is its ports. These come in four flavors – ProvidedRequestPort, RequiredRequestPort, InEventPort and OutEventPort. See the definitions of these IDML types for more details about each port type. Any ports a Component may have are displayed as GME ports inside the Component icon in the Model Editor (see the GEM manual for more info about GME ports), as miniature versions of the regular port icons. When IDML is used in integration with PICML, connections can be drawn between ports in the various Components in the model. These connections can be between ProvidedRequestPorts and RequiredRequestPorts, and between InEventPorts and OutEventPorts. In the former case, the type of Object referred to by each endpoint port of the connection must be the same, for the latter case, the type of Event referred to must be the same.

The four types of ports in IDML can be conceptually divided into two pairs of types – the request ports (with ‘provided’ and ‘required’ varieties) and the event ports (with ‘in’ and ‘out’ varieties). The request ports are conceptually connected one-to-one (even though a RequiredRequestPort may have multiple connections, they are separate connections, each with two endpoints, and the ‘requests’ are still sent out one at a time in the application). The event ports, on the other hand conform conceptually more to a publish/subscribe mechanism. Events are ‘published’ anonymously by an OutEventPort, and ‘consumed’ anonymously by any InEventPort that ‘subscribes’ to that type of Event. In this context, ‘anonymous’ means that one end has no idea who is at the other end, unlike with request ports, where there are binary ‘connections’ in which each end must know about the other end before anything can happen.

ComponentFactory

A Component may optionally be ‘managed’ by a ComponentFactory, although if a ComponentFactory exists in the model, it must ‘manage’ a Component. This is indicated in IDML by a connection and represented by a green line. If the ComponentFactory and its managed Component are declared in the same Model Editor screen, they can be connected directly. If they do not appear on the same screen (in the same ‘scope’), a ComponentRef icon can be dragged in from the Part Browser instead, a connection drawn from the ComponentFactory, and the Component that the ComponentRef refers to assigned in the manner described in the *Using the Model Browser* subsection earlier in this document.

A ComponentFactory may optionally inherit from a single ComponentFactory, and may also support any number of non-abstract, non-local Objects, similarly to a Component. Unlike a Component, however, a ComponentFactory may define its own operations, and declare any new types that would be legal to declare in an Object, ValueObject, or Event. It may also have Attributes and ReadonlyAttributes.

A ComponentFactory ‘manages’ a Component in an application by either creating an instance of the Component type from scratch, or by retrieving it from a repository, perhaps after first looking it up in some sort of database. To help with the latter task, a ComponentFactory may optionally contain a LookupKey, which is a reference to a ValueObject. For additional help with looking up a Component, a ComponentFactory may contain any number of LookupOperations. To create a Component from scratch, a ComponentFactory may contain any number of FactoryOperations.

ComponentRef

A reference to a Component, used when a ComponentFactory ‘manages’ a Component that does not appear in the same Model Editor screen (conceptually, not in the same ‘scope’). The Component it refers to is set as described in *Using the Model Browser* subsection earlier in this document.

Constant

A type not found on all middleware platforms, but included in IDML to facilitate the importing of CORBA IDL. Constants may be of the following types:

- Boolean
- Byte
- String
- ShortInteger
- LongInteger
- RealNumber
- Enum

Constants are references in IDML, so the type of the constant is assigned in the manner described for references previously in this manual. When the reference assignment is made, the Constant's icon does not get replaced as it does for assignment to Member icons, since we still want to know at a glance that this thing is a Constant, and not something else.

The value of the constant is entered in the Attribute Browser, and IDML constraints check that the value entered matches the type and, as far as possible, the range of the type of the constant declaration.

Discriminator

Contained by a SwitchedAggregate, the Discriminator refers to the type that the Label (see below) values must have. A Discriminator may be of the following types:

- Boolean
- ShortInteger
- LongInteger
- Enum

Enum

The same as the 'enum' type you'll find in many programming languages, except that you can't make explicit assignments to the Enum's contained values.

EnumValue

The only things that can be contained in an Enum.

Event

Events are very similar to ValueObjects, as the icon reflects. Events are used in a special way by Components, and in actual middleware implementations, they often have a different base class than a ValueObject, so in IDML they are a separate type. In component applications, Events are often used for things that happen periodically, or used as a timing mechanism, hence the little clock in the icon that distinguishes it from the icon for ValueObject. A ValueObject may be a LookupKey and an Event may not. Otherwise they are the same in IDML.

Exception

Again, the term means what you would expect from all the other places you may have seen it: a way to gracefully handle things that may go wrong while executing an operation, and a way to pass information about what went wrong to the exception handler. An exception may be empty or it may contain Members (see above) but it cannot itself be referred to by a Member, Alias, Constant, or operation parameter type. There are several exception reference types in IDML, one or more of which may be contained in operation types (exception OnewayOperation) or attribute types (see below). Currently, exceptions in IDML cannot inherit from other exceptions, but inheritance may be added to exceptions in a future version of IDML.

ExceptionRef

As the name indicates, a reference to an Exception. ExceptionRefs can be found inside a TwowayOperation, FactoryOperation, or LookupOperation. An ExceptionRef refers to an Exception in the model, and the reference is assigned as described earlier in the subsection *Using the Model Browser.*

FactoryOperation

A type of operation that creates something and returns it. In IDML, FactoryOperations are found in ValueObjects, Events, and ComponentFactories. A FactoryOperation has no return type, since it is implicit, and has only InParameters. It may also, however, have zero or more ExceptionRefs. When found in a ValueObject or Event, the implicit return type is the type of the ValueObject or Event. When found in a ComponentFactory, the implicit return type is the type of the managed Component. ValueObjects, Events and ComponentFactories may contain any number of FactoryOperations.

File

Right click on an InterfaceDefinitions folder and move the cursor down to Insert. The only item appearing in the menu extension will be File. All File names in a project must be unique.

FileRef

Can be inserted only into a File and represents an included file. The IDL generator (see the last section of this manual) will keep track of file dependencies and automatically generate include statements, but this may not be possible for all model interpreters, so it is kept as part of IDML. All FileRefs in a single File must refer to unique Files. It doesn’t matter if a File includes itself, either directly or indirectly. IDML interpreters will generate guards to handle such things.

GetException

Can be declared only inside an Attribute or ReadonlyAttribute. It is a reference to an Exception, so what it refers to must be assigned in the manner described in the subsection *Using the Model Browser*. When the Attribute or ReadonlyAttribute is ultimately translated into software, the Exception referred to will be associated with the "get operation" part of the translation.

InEventPort

This type of port refers to an Event, and its function is to 'consume' Events of this type that come from an OutEventPort.

Inherits

The icon for Inherits is intentionally similar to its counterpart in UML, but with the 'shortcut' arrow in the lower left to indicate that it is a reference. Inherits can be found inside Objects, Value Objects, Events, Components and ComponentFactories. Each of these types can inherit only from the same type.

InOutParameter

An operation parameter used when the caller wants to send some value and also be aware of any modifications to its value by the callee. In distributed applications, this scenario is analogous to passing a parameter by reference. This type of parameter is found in IDML only in TwowayOperations.

InParameter

An operation parameter in which the caller has no interest once it is passed in an operation call. This type of parameter can appear in any kind of IDML operation.

InterfaceDefinitions

This is not strictly a model element, but rather a GME type known as a Folder. It must be inserted at the top level in any model, since all other model elements must be contained by it (except predefined types, which have their own folder PredefinedTypes). You may insert as many of these in your model as you want (they cannot be nested so they'll all be at the top level) and each one may be renamed. However, when the model is translated to software artifacts or to some middleware specification language, these folders will not appear in the output and have no effect on it. No icon is associated with the folder, since it appears only in the Model Browser. It was added mainly for ease of integration with PICML, but it can also be used to organize File declarations (see below) when modeling a very large project.

To insert one, right click on the top-level element in the Model Browser and move the cursor down to Insert. InterfaceDefinitions is the only item that will appear in the menu extension.

Label

Contained by a SwitchedAggregate, one or more Labels must be associated with each Member. This association is created in an IDML model by drawing a connection from a Member to a Label. One Member may optionally be connected to a Label named "default". If this is the case, no other Labels may be connected to that Member. This default Label indicates which Member is active if, when evaluated at runtime, the Discriminator's value does not match with any of the other Labels.

A Label must be connected to exactly one Member. Label connections in IDML are shown as blue lines.

LookupKey

A reference to a ValueObject (as its icon shows), optionally contained by a ComponentFactory. Its intended use in an application is as an aid to a ComponentFactory when lookup up its managed Component in a database or repository.

LookupOperation

Optionally found in a ComponentFactory, a LookupOperation is intended to function in an application by looking up the ComponentFactory's managed Component (which is the implicit return type) in a database or repository. A LookupOperation may have zero or more InParameters and zero or more ExceptionRefs.

Member

Represents a named reference contained by some type. As mentioned previously, the reference cannot be null, and assigning the reference replaces the generic Member icon with the reference version (or predefined type) icon of the type referred to. The following IDML types can be referred to by Member:

- any predefined type
- Aggregate
- Alias
- Boxed
- Collection
- Component
- ComponentFactory
- Enum
- Event
- Object
- SwitchedAggregate
- ValueObject

Object

Along with Component, the most important part of a middleware application design. It is through the operations defined inside an Object that much of the behavior of the application is realized.

An Object may contain attributes, operations, and declarations of any type that it is legal to declare in a Package, except Component, ComponentFactory, ValueObject and Event.

An Object may optionally inherit from any number of other Objects.

Objects have two attributes in IDML (not to be confused with Attributes that may be defined inside an Object. For more about GME attributes, see the GME manual). They are called 'abstract' and 'local'. They are Boolean values, and they are set, like all IDML attributes, in the GME Attribute Browser. The default value for both these attributes is FALSE. If an Object is ‘abstract’ it means that, in an actual application, it is not intended to be instantiated directly. However, its operations and attributes (IDML attributes this time) may be inherited by other Objects. Using abstract Objects is a convenient way to efficiently bundle operations that you want to be common to several other Objects. If an Object is ‘local’, it means that, in an application, it will never interact with other Objects remotely (over a network or something similar) and will never itself be passed in an operation parameter. In an application, local Objects are usually much more lightweight than regular ones, so declaring an Object ‘local’ in the model whenever possible will result in smaller applications.

A ‘local” Object may inherit from a non-local one, but not vice versa. Also, an ‘abstract’ Object may not inherit from a non-abstract one.

OnewayOperation

In this type of operation, the caller will know nothing about the outcome of the operation call once it is made, even if an exception is thrown. If this restriction is acceptable to the caller, using a OnewayOperation can be more efficient, since the caller will not have to wait for a reply. In IDML a OnewayOperation can appear in any place that a TwowayOperation can. A OnewayOperation can have no return type, and may contain only zero or more InParameters. It may not contain any ExceptionRefs.

OutEventPort

This type of port refers to an Event, and its function is to ‘publish’ Events of this type to any and all ‘subscribers’ (namely InEventPorts) to this type of Event. However, it also has a GME attribute called “single destination” (which is FALSE by default), but if set to TRUE, means that the port will take on some of the characteristics of a request port, in that it will communicate with only one InEventPort. The motivation for this option is efficiency. The default anonymous publish/subscribe mechanism often requires in practice a separate ‘event channel’ entity that manages everything, which can be more heavyweight than we want. Where it is known that one-to-one Event passing is sufficient, or even required, this option allows us to be more lean and mean.

OutParameter

This type of parameter is used when the caller wants to send an uninitialized reference to be initialized by the callee, and is found in IDML only in TwowayOperations.

Package

A familiar term in the Java world, as well as a few other places, and it means pretty much the same thing: a way to organize, both horizontally and vertically, related definitions. Packages may nest but may otherwise appear only inside Files. They are also partially exempt from the name uniqueness constraint (see the previous section). The same package name may appear any number of times in a given scope, but no package name may clash with other types declared in the same scope.

Predefined Types

Some of these represent the basic arithmetic and string types found in most programming languages. Others are generic versions of common middleware entities, which may be used to pass such entities around as a base class in an application. Others are not common to all kinds of middleware, but are included to make importing this kind or that kind of middleware more convenient.

- **Boolean –** You know, TRUE and FALSE. ‘Nuff said.
- **Byte** – Also self-explanatory.
- **String –** Also self-explanatory, but worth mentioning that it need not always be translated to an array of bytes by a model interpreter.
- **ShortInteger -**
- **LongInteger –** When generated, the actual sizes corresponding to ShortInteger and LongInteger will depend on the platform. Two sizes are provided here so the smaller one can be used for efficiency if desired.
- **RealNumber –** Floating point number. The precision of the generated quantity depends on the platform, but will usually be 8 bytes.
- **GenericObject** – A placeholder for any kind of Object. Usually generated as a common base class.
- **GenericValueObject** – Same function as GenericObject but for ValueObjects.
- **TypeEndcoding** – Holds type information, to be decoded at runtime.
- **TypeKind –** Represents a data type enumerating the categories of types found in a type system. Probably used by TypeEncoding.
- **GenericValue –** Container for any type of value, perhaps of a type to be determined at runtime. May also contain a TypeEncoding to describe the type of the value contained.

The last three types in the above list are not common to all kinds of middleware, but are included to facilitate the anticipated import of legacy CORBA IDL files.

Icons for these types do not appear anywhere in the Part Browser. Since they already exist, they cannot be “declared” as such, but are instead dragged over from the Model Browser to be assigned to a reference. They may be loaded into the Model Browser by one of the methods described in the subsection “Three Ways to Start a New Model”.

Private

Members of ValueObjects and Events are a little different than other Members, in that they may be public or private. In IDML they are public by default, but if you want to make them private, you drag in a Private icon from the Part Browser and connect the member you want to be private to it. A Private must be connected to exactly one Member, and the connection is shown in IDML as a red line. A member cannot be connected to more than one Private icon.

ProvidedRequestPort

This is a reference to an Object, and must be set as any other reference is set in IDML. A ProvidedRequestPort “provides” the operations of the Object to the RequiredRequestPorts (in other Components) that refer to the same Object. The Object referred to by ProvidedRequestPort cannot be abstract or local.

A ProvidedRequestPort may also refer to the predefined type GenericObject, which means that

- The actual type of Object provided by the port will be determined either in the last stages of implementation or at runtime.
- The specific type of Object provided by the port may not always be the same.

ReadonlyAttribute

Same as Attribute, except that when translated to software, it becomes only a single get operation. It contains exactly one AttributeMember and zero or more GetExceptions.

RequiredRequestPort

Like ProvidedRequestPort, this type refers to an Object (which cannot be abstract or local), and must be likewise set as described in *Using the Model Browser* earlier in this document. Its name stems from the fact that the Object referred to (more specifically, access to the Object’s operations) is “required” by the Component.

If you look in the Attribute Browser after selecting a RequiredRequestPort in the Model Editor, you’ll see that it has a GME attribute called ‘multiple_connections’. It’s Boolean and FALSE by default, but if set to TRUE, it means that this type of port may optionally need to connect to more than one ProvidedRequestPort at a time.

ReturnType

Optionally appearing in a TwowayOperation. It is a reference to some other type appearing in the model. As with all references in IDML, once dragged into the Model Editor, it must be set to refer to something (or deleted). If your intention is for the TwowayOperation to have no return type, just leave it out.

SetException

Can be declared only inside an Attribute. It is a reference to an Exception, so what it refers to must be assigned in the manner described in the subsection *Using the Model Browser*. When the Attribute is ultimately translated into software, the Exception referred to will be associated with the “set operation” part of the translation.

Supports

Can be defined inside a ValueObject, Event, Component, or ComponentFactory. The only things that can be ‘supported’ are Objects, so that is reflected in the Supports icon. You can also see from the icon that it is a reference, so it must refer to some Object in the model.

SwitchedAggregate

Similar to Aggregate, in that it represents a data type whose declaration contains one or more Members, but which, when evaluated at runtime, contains a single value which must be of one of the Member types. A SwitchedAggregate also contains exactly one Discriminator (see below) which, when evaluated at runtime, indicates which of the declared members is “active”, or the one actually contained. Each Member is associated with one or more Labels (see below), which are also contained in the SwitchedAggregate.

TwowayOperation

This is the most common kind of operation, and is called TwowayOperation because, even if it has no return type, in the application it may throw an exception to be caught. Zero or more ExceptionRefs may be contained in a TwowayOperation. The TwowayOperation itself may be found inside an Object, ValueObject, Event, or ComponentFactory.

A TwowayOperation may optionally contain a ReturnType, and zero or more parameters, divided into three categories – InParameter, InoutParameter and OutParameter. All four of these IDML types are references, and may refer to any type a Member refers to. The parameter types have names, but the ReturnType does not, so it has no name displayed when it is dragged into the Model Editor. A TwowayOperation may contain no more than one ReturnType, but any number of other parameters. All the parameters must have unique names.

If a TwowayOperation has more than one ExceptionRef, each one must refer to a different Exception.

ValueObject

Look at the icon for ValueObject, and you’ll see that it looks like a combination of the icons for Object and Aggregate, and that pretty much describes ValueObject itself. Why do we need such an animal in IDML, you ask? Well, for several reasons.

First, consider an Object’s Attributes or ReadonlyAttributes. They can be accessed only remotely, i.e., over a network. Even if the Object is ‘local’, in the middleware world it is still likely that accessing the Objects attributes will not be as efficient as a local operation call in some programming language. With a ValueObject’s Members, however, this is guaranteed to be the case, since wherever a ValueObject goes, its Members go with it, even over a network. An Object’s Attributes and ReadonlyAttributes stay where the Object is created. But what about Aggregates, you ask? Don’t they provide the same thing? Yes, they do, but ValueObjects have the additional benefit of inheritance, which spares you from having to repeat Member declarations when you want to share them among several types.

Second, suppose we want the efficiency and speed of operations that are completely ‘local’? Operations on a ‘local’ Object may not be going over a network, but there is often extra overhead still associated with them. With a ValueObject, like its Members, its operations go where it goes – there will be copies of these operations everywhere the ValueObject appears.

A ValueObject does not have the IDML attribute ‘local’ as Objects do, but it does have the ‘abstract’ attribute. If a ValueObject is ‘abstract’ it means that it is essentially a bundle of operations. An ‘abstract’ ValueObject cannot have any Members.

A ValueObject may optionally inherit from one other ValueObject in the model. An 'abstract' ValueObject may not inherit from a non-abstract one.

A ValueObject may also optionally 'support' any number of Objects. This means that, in the application, the ValueObject will have the Object's operations. If a ValueObject supports more than one Object, at most one of these Objects can be non-abstract.

A ValueObject may contain declarations of any types that it is legal to declare in Objects.

IDLGenerator

Overview

This model interpreter for IDML generates CORBA IDL which conforms to the OMG CORBA 3.x specification. The contents of each File in the model will be generated as a separate IDL file, having the name of the File model element with .idl appended. To launch IDLGenerator, left click on the **i** icon, second from left on the toolbar above the Model Editor. If more than one interpreter is registered with IDML, this action will bring up a list of interpreters to choose from.

Before interpretation starts, a dialog box will pop up for you to select the directory in which to output the IDL files.

Insofar as possible, the constraints in IDML will prevent you from creating a legal model that will result in illegal IDL from IDLGenerator. However, such protection can never be completely foolproof. For example, if constraints were added to IDML to prevent model element identifiers from clashing with IDL keywords, such constraints would also have to be added for the keyword set of every platform/specification language/programming language for which there is an interpreter for IDML. However, adding such checks to each interpreter, while being less than ideal since you won't find out until relatively late in the process if your model generates something problematic, is still a possibility for a future enhancement of IDLGenerator.

IDL Features Not Supported

For a platform-independent modeling language such as IDML, supporting every last nook and cranny of IDL or any other specific middleware platform is, if not impossible, certainly very difficult and guaranteed to make the modeling language unfriendly enough to make everyone suffer. Therefore, some features of IDL are not supported. Some of these features are deprecated by the OMG itself, some are not found in any other middleware platform but CORBA (and are currently under review by the OMG for possible future revision) and some are useful and will be supported in IDML at some future date. Most of them will not cause a problem when building an IDML model from scratch, since they'll be "out of sight, out of mind". If a model is imported into IDML from an existing IDL file (either by hand or with the planned IDL importing interpreter), some things will be lost, but hopefully not missed too much. Anyway, here's the list, with short comments about each one.

- CORBA::Char and CORBA::WChar – some the CORBA basic types that are under review by the OMG for possible rethinking. If these are a must-have, they can be cast from what’s generated for Byte (CORBA::Octet) and ShortInteger (CORBA::Short) respectively.

- Wide strings – IDLGenerator maps String to char*. This may change in the future, since some new middleware platforms are going with Unicode-only strings.
- Float – Maybe there is some efficiency gain for very long sequences of these things. It has been suggested, however, that a single size of floating point type is enough.
- Long double – again, under review by the OMG for rethinking.
- Unsigned integer types – ditto
- Bounded strings – the small gains in efficiency you might get from these doesn’t seem worth the trouble.
- Bounded sequences – it’s possible to optimize these to reduce the number of allocations and reallocations as the sequence grows and shrinks, but, like bounded strings, doesn’t seem to be worth the trouble.
- Arrays – arrays, along with the two bounded types above, are also under review by the OMG to reevaluate their worth.
- Anonymous declarations – now deprecated by the OMG, these existed only to support recursive types before forward declaration of structs and unions was added to IDL. IDLGenerator will avoid this construction by adding a typedef in an outer scope if necessary.
- Custom value types and truncatable valuetype inheritance – a goodly number of CORBA vendors don’t support custom marshaling or truncatable inheritance for value types.
- Members as declarations – for example

  ```
  struct foo
  {
     struct bar { short s; } foo_member;
  };
  ```

  Legal, but has anyone ever seen this in a real-world IDL file?
- Recursive types – these are admittedly useful, and support for them will be added to IDML in the future.
- ‘import’ keyword – not supported by most, if not all, C++ ORB vendors, since it’s essentially a Java notion that has no counterpart in C, C++, and other languages for which an IDL mapping is specified.

IDML to IDL Mapping

IDML	IDL	Comments
PrefixTag	`typeprefix`	Not generating `#pragma prefix`, since it seems typeprefix is intended to replace it.

VersionTag	`#pragma version`	
SpecifyIdTag	`typeid`	Not generating `#pragma ID`, since it seems typeid is intended to replace it
Aggregate	`struct`	
Alias	`typedef`	
Attribute	`attribute`	
AttributeMember		No explicit IDL – the model element referred to by AttributeMember is used to generate the type reference that follows the ‘attribute’ keyword.
Boolean	`boolean`	
Byte	`octet`	
Boxed	`valuetype`	Generates a boxed valuetype declaration, of the form ‘valuetype <identifier> <type>;’
Collection	`typedef sequence <…>`	If a Collection is referred to by a Member or AttributeMember, a typedef is generated in the scope enclosing the scope of the member, and the Collection’s type is generated just before the member name. Otherwise, inside the template brackets, IDLGenerator outputs the name of the type this Collection refers to. Finally, the name of the Collection is generated.
Component	`component`	
ComponentFactory	`home`	In IDML, it is not required to have a ‘home’ for every ‘component’. If necessary, IDLGenerator will generate a default home declaration, appending ‘IDMLDefaultHome’ to the Component’s name to get the home name. For an existing ComponentFactory, the name of the Component it connects to will follow generation of the `manages` keyword.
ComponentRef		The name of the Component this reference refers to will follow generation of the

		`manages` keyword for the ComponentFactory.
Constant	`const`	Followed by the name of the type the Constant refers to, followed by ‘ = ‘, followed by the string in the ‘value’ attribute.
Discriminator	`switch (…)`	The name of the type referred to by Discriminator is generated inside the parentheses.
Enum	`enum`	
EnumValue		Generated inside the braces of the enum declaration and separated by commas.
Event	`eventtype`	
Exception	`exception`	
ExceptionRef		The name of the type ExceptionRef refers to is generated in the list inside a `raises (…)` clause.
FactoryOperation	`factory`	
File		‘.idl’ is appended to File’s name and used to create an output stream.
FileRef	`#include`	
GenericObject	`Object`	
GenericValue	`any`	
GenericValueObject	`ValueBase`	
GetException	`getraises`	Unless GetException is found in a ReadonlyAttribute, in which case we generate `raises`.
InEventPort	`consumes`	
Inherits		The name of the type referred to by Inherits is generated in the comma-separated list following ‘`:`‘.
InoutParameter	`inout`	
InParameter	`in`	
InterfaceDefinitions		No IDL generation, this is at a higher level than File.
Label	`case`	The name of the Label is generated next, followed by a colon, unless the Label’s name is ‘default’, in which case `default:` is generated.
LongInteger	`long`	

LookupKey	`primarykey`	
LookupOperation	`finder`	
Member		Generates the name of the type that Member refers to, followed by Member's name. If the Member occurs in a ValueObject, `public` is generated first by default, unless the Member is connected to a Private model element.
Object	`interface`	First generates `local` or `abstract` if either of these GME attributes is set to TRUE.
OnewayOperation	`oneway`	
OutEventPort	`publishes`	If the 'single_destination' attribute is set, then `emits` is generated instead.
OutParameter	`out`	
Package	`module`	Packages with the same name in the same scope will map to a reopened module.
Private	`private`	Overrides the default `public` generated in front of a ValueObject's Member.
ProvidedRequestPort	`facet`	
ReadonlyAttribute	`readonly attribute`	
RealNumber	`double`	
RequiredRequestPort	`uses`	If the 'multiple_connections' attribute is set, then `uses multiple` is generated.
ReturnType		Only the name of ReturnType's reference is generated.
SetException	`setexception`	
ShortInteger	`short`	
String	`string`	
Supports	`supports`	
SwitchedAggregate	`union`	
TwowayOperation		The name of the ReturnType is generated (or if there is none, `void`) then TwowayOperation's name.
TypeEncoding	`CORBA::TypeCode`	
TypeKind	`CORBA::TCKind`	
ValueObject	`valuetype`	Preceded by `abstract` if the 'abstract' attribute is set.

Other Features

IDLGenerator makes two passes over the model when it is executed. In the first pass, a number of things happen.

- Dependency information is cached.
- A flag is set to generate `#include <Components.idl>` if any component-related type is seen in a File.
- A flag is set to generate `#include <orb.idl>` if a TypeEncoding or TypeKind is seen in a File.
- If a model element in File A refers to a model element in File B, a flag is set to generate an include directive for B.idl in A.idl.

If a FileRef model element calls for the generation of an include directive that has already been cached in the first pass, it will not be generated twice. However, if it has not already been cached, it will be generated regardless of whether the dependency is necessary or not.

On the second pass, IDLGenerator generates, for each IDL file, `#ifdef` guards followed by a list of file include directives, followed by the contents of the file, followed by an `#endif`. For each scope, IDLGenerator will order the contents of the scope based on the cached dependency information, then generate the contents. For types that can be forward declared (`interface, valuetype, eventtype,` and `component`), they will not be reordered but simply forward declared at the beginning of the scope in which they appear. If dependencies reach across module reopenings, a module will be reopened, to whatever depth of nesting necessary, to make a forward declaration. Forward declarations are not generated for IDL structs or unions, since the CORBA spec mandates the forward declaration of these types be used only to declare a recursive type. Recursive structs and unions are not yet supported in IDML.

Of the three ‘anonymous’ types in IDL – sequences, arrays and bounded strings, IDML does not support the last two. For sequences, it is not necessary to use Alias to create a ‘typedef’ of a sequence type (the way preferred by the OMG). If a Collection is seen by IDLGenerator, an IDL typedef is automatically generated (see the table above).

Known Issues

In a graphical model such as IDML, there is no inherent order of model elements in a given window of the Model Editor. In GME’s MultiGraph Architecture (MGA), there is no guaranteed order in which child elements are stored or traversed, not even the order displayed in the Model Browser. You should not assume that the elements of your model will be generated as IDL in any particular order by IDLGenerator, or even that the order will be the same in successive executions of IDLGenerator.

Since recursive types are not yet supported by IDML, a recursive Aggregate or SwitchedAggregate in an IDML model will cause undefined behavior in IDLGenerator. In addition, it is also possible to have circular dependency at the File level. This will not cause IDLGenerator any problems, but the resulting IDL files may not be compileable without alteration.