High-Level Java Interface to GME

Users Manual

Version 1.0

February 2004

Institute for Software Integrated Systems
Vanderbilt University

http://www.isis.vanderbilt.edu

Introduction to the Java Interface

The process of accessing GME models and generating useful information, e.g. configuration files for COTS software, database schema, input for a discrete-event simulator or even source code, is called ***model interpretation***. GME provides two interfaces to support model interpretation. The first one is a COM interface that lets the user write these components in any language that supports COM, e.g. C++, Visual Basic or Python. The COM interface provides the means to access and modify the models, their attributes and connectivity. In short, the user can do everything that can be done using the GUI of the GME. The other interface is a high-level C++ interface (builder object network or BON) that takes care of a lot of lower level issues and makes component writing much easier. This document describes a new addition, a high-level java interface for GME.

Interpreters are typical, but not the only components that can be created using this technology. The other types are ***plugin***s, i.e. paradigm-independent components that provide some useful additional functionality to ease working in GME. These components are very similar to interpreters. For example, a plugin can be developed to search or locate objects based on some user-defined criteria, like the value of an attribute.

What Does the Java Interface Do?

The Java interface is implemented on the top of the COM interface. GME objects can be accessed in java through a low-level API which is the direct wrapping of the COM interfaces or a high-level API which is similar to the C++ builder object network (BON) API. When the user initiates model interpretation, the Java interface creates the builder object network. The builder object network mirrors the structure of the models: each model, atom, reference, connection, etc. has a corresponding builder object. This way the interface shields the user from the lower level details of the COM interface and provides support for easy traversal of the models along the containment hierarchy, the connections, or the references. The builder classes provide general-purpose functionality. The builder objects are instances of these predefined paradigm-independent classes. For simple paradigm-specific or any kind of paradigm-independent components, they are all the user needs. For more complicated components, the builder classes can be extended with inheritance. By implementing a function of the BONComponent interface, the user can have the component interface automatically instantiate these paradigm-specific classes instead of the built-in ones. The builder object network will have the functionality provided by the general-purpose interface extended by the functionality the component writer needs.

Writing a Java interpreter

First of all the interpreter writer should decide which java API will be used. If the user wants to use the low-level API the org.isis.gme.bon.Component must be implemented. For the high-level API the org.isis.gme.bon.BONComponent must be implemented. Here is a sample class writing out the project name using the low-level API:

```
package org.isis.gme.bon;

import javax.swing.JOptionPane;
import org.isis.gme.mga.MgaFCO;
import org.isis.gme.mga.MgaFCOs;
import org.isis.gme.mga.MgaProject;

public class TestComponent implements Component {
    public void invokeEx(MgaProject project, MgaFCO currentObj,
                         MgaFCOs selectedObjs, int param)
    {
        String msg = new String();
        msg = "Project name: " + project.getName() + "\n";
        JOptionPane.showMessageDialog(null, msg, "Java Interpreter Test",
                                      JOptionPane.ERROR_MESSAGE);
    }
}
```

When the user starts the interpreter, a java virtual machine is started, the java wrapper classes of the gme objects are created than the invokeEx function of the interpreter is called.

The following code shows an example of an interpreter using the high-level API:

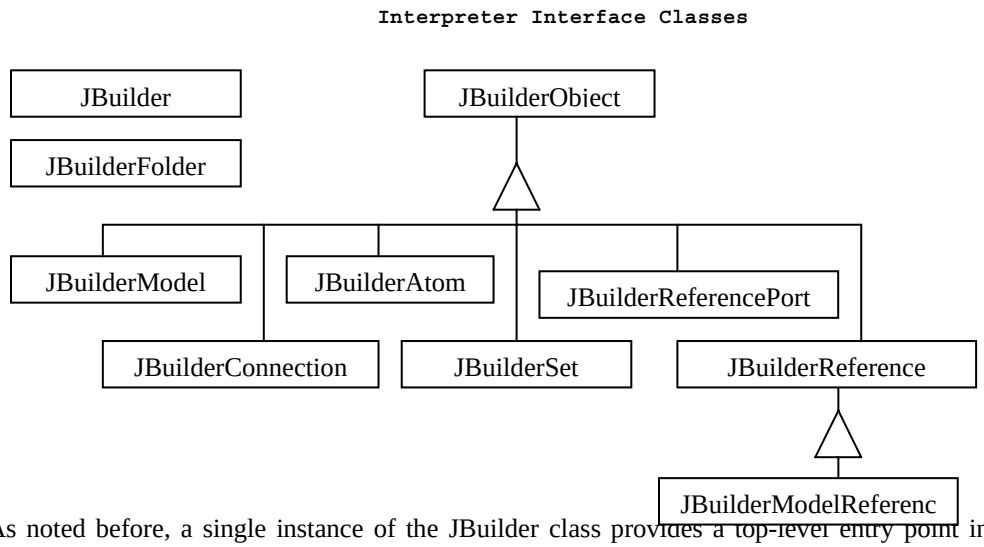
```
package org.isis.gme.bon;

import javax.swing.JOptionPane;

import org.isis.gme.mga.MgaFCO;
import org.isis.gme.mga.MgaFCOs;
import org.isis.gme.mga.MgaProject;

public class TestComponent implements BONComponent
{
        public void invokeEx(JBuilder builder, JBuilderObject focus,
                             Collection selected, int param)
        {
        }
}
```

When the user initiates interpretation, first the builder object network is created then the above function is called. The first three parameters provide ways of traversing the builder object network. Using the first parameter 'builder' it is possible for the user to start interpretation from the root folder of the models. Using the root folder one can access the list of folders. Each folder provides a list of builder objects corresponding to the root models and subfolders. Any builder object can then be accessed through recursive traversal of the children of model builders.

The second parameter 'focus' provides the object in focus when the interpretation was started, thus the user can interpret from this stage. The Next parameter 'selected' contains builders that were selected when interpretation was started. If the interpretation was started through a context menu (i.e. right click) then the list contains items for all the selected objects in the given window. If the interpretation was started through the context menu of the Model Browser, then the list contains the builders for the selected models in the browser. Using this list parameter of the Invoke function makes it possible to start the interpretation at models the user selects. The long parameter is unused at this point.

High-level Java Interface

The simple class structure of the component interface is shown below.



As noted before, a single instance of the JBuilder class provides a top-level entry point into the builder object network. It provides access to the model folders and supplies the name of the current project. The public interface of the JBuilder class is shown below.

```
public class JBuilder
{       String getProjectName();
        JBuilderFolder getRootFolder();
        Vector getFolders();   // the Vector contains JBuilderFolder objects
        JBuilderFolder getFolder(String name);
}
```

The JBuilderFolder class provides access to the root models of the given folder. It can also be used to create new root models.

```
public class JBuilderFolder
{       String getName();
        Vector getRootModels();       // the Vector contains JBuilderModel objects
        Vector getSubFolders();       // the Vector contains JBuilderFolder objects
        JBuilderModel getRootModel(String name);
        JBuilderModel createNewModel(String kindName);
}
```

The JBuilderObject is the base class for several other classes. It provides a set of common functionality for models, atoms, references, sets and connections. Some of the functions need some explanation. The getAttribute() functions return true when they successfully retrieved the value of attribute whose name was supplied in the name argument. If the type of the val argument does not match the attribute or the wrong name was provided, the function returns false. For field and page attributes, the type matches that of specified in the meta, for menus, it is a String and for toggle switches, it is a bool. The getxxxAttributeNames functions return the list of names of attributes the given object has. This helps writing paradigm-independent components (plugins). The getReferencedBy function returns the list of references that refer to the given object. The getInConnections (getOutConnection) functions return the list of incoming (outgoing) connections from the given object. The string argument specifies the name of the connection kind as specified by the modeling paradigm. The getInConnectedObjects (getOutConnectedObjects) functions return a list of objects instead. The traverseChildren functions provide a ways to traverse the builder object network along the containment hierarchy. The implementation provided does not do anything; the component writer can override it to implement the necessary functionality. As we'll see later, the JBuilderModel class does override this function. It enumerates all of its children and calls their Traverse method.

```
public class JBuilderObject extends Object
{       String getName();
        boolean setName(String newname);
        void getNamePath(String namePath);
        String getKindName();
        String getPartName();
        JBuilderModel getParent();
        JBuilderFolder getFolder();
        boolean getAttribute(String name, String val[]); //uses arrays because java
        boolean getAttribute(String name, int val[]); //does not have a true pass
        boolean getAttribute(String name, bool val[]); //by reference, the result will
        boolean setAttribute(String name, String val[]); //be in val[0]
        boolean setAttribute(String name, int val[]);
        boolean setAttribute(String name, bool val[]);
        Vector getStrAttributeNames();               // the Vector contains String objects
        Vector getIntAttributeNames();               // the Vector contains String objects
        Vector getBoolAttributeNames();      // the Vector contains String objects
        Vector getFloatAttributeNames();     // the Vector contains String objects
        Vector getRefAttributeNames();               // the Vector contains String objects
        Vector getReferencedBy();     // the Vector contains JBuilderObject objects
        Vector getInConnections(String name); // the Vector contains JBuilderConnection
        Vector getOutConnections(String name);// the Vector contains JBuilderConnection
        boolean getInConnectedObjects(String name, Vector list[]);
                                             //list[0] contains JbuilderObject objects
        boolean getOutConnectedObjects(String name, Vector list[]);
                                             //list[0] contains JbuilderObject objects
        void traverseChildren(); //extend in custom classes
}
```

The JBuilderModel class is the most important class in the component interface, simply because models are the central objects in the GME. They contain other objects, connections, sets and have aspects etc. The getChildren function returns a list of all children, i.e. all objects the model contains (models, atoms, sets,

references and connections). The getModels method returns the list of contained models. If a role name is supplied then only the specified part list is returned. The getAtoms, getReferences, getAtomReferences and getModelReferences, getSets() functions work the same way except that a part name must be supplied to them. The getConnections method returns the list of the kind of connections that was requested. These are the connections that are visible inside the given model.

The getAspectNames function returns the list of names of aspects the current model has. This helps in writing paradigm-independent components. Children can be created with the appropriate creation functions. Similarly, connections can be constructed by specifying their kind and the source and destination objects. Please, see the description of the JBuilderConnection class for a detailed description of connections. The traverseModels function is similar to the traverseChildren but it only traverses models.

```
public class JBuilderModel extends JBuilderObject
{       Vector getChildren();         // the Vector contains JbuilderObject objects
        Vector getModels();           // the Vector contains JbuilderModel objects
        Vector getModels(String partName); // the Vector contains JbuilderModel objects
        Vector getAtoms(String partName); // the Vector contains JbuilderAtom objects
        Vector getReferences(String refPartName);
                              // the Vector contains JbuilderReference objects
        Vector getModelReferences(String refPartName);
                              // the Vector contains JbuilderModelReference objects
        Vector getAtomReferences(String refPartName);
                              // the Vector contains JbuilderAtomReference objects
        Vector getConnections(String name);
                              // the Vector contains JbuilderConnection objects
        Vector getSets(String name); // the Vector contains JbuilderSet objects
        void getAspectNames(Vector list);
                              // the Vector should be created and passed, the list will
                              contain String objects of all the Aspects on returning
        JBuilderModel createNewModel(String partName);
        JBuilderAtom createNewAtom(String partName);
        JBuilderModelReference createNewModelReference(String refPartName,
                                                       JBuilderObject refTo);
        JBuilderAtomReference createNewAtomReference(String refPartName,
                                                       JBuilderObject refTo);
        JBuilderSet createNewSet(String partName);
        JBuilderConnection createNewConnection(String connName, CBuilderObject *src,
                                            CBuilderObject *dst);
        void traverseModels();
        void traverseChildren();
}
```

The JBuilderAtom class does not provide any new public methods.

```
public class JBuilderAtom extends JBuilderObject
{
}
```

Even though the GME deals with ports of models (since connection are usually made to these instead of the model itself), the component interface avoids using ports for the sake of simplicity. However, model references mandate the introduction of a new kind of object, model reference ports. A model reference contains a list of port objects. The getOwner method of the JBuilderReferencePort class returns the model reference containing the given port.

```
public class JBuilderReferencePort extends JBuilderObject
{     JBuilderModelReference getOwner();
}
```

The JBuilderModelReference class provides the getReferred function that returns the model (or model reference) referred to by the given reference. The getReferencePorts return the list of JBuilderReferencePorts.

```
public class JBuilderModelReference extends JBuilderObject
{       Vector getReferencePorts();   // the Vector contains JBuilderReferencePort objects
        JBuilderObject getReferred();
}
```

A JBuilderConnection instance describes a relation among three objects. The owner is the model that contains the given connection (i.e. the connection is visible in that model). The source (destination) is

always JBuilderObject. If it is a regular object (i.e. not a reference port) then it is either contained by the owner, or it corresponds to a port of a model contained by the owner. So, in case of regular objects, either the source (destination) or its parent is a child of the owner. In case of a reference port, its owner must be a child of the owner of the connection.

```
public class JBuilderConnection extends JBuilderObject
{       JBuilderModel getOwner();
        JBuilderObject getSource();
        JBuilderObject getDestination();
}
```

The JBuilderSet class member function provides straightforward access to the different components of sets.

```
public class JBuilderSet extends JBuilderObject
{       JBuilderModel getOwner();
        Vector getMembers();          // the Vector contains JbuilderObejct objects
        boolean addMember(JBuilderObject part);
        boolean removeMember(JBuilderObject part);
}
```

Example

The following simple paradigm-independent interpreter prints a message for each model in the project. For the sake of simplicity, it assumes that there is no folder hierarchy in the given project. The JComponent class file is shown below.

```
import java.util.*;
import org.isis.gme.bon.*;
public class Example2 extends BONComponent
{
        public void invokeEx(JBuilder builder, JBuilderObject focus, Collection selected,
                             int param)
  {
        Vector folds = builder.getFolders();
        int foCount = folds.size();
        for(int i=0; i<foCount;i++)
        {       JBuilderFolder fold = (JBuilderFolder)folds.elementAt(i);
                Vector roots = fold.getRootModels();
                int roCount = roots.size();
                for(int j=0; j<roCount;j++)
                {       JBuilderModel root = (JBuilderModel)roots.elementAt(j);
                        scanModels(root,fold.getName());
                }
        }
  }
  public void scanModels(JBuilderModel model, String fName)
  {     system.out.println(model.getName()+" model found in the
                                                                          "+fname);
        Vector models = model.getModels();
        Int moCount = models.size();
        For(int i=0;i<moCount;i++)
        {       JBuilderModel subModel = (JBuilderModel)models.elementAt(i);
                scanModels(subModel,fName);
        }
  }
}
```

Extending the Java Interface

The previous example used the built-in classes only. The component writer can extend the component interface by his own classes. In order for the interface to be able to create the builder object network instantiating the new added classes before the user-defined interpretation actually begins, a couple of steps must be done. The derived class declaration must contain a constructor identical to the super class and should call the constructor of the super class. Then the user should add the function calls for the specific custom class in the interpreter's registerCustomClasses method. The function classes are given below.

```
addCustomModel(<kindName>,<Class Name>);
addCustomModelRef(<kindName>,<Class Name>);
addCustomAtom(<kindName>,<Class Name>);
```

```
addCustomAtomRef(<kindName>,<Class Name>);
addCustomConnection(<kindName>,<Class Name>);
addCustomSet(<kindName>,<Class Name>);
```

Here, the <Class Name> is the name of the new class, while the <kindName> is the kind name of the object that should instantiate your class. (The user can create abstract base classes as discussed later.). For example, if we have a "Compound" model in our paradigm, we can create a builder class for it the following way.

```
// JCompoundBuilder class
class JCompoundBuilder extends JBuilderModel
{       public JCompoundBuilder (IMgaModel iModel, JBuilderModel parent)
        {       super(iModel,parent);
        }
        // user defined code and variables;

}

// Interpreter class
public class TestInterpreter
{
        public void invokeEx(JBuilder builder, JBuilderObject focus, Collection selected,
                             int param)
        {
        …
        }

        public void registerCustomClasses()
        {
                …
                AddCustomModel("Compound","JCompoundBuilder");
                …
        }
}
```

Do not define your own constructors apart from the one required, you have to call the base class implementation.

If you want to define abstract base classes that are not associated with any of your models, use "*" in the kindName field.

Example

Let's assume that our modeling paradigm has a model kind called Compound. Let's write a component that implements an algorithm similar to the previous example. In this case, we'll scan only the Compound models. Again, the folder hierarchy is not considered. Here is the Component.java file:

```
// JCompoundBuilder class
class JCompoundBuilder extends JBuilderModel
{       public JCompoundBuilder (IMgaModel iModel, JBuilderModel parent)
        {       super(iModel,parent);
        }

        public void scan(String fName)
        {       System.out.println(model.getName()+" Compound found
                                                                    in The
                                                              +fname);
                Vector models = model.getModels();
                Int moCount = models.size();
                For(int i=0;i<moCount;i++)
                {       JBuilderModel subModel = (JBuilderModel)models.ellementAt(i);
                        if(root.getClass().toString().equals(
                                        "class JCompoundBuilder"))
                                ((JCompoundBuilder)subModel).scan(fold.getName);

                }
        }

}

// Interpreter class
public class TestInterpreter
```

```
{
        public void invokeEx(JBuilder builder, JBuilderObject focus,
                        Collection selected, int param)
        {
                Vector folds = builder.getFolders();
                int foCount = folds.size();
                for(int i=0; i<foCount;i++)
                {       JBuilderFolder fold = (JBuilderFolder)folds.elementAt(i);
                        Vector roots = fold.getRootModels();
                        int roCount = roots.size();
                        for(int j=0; j<roCount;j++)
                        {       JBuilderModel root = (JBuilderModel)roots.elementAt(j);
                                if(root.getClass().toString().equals(
                                                        "JCompoundBuilder"))
                                        ((JCompoundBuilder)root).scan(fold.GetName);
                        }
                }
        }

        public void registerCustomClasses()
        {
                addCustomModel("Compound","JCompoundBuilder");
        }
}
```

Registering a Java interpreter

Once the Java interpreter is implemented it must be registered before using it. The registration can be done with the JavaCompRegister utility included in the distribution.