.NET Extensions for Generic Modeling Environment

Gabor Simko

2010 October

Contents

[Introduction 3](#_Toc275275322)

[GME Architecture 3](#_Toc275275323)

[C# and COM 4](#_Toc275275324)

[A Sample DSML 4](#_Toc275275325)

[GME Application Programming Interfaces in .NET 4](#_Toc275275326)

[The Meta Library 4](#_Toc275275327)

[The MGA Library 4](#_Toc275275328)

[The GME Library 5](#_Toc275275329)

[The Interpreter interface 5](#_Toc275275330)

[The Add-on Interface 6](#_Toc275275331)

[The Decorator Interface 6](#_Toc275275332)

[C# Component Wizard 6](#_Toc275275333)

# Introduction

“Describing the world, or a segment of it, is one of the key problems in communication. People of all historic eras have been looking for solutions. An array of remarkable approaches has emerged, like Egyptian engravings used to record historical events, Greek and medieval astronomers' survey of the Universe, or Carl von Linne's taxonomy of the world's living creatures.

There is no "best" technique, especially because different human activities call for different approaches. Art deliberately maintains a wide variety of techniques to communicate subtle impressions, beliefs, or affections; the style of the description is often equally as important as the subject itself. On the other hand, science, especially the natural sciences and technology, is looking for *precision*, *conciseness*, and *unambiguity* instead.

**M**odeling is one of the most universal scientific description techniques. While it is a rather broadly used and thus ill-defined term itself, it generally means describing the world in terms of *pre-established concepts and rules*. These rules and concepts govern the description process and also make it concise, since details implied by them can be omitted without decreasing clarity.

Most scientific description techniques like databases, computer programming languages, or even formal description methods are based on the modeling approach. Still, the most well-known representatives of modeling are graphical description systems. These tools enhance the illustrative, easy-to-understand nature of charts and diagrams with the precision of modeling concepts and rules. Things represented by these models can be as diverse as organizational hierarchy charts, genetic maps, and community sewer network diagrams.

Modeling is not only about description and illustration for human use. Their adherence to rules and patterns makes models different from drawings, figures, or free format textual descriptions. This not only reduces data size and disambiguates interpretation, but also makes the model suitable for automatic processing. For example, given an up-to-date company hierarchy chart, ordering correct business cards for each employee is a very straightforward process.”

The Generic Modeling Environment (GME) is a Domain-Specific Modeling environment, which makes this modeling process possible. It is a generic environment, because the same modeling environment is used for both modeling the domain-specific meta-model, and the specific models based on this meta-model.

## GME Architecture

The heart of the GME architecture is the MGA (Multi-Graph Architecture) ­­which handles and stores the graphs and provides a common interface for modifying these data. A graph is called multi-graph, if it contains edge-loops or parallel edges, and these graphs are representable by the MGA architecture.

MGA is extendable by user provided components, one of which is GME User Interface, which provides a standard user interface for defining and manipulating graphs. Other classes of user components are interpreters and add-ons:

* Interpreters are generic components, which are executed on demand and which may transform/interpret the graph. They have to entry points: initialization and running. The former is called once, when the interpreter is loaded, while the latter is called whenever you request the interpreter to run.
* Add-ons are components, which are called whenever a given event happens in the graph. They can response to data changes, and may be used to augment GME with additional functions. They can handle global and object events, where global is any event connected to projects, and object events are called on accessing specific objects.

Table MGA is the base of GME/interpreters/addons; all of them communicate through MGA.

|  |  |  |
| --- | --- | --- |
| MGA | | |
| GME | Interpreter | Addon |

Basically, MGA is a data store, which is responsible for the data management. GME is the user interface, which makes the manipulation of these data easier. Interpreter and addons are used for building specific functions.

To support different programming languages MGA and the whole underlying architecture is written using the Component Object Model (COM), which provides a language independent way to accessing the functions of MGA.

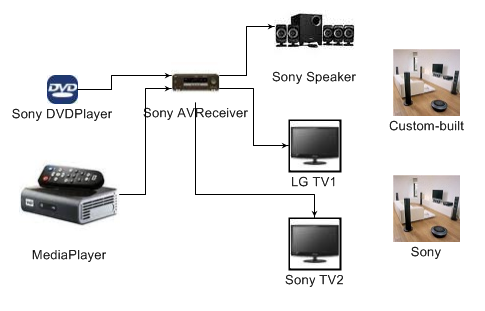
In order to keep the model consistent, any process accessing the model is obligated to initiate a transaction. When the process finished its job, the transaction must be committed or aborted. In case you forget to do so, the model may remain in an incorrect state, so exception handling and aborting on failure are absolutely necessary.

## C# and COM

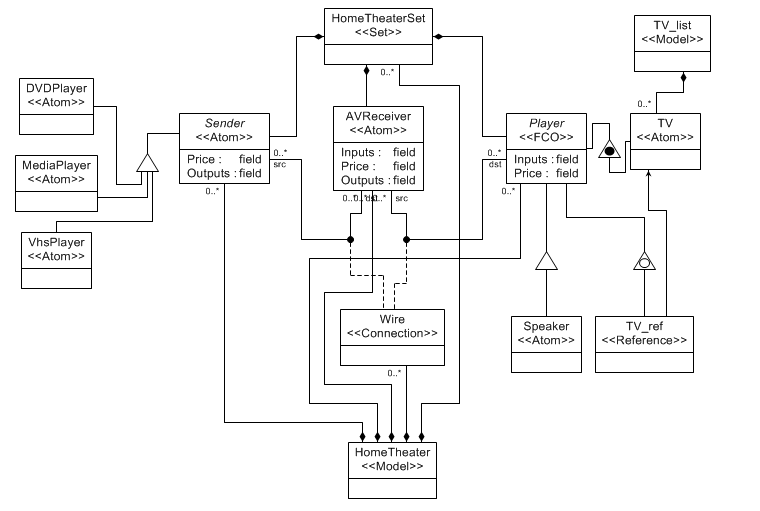
Using a Component Object Model is very straightforward and easy in C#. Basically, after adding the COM component to the project References, one may instantly use all the functions and variables stored in the COM. The Microsoft Visual Studio IDE also provides a way to access all the imported symbols through its object browser.

It is really important to understand how COM is handled by .NET, because It is different to how DLL’s are loaded and not that self-evident. When you use a COM object, the .NET framework will automatically find and load the appropriate DLL, or if it is already loaded, just use it. There is always one and only one COM object in the memory and the COM manager does not allocate separate data memories for different processes. Consequently, you should never use static variables in COM components, because this may lead to hard to detect problems.

## A Sample DSML

In the following example we are building a very basic model for home theater stores. We assume that the center of every home theater system is the AV receiver unit, and we have input and output units “Wire”-d to this unit: Senders are reading and sending the data to the AVReceiver, which is transmitting it to the Players. We define three kinds of inputs, DVDPlayer, MediaPlayer and VhsPlayer, while the Player might be a Speaker or a TV. 

The store want to build different configurations from the above components called HomeTheaterSet, and they also want to have a separate list of the TVs in order to easily manage the available TVs. The latter is implemented by adding only TV references to the HomeTheater models, while the real TVs are stored in TV\_list models.



# A Sample Interpreter

We are going to build an interpreter, which traverses the HomeTheater sets. For each HomeTheaterSet it prints all the contained devices, and calculates the sum of their cost.

Moreover, it collects information about the possible connections to and from AVReceivers, and displays all the devices which are connected to the AVReceiver in any set.

private string Interpret()

{

StringBuilder result = new StringBuilder();

// always handle exceptions, the transaction HAS TO BE aborted if an error occurs

try

{

// Begin a Read-only transaction, since we don't modify anything

MgaGateway.BeginTransaction(MGALib.transactiontype\_enum.TRANSACTION\_READ\_ONLY);

// stores the devices connected to any AVReceiver (AVReceiver -> set of devices mapping)

Dictionary<string, SortedSet<string>> avr\_connections\_src = new Dictionary<string,SortedSet<string>>();

Dictionary<string, SortedSet<string>> avr\_connections\_dst = new Dictionary<string,SortedSet<string>>();

// for all HomeTheater models

foreach (IMgaModel setlibs in project.RootFolder.GetChildrenOfKind("HomeTheater"))

{

// Print the name of the HomeTheater

result.AppendLine(setlibs.Name);

// for all HomeTheaterSet in the HT, list its components and the price of the set

foreach (IMgaSet set in setlibs.GetChildrenOfKind("HomeTheaterSet"))

{

StringBuilder info = new StringBuilder(set.Name + ": ");

double sumprice = 0;

foreach (IMgaFCO device in set.Members)

{

// if the device is TV\_ref, follow the reference to obtain its price

if (device.MetaBase.Name.Equals("TV\_ref"))

{

IMgaReference tv\_ref = (IMgaReference)device;

sumprice += tv\_ref.Referred.FloatAttrByName["Price"];

}

// else the price is its direct attribute

else

sumprice += device.FloatAttrByName["Price"];

info.Append(device.Name + ", ");

}

info.Append(sumprice.ToString());

result.AppendLine(info.ToString());

}

// for all AVReceivers in the HT store its connections

foreach (IMgaFCO receiver in setlibs.GetChildrenOfKind("AVReceiver"))

{

avr\_connections\_src[receiver.Name] = new SortedSet<string>();

avr\_connections\_dst[receiver.Name] = new SortedSet<string>();

// traverse all the connections for the receiver

IMgaConnPoints cp = receiver.PartOfConns;

foreach (IMgaConnPoint connPoint in cp)

{

IMgaSimpleConnection conn = (MGALib.IMgaSimpleConnection)connPoint.Owner;

// if the receiver is the src, we store the destination

if (conn.src == receiver)

avr\_connections\_dst[receiver.Name].Add(conn.dst.Name);

// if the receiver is the dst, we store the source

if (conn.dst == receiver)

avr\_connections\_src[receiver.Name].Add(conn.src.Name);

}

}

}

// list all the AVReceivers with their output connections

foreach (KeyValuePair<string, SortedSet<string>> kvp in avr\_connections\_src)

{

StringBuilder sb = new StringBuilder("To " + kvp.Key + ": ");

foreach (string s in kvp.Value)

sb.Append(s + "; ");

result.AppendLine(sb.ToString());

}

// list all the AVReceivers with their input connections

foreach (KeyValuePair<string, SortedSet<string>> kvp in avr\_connections\_dst)

{

StringBuilder sb = new StringBuilder("From " + kvp.Key + ": ");

foreach (string s in kvp.Value)

sb.Append(s + "; ");

result.AppendLine(sb.ToString());

}

}

finally

{

MgaGateway.AbortTransaction();

}

return result.ToString();

}

# GME Application Programming Interfaces in .NET

## The Meta Library

## The MGA Library

As GME handles every object through MGA, we can access, traverse and manipulate all kind of objects using the MGA interface. Every GME meta-model object has a corresponding interface in MGA, which are generally named IMgaXXX, where XXX is the object name. For instance, we can find IMgaFCO, IMgaModel, IMgaReference, IMgaSet, etc.

Usage of Microsoft Visual Studio is highly recommended, because Intellisense displays all the available functions and properties, which makes the development very straightforward and easy. There are several use scenarios which I will detail here:

1. The RootFolder of an IMgaProject project can be obtained by project.RootFolder
2. To retrieve all the children of a model, set, or folder, use the GetChildrenOfKind(String kind) function, where kind tells the kind of the retrieved children
3. Every IMgaFCO object has a name, accessible through object.Name
4. The attributes of an IMgaFCO object are accessible through object.StrAttrByName, object.IntAttrByName, and object.FloatAttrByName. For example the string attribute called “value”: object.StrAttrByName["value"]
5. To access the connections of an IMgaFCO object:

foreach (IMgaConnPoint connPoint in object.PartOfConns)

{

IMgaSimpleConnection conn = (MGALib.IMgaSimpleConnection)connPoint.Owner;

// IMgaFCO conn.src is the source of the connection

// IMgaFCO conn.dst is the destination of the connection

}

1. The referred value of an IMgaReference reference is reference.Referred.

## The GME Library

The most used GME library calls are for the usage of the GME console. The GMEConsole class in the namespace of GME.CSharp automatically handles this, so you will not have to bother with it. Instead of System.Console.WriteLine(), System.Console.Clear(), and System.Console.Error.WriteLine() use GME.CSharp.Console.WriteLine(),GME.CSharp.Console.Clear(),GME.CSharp.Console.Error.WriteLine().

## The Interpreter interface

The interpreter is automatically generated by the wizard, so you do not need to understand its functioning. Actually, writing an interpreter needs just the same efforts as writing a standalone application for GME.

You have to provide a function, which will accept four arguments:

IMgaProject project

IMgaFCO currentobj

IMgaFCOs selectedobjs

int param

where :

* project contains the whole project with all the objects,
* currentobj is the object I DUNNO
* selectedobjs are the currently selected objects
* param is not used.

## The Add-on Interface

The wizard generated AddOn will contain the GlobalEvent and the ObjectEvent functions. The first is used for responding global events (e.g. project open/close), while the latter one is called when objects are manipulated. Using the wizard you can sign up for different events, which will trigger the ObjectEvent function. Most of the time you should only modify this function. It accepts three parameters:

MgaObject subject /\* the object the event(s) happened to \*/

uint eventMask /\* events ORed together \*/

object param /\* not used \*/

## The Decorator Interface

# C# Component Wizard