How to add a new generator…?
Technical interface description

Diploma thesis with working title Open DB Designer

dataMagus
Monika Schwitter & Jürg Zgraggen, 8Ibb-a

Revision

Version	Date	Comment	Author
1.0	18.11.2007	First version created	MOS

Table of contents

1 Introduction

dataMagus is a software to learn database modelling. The goal is to help students to understand the concepts of various databases. Therefore the software also supports source code generation of different targets (PostgreSQL, SQLServer2000, ODL). Because there are lots of other possible SQL or ODL dialects we decided to use the concept of plug-ins to gain a high flexibility.

The main goal of this document is to describe the interfaces needed for a generator plug-in. It should give you a short overview about how to write a new generator for the dataMagus software. The target group it has been written for are developers.

2 A brief excursion about the conception

In dataMagus only one logical schema (also known as layer) exists. The user can model its database here. Furthermore, there are two different physical schemas (a relational one and a object-oriented one) at the moment. These schemas will be created out of the logical one. So, if you are about to write a generator, you must decide for which physical schema you'd like to write it for.

3 How to write your own generator

There are several main interfaces which you must understand quite well before writing a new code generator. The highest abstract *IGenerator* interface is used to define the main functionality every generator must have. This interface would just be adequate to start generating source code. However, there are some sub interfaces like *IRelationalGenerator* and *IObjectorientedGenerator* which are marking the generator for the specified schema and are defining some additional functionality the generator may be interested in implementing.

Furthermore, every generator can have additional extended data to nearly every physical element in the schema. This could, for example, be special options on tables, classes, columns, properties, etc., which are generator specific and not globally supported. These data has to be semi-managed by the generator. The persistent saving of that data will be controlled by dataMagus whereas the serialization must be implemented by the generator itself. To make additional generator data persistent the plug-in will have to implement the marker interface *IExtendedData*. This interface also instructs to implement the *IXmlSerializable*. During the serialization process the *WriteXml()* and *ReadXml()* methods will be called. So the plug-in developer can implement them to serialize and deserialize its data to be stored or restored.

With the *CleanExtendedData()* method the generator gets called at predefined times to check and clean up probable orphans in its data. This can happen when the user deletes or modifies data in the logical schema. The clean up will be done every time the user switches from the logical layer to the physical one. For example if the generator stores data about a relationship the user deleted, it has to clean up this data, too.

Fact is that the generator can specify additional data to nearly every physical element. To give the user the chance to fill in those data, the generator can implant several user controls (diverted from the *GeneratorControl*) for each physical element which needs extensions. This control will be requested by dataMagus when the additional data to a physical element is shown. The given control will be integrated in the *Extended Property* dialog window and gets loaded as soon as possible. For the saving of the data the closing or leaving events on the control itself can be used.

Very important: The generator is responsible for all its data.

If the generation of source code is requested, the *Generate()* method will be called. Before that, the generator gets a clone of the current schema with all data on it.
The generator now can iterate through the schema, generate the source code out of the schema and write the results into the stream which will be passed by reference with the *Generate()* method.

There are no limits about how the source code is generated. As a help, a plug-in developer can call the *WriteXml()* method on the current schema to get the whole schema definition in XML. This could be helpful when the developer wants to use XPath of XSLT for the generation.

With the following few steps a new generator can be implemented:

- Create a new project with the namespace dataMagus.PlugIns.Generators.<yourGenerator>.
- Add the dataMagus.Common assembly to your references.
- Create a new class (which must have a default constructor) by implementing the *IRelationalGenerator* or *IObjectorientedGenerator*.
- Optional: Create a new UserControl which extends the *GeneratorControl* class to create an input mask for additional data. You can create such a generator control for every physical element and return it by the specified method created by inheriting the specified interface.

4 Important interfaces to know about

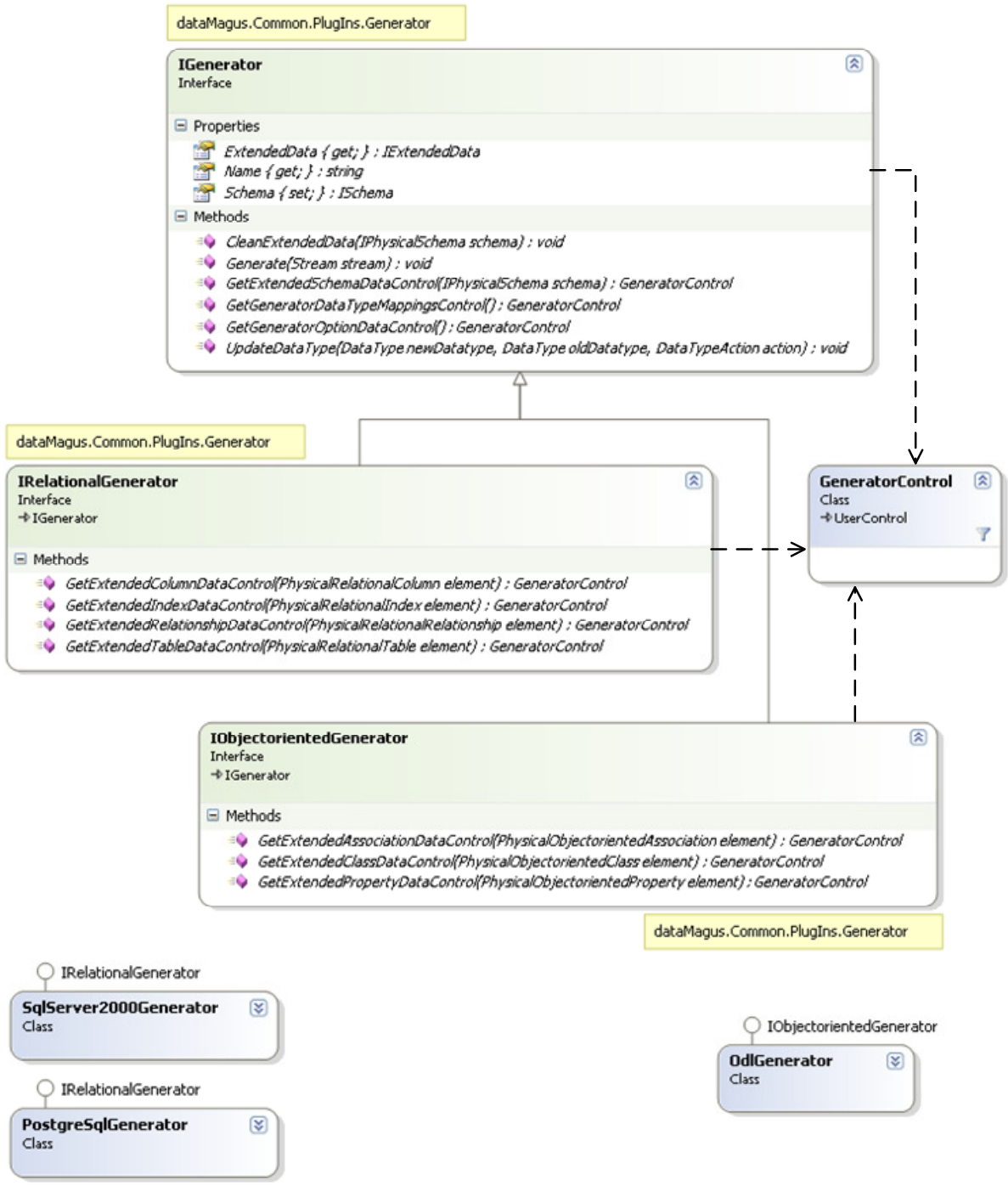

Figure 1 - Code generators

The description of the interfaces is not complete and will only include important things. Please seek advice in the source code documentation.

IGenerator
This interface is the root interface of all physical generators in the system.

Class member	Description
Name	Name of the generator
Schema	Clone of the schema to be generated.
ExtendedData	Extended data from the generator.
CleanExtendedData	Clean all extended generator data. Parameters: **schema**: Schema to clean the data for.
UpdateDataType	Update a datatype on the schema. Parameters: **newDatatype**: New datatype **oldDatatype**: Old datatype **action**: Action happend
GetGeneratorOptionDataControl	Get the generator control for the options tab on the generator dialog. Returns an instance of a generator control.
GetGeneratorDataTypeMappingsControl	Get the generator control for the datatype mapping. Returns an instance of a generator control.
GetExtendedSchemaDataControl	Get the generator control for the extended schema data. Returns an instance of a generator control.

Table 1 - Description of IGenerator

IRelationalGenerator
This interface is the root interface of all physical relational generators in the system.

Class member	Description
GetExtendedTableDataControl	Get the generator control for the extended table data. Parameters: **element**: Element to store the extended data for. Returns an instance of a generator control.
GetExtendedColumnDataControl	Get the generator control for the extended column data. Parameters: **element**: Element to store the extended data for. Returns an instance of a generator control.

GetExtendedRelationshipDataControl	Get the generator control for the extended relationship data. Parameters: **element**: Element to store the extended data for. Returns an instance of a generator control.
GetExtendedIndexDataControl	Get the generator control for the extended index data. Parameters: **element**: Element to store the extended data for. Returns an instance of a generator control.

Table 2 - Description of IRelationalGenerator

IObjectorientedGenerator
This interface is the root interface of all physical objectoriented generators in the system.

Class member	Description
GetExtendedClassDataControl	Get the generator control for the extended class data. Parameters: **element**: Element to store the extended data for. Returns an instance of a generator control.
GetExtendedPropertyDataControl	Get the generator control for the extended property data. Parameters: **element**: Element to store the extended data for. Returns an instance of a generator control.
GetExtendedAssociationDataControl	Get the generator control for the extended association data. Parameters: **element**: Element to store the extended data for. Returns an instance of a generator control.

Table 3 - Description of IObjectorientedGenerator

5 Important classes to know about

GeneratorControl
This class can be used by generators for specific extended properties. At the moment it's just a marker class on the base of a user control.

6 How and when the plug-ins will be loaded

The singleton class GeneratorManager cares about all generators in the application. At the start of the software the root directory of the application will be scanned for assemblies implementing the IGenerator interface. Instances of all found assemblies will be created, kept in memory and invoked on demand.

7 Good luck

The challenge of a new generator lies in the specific representation of the specified dialect. You must know quite much about the internal business model of dataMagus you have to parse.
Good luck.

Index of figures

Index of tables