How to add a new notation…?
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	JAZ

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 different notations on the different layers (logical, physical relational and physical object-oriented). Because there exists a variety of possible notations we've 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 notation plug-ins. It should give you a short overview about how to write a new notation 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 notation you must decide for which schema you'd like to write it for.

3 How to write your own notation

Before you can start writing a new notation you must understand several main interfaces. The highest abstract *INotation* interface is used to define the main functionality every notation must have. However, you shouldn't use it straight away.
Because all notations have a close relationship to its underlaying schema there are some important sub interfaces. The *ILogicalNotation* and *IPhysicalNotation* are the global interfaces to differ the two principal schemas. The *ILogicalNotation* interface directly defines the methods and properties to be implemented whereas the *IPhysicalNotation* is only a general marker interface for physical notations. So, if you'd like to write a notation for a relational schema you have to implement *IphysicalRelationalNotation*. If you'd like to write a notaion for a objectoriented schema you must implement *IPhysicalObjectorientedNotation*.

The conception for implementing new notations is based on the UserControls of the Microsoft .NET Framework and the System.Drawing namespace. Entities, tables or classes (depending on the physical schema) will be realized as UserControls by extending the *ElementControl*, whereas all relationships or associations must be drawn into the graphical context. Furthermore, there exists a *NotationHelper* class which helps to calculate the needed points in a notation very easily. However, the developer of the notation can also calculate the points at its own if he/she wants to. The event handling on the UserControl can be done directly by the plug-in developer.

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

- Create a new project with the namespace dataMagus.PlugIns.Notations.<yourNotation>.
- Add the dataMagus.Common assembly to your references.
- Create a new class (which must have a default constructor) by implementing the *ILogicalNotation*, *IPhysicalRelationalNotation* or *IPhysicalObjectorientedNotation*.
- Create a new UserControl which extends the *ElementControl* class.
- Optional: Create a class only used for drawing relationships
- Now read each section on the interfaces to be implemented and deal with it.
- Compile and copy your new notation into the root of the software directory.

- Now when you've done every thing right, your notation should be loaded automatically into the dropdown on the specified schema. If not, please look at one of the four existing notations to find and fix your problem.

4 Definition of elementary points

In the next sections we're going to use several expressions which must be understood.

4.1 The different points

There are several different points.

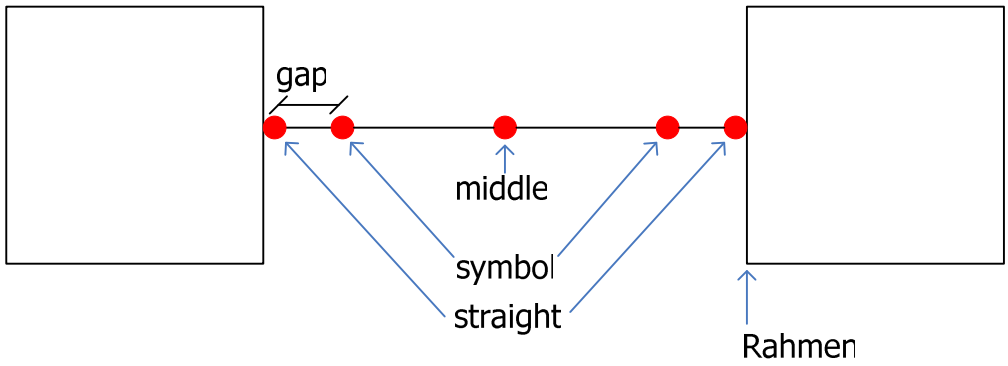

Figure 1 – Points within a regular relationship

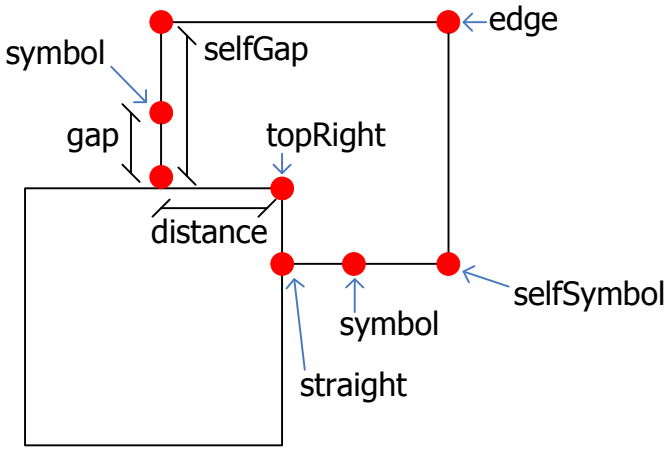

Figure 2 – Points within a self-relationship

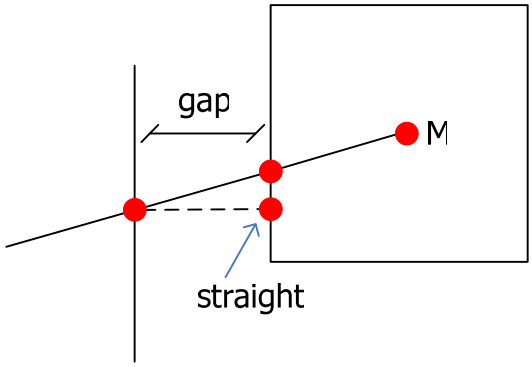

Figure 3 – Calculation of a straight point

The symbol point
When we talk about a symbol point on regular relationships between two different controls we mean the location where the straight line between the middle point of the two controls subtends the controls border. In nearly all cases this point will be calculated with a margin (gap) around the control. Normally, this point can be used to draw the literal cardinalities or role names.

The straight point
A straight point names the point that stands in a 90 degree angle from the symbol point to the control. It will mostly be used to draw the notations cardinality sign. This point depends always on its symbol point and will also be moved when the symbol point moves.

5 Important interfaces to know about

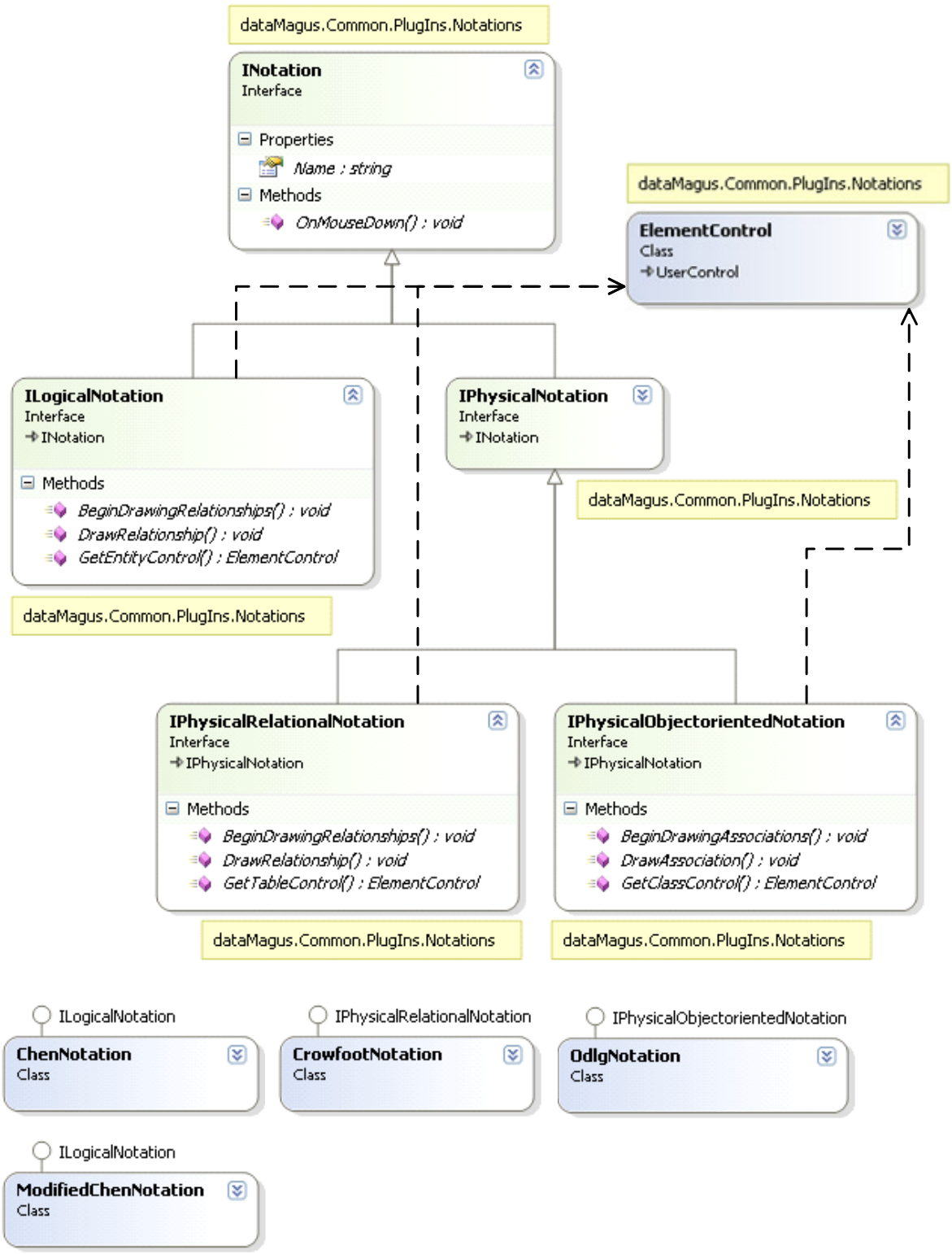

Figure 4 - Class diagram of the notations

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

INotation
This interface is the root interface of all notations in the system.

Class member	Description
Name	Name of the notation to be shown in the dropdown list.
OnMouseDown	Tells the notation that a MouseDown event has happend on the schema control. This can be important when relationships (which will be drawn) also have (drawn) controls to react. Parameters: **e**: MouseEventArgs

Table 1 - Description of *INotation*

ILogicalNotation
This interface is the root interface of all logical notations in the system.

Class member	Description
BeginDrawingRelationships	Tells the notation that the drawing of the relationships will begin. This can be helpful, if the notation wants to cleanup before starting the draw.
GetEntityControl	Gets the control for an entity. Parameters: **entity**: Logical entity on which the control is based. Returns an *ElementControl* if successful or null to ignore.
DrawRelationship	Tells the notation to draw a relationship. Parameters: **relationship**: Logical relationship to draw. **element1**: Control of the first element of the relationship. **element2**: Control of the second element of the relationship. **showRelationshipName**: Indicator if the name of the relationship should show up. **showRolenames**: Indicator if the name of the roles should show up. **showCardinalities**: Indicator if the cardinalities should show up. **context**: Graphics context to draw in.

Table 2 - Description of *ILogicalNotation*

IPhysicalNotation
This interface is the root marker interface of all physical notations in the system.

IPhysicalRelationalNotation
This interface is the root interface of all physical relational notations in the system.

Class member	Description
BeginDrawingRelationships	Tells the notation that the drawing of the relationships will begin. This can be helpful if the notation wants to cleanup before starting to draw.
GetTableControl	Gets the control for a table. Parameters: **table**: Physical table the control is based on. Returns an *ElementControl* if successful or null to ignore.
DrawRelationship	Tells the notation to draw a relationship. Parameters: **relationship**: Logical relationship to draw. **element1**: Control of the first element of the relationship. **element2**: Control of the second element of the relationship. **showRelationshipName**: Indicator if the name of the relationship should show up. **showRolenames**: Indicator if the name of the roles should show up. **showCardinalities**: Indicator if the cardinalities should show up. **context**: Graphics context to draw in.

Table 3 - Description of *IPhysicalRelationalNotation*

IPhysicalObjectorientedNotation
This interface is the root interface of all physical object-oriented notations in the system.

Class member	Description
BeginDrawingAssociations	Tells the notation that the drawing of the relationships will begin. This can be helpful if the notaion wants to cleanup before start the drawing.
GetClassControl	Gets the control for a class. Parameters: **classElement**: Physical class the control is based on. Returns an *ElementControl* if successful or null to ignore.
DrawAssociation	Tells the notation to draw a relationship. Parameters: **association**: Physical association to draw. **element1**: Control of the first element between the association. **element2**: Control of the second element between the **association**. **showAssociationName**: Indicator if the name of the association should show up. **showRolenames**: Indicator if the name of the roles should show up. **showCardinalities**: Indicator if the cardinalities should show up. **context**: Graphics context to draw in.

Table 4 - Description of *IPhysicalObjectorientedNotation*

6 Important classes to know about

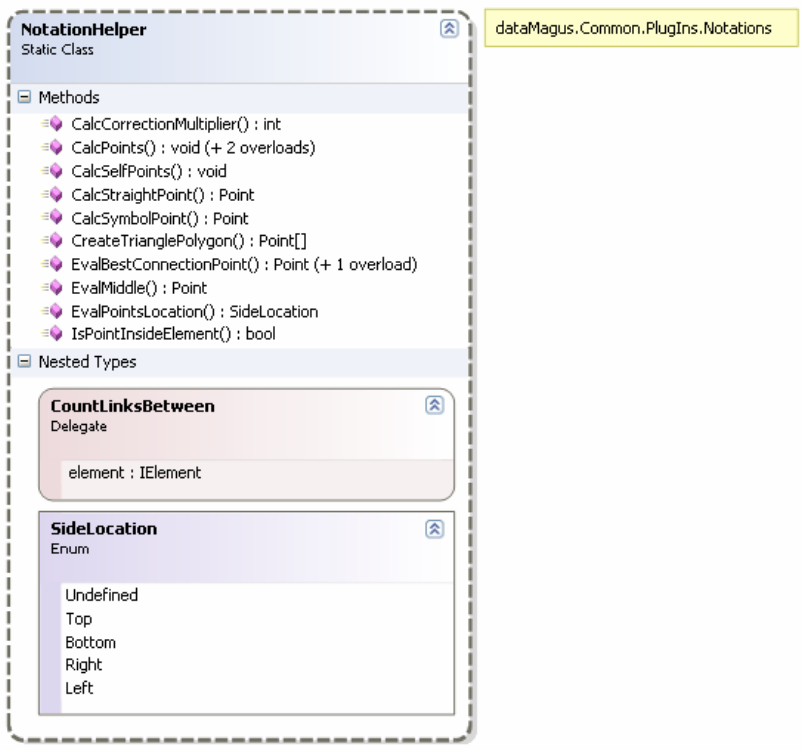

Table 5 - Class *NotationHelper*

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

ElementControl
This class can be used by notations for elements on the schema. It implements a small communication between the element and the schema. Basically, you don't have to care about that. However you should override the following methods:

Class member	Description
DrawControl	Draws the whole control into a graphics context. Parameters: **context**: Graphics context
CalcControlSize	Calculate the whole size of the control Returns: Size of the control.

Table 6 - Description of *ElementControl*

NotationHelper
This static class contains helper functionality for the notations.

Class member	Description
CalcCorrectionMultiplier	Calculates a multiplier which then will be multiplied with the X or Y axis of the point. This will help to correct the position of overlaying points. Each call will decrease the multiplier and hop between the negative and positive numerical range. Parameters: **master**: Element of the first control. **masterLinksCounter**: Method implementing the CountLinksBetween delegate. **slave**: Element of the second control. **betweenCounter**: Dictionary<long, int>. The helper method can store data in. Returns the multiplier (or factor) to be used for correction of the needed points. Examples: If an element has 4 relationships (even) with another element, this method will return -2 / 2 / -1 / 1 / 0 If an element has 3 relationships (odd) with another element, this method will return -1 / 1 / 0
CalcPoints	Calculates helpful points between two element controls. Parameters: **master**: Master control. **slave**: Slave control. **gap**: Gap around the controls. **divergency**: Divergency on the X or Y axis depending on the location of the points. Out Parameters: **symbol1**: The symbol point of the master. **symbol2**: The symbol point of the slave. **straight1**: The straight point of the master. **straight2**: The straight point of the slave. **middle**: The middle point between master and slave.
CalcSelfPoints	Calculates helpful points for element controls having a self relationship. The calculation is based on the top right of the control. Parameters: **ctrl**: Control with the self relation. **gap**: Gap around the control. **selfGap**: Enlarge gap to calculate the selfSymbol point. **distance**: Distance from the top right of the control. Out Parameters: **symbol1**: The symbol point on the top. **symbol2**: The symbol point on the right.

	straight1: The straight point on the top. **straight2**: The straight point on the right. **selfSymbol1**: The self symbol point on the top. **selfSymbol2**: The self symbol point on the right. **edge**: The edge where the self relation gets together.
IsPointInsideElement	Checks if a point is still inside an element control. This method can be used to check if a point is leaving the element. Parameters: **element**: Element control. **p**: Point to be checked. **gap**: Gap around the control. Returns true if point is still inside, otherwise false.
EvalBestConnectionPoint	Evaluates the best connection point of four points (ex. diamond). Parameters: **ctrl**: Control which will be linked to the connection point. **top**: Top point of the figure. **bottom**: Buttom point of the figure **left**: Left point of the figure **right**: Right point of the figure Returns the best point to link to.
EvalPointsLocation	Evaluates where the symbol and straight point is located. These two points are enough for the evaluation because they have a functional dependency on each other. If symbol point is equal the straight point, *SideLocation.Undefined* will be returned. Parameters: **symbolPoint**: Symbol point. **straightPoint**: Straight point Returns the location of the points.
CreateTrianglePolygon	Creates a triangle polygon where the height is calculated by the difference between the start and help point. Parameters: **start**: Start point of the triangle. **help**: Help point of the triangle. **width**: Width of the triangle. Returns *Point[]* containing the polygon.
EvalMiddle	Evaluates the middle between two points. Parameters: **p1**: Point 1. **p2**: Point 2. Returns the middle point.

Table 7 - Description of *NotationHelper*

7 How and when the plug-ins will be loaded

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

8 Good luck

The challenge of a new notation lies in the specific representation of the logical or physical layer. The *NotationHelper* class can help you solving rudimental things. Although these little helpers are not extremely good and show no high-performance at drawing, it can help to concentrate on the essentials of the new notation. We've chosen the pragmatic approach. Good luck.

Index of figures

Index of tables