

# Visual Studio

## Visualization and Modeling SDK Lab – Part 1 / 6

### 1 Introduction

#### 1.1 Objective of this Lab.

The objective of this Lab is to create a domain-specific language (DSL) by using the Visual Studio Visualization and Modeling SDK (VMSDK, formerly known as DSL Tools), and to customize it by writing code. The VMSDK is especially beneficial when you want to create a vertical language that is suitable for your business, and from the models that the language manipulates, generate the code for your business framework. Nevertheless, because it is difficult to ensure that everyone who takes this training knows the professional tasks that are addressed by the targeted business framework, we will settle for a horizontal (that is, technical) DSL that lets us learn how to use different aspects of VMSDK.

In addition to providing training, this document also shows an approach to DSL design.

#### 1.2 Prerequisites

Before you start this Lab, ensure that you have these tools installed on your computer:

- [Visual Studio 2010](#) – any edition except Express
- [Visual Studio 2010 SDK](#)
- [Visualization and Modeling SDK](#)

In addition, to simplify the creation of your code generators, download and install a T4 editor from <http://www.t4editor.net/>. A T4 editor provides syntactic coloring to aid in the editing of your T4 templates.

Other resources:

- [Samples, updates, and more labs at the VMSDK Home](#)
- [Discussion and problem-solving at the VMSDK Forum](#)
- [Visualization and Modeling User Guide](#)
- [Jean-Marc Prieur's blog](#)

#### 1.3 Plan

The approach adopted in this Lab is the following:

1. **Creation of a DSL from an "Empty" DSL model. (45 steps: this chapter)**
2. Creation of a metamodel for finite-state automata: this metamodel includes the StateMachine, State, and Transition concepts. For now we will assume that the states do not have substates (27 steps).
3. Specification of graphical notation with our DSL (51 steps).
4. Test of our DSL in the test version of Visual Studio.
5. Improvement of the MMI of the DSL.
6. Addition of consistency rules to the model.
7. Addition of validation rules to the model.
8. Creation of a code generator that targets a Framework for implementation of finite-state automata.
9. Test of our DSL on a few amusing examples.
10. Addition of a setup to distribute our DSL.

#### Acknowledgements

Many thanks to all the people who contributed to this Lab, especially to those who have used it and have given us feedback. The principal author of this lab is [Jean-Marc Prieur](#).

## 2 Let's go!

### 2.1 Creating a DSL

#### 2.1.1 Using the DSL creation wizard

1. In Visual Studio, create a project.

**File -> New -> Project.**

The **New Project** dialog box appears.

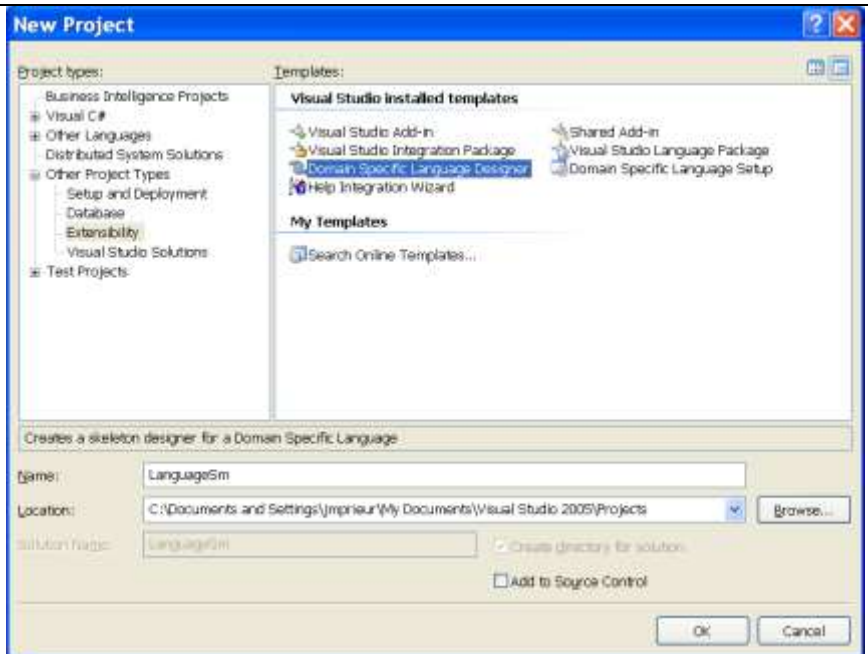
2. In the projects category "Other Project Types" and the sub-category "Extensibility", **choose to create a project of the following type:**

**Domain Specific Language Designer**

3. Give it the name **LanguageSm**.

4. Finally, click **OK**.

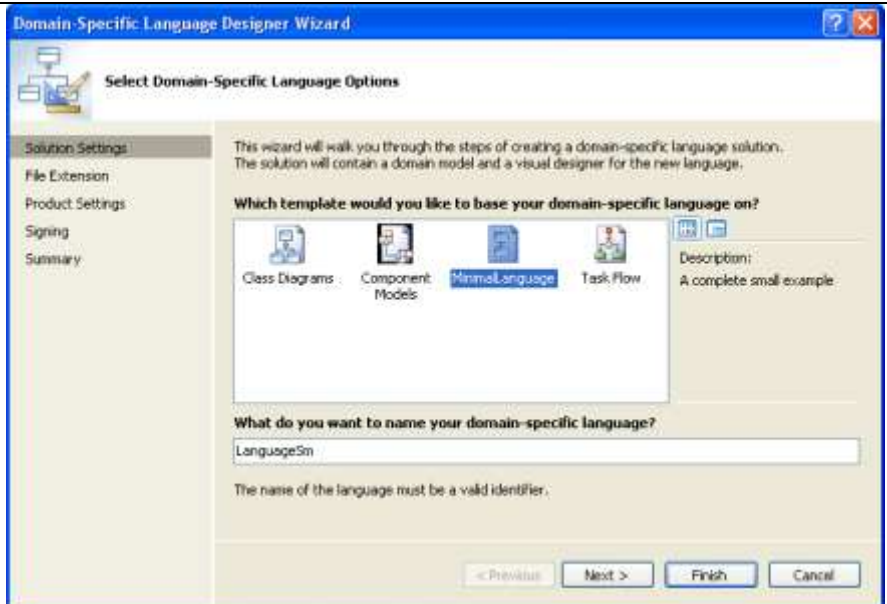
Note: Because this template creates a complete solution containing more than one project, you must start the New Project dialog from the File menu. You cannot add to an existing solution.



The wizard lets you create a DSL by using one of the DSL models. It offers you several models, including:

- **Class Diagrams** constitute a diagram of the diagram type UML classes.
- **Component Models** lets you create designers to assemble components from sub-components.
- **Minimal Language** is a language that includes a single concept. It is a virtually empty language just to let you get started.
- **Task Flow** creates a language of the type UML status-activity diagrams.

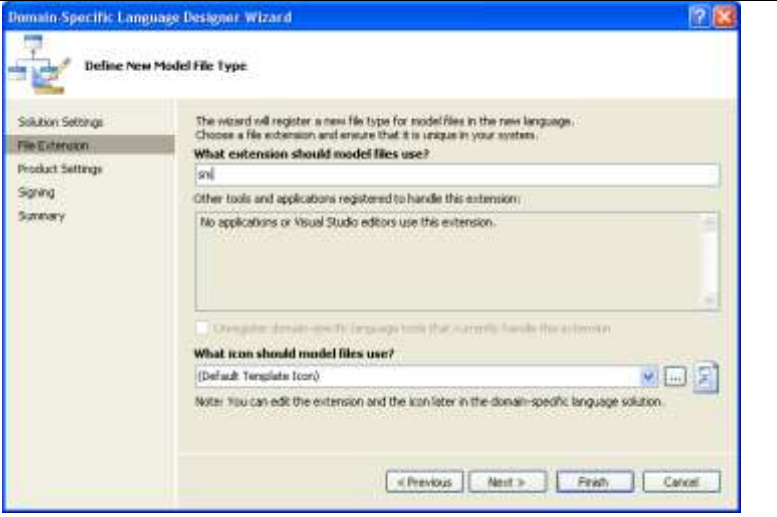
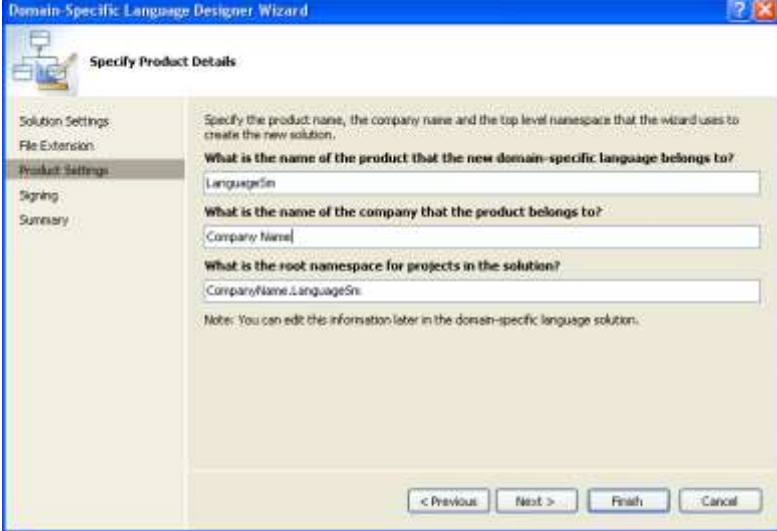
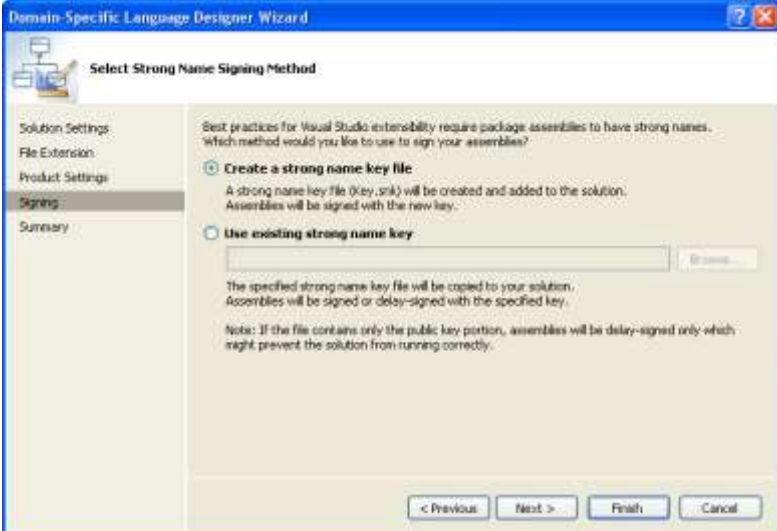
5. **Choose the Minimal Language** and **click Next >**



The next step consists in giving a file name extension for the files that will contain the concepts of your language.

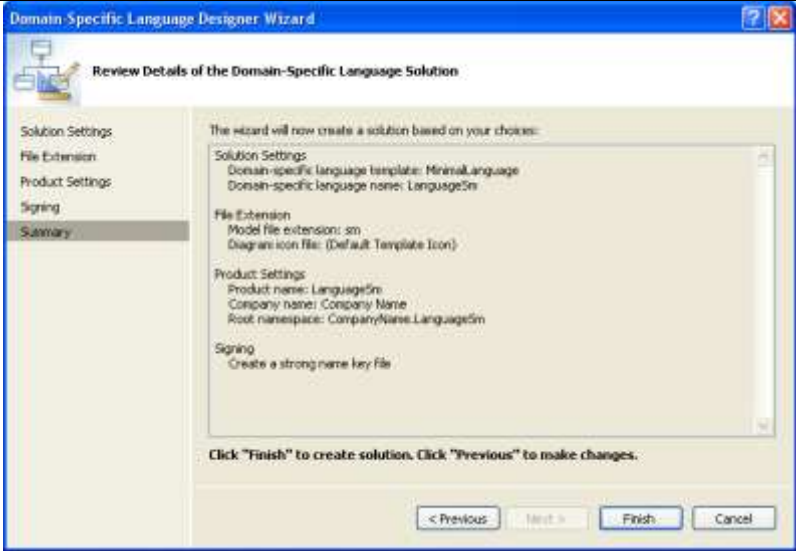
A check is carried out to find out if the extension is already used by other DSLs.

6. **Choose .sm** as an extension for our model and **click Next >**

	 <p>The wizard will register a new file type for model files in the new language. Choose a file extension and ensure that it is unique in your system.</p> <p><b>What extension should model files use?</b></p> <p>snl</p> <p>Other tools and applications registered to handle this extension:</p> <p>No applications or Visual Studio editors use this extension.</p> <p><input type="checkbox"/> Unregister domain-specific language tools that currently handle this extension</p> <p><b>What icon should model files use?</b></p> <p>(Default Template Icon)</p> <p>Note: You can edit the extension and the icon later in the domain-specific language solution.</p> <p>&lt; Previous Next &gt; Finish Cancel</p>
<p>The following dialog box lets us specify the name of the DSL and the name of your company.</p> <p>By default, the concatenation of the two gives the namespace in which the code that comprises the DSL will be generated.</p> <p>7. Accept the suggestion by <b>Clicking on the button <u>N</u>ext &gt;</b></p>	 <p>Specify the product name, the company name and the top level namespace that the wizard uses to create the new solution.</p> <p><b>What is the name of the product that the new domain-specific language belongs to?</b></p> <p>LanguagePrism</p> <p><b>What is the name of the company that the product belongs to?</b></p> <p>Company Name</p> <p><b>What is the root namespace for projects in the solution?</b></p> <p>CompanyName.LanguageSnl</p> <p>Note: You can edit this information later in the domain-specific language solution.</p> <p>&lt; Previous Next &gt; Finish Cancel</p>
<p>The assistant will generate two compilations for your DSL. As these are parts of a Visual Studio 2005 Package, they must be signed (they will go elsewhere in the GAC).</p> <p>The following step consists, therefore, of supplying a key for signing the compilations, or letting the assistant create one.</p> <p>8. Let the assistant generate a key and click <b><u>N</u>ext &gt;</b></p>	 <p>Best practices for Visual Studio extensibility require package assemblies to have strong names. Which method would you like to use to sign your assemblies?</p> <p><input checked="" type="radio"/> <b>Create a strong name key file</b></p> <p>A strong name key file (key.snk) will be created and added to the solution. Assemblies will be signed with the new key.</p> <p><input type="radio"/> <b>Use existing strong name key:</b></p> <p><input type="text"/> Browse...</p> <p>The specified strong name key file will be copied to your solution. Assemblies will be signed or delay-signed with the specified key.</p> <p>Note: If the file contains only the public key portion, assemblies will be delay-signed only which might prevent the solution from running correctly.</p> <p>&lt; Previous Next &gt; Finish Cancel</p>

The wizard summarizes your choices, and you can change them if you like. (Which is not advised if you want your experiment to be consistent with the results of this Lab).

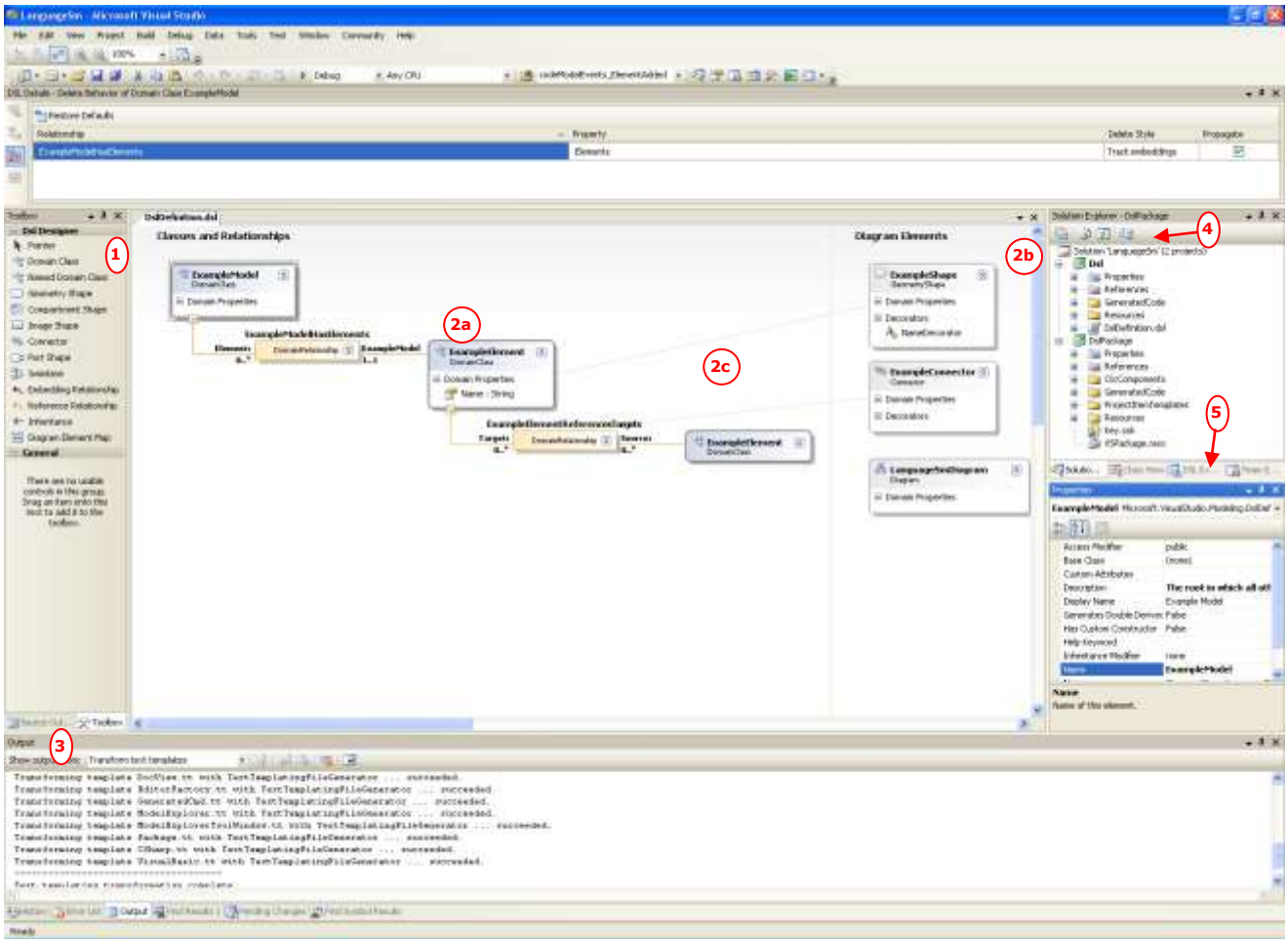
9. Leave the two boxes checked and **click** the **Finish** button



Once the assistant has finished, a Visual Studio 2008 application is created. It consists of two projects:

- **Dsl**, which contains the customized code by freely using the partial class mechanism.
- **DslPackage**, which contains the code that enables our DSL to be hosted and to interact with Visual Studio 2008. A part of its code is also generated from the file `DslDefinition.dsl`

A great deal of code is, as we have seen, generated automatically from the DSL description file by applying a *Custom Tool* called `TextTemplatingFileGenerator` to the files in **T4 format** (a textual language that resembles ASP). T4 files have the file name extension `.tt`.



## 10. Note on the window above:

1. The **toolbox** is associated with the file extension `.dsl`. It also contains the domain elements and domain relationships of the metamodel and their graphical representations.

The **domain model** is manipulated by using the concepts of:

- **Domain Class**, which represents the concepts handled in our language. There is also a current specialization of Domain Class that bears the name: **Named Domain Class**
- There are three types of domain relationships:
  - Aggregation (**Embedding Relationship**), to specify that a concept has a lifespan included in that of another.
  - simple association between two concepts (**Reference Relationship**),
  - and the inheritance relationship (**Inheritance**)

We note that **Domain Class**, like **domain relationships**, has **Properties**. In the metamodel of the DSLs, the relationships are thus attributed (attributes carriers), which constitutes a difference compared to the MOF (basic metamodel for UML).

The **graphical notation** is specified by the following concepts:

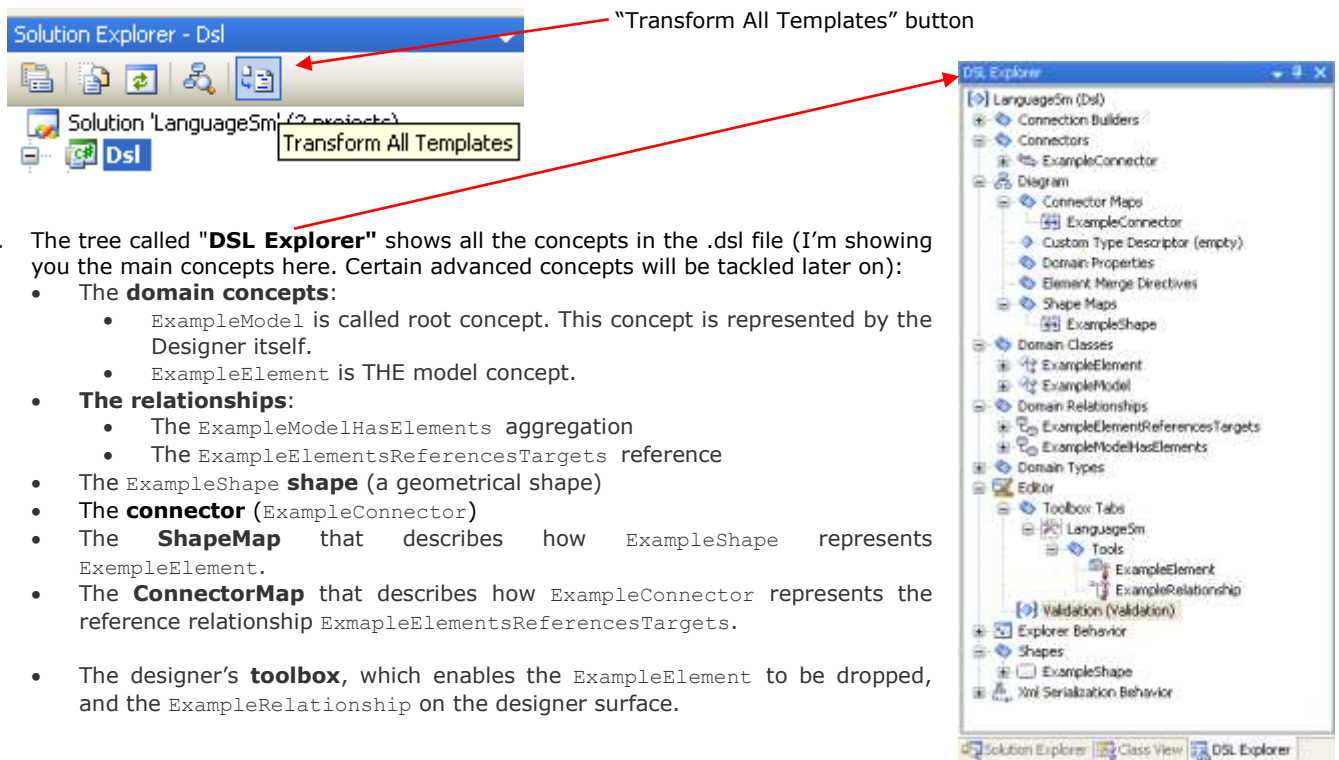
- **Geometry Shape**, which represents a concept, in the form of a geometrical figure (rectangle, rectangle with rounded corners, ellipse, etc.)
- **Image Shape**, which represents a concept in the form of an image
- **Compartment Shape**, which represents a concept in the form of a geometrical shape that has compartments
- **Connector**, which represents a relationship (in the form of a line that connects the *shapes*)

Moreover:

- the connectors can be forced to return to the shapes under the **Port shapes** (small shapes that are positioned along the edges of other shapes)
- the shapes themselves may be forced to be placed in the water lanes (**Swimlane**), just like in a swimming pool!

Finally, the relationship between a Domain Class and a shape (or a relationship and a connector) is specified by the concept of **Diagram Element Map** (in fact, by their specializations, **ShapeMap** and **ConnectorMap**).

2. Representation of the metamodel (2a), of the graphical notation (2b), and the element map (2c)
3. The **Output window**, which shows the result of the transformation of the templates. At the time the solution is created, C# files are automatically generated from the `.dsl` file. This is the summary of the result of this transformation, which is seen in the Output window, in the Transform Text templates tab.
4. The **"Transform All Templates" button**, which enables all of the solution Custom Tools to be applied (this function was automatically applied when the project was loaded).



The image shows two windows from the Visual Studio IDE. On the left is the 'Solution Explorer - Dsl' window, which displays a project named 'LanguageSm' with a sub-project 'Dsl'. A button labeled 'Transform All Templates' is visible. A red arrow points from this button to the 'DSL Explorer' window on the right. The 'DSL Explorer' window shows a hierarchical tree of DSL concepts. The tree structure is as follows:

- LanguageSm (Dsl)
  - Connection Builders
  - Connectors
    - ExampleConnector
  - Diagram
    - Connector Maps
      - ExampleConnector
    - Custom Type Descriptor (empty)
    - Domain Properties
    - Element Merge Directives
    - Shape Maps
      - ExampleShape
  - Domain Classes
    - ExampleElement
    - ExampleModel
  - Domain Relationships
    - ExampleElementReferencesTargets
    - ExampleModelHasElements
  - Domain Types
  - Editor
    - Toolbox Tabs
      - LanguageSm
        - Tools
          - ExampleElement
          - ExampleRelationship
  - Validation (Validation)
  - Explorer Behavior
  - Shapes
    - ExampleShape
  - Xml Serialization Behavior

5. The tree called **"DSL Explorer"** shows all the concepts in the `.dsl` file (I'm showing you the main concepts here. Certain advanced concepts will be tackled later on):
  - The **domain concepts**:
    - `ExampleModel` is called root concept. This concept is represented by the Designer itself.
    - `ExampleElement` is THE model concept.
  - The **relationships**:
    - The `ExampleModelHasElements` aggregation
    - The `ExampleElementsReferencesTargets` reference
  - The `ExampleShape` **shape** (a geometrical shape)
  - The **connector** (`ExampleConnector`)
  - The **ShapeMap** that describes how `ExampleShape` represents `ExampleElement`.
  - The **ConnectorMap** that describes how `ExampleConnector` represents the reference relationship `ExampleElementsReferencesTargets`.
  - The designer's **toolbox**, which enables the `ExampleElement` to be dropped, and the `ExampleRelationship` on the designer surface.



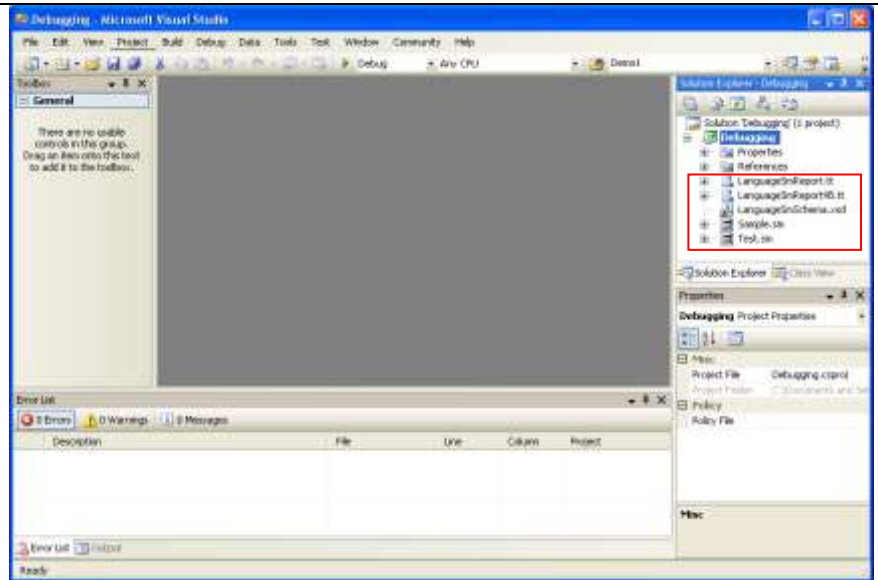
## 2.1.2 First run of the DSL

12. Run your DSL from the application: (**Debug -> Start without debugging**).

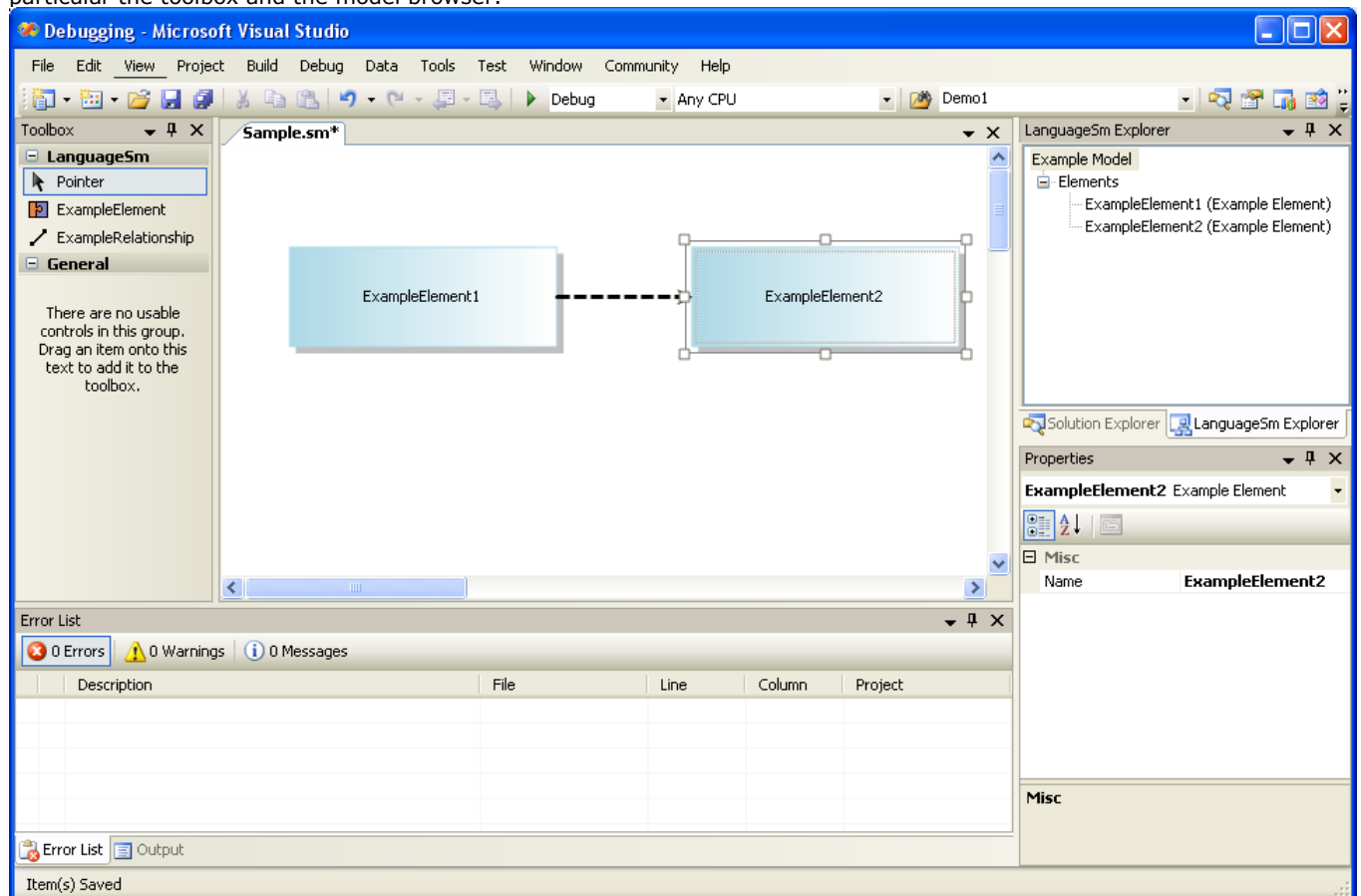
This is compiled and a third project created by the DSL Tools assistant (a debugging application) is shown in its own application, which is launched in the **test version** of VS 2008.

There we find in particular:

- A `Sample.sm` file that contains an example of your DSL.
- The `.XSD` that describes our `.sm` files.
- Two `.tt` files that contain the artifact generators (here enabling you to generate HTML files that document the `Sample.sm` model). These files are both in T4 format but the code used to describe them is C# in the first case and Visual Basic in the second case.

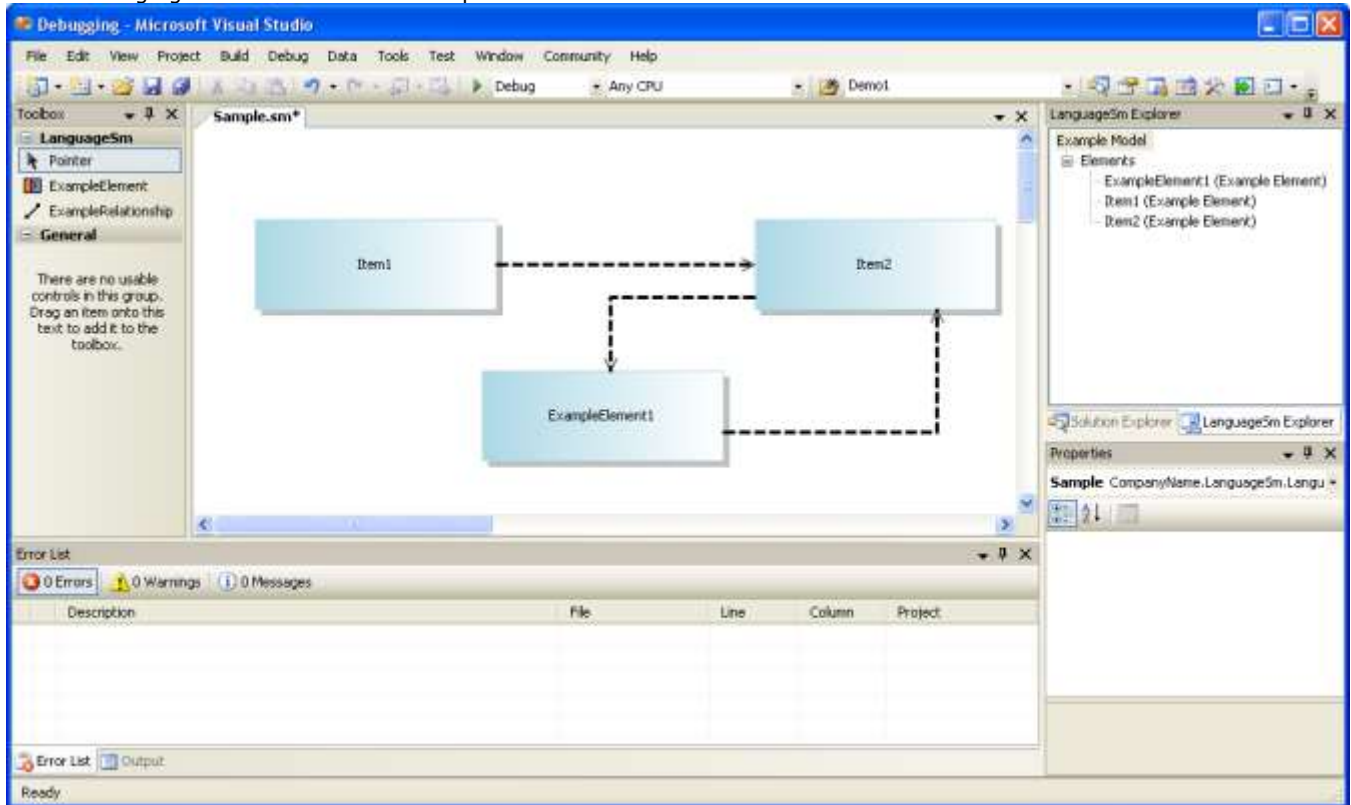


13. **Open the `Sample.sm` file** (by double-clicking it in the browser application). The corresponding diagram appears. Note in particular the toolbox and the model browser.



14. **Play around a little** with the model: add a box, and connections, and look at the model browser and the properties of the boxes and connections. **Save your model**. In a moment, we will look at all this again from the metamodel perspective.

The following figure shows the result of experimentation:



<p>15 From the browser application, right-click the LanguageSmReport.tt file and select the contextual command <b>"Run Custom Tool"</b>. The file LanguageSmReport.txt is generated. (Ignore the security warning.)</p> <p>16. Expand the LanguageSmReport.tt node; the LanguageSmReport.txt node appears. Double-click above to view the contents.</p>	<p>The screenshot shows the 'LanguageSmReport.txt' file open in Visual Studio. The content is as follows:</p> <pre>Generated material. Generating code in C#.  Item1 Item2 ExampleElement1</pre>
<p>17 Close the version of Visual Studio that contains the Debugging application.</p>	

We are now going to look at the structure of the DSL that is generated by the DSL Tools wizard, from the code generator.

## 2.1.3 Understanding

At the start of the Lab we created a DSL (very modest) without coding anything. We then experimented by creating three shapes and three connectors, and then generated a .txt file that has content that corresponds to the model.

We are now going to gain a greater understanding of what was generated by the DSL Tools assistant. The metamodel has just these few concepts:

- the metamodel
- the graphical notation
- the connection between the metamodel and the graphical notation
- the persistence of the model and the diagram
- the artifacts generator

In the following chapter, we will develop the metamodel of our DSL to be able to model simple state machines.

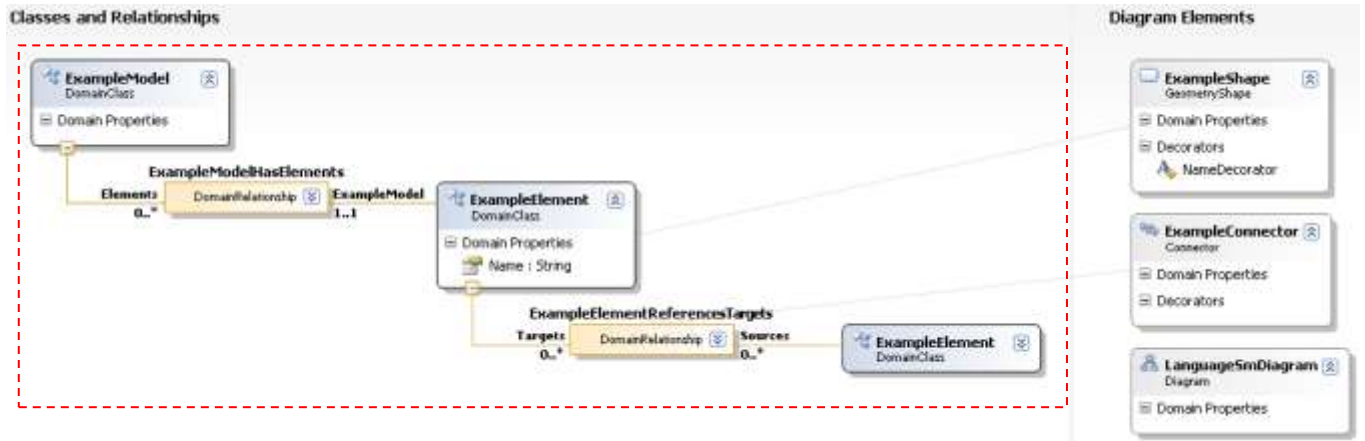
### 2.1.3.1 Understanding the metamodel

- 18 In the version of Visual Studio that contains the `LanguageSm` application, look at the content of the description of the DSL: the `DslDefinition.dsl` file. It is, in a sense, a "DSL for creating DSLs".

The DSL has two parts (which are in two Swimlanes):

- the metamodel, (or Domain Model) in the left section called "Classes and Relationships"
- the graphical notation, in the right section called "Diagram Elements"

The connection between the Domain Model and the graphical notation is represented in the form of connectors that link the classes or relationships, in shapes or connectors.



The Domain Model has **two concepts**:

- `ExampleModel` which is the *Root concept*, that is, the concept that is associated with the Designer (`LanguageSmDiagram`). This concept will not be shown in the form of a box, but rather, in the shape of the **diagram itself**.
- `ExampleElement`, which is the only true concept of the model. `ExampleElement` possesses a `Name` property of the `String` type.

We also find **two domain relationships** here:

- an embedding relationship called `ExampleModelHasElements` that links an `ExampleModel` to (see the cardinality `0..*`) `ExampleElement` whose function is to serve as the `Elements` of the `ExampleModel`. The `ExampleModel` is a sample of the `ExampleElement` (above the cardinality `1..1`)
- A single reference relationship called `ExampleElementReferencesTargets` that links two `ExampleElements`. An `ExampleElement` has as its image, through the relationship of several `ExampleElements` whose functions are to be the `Targets`, for several precedent `ExampleElements`, whose function is to serve as the `Sources` of the relationship.

- 19 Select the relationship between `ExampleModel` and `ExampleElement` and look at its properties. This relationship is an *Embedding*, and is called `ExampleModelHasElements`.

It is important to understand that, in the generated DSL, the relationship will be carried out in two ways:

- by the **functions** `Elements` in `ExampleModel` and `ExampleModel` in `ExampleElement`.
- by a C# `ExampleModelHasElements` class (there will be as many instances as there are instances of the relationship) whose characteristics are defined in the categories of "Definition" and "Code" properties. Note that it is possible to specify whether we can have two instances of the relationship between the same concepts (*Allow Duplicates*, which must not be True here, because it is the case of an embedding relationship). We can also have a basic relationship with the relationship. The "Code" category specifies the visibility of the class, and specifies (by *GeneratesDoubleDerived*) whether we can generate two classes (`ExampleModelHasElementsBase` that contain any implementation, and `ExampleModelHasElements` in the partial classes in which we can alter the implementations generated), or a single one of them.

Properties

**ExampleModelHasElements** Domain Relationship

**Code**

Access Modifier	public
Custom Attributes	
Generates Double Derived	False
Has Custom Constructor	False
Inheritance Modifier	none

**Definition**

Allows Duplicates	False
Base Relationship	(none)
Is Embedding	True
Name	ExampleModelHasElements
Namespace	CompanyName.LanguageSm

**Documentation**

Notes

**Resources**

Description	Embedding relationship bet
Display Name	Example Model Has Elements
Help Keyword	

**Name**

Name of this element.

- 20 Now select the **Elements** function of `ExampleModel` in the aggregation.



The function of the `ExampleElement` in the relationship is to serve as the elements of the `ExampleModel`. There are several of them because the cardinality shows (0..\*). In the properties of the "Definition" category, we will note in particular:

- The **PropertyName** property (being `Elements`). This is the name that we have chosen. This property defines the name (because the property "IsPropertyGenerator" is True), of the property of the class `ExampleModel` and its type is a collection of `ExampleElement`.
- The Property Name (in this case, `ExampleModel`) defines the name of a property of the class that implements the relationship (`ExampleModelHasExampleElements`). Therefore, the type will be an `ExampleModel`.
- **Is property generator** means that in the class `ExampleModel`, a property `Elements` will be generated, whose type is a collection of `ExampleElement`. **Property Getter Access Modifier** and **Property Setter Access Modifier** informs us that this property will have public settings and will obtain accessors. **Is property browsable** indicates that the property will be visible in the properties window of the `ExampleModel` instances.

Select the opposite function in the relationship (called `ExampleModel`). Look at its properties.

- 21 Now select the other relationship (`ExampleElementReferencesTargets`) and its functions (`Targets` of multiple cardinality and also `Sources` of multiple cardinality).

Make sure that the property **Name** of the `Targets` function is called `Source`: this function will be generated in the form of a `Targets` collection from `ExampleElement` in the `ExampleElement` class, but also in the form of a `Source` property of the `ExampleElement` type in the class that implements the (`ExampleElementReferencesTargets`) relationship.

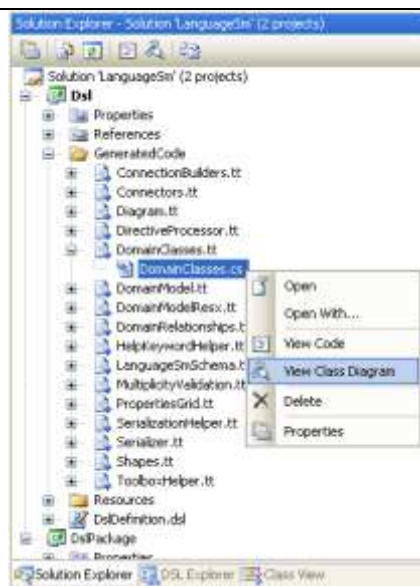
The same is true for the `Sources` function.



At this stage, we have looked at the metamodel that was created. We will now see what the classes that are generated from this metamodel look like.

- 22 In the `DomainClasses.cs` file that is generated from the description of the DSL by the text template called `DomainClasses.tt`, request the command "**View Class Diagram**" from the shortcut menu.

We obtain the two classes that have been generated:

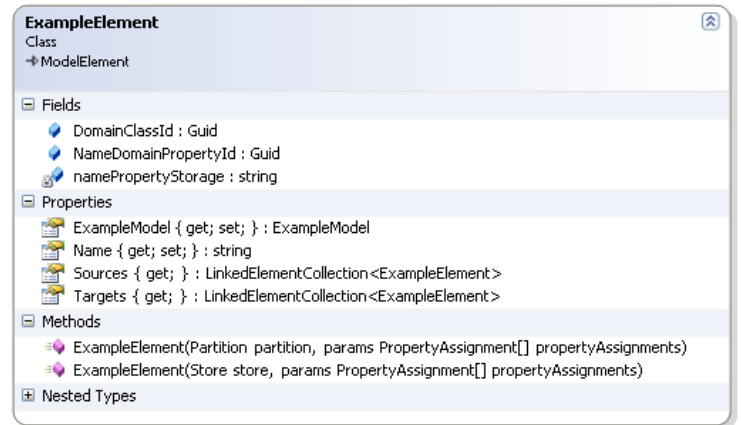


23 "Open" the `ExampleElement` class.

Here we find:

- public fields that are the GUID. The metamodel is such that the associations are bi-directional (for example, if by program we add an `ExampleElement e1` to the `Targets` property (which is a collection from `ExampleElement`) from another `ExampleElement e2`, then automatically `e2` will be found in the `Sources` collection of `e1`). The GUIDs appear in the implementation of this functionality. We will also find them in the validation rules in section 2)
- of the private fields that store the properties of the class (`namePropertyStorage`)
- of properties that correspond to the attributes of the meta-class and the functions that are navigable from this class
- the builders.

- 24 Request to hide the content of the `Fields` and `Methods` compartments of each of the classes and use the **Show as Association** and **Show as Association Collection** commands to find the following diagram.



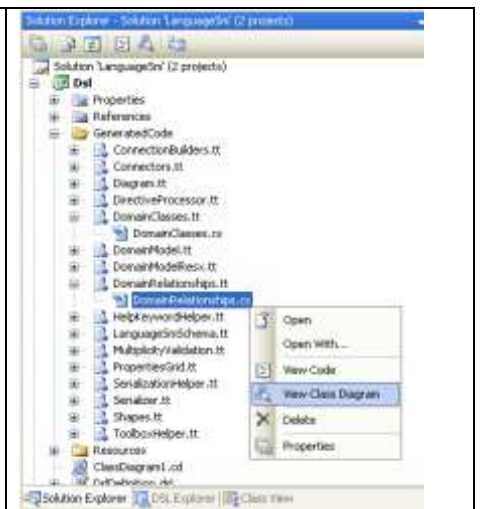
Here we see that an `ExampleModel` is in relationship with the `ExampleElement` by way of the `Elements` function. An `ExampleElement` is linked to just one `ExampleModel` (`ExampleModel` function). One `ExampleElement` is in relationship with other `ExampleElements` (two functions `Targets` and `Sources`).

Elsewhere the model contains the domain relationships themselves (`ModelHasElements` and `ExampleElementReferencesTargets`). This enables access to the two players in the relationships.

- 25 In the `DomainRelationships.cs` file that is generated from the description of the DSL by the text template called `DomainRelationships.tt`, request the command "View Class Diagram" from the shortcut menu.

Two new classes are added to the classes diagram. After a little work (hiding the compartments of the fields), transforming the properties in associations, you will obtain the diagram from the following page.

It will be very useful for us to have such a model of the classes at hand when we write our code generators, or look at the model.

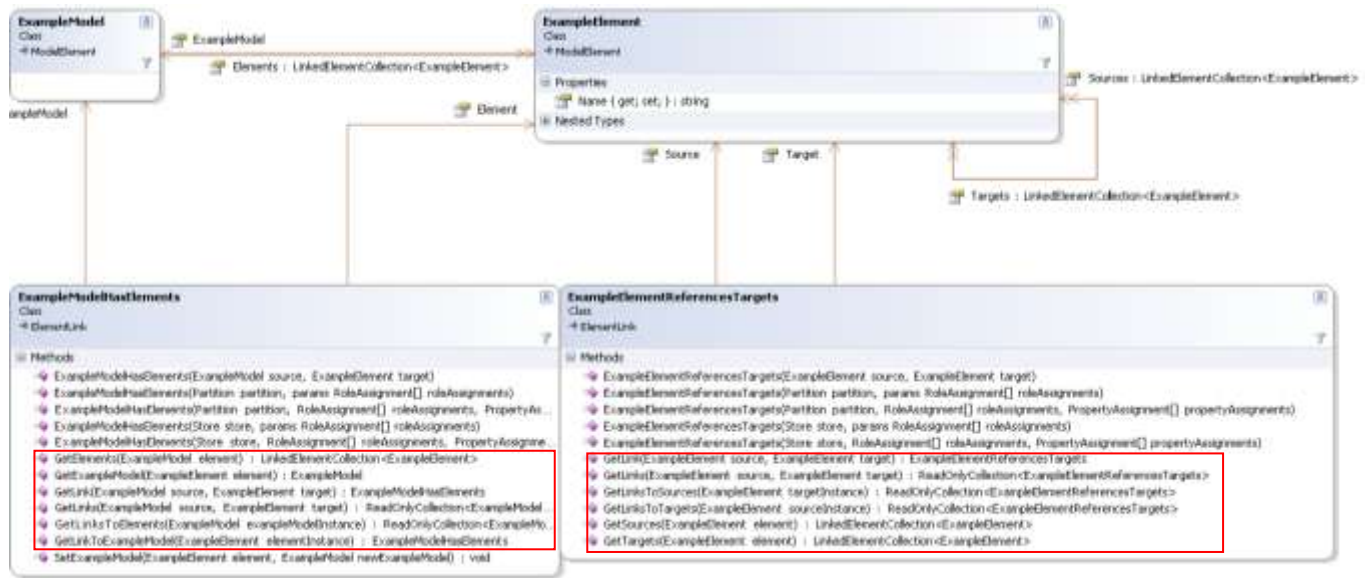


In the diagram above, note that the `ExampleModelHasElements` class possesses, in addition to the builders, static methods that enable us to find:

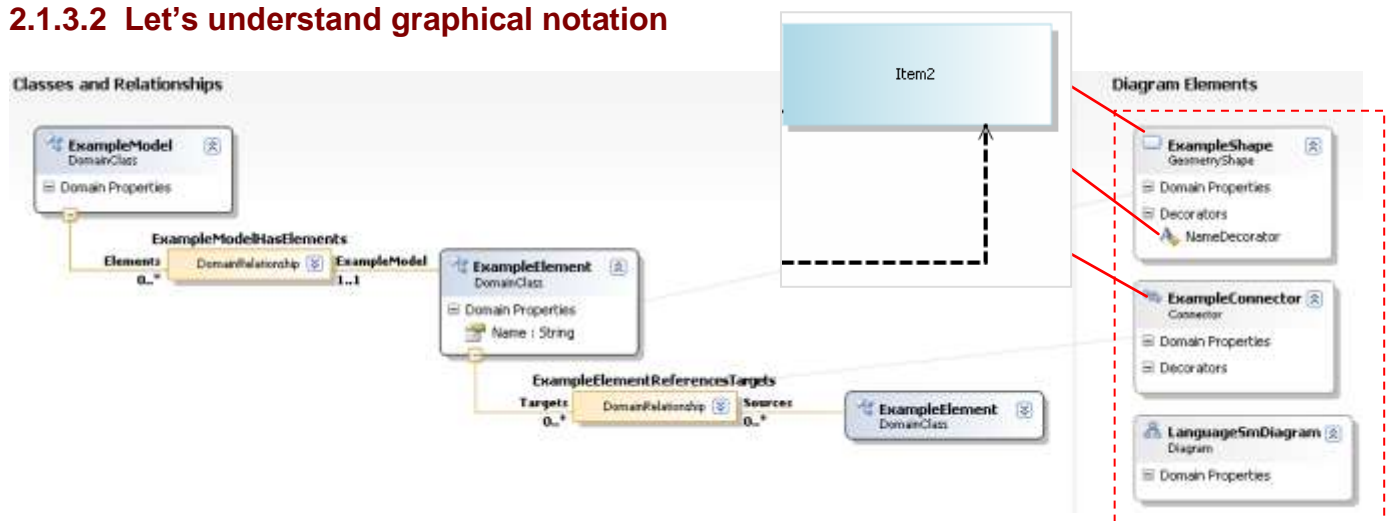
- all of the instances of `Element` in relationship with an `ExampleModel`, that is one which plays the **role** of its elements: `(GetElements(ExampleModel))`.
- all of the instances of `ExampleModel` in relationship with an `Element` by means of this relationship: `(GetExampleModel(ExampleElement))`.
- all of the instances of the relationship from an `ExampleModel`, an `ExampleElement`, or both: they are the `GetLink()` and `GetLinks()`, `GetLinksTo...()` methods.

Note that the names of these methods are generated automatically by the DSM Tools code generators, from the names of the functions in the relationship, and we therefore give very typical access, including in the case of reflexive relationships (as is the case for the `ExampleElementReferencesTargets` relationship that links two `ExampleElements` and whose reading meaning is determined uniquely by the names of the functions).

To summarize, with all these methods, we can obtain the relationships between the elements in both directions, through domain relationships (which is interesting because the relationships are attributed (carriers of properties) in the DSL Tools metamodel why using methods that get the links (instances of domain relationships), sources, and targets).



### 2.1.3.2 Let's understand graphical notation



Graphical notation is also explained in the `DslDefinition.dsl` file. It introduces:

- A shape (**ExampleShape**) that is a geometrical shape (here a blue rectangle). This geometrical shape has a textual decorator (**NameDecorator**). This does not appear on this diagram, but the **NameDecorator** shows and enables entry of the **Name** property of the **ExampleElement** that is represented by the **ExampleShape**.
- (the gray link connects shape **ExampleElement** to shape **ExampleShape** representing the notion of **ShapeMap**: the relationship between the two)
- A connector (**ExampleConnector**) is dotted black. It is a link that connects the shapes. In this case, it does not have decorators (but could have them).
- The diagram itself (**LanguageSmDiagram**) houses all the shapes and connectors.

- 26 Click the shape `ExampleShape` and note the properties of this shape. In particular: the graphical characteristics of this geometrical shape:

Geometry	Rectangle
Has Default Connection Points	Rectangle
Outline Color	RoundedRectangle
Outline Dash Style	Ellipse
Outline Thickness	Circle

Fill Gradient Mode	Horizontal
Geometry	Horizontal
Has Default Connection Points	Vertical
Outline Color	ForwardDiagonal
Outline Dash Style	BackwardDiagonal
Outline Thickness	None

Outline Dash Style	Solid
Outline Thickness	Solid
Text Color	Dash
Code	Dot
Access Modifier	DashDot
Custom Attributes	DashDotDot
Generates Double Derived	Custom

Also note that there are several properties that let you create more sophisticated behaviors (for example `Has Default Connection Points`, `Has Custom Constructor`, etc.). If these properties are moved to `true`, the DSL designer should devise certain methods (in the partial classes of those generated) for supplying, for example, particular connection points, etc.

**Properties** Microsoft.VisualStudio.Modeling.DslDefinition.GeometryShape

**ExampleShape**

**Appearance**

Fill Color: LightBlue

Fill Gradient Mode: Horizontal

Geometry: Rectangle

Has Default Connection Points: False

Outline Color:

Outline Dash Style: Solid

Outline Thickness: 0,03125

Text Color: Black

**Code**

Access Modifier: public

Custom Attributes:

Generates Double Derived: False

Has Custom Constructor: False

Inheritance Modifier: none

**Definition**

Base Geometry Shape: (none)

Name: **ExampleShape**

Namespace: CompanyName.Language5m

Tooltip Type: None

**Documentation**

Notes: **The shape has a text decorator used to display the**

**Exposed Style Properties**

**Layout**

Initial Height: **0,75**

Initial Width: **2**

**Resources**

Description: **Shape used to represent ExampleElements on a Di**

Display Name: Example Shape

Fixed Tooltip Text: Example Shape

Help Keyword:

**Name**

Name of this element.

- 27 Now click the connector `ExampleConnector` and look at its properties.

Among the configurable graphics properties, we see the style of the ends of the connector:

Source End Style	None
Target End Style	EmptyArrow
Text Color	None
Thickness	HollowArrow
Code	EmptyArrow
Access Modifier	FilledArrow
Custom Attributes	EmptyDiamond
Generates Double Derived	FilledDiamond

And the routing style of the connectors (whether they will go in a straight line or horizontally and vertically).

Routing Style	Rectilinear
Resources	Rectilinear
Description	Straight
Display Name	Example Connector

**ExampleConnector** Microsoft.VisualStudio.Modeling.DslC

**Appearance**

Color: Black

Dash Style: Dash

Source End Style: None

Target End Style: EmptyArrow

Text Color: Black

Thickness: 0,03125

**Code**

Access Modifier: public

Custom Attributes:

Generates Double Derived: False

Has Custom Constructor: False

Inheritance Modifier: none

**Definition**

Base Connector: (none)

Name: **ExampleConnector**

Namespace: CompanyName.Language5m

Tooltip Type: None

**Documentation**

Notes:

**Exposed Style Properties**

**Layout**

Routing Style: Rectilinear

**Resources**

Description: **Connector between the**


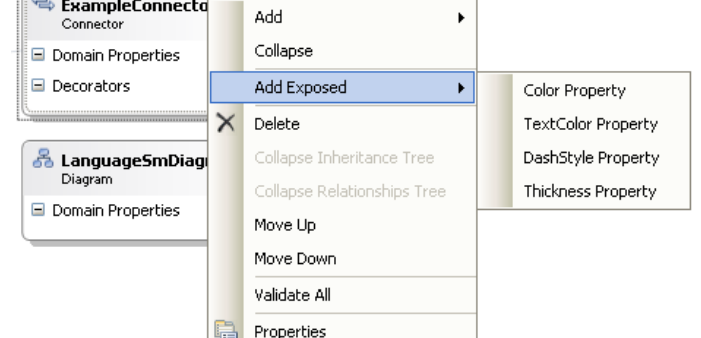
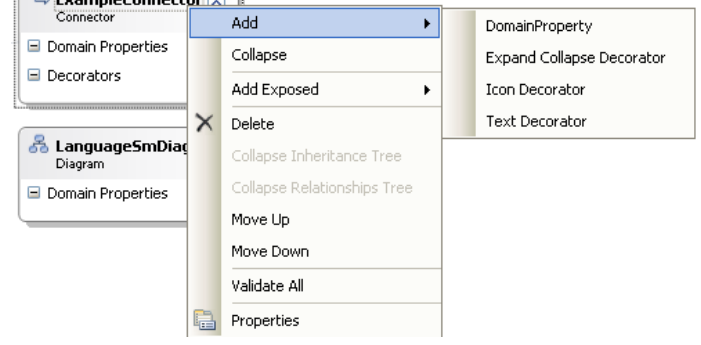
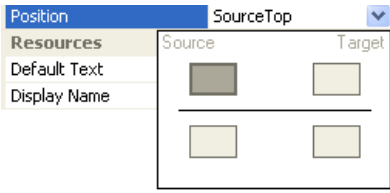
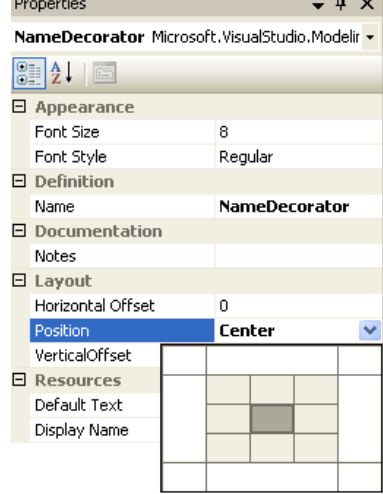
Display Name: Example Connector

Fixed Tooltip Text: Example Connector

Help Keyword:

**Name**

Name of this element.

<p>28 Click the "+" that is in front of the "Exposed Style Properties" category.</p> <p>They are in gray! (Elsewhere we could have done the same thing with the preceding shape.)</p> <p>This means that the graphical characteristics of the connector are fixed, and cannot be modified by the program (they are not displayed).</p>	
<p>29 Right-click the shape "ExampleConnector", and open the submenu <b>Add Exposed</b>. Here you find the commands that let you display the graphical style properties, shaded if necessary.</p>	
<p>30 Right-click the "ExampleConnector" shape, but this time open the submenu <b>Add</b>. Here we find the commands that let you add:</p> <ul style="list-style-type: none"> <li>- A specific property of the shape.</li> <li>- A decorator that lets you unfold/re-fold the shape (📁).</li> <li>- A decorator in the form of an icon.</li> <li>- A textual decorator.</li> </ul>	
<p>31 In the shape "ExampleShape", <b>click the decorator "NameDecorator"</b> and notice its properties.</p> <p>The decorators only show the graphical properties, and especially the position. A decorator may be positioned inside or outside a shape (here in the center), above, below, to the right, to the left, or in the middle/center. Elsewhere, compared to a position of reference, we can supply offsets (shown in <b>inches</b>).</p> <p>We have seen the textual decorators for shapes.</p> <p>We would have similar properties for the iconic decorators, and also for the connector decorators (in particular, with a movement from one side to the other of the connector, above or below )</p> 	

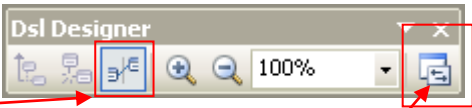

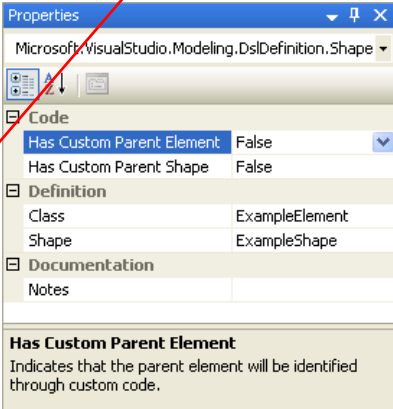
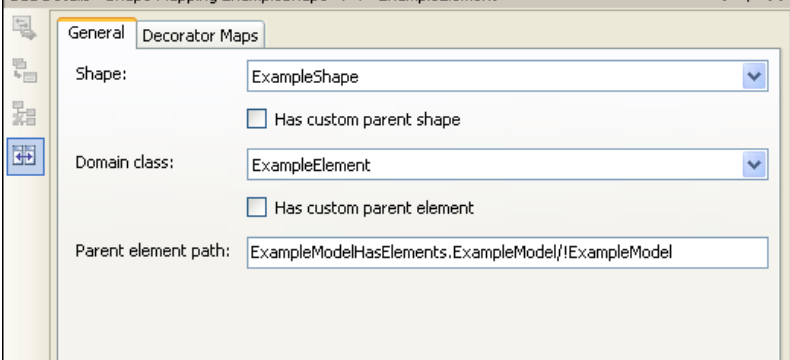
### 2.1.3.3 Understanding the connection between the metamodel and the graphical notation

The **domain elements of the domain model** are represented graphically by **shapes**. The domain **relationships** are represented by **connectors**. Likewise, the **decorators** are represented by the **properties** of the domain elements and domain relationships. The textual decorators display the property values and let you edit them. The icon decorators display the icons.

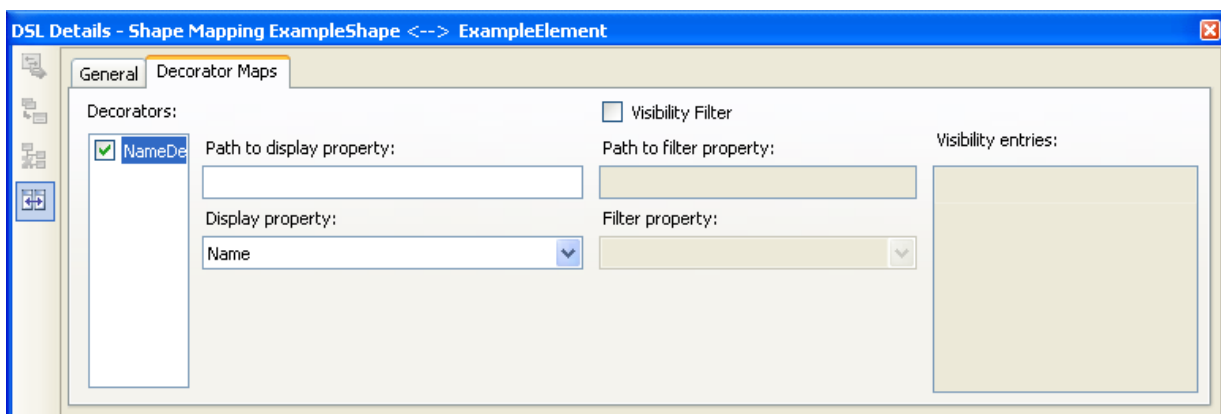
Textual and icon decorators can be either displayed or not displayed, depending on the property values.



This representation, and this configurability for the display of decorators are described in the ElementMaps (ShapeMap and ConnectorMap).

<p>32 Make sure that the links between the boxes that represent the concepts, and the boxes that represent the shapes are correctly displayed. If this is not the case you have to request display by clicking the <b>"show map lines"</b> button on the <b>"Dsl designer"</b> toolbar.</p>	
<p>33 Click the "map line" that links the "ExampleElement" box to the "ExampleShape" shape. This "map line" represents the ShapeMap.</p>	
<p>34 Look at the properties of this ShapeMap in the properties window. The properties of the "Code" category correspond to the advanced aspects.</p>	
<p>35 Now look at the <b>"DSL Details"</b> window. If it is not displayed, you can request its display by clicking the <b>"DSL details"</b> button on the <b>"Dsl designer"</b> toolbar.</p> <p>This window has two tabs. The General tab effectively concerns the connection between concept and shape. The "Parent element path" expresses the relationship that will receive a new instance of <code>ExampleElement</code> during creation of an <code>ExampleShape</code>: here, we specify that we are adding the <code>ExampleElement</code> to the function next to <code>ExampleModel</code> in the <code>ExampleModelHasElements</code> relationship.</p>	

36 Click the Decorator Maps tab. It describes the relationship between the properties and the decorators:



Here, the `NameDecorator` is associated with the `Name` property (of `ExampleElement`). It would also be possible to create a visibility filter, and specify for which values ("Visibility entries") of which property ("Filter property") the decorator is displayed. We will revisit all of this in detail in other chapters.

Putting aside the typical image shapes, compartments (which we will revisit after this Lab), ports, and swimlanes, we have looked at all of the present standard graphical possibilities of the DSL Tools.

It is also possible to specify more sophisticated behaviors such as:

- the end control of the connections (when you can create a connector).
- control of integrity constraints during deletion of a concept.

All this will be seen at a later stage.

### 2.1.3.4 Understanding serialization of the model and the diagram

The model that you are completing is serialized; we defined it during the creation of our DSL, in a `.sm` file. We will see in this paragraph:

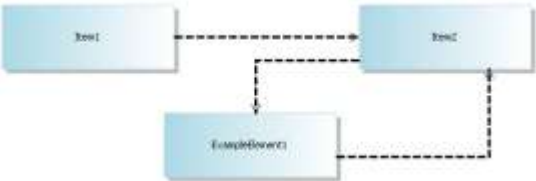
- How the **model** is serialized (in particular the XML diagram (`.xsd`) that is created by the DSL Tools for our attention).
- How the **diagram** (the graphical layout of it) is serialized.
- How we can **manage serialization** by using the DSL browser.

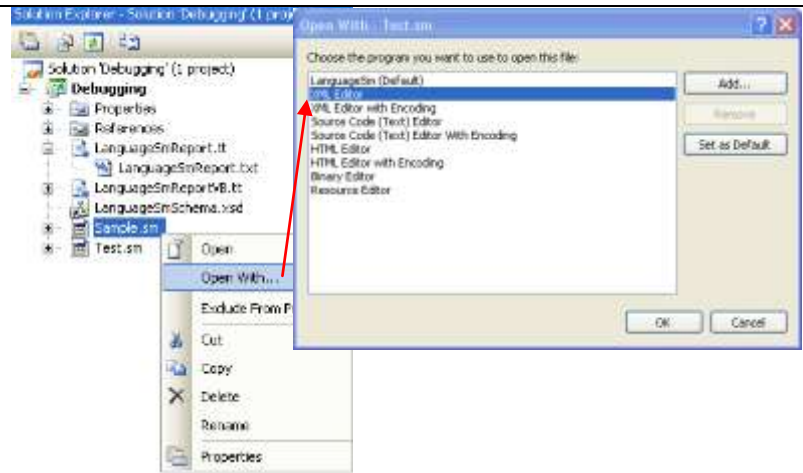
#### 2.1.3.4.1 Serialization of the model

37 Run the test version of Visual Studio: **Debug** -> **Run without debugging**.

38 **Open the Sample.sm model** that we created previously, by using the contextual command "Open with ..." in `Sample.sm`, and choosing "XML editor", instead of our DSL specialized editor.

(corresponding model)





39 Examine the serialized form of your model. It is relatively easy and comprehensible. Here we find the `ExampleModel`, and the 3 `ExampleElements`.

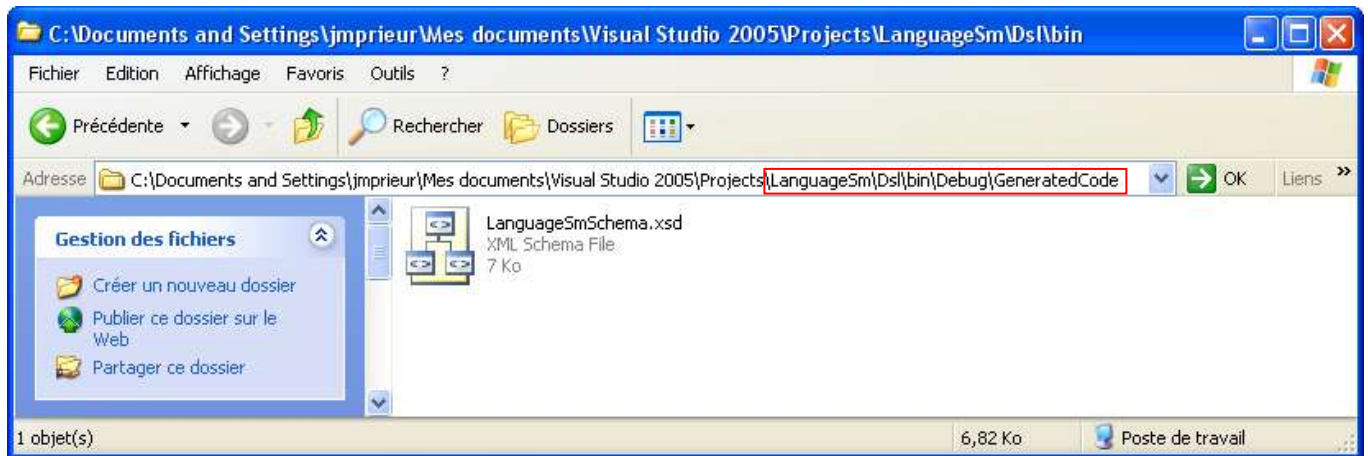
```
<?xml version="1.0" encoding="utf-8"?>
<exampleModel dslVersion="1.0.0.0" Id="84e1905c-7882-487b-8f13-7e67a92e4cfa"
  xmlns="http://schemas.microsoft.com/dsltools/LanguageSm">
  <elements>
    <exampleElement name="Item1">
      <targets>
        <exampleElementMoniker name="/84e1905c-7882-487b-8f13-7e67a92e4cfa/Item2" />
      </targets>
    </exampleElement>

    <exampleElement name="Item2">
      <targets>
        <exampleElementMoniker name="/84e1905c-7882-487b-8f13-7e67a92e4cfa/ExampleElement1" />
      </targets>
    </exampleElement>
    <exampleElement name="ExampleElement1">
      <targets>
        <exampleElementMoniker name="/84e1905c-7882-487b-8f13-7e67a92e4cfa/Item2" />
      </targets>
    </exampleElement>
  </elements>
</exampleModel>
```

Notice that the elements of the models have a unique identifier that we call a moniker, which is used in the relationships.

Also notice that the XML tag corresponding to the `ExampleElement`. **Item2** contains this, corresponding to **ExampleElement1**. We will take a closer look at the `exampleElement`, and will see the references between them.

The `.sm` files are XML files that follow an XML diagram that is generated by the DSL Tools, and installed with Visual Studio when we install the DSL (or when we develop it).

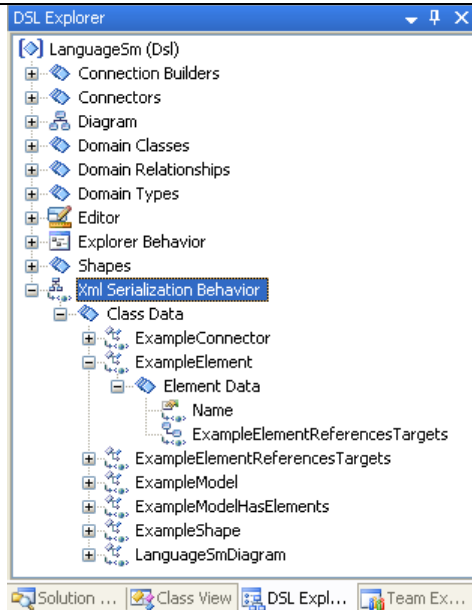


40 **Now open the** `Sample.sm.diagram` model (which is located as subordinate to `Test.sm`) by using, as in the previous case, the contextual command **"Open with ..."** on `Sample.sm.diagram`, and choosing **"XML editor"**, instead of our DSL specialized editor.

41 Examine the serialized form of the diagram of your model. It is also relatively easy and comprehensible. We find the monikers of the Examples here.

```
<?xml version="1.0" encoding="utf-8"?>
<minimalLanguageDiagram dslVersion="1.0.0.0" absoluteBounds="0, 0, 11, 8.5" isCompleteView="false"
  name="Sample">
  <exampleModelMoniker Id="84e1905c-7882-487b-8f13-7e67a92e4cfa" />
  <nestedChildShapes>
    <exampleShape Id="08e7d150-d4e4-451f-a7d6-8298e1d77788" absoluteBounds="0.5, 0.75, 2, 0.75">
      <exampleElementMoniker name="/84e1905c-7882-487b-8f13-7e67a92e4cfa/Item1" />
    </exampleShape>
    <exampleShape Id="d39030b5-4b9a-41d1-bb71-3014d2d4ed6e" absoluteBounds="4.625, 0.75, 2, 0.75">
      <exampleElementMoniker name="/84e1905c-7882-487b-8f13-7e67a92e4cfa/Item2" />
    </exampleShape>
    <exampleConnector edgePoints="[(2.5 : 1.125); (4.625 : 1.125)]" manuallyRouted="false"
      fixedFrom="Algorithm" fixedTo="Algorithm"
      TargetRelationshipDomainClassId="34cd2aac-61ef-4db8-9305-ad95ba477805">
      <nodes>
        <exampleShapeMoniker Id="08e7d150-d4e4-451f-a7d6-8298e1d77788" />
        <exampleShapeMoniker Id="d39030b5-4b9a-41d1-bb71-3014d2d4ed6e" />
      </nodes>
    </exampleConnector>
    <exampleShape Id="59c8cc39-05e2-4aad-858a-c1f0e659cda1" absoluteBounds="2.375, 2, 2, 0.75">
      <exampleElementMoniker name="/84e1905c-7882-487b-8f13-7e67a92e4cfa/ExampleElement1" />
    </exampleShape>
    <exampleConnector
      edgePoints="[(4.625 : 1.395833333333333); (3.4375005 : 1.395833333333333); (3.4375005 : 2)]"
      manuallyRouted="false" fixedFrom="Caller" fixedTo="Algorithm"
      TargetRelationshipDomainClassId="34cd2aac-61ef-4db8-9305-ad95ba477805">
      <nodes>
        <exampleShapeMoniker Id="d39030b5-4b9a-41d1-bb71-3014d2d4ed6e" />
        <exampleShapeMoniker Id="59c8cc39-05e2-4aad-858a-c1f0e659cda1" />
      </nodes>
    </exampleConnector>
    <exampleConnector
      edgePoints="[(4.375 : 2.510416666666667); (6.135416666666667 : 2.510416666666667); (6.135416666666667 :
1.5)]"
      manuallyRouted="false" fixedFrom="Caller" fixedTo="Caller"
      TargetRelationshipDomainClassId="34cd2aac-61ef-4db8-9305-ad95ba477805">
      <nodes>
        <exampleShapeMoniker Id="59c8cc39-05e2-4aad-858a-c1f0e659cda1" />
        <exampleShapeMoniker Id="d39030b5-4b9a-41d1-bb71-3014d2d4ed6e" />
      </nodes>
    </exampleConnector>
  </nestedChildShapes>
</minimalLanguageDiagram>
```

- 42 Now look at the LangageSm application, the DSL Explorer, and more specifically, the nodes under "Xml Serialization Behavior". Here we find all of the domain elements, domain relationships, shapes, and connectors, together with their properties and the way in which these properties will be serialized in the form of elements or XML attributes.



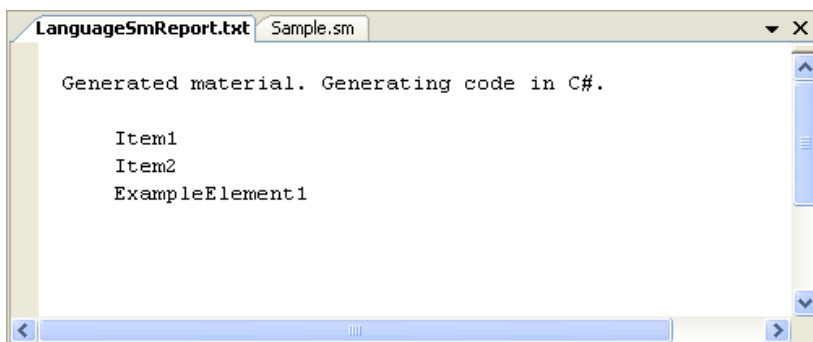
### 2.1.3.5 Understanding the artifact generators (Text Transformation templates)

We complete our study of the generated DSL, with the artifact generation templates.

- 43 Run the test version of Visual Studio: **Debug -> Run without debugging**.

- 44 From the browser window of the Debugging application, open the file `LanguageSmReport.tt` and get to know its structure.

The following page shows the generated documentation.



**<#@** : Opens a directive for the T4 processor

**#>** : Closing

```

<#@ template inherits="Microsoft.VisualStudio.TextTemplating.VSHost.ModelingTextTransformation" #>
<#@ output extension=".txt" #>
<#@ LanguageSm processor="LanguageSmDirectiveProcessor" requires="fileName='Sample.sm'" #>

Generated material. Generating code in C#.

<#
// When you change the DSL Definition, some of the code below may not work.
foreach (ExampleElement element in this.ExampleModel.Elements)
{
    <#= element.Name #>
}
#>

```

The code generator will be a .NET class that inherits from `ModelingTextTransformation` and is generated just in time.

The **output** directive lets us specify the output file name extension.

Next directive defines the name of a class that makes the link between our model and the T4 engine (`LanguageSmDirectiveProcessor`), the file that contains the model that will be loaded to generate the artefact (`Sample.sm`), and the name of the property from which our model will be accessible (Property `ExampleModel` which type is `ExampleModel` which is the name of the concept that is associated with the graphical designer).

In this file the brown text is the text that will be generated as is. The code in black / blue / green is C# control code that accesses the model.

Takes the value and inserts it into the text in brown (which is copied as is).

Control code C# of artifact generation.

45 Close the test version of Visual Studio.

This concludes the first phase of the Lab.

In the next phase of the Lab, we will see how to create and modify the metamodel for creating a StateMachine, and how to specify its graphical notation.