Tutorial: Applying Domain-Specific Modeling to Game Development with the Microsoft DSL Tools

André W. B. Furtado, André L. de M. Santos

Microsoft Innovation Center at Recife / Informatics Center (CIn) - UFPE
Rua do Apolo, 181, CEP 50030-220, Recife/PE/Brazil

{awbf,alms}@cin.ufpe.br

Abstract. *This tutorial introduces the concepts of domain-specific modeling (DSM) and domain-specific languages (DSLs), presenting how the theory can be productively put into practice with the Microsoft Visual Studio DSL Tools. The required steps for applying DSM to the game development domain are specified, illustrated with the creation of a visual DSL for modeling 2D adventure games. The final intention is to allow game developers and designers to work more intuitively, with a higher level of abstraction and closer to their application domain.*

Keywords: *domain-specific modeling (DSM), domain-specific languages (DSLs), digital games development.*

1. Introduction

Digital games are one of the most profitable industries in the world, being a match even for the movie and music industries [1]. However, software development industrialization, an upcoming tendency entailed by the exponential growth of the total global demand for software, will present many new challenges to game development.

Studies reveal that there is evidence that the current development paradigm is near its end, and that a new paradigm is needed to support the next leap forward in software development technology [2]. For example, although game engines [3] brought the benefits of Software Engineering and object-orientation towards game development automation, the abstraction level provided by them could be made less complex to consume by means of language-based tools, the use of visual models as first-class citizens (in the same way as source code) and a better integration with development processes.

This tutorial intends, therefore, to anticipate to its audience one of the most important software industrialization fundamentals: Domain-Specific Modeling (DSM) [4]. DSM raises the level of abstraction beyond programming by specifying (diagramming) the solution directly using domain concepts. The final products are generated from these high-level specifications. This automation is possible because both the language and generators need to fit the requirements of only one company and domain. In other words, DSM does to code what compilers did to assembly language, and industrial experiences of DSM consistently show it to be 5-10 times faster than current practices.

By using the Microsoft Domain-Specific Language (DSL) Tools [5], the tutorial will guide the audience through the practical steps of a DSM experience in game development. It will explore how to properly choose a game development domain, how to model such a domain with tool support, how to create visual editors to help game designers intuitively manipulate the concepts of the domain, how to build automatic code generators and, finally, how to embed the developed DSL into a real-world development environment.

The remainder of this document is organized as follows: Section 2 introduces the concept domain-specific languages. Section 3 provides an overview of the Microsoft DSL Tools. Sections 4 to 14 detail the required steps to apply DSM in the game development domain, while Section 15 concludes about the proposed approach.

2. Domain-Specific Languages

In all branches of science and engineering one can distinguish between approaches that are generic and those that are specific [6]. A generic approach provides a general solution for many problems in a certain area, but such a solution may be suboptimal. A specific approach provides a much better solution for a smaller set of problems. One of the incarnations of this dichotomy in computer science is domain-specific languages versus generic programming languages.

Of course, this is not a new topic. The older programming languages (Cobol, Fortran, Lisp) all came into existence as dedicated languages for solving problems in a certain area (respectively business processing, numeric computation and symbolic processing). Gradually, they have evolved into general purpose languages and over and over again the need for more specialized language support to solve problems in well-defined application domains has resurfaced. Over time, the following solutions have been tried [6]:

- Subroutine libraries contain subroutines that perform related tasks in well-defined domains like, for instance, differential equations, graphics, user-interfaces and databases. The subroutine library is the classical method for packaging reusable domain-knowledge.
- Object-oriented frameworks and component frameworks continue the idea of subroutine libraries. Classical libraries have a flat structure, and the application invokes the library. In object-oriented frameworks it is often the case that the framework is in control, and invokes methods provided by the application-specific code [7, 8].
- A domain-specific language (DSL) is a small, usually declarative, language that offers expressive power focused on a particular problem domain. In many cases, DSL programs are translated to calls to a common subroutine library and the DSL can be viewed as a means to hide the details of that library.

A domain-specific language is a limited form of computer language designed for a specific class of problems [9]. It is a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain.

The key characteristic of DSLs is their focused expressive power. DSLs are usually small, offering only a restricted set of notations and abstractions. Domain-

specific languages are usually declarative. Consequently, they can be viewed as specification languages, as well as programming languages. Examples of popular domain-specific languages include:

- SQL (Structured Query Language), a language that provides an interface to relational database systems;
- HTML (Hypertext Markup Language), a markup language designed for the creation of web pages and other information viewable in a browser;
- TeX, a typesetting language used to create highly structured documents, especially good for mathematical notation;
- BNF (Backus-Naur Form), a meta-language for describing other languages, particularly computer languages.

XML configuration files and GUI builders, in which the user experience is quite different from textual programming languages, can also be pointed as examples, in spite of not being usually perceived as DSLs.

Today, DSLs span many diverse domains, such as financial products, software architectures, databases, video device driver specifications, operating system specialization, web computing, image manipulation, 3D animation, drawing, communication protocols, telecommunication switches, simulation, mobile agents, robot control, partial differential equations and digital hardware design, just to mention a few.

Adopting a DSL approach to Software Engineering (i. e., adopting a language oriented programming approach) involves both risks and opportunities. Well-designed DSLs manage to find the proper balance between these two. The benefits of DSLs include:

- DSLs allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain; consequently, domain experts themselves can understand, validate, modify, and often even develop DSL programs;
- DSL programs are concise, self-documenting to a large extent, and can be re-used for different purposes [10];
- DSLs enhance productivity, reliability, maintainability [11, 12], and portability [13];
- DSLs embody domain knowledge, and thus enable the conservation and reuse of this knowledge;
- DSLs allow validation and optimization at the domain level [14, 15, 16];
- DSLs improve testability following approaches such as [17];
- Finally, the Microsoft Software Factories Initiative [18] argues that a well-defined DSL is a powerful implementation language, providing much greater rigor than a general purpose modeling language like UML [2].

On the other hand, some disadvantages of using DSLs can be also pointed out:

- The high costs of designing, implementing and maintaining a DSL;

- The high costs of education for DSL users;
- The limited availability of DSLs [19];
- The difficulty of finding the proper scope for a DSL;
- The difficulty of balancing between domain-specificity and general-purpose programming language constructs;
- The potential loss of efficiency when compared with hand-coded software.

A deeper discussion about the trade-offs of using a DSL may depend on its “internal” or “external” style. Internal DSLs morph the host language into a DSL itself. External DSLs, on the other hand, are written in a different language than the main (host) language of the application and are transformed into it using some form of compiler or interpreter [9].

Graphical or visual DSLs, which are used by means of graphical notations instead of text, are an evolution of external DSLs. They are used in domain-specific modeling (DSM).

Two examples of graphical DSLs are illustrated in Figure 1, which is a screenshot from the Microsoft Visual Studio 2005 Team System (VSTS) [20]. The DSL on the left describes software solution components, such as web services. It is used to automate component development and configuration. The DSL on the right describes logical server types in a data center. It is used to design and implement data center configurations. Web service deployments are described by dragging service components onto logical servers. Differences between the resources they require and the resources services available on the logical servers onto which they are deployed are flagged as validation errors by the IDE.

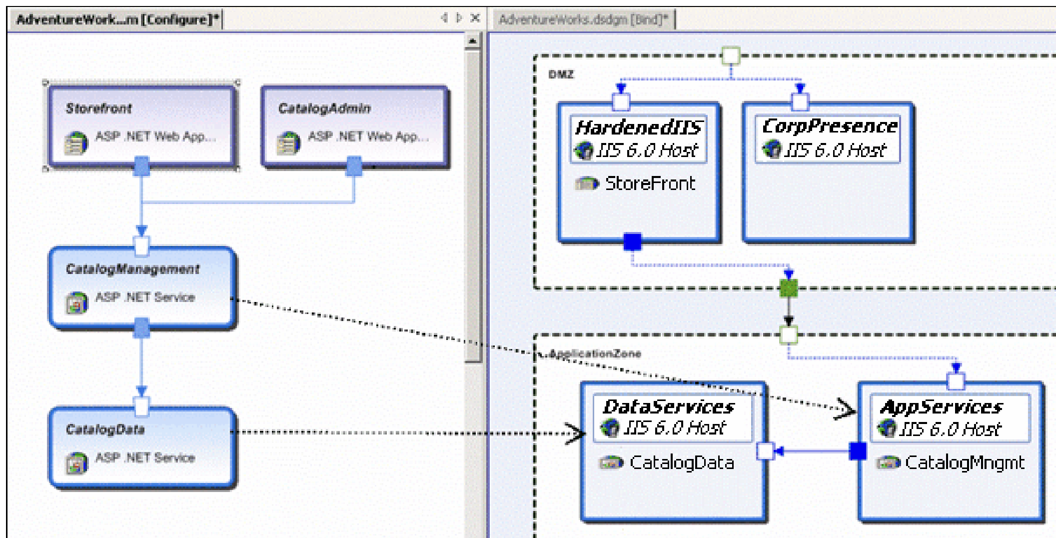


Figure 1 – Visual DSLs in Microsoft Visual Studio Team System

Another example of a visual DSL is illustrated by Tolvanen [21], in Figure 2. It presents a simple domain-specific modeling language used to model a conference registration application that runs on a smart phone. The design is expressed in a language created specially for defining enterprise applications on smart phones, and

would be useless to other domains. It includes modeling concepts such as notification, query, pop-up, and text message (the actual UI widgets and services already found on a phone).

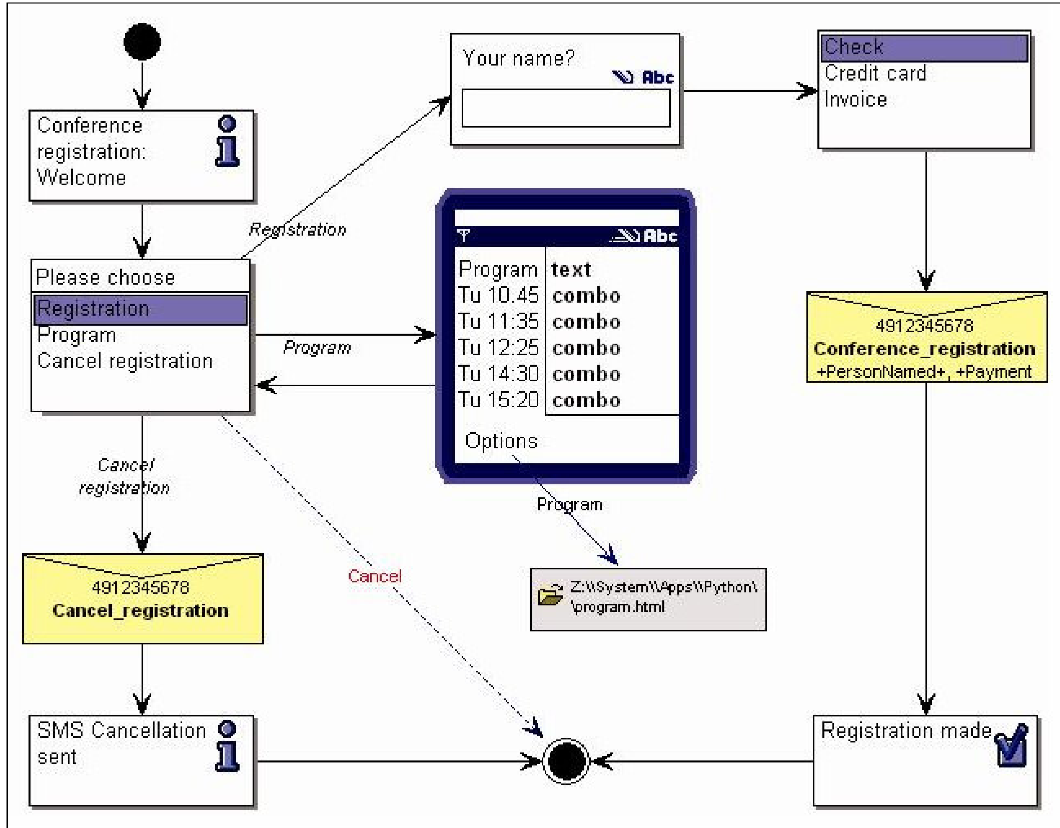


Figure 2 – DSL for smart phone applications

With these and other similar modeling concepts, it is possible to specify both the static structure of applications and their dynamic behavior, and the design provides enough information to automatically generate full code for this application. This sample language includes the design rules for phone application development and completely hides the implementation details. No programming concepts (classes, interfaces, structs, etc.) can be found in the model.

By using DSM in a development process, the created models are not only used for documentation. Actually, they are live artifacts, which can be transformed into other artifacts (such as other diagrams or source-code). This is possible once transformers (such as code generators) are created, along with a domain framework (such as a game engine) which is consumed by the generated artifacts, as shown in Figure 3. Items in the left side of the figure are created only once, by the language designer. Items on the right, on the other hand, provide abstraction to developers and improve their productivity, being created each time a new product needs to be generated.

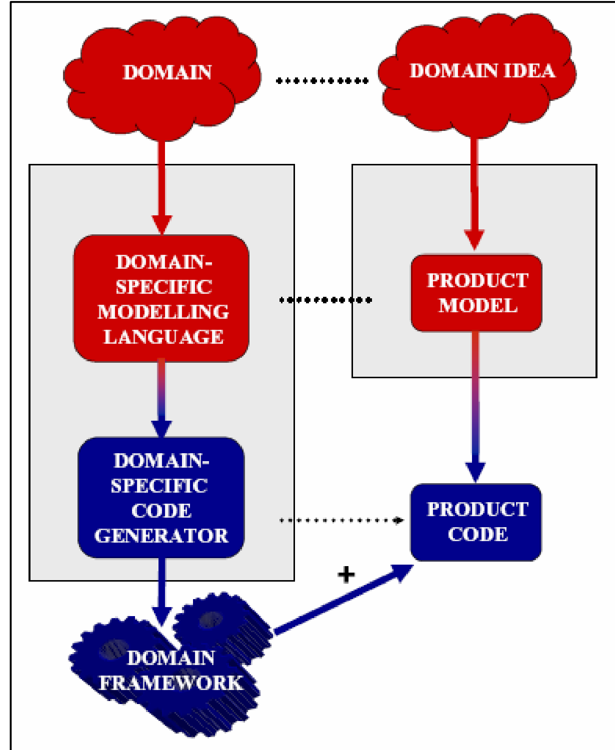


Figure 3 – DSM artifacts

Using internal DSLs reduces the tool cost, but the resulting constraints on the DSL itself can also greatly reduce the benefits [9]. An external DSL, on the other hand, gives the greatest potential to realize benefits, but comes at a greater cost to design the language, build the translator, and consider tools to support programming. This is especially true for visual DSLs, which demand more tooling such as graphical editors. In other words, unless there are ways to make it faster, cheaper and easier to develop external/visual DSLs, it will not be cost-effective to provide automatic refinement or other forms of automation for narrow domains.

This is where **language workbenches** come into play. They contrast the early days of domain-specific modeling, where no tools were available to create domain-specific languages and support modeling with them in a cost effective manner [21]. Those wishing to adopt DSM were left to develop their own tools from scratch based on generic graphics libraries. Considering that building a modeling tool is an effort that requires many man-years of development, DSM was left as an option only for large organizations that could commit to such an undertaking.

Nowadays, this situation has changed. A growing number of experts, as well as tool vendors, see DSM as a viable solution that offers developers better tools to deal with the growing complexity of the problem domains surrounding them. Besides the establishment of an independent organization (DSM Forum) [4] to spread DSM knowledge and know-how, this also lead to the creation of a new category of tools, named language workbenches [9]. Such tools use IDE tooling in a bid to make language oriented programming a viable approach.

Essentially, the promise of language workbenches is that they provide the flexibility of external DSLs without a semantic barrier. Furthermore, they make it easy to build tools that match the best of modern IDEs. The result makes language oriented programming much easier to build and support, lowering the barriers that have made language oriented programming so awkward for so many.

Examples of language workbenches include the Meta Programming System [22], Intentional Software [23] and Microsoft Visual Studio Team System (VSTS) [20]. However, VSTS is more deeply aligned with the software factories initiative, where DSLs are not only used to automate programming but also for other areas of software development that often do not get automated, such as deployment, testing, and documentation. It also explores simulators for situations where DSLs are not meant to be executed directly in development, such as deployment DSLs.

3. Visual Studio Team System DSL Tools

VSTS is a software development life-cycle management tools platform that helps software teams collaborate to reduce the complexity of delivering modern service-oriented solutions. It provides functionalities than spans all software development process, which includes analysis and design activities, such as application modeling.

VSTS delivers a set of designers (built-in visual editors), as part of Visual Studio 2005 Team Architect Edition, that enable architects and developers to design service-oriented applications and operations infrastructure simultaneously. Examples of such designers are the Class Designer, Application Designer, System Designer, Logical Datacenter Designer and the Deployment Designer. Since much functionality are common to all VSTS designers, such as zooming, multiple selecting, dragging items from the Toolbox and so on, Microsoft has decided to implement a common platform (suite of tools), upon which designers can be built. This platform, named **DSL Tools** [5], is not only internally used by built-in VSTS designers but is also delivered to end-users as part of Visual Studio Team System. It provides a framework and toolset that enable partners to build custom visual designers and domain-specific language designers using Visual Studio. Figure 4 illustrates where the DSL Tools are situated in Visual Studio modeling strategy.

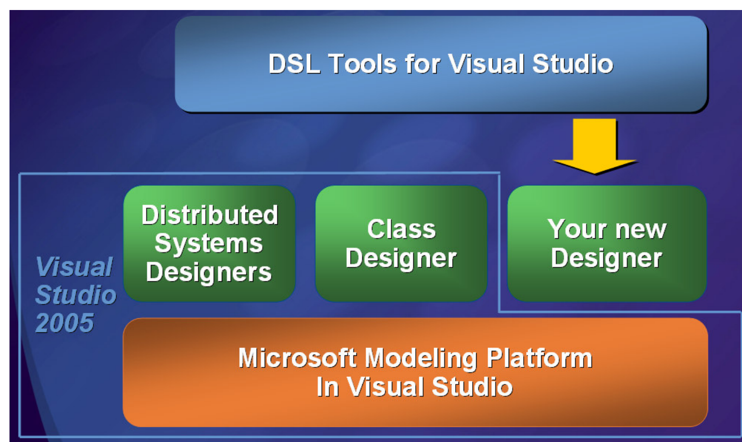


Figure 4 – DSL Tools enable new designers to be plugged in Visual Studio

In other words, it is possible to extend VSTS by creating and plugging into it a new designer, based on a visual domain-specific language. Through the DSL Tools, one can create, edit and visualize metadata that is underpinned by a code framework, which makes it easier to define domain-specific schemas for metadata, and then to construct a custom graphical designer hosted in Visual Studio. The suite consists of:

- A project wizard for creating a fully configured solution in which it is possible to define a domain model that consists of a designer and a textual artifact generator. Running a completed solution from within Visual Studio opens a test solution in a separate instance of Visual Studio, making it possible to test the designer and artifact generator;
- A format and a graphical designer for defining and editing domain models;
- A graphical designer for creating designer definitions, from which the code for implementing designers is generated. This makes it possible to define a graphical designer hosted in Visual Studio without any hand coding;
- A set of code generators, which take a domain model definition and a designer definition as input, and produce code that implements both of the components as output;
- A framework for defining template-based artifact generators, which takes data (models) conforming to a domain model as input, and outputs text based on the template. Parameters in the template are substituted using the results of running a C# [24] script embedded in the template.

Figure 5 presents the overall user experience when dealing with a DSL created with DSL Tools and hosted in Visual Studio .NET. The Toolbox (at the left) presents some domain concepts that can be dragged and dropped to the designer (at the middle). The Error List (at the bottom) presents errors raised from semantic validators. The Properties window (at the right bottom) makes it possible to edit properties of the selected item in the diagram, eventually launching advanced property editors. By using menu commands, users can launch a code generator as well as create their own code. Finally, users can use an Explorer Window (top right) to hierarchically browse through concept instances added to the model.

4. Domain Definition

Once enough background is provided about domain-specific languages, language workbenches and the Microsoft Visual Studio Team System DSL Tools, some suggested steps required to apply DSM to game development can be detailed. Defining the target domain is the first task to be performed. Its purpose is to decide which types of games will be modeled and generated.

The great diversity of games created so far has turned the digital games universe into a very broad domain. Therefore, using DSM towards computer games development in general, ranging from 2D platform games to 3D flight simulators, constitutes a too broad and ineffective endeavor. In such a scenario, the production process and its tools would not be able to fully exploit DSM benefits such as component reuse and assemblage. In other words, a narrower subset of games should be chosen.

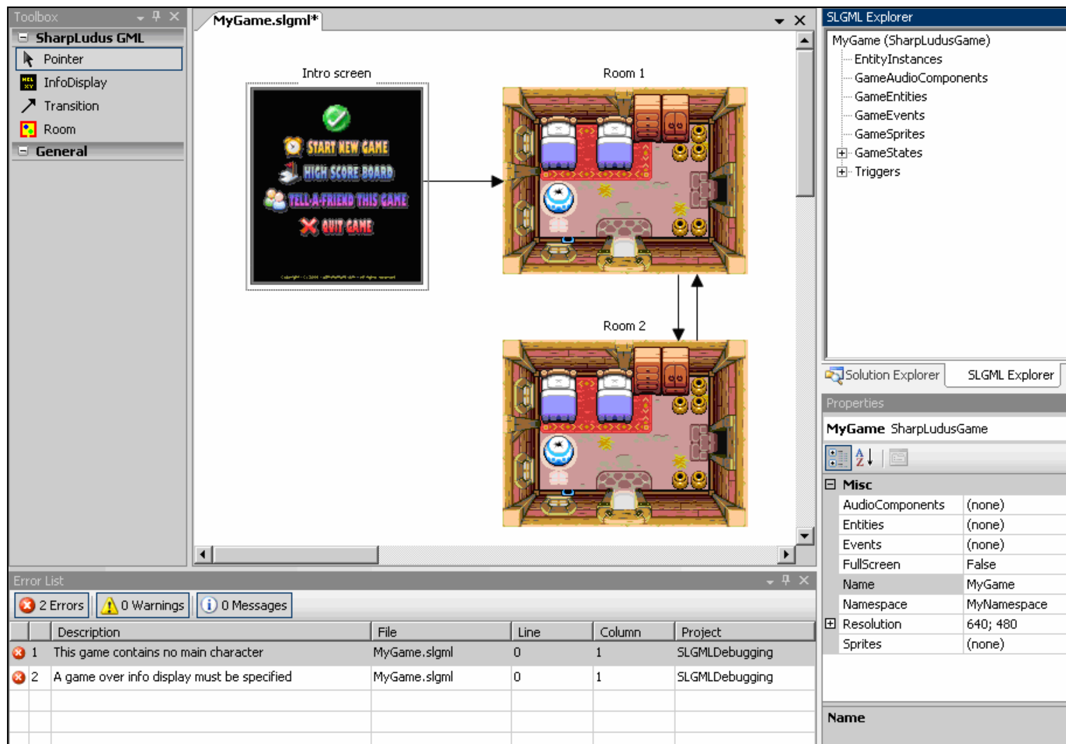


Figure 5 – Overall DSL consumer experience in VS.NET

One of the most often used attempts to classify computer games is to define game genres, which together compose a game taxonomy [25]. Some of the most popular game genres [25, 26, 27] are summarized in Table 1.

However, this tutorial suggests that in spite of solely relying on a game genre name to define a domain, a description of what is understood from such genre should also be provided, as well as any relevant information such as dimension (2D, 2D ½, 3D), sound support, input handling, networking support and so on. Therefore, a suggestion of a product line (domain) definition template for a DSM process is presented in Table 2, already filled with an example.

Table 1 – Popular game genres

Genre Name	Description	Examples
Adventure	Games which are set in a "world" usually made up ofmultiple, connected rooms or screens, involving an objective which is more complex than simply catching, shooting, capturing, or escaping, although completion of the objective may involve several or all of these.	Berzerk, Adventure (Atari), Myst, Tomb Raider, Space Quest, Indiana Jones
Board	Adaptation of existing board games or games which are similar to board games in their design and play even if they did not previously exist as board games.	Backgammon, Othello, Checkers, Chess
Fighting	Games involving characters who fight usually hand-to-hand, in one-to-one combat situations. In most of these games, the fighters are represented as humans or anthropomorphic characters.	Street Fighter, Mortal Kombat, Dragon Ball Z
Platform	Consists of animated objects running, climbing and jumping on platforms. Characters and settings are seen in side view as opposed to top view, and most games scroll the screen while the main character moves.	Super Mario Bros., Alex Kid, Comander Keen
Role-Playing	Games in which players create or take on a character represented by various statistics, which may even include a developed persona.	Final Fantasy, Dungeons & Dragons, Diablo
Shooter	Games involving shooting at, and often destroying, a series of opponents or objects, usually requiring quick reflexes.	Doom, Half-Life, Halo
Simulation	Games or programs which attempt to simulate a realistic situation, for the purpose of training, and usually the development of some physical skill such as steering (as in driving and flight simulators).	Flight Simulator, Apache, F-22, The Sims
Sports	Games which are adaptations of existing sports or variations of them.	NHL Hockey, FIFA Soccer, NBA Basketball, Top Spin
Strategy	Games which require planning, complex decisions and balancing the use of limited resources towards a higher-lever goal (improve a city, evolve a civilization, etc.), while dealing with internal forces (crime, pollution, etc.) or external forces (natural disasters, enemies, etc.).	Age of Empires, WarCraft, Civilization, SimCity

Table 2 – Detailed Domain Definition

Product Line (Domain) Definition	
Related game genre(s): adventure	
Description: The modeling process will produce computer games in which the player control a main character in a world composed by connected rooms. Rooms may contain items to be collected, such as keys and weapons. Enemies may also be present in a room; they must be avoided or defeated. Victory condition is specified by the game designer (a specific room is reached, a number of enemies is defeated, an object is collected, etc.).	
Target Platforms: PCs	
Feature Overview	
Feature	**Description**
Dimensionality	Two-dimensional (2D). World rooms are viewed from above.
User interface	Information display screens containing texts, radio buttons and graphical elements are supported. HUDs (heads-up display) can also be configured and displayed.
Game flow	Each game should have, at least, a main character, an introduction screen, one room and a game over screen (this last one is reached when the number of lives of the main character becomes zero).
Sound/Music	Games will be able to reproduce sound effects (wav files) as event reactions. Background music (mp3 files) can be associated with game rooms or information display screens.
Input handling	Keyboard only.
Networking	High scores can be uploaded to and retrieved from a web server.
Artificial Intelligence	Enemies can be set to chase the player within a room. More elaborated behaviors can be created visually by combining predefined event triggers and event reactions, or programmatically by developers.
Multiplayer	Online multiplayer is not supported by the factory. Event triggers and reactions can be combined, however, to allow two-player mode in a single computer.
End-user editors	Not supported by the factory. Once created, a game cannot be customized by its players.

The next sections present the tasks required to create and embed a DSL into VS.NET. The tasks are presented as a set of steps, grouped in major activities.

5. Environment Setup

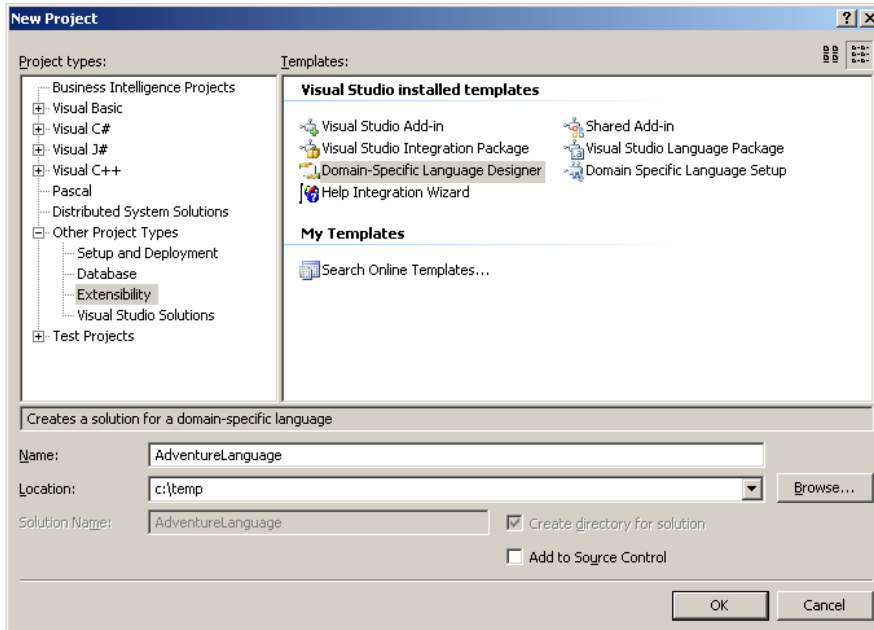
5.1. Install Visual Studio .NET 2005.

5.2. Install Visual Studio SDK (Software Development Kit)[1], which contains the DSL Tools plug-in. This tutorial uses the September 2006 version of the VS SDK, also known as "Visual Studio 2005 SDK version 3.0".

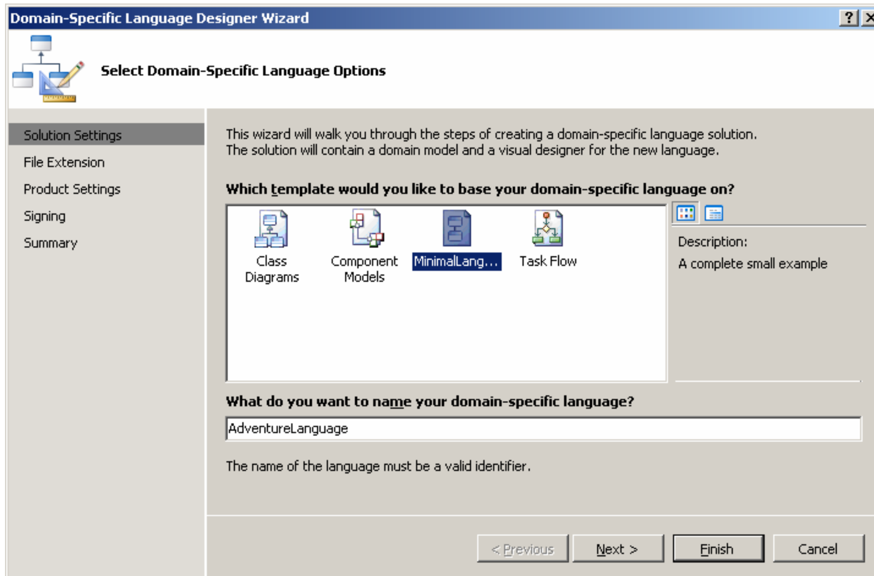
[1] Available at *http://go.microsoft.com/fwlink/?LinkId=73702*.

6. Creating a new DSL Tools Project

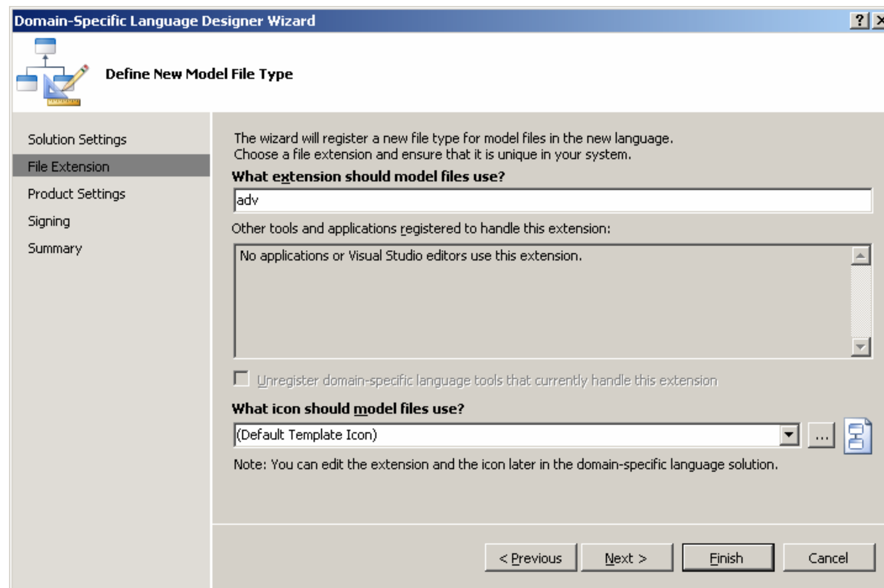
6.1. Open VS.NET and select menu *File* > *New Project*. In the left panel, select *Other Project Types* > *Extensibility*. In the right panel, select *Domain Specific Language Designer*. Provide a language name (such as "AdventureLanguage") in the *Name* field.



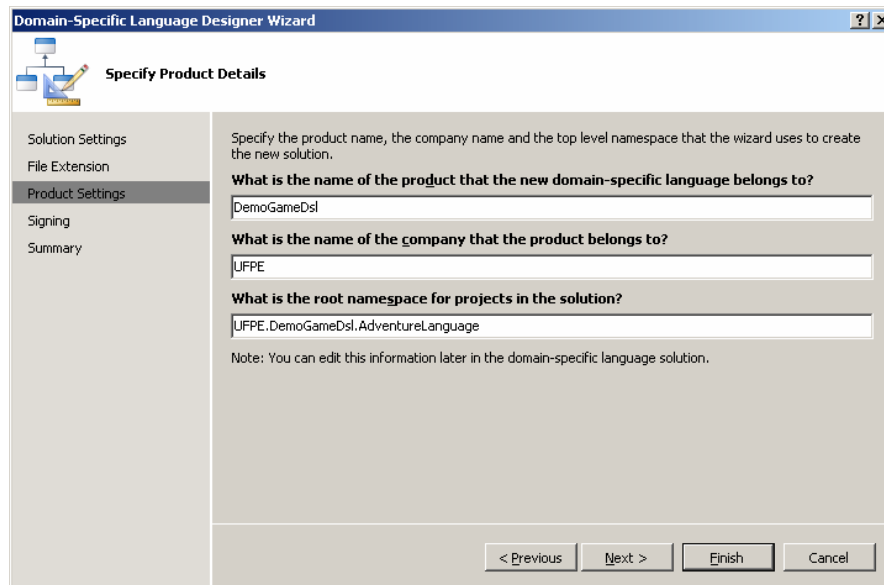
6.2. A wizard will appear. In its first screen, select *Minimal Language* and click *Next*.



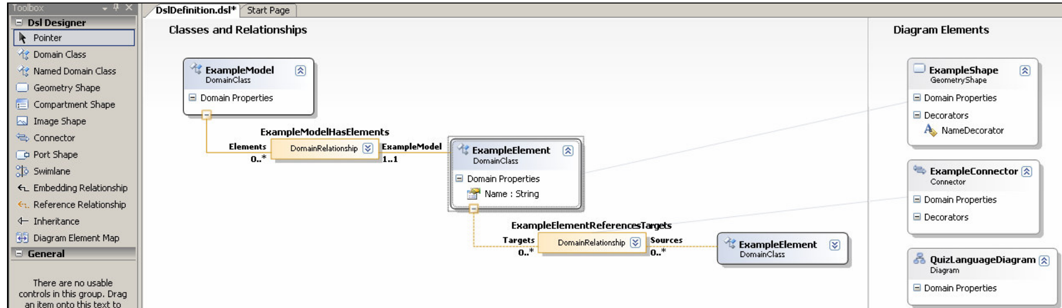
6.3. In the next wizard screen, enter in the first field an extension for the DSL diagram files (such as “adv”). Click *Next*.



6.4. In the next wizard screen, enter in the field a name for the product that the new DSL belongs to (such as “DemoGameDsl”), an organization name (such as “UFPE”) and a namespace for the projects which will be generated (you can leave the default option, such as “UFPE.DemoGameDsl.AdventureLanguage”). Click *Finish*.

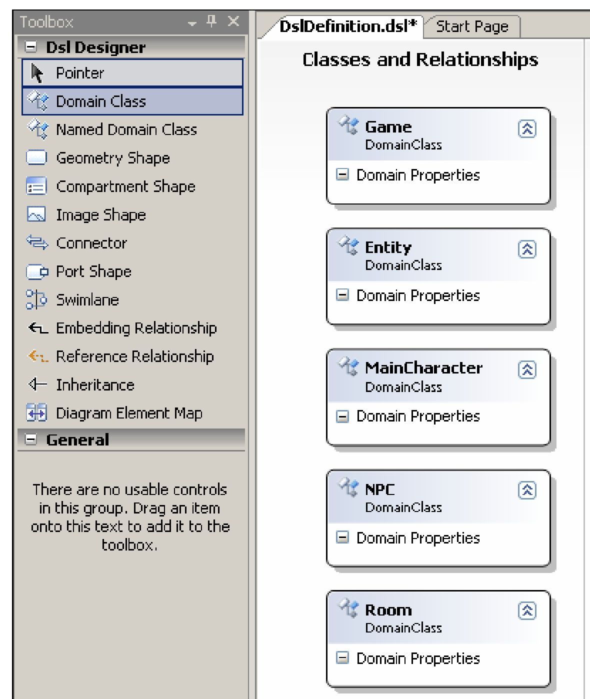


6.5. A new solution will be created in VS.NET, and a designer with some sample elements will be displayed, as shown in the picture below. Delete these elements by pressing *CTRL + A* then *DEL*.

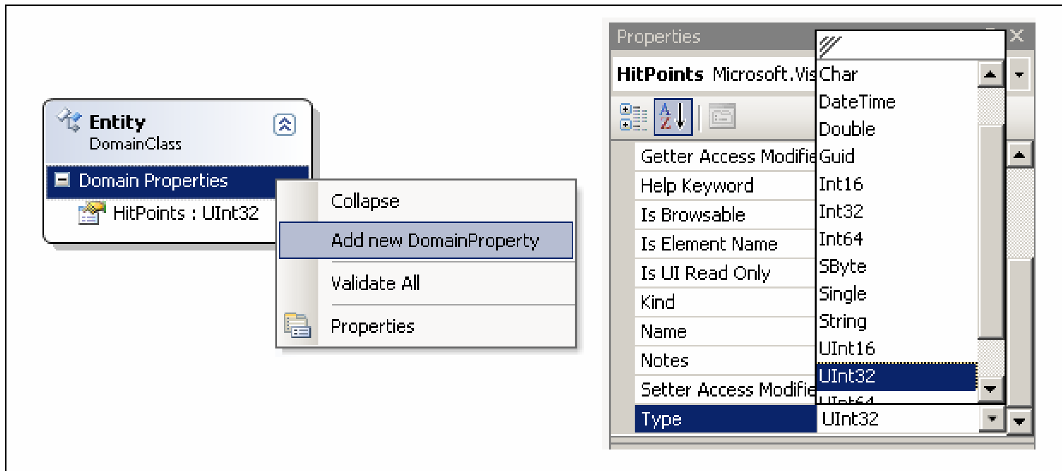


7. Modeling Domain Concepts

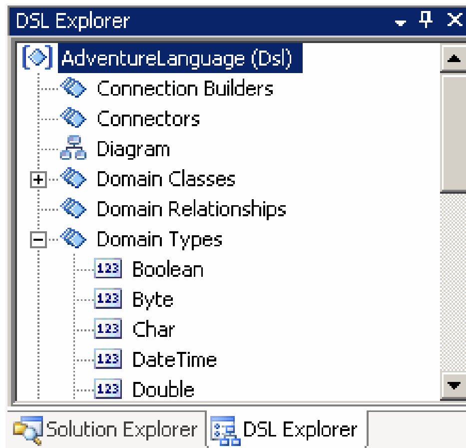
7.1. Drag and drop domain concepts from the Toolbox (*Domain Class* elements) to the main designer. Suggested domain concepts are *Game* (the root concept), *Entity*, *MainCharacter*, *NPC* (non-playable character) and *Room*.



7.2. Provide more detail for the domain concepts by adding domain properties to them. For example, a game entity may contain properties to indicate its hit points, position and image, while a room may have its width and height. To add domain properties, right-click the correspondent *Domain Properties* compartment of the concept and then in *Add new DomainProperty*. Give a name to the property and specify its type in the Properties window.

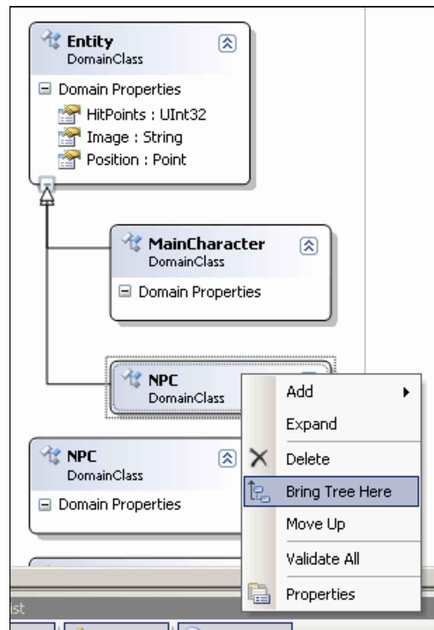


7.3. Some built-in types (*String*, *Boolean*, *Double*, etc.) may not satisfy the language designer needs. For example, an entity position should better be represented by a *Point* type, which is not built-in (it is defined in the namespace *System.Drawing*). To add such external types or to create your own enumeration types, right click the root node of the DSL Explorer window and select the desired option (*Add New External Type*, *Add New Enumeration*, etc.).



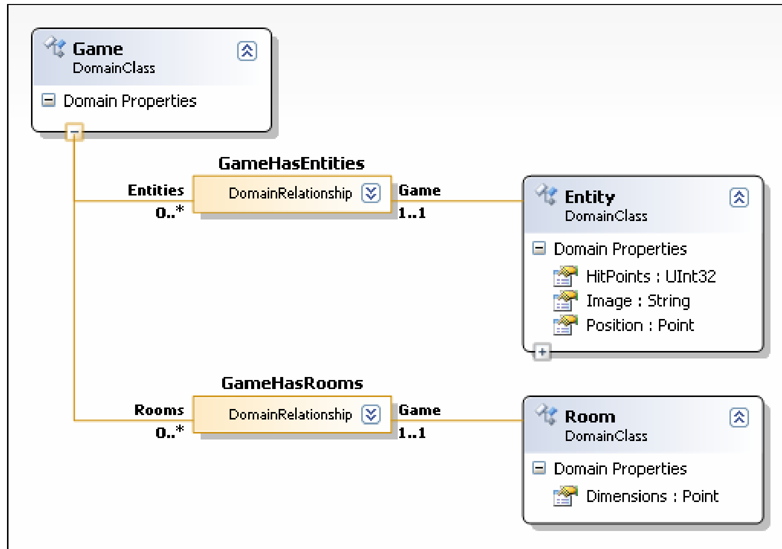
8. Modeling Domain Relationships

8.1. Use the *Inheritance* Toolbox element to say that *NPCs* and *MainCharacters* are specializations of the *Entity* base concept. To avoid a concept to appear twice in the diagram (both in the inheritance relationship and in its original definition) right-click the concept and select *Bring Tree Here*.



8.2. Set the *Inheritance Modifier* property of the Entity concept to *abstract*.

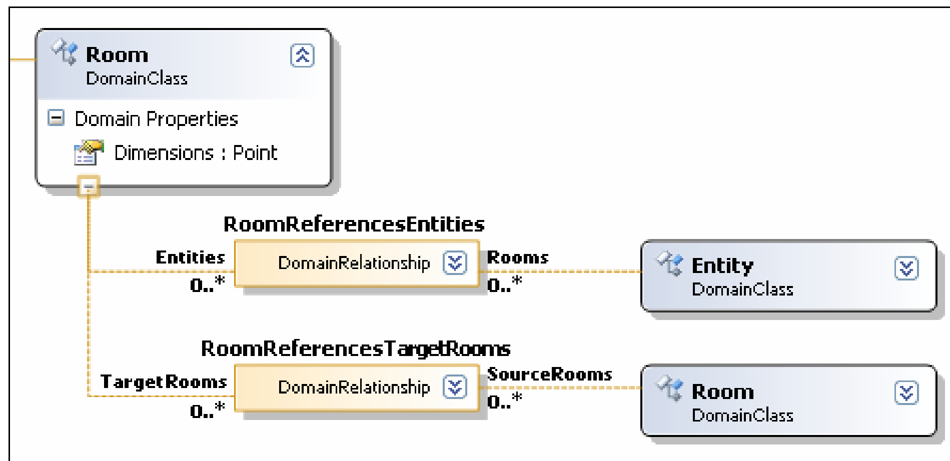
8.3. Use the *Embedding Relationship* Toolbox element to say that the *Game* concept is strongly related to *Rooms* and to *Entities*[2]. It is possible to adjust relationship name and multiplicity as desired through the *Property* window.



8.4. Use the *Reference Relationship* Toolbox element to say that the *Room* concept is weakly related to *Entities*. Use this same Toolbox element to say that a *Room* refers to other *Room* (a subsequent room in the game). To create such a relationship from a concept to itself, double-click the desired concept after selecting the relationship in the Toolbox.

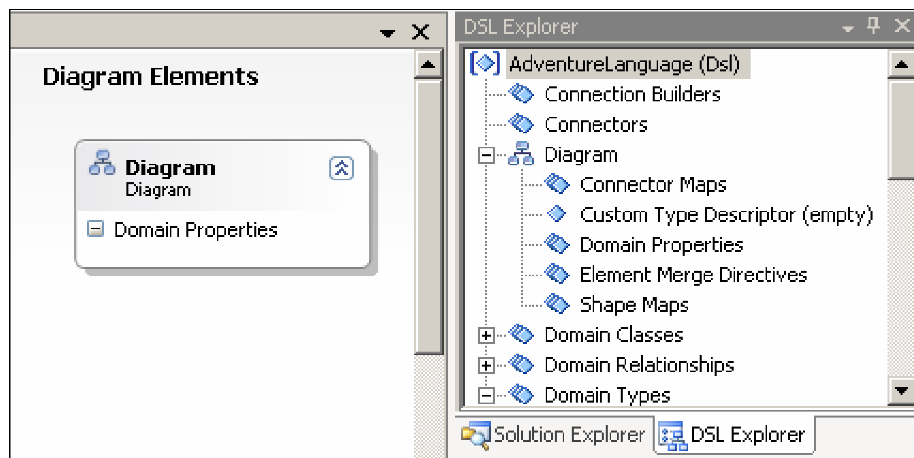
---

[2] A "strong" relationship here means that if a *Game* concept instance ceases to exist, its rooms and entities are also destroyed. "Weak" relationships are modeled through the *Reference Relationship* Toolbox item.



9. Defining DSL Visual Syntax

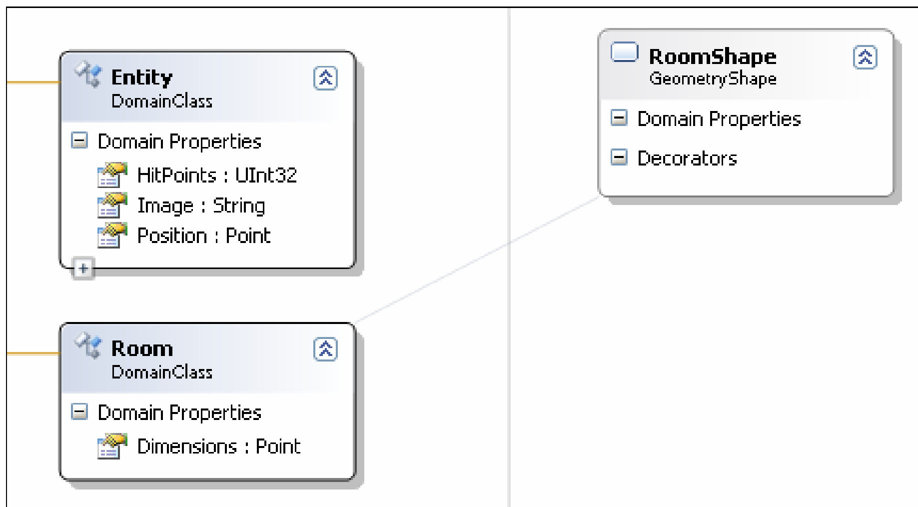
9.1. Right-click the root node of the DSL Explorer window (*AdventureLanguage*) and select *Add New Diagram*. A new element named "Diagram1" will be added to the *Diagram Elements* section in the main designer. Rename the element to "Diagram".



9.2. Click on the *Diagram* element in the designer and, through the Properties window, change the value of the *Class Represented* property to *Game*.

9.3. Drag a *Geometry Shape* from the Toolbox do the main diagram. Rename it to "RoomShape". Change its *Fill Color* property to *LightGray*. Then select the *Diagram Element Map* element in the Toolbox to map the *Room* concept to the *RoomShape*. A line will appear, linking both elements.

9.4. Drag an *Image Shape* from the Toolbox do the main diagram. Rename it to "MainCharacterShape". Then select the *Diagram Element Map* element in the Toolbox to map the *MainCharacter* concept to the *MainCharacterShape*. A line will appear, linking both elements. Set the *MainCharacterShape Image* property to an image of your choice. This image will be used to visually represent the main character concept in diagrams modeled by developers using the AdventureLanguage.



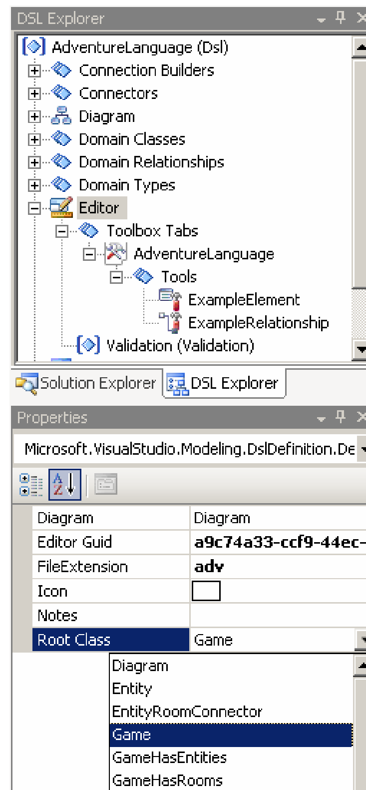
9.5. Repeat the previous step to create a *NPCShape* and link it to the *NPC* concept.

9.6. Drag a *Connector* element from the Toolbox to the main diagram. Rename it to "NextRoomConnector". Change its *Target End Style* property to *FilledArrow*. Then select the *Diagram Element Map* element in the Toolbox to map the *RoomReferencesTargetRooms* relationship to the *NextRoomConnector*.

9.7. Repeat the previous step to create an *EntityRoomConnector* and link it to the *RoomReferencesEntities* relationship.

10. Customizing the Developer Toolbox

10.1. Select the *Editor* node in the DSL Explorer window. Change its *Root class* property to *Game*.



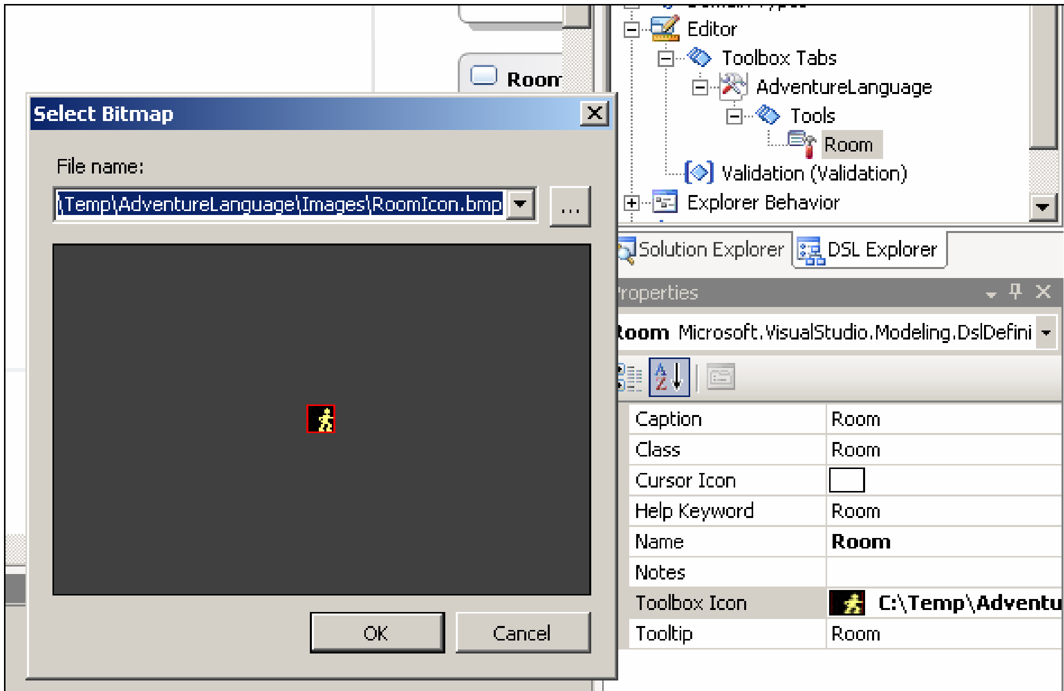
10.2. In the DSL Explorer window, delete the *Editor* > *Toolbox Tabs* > *AdventureLanguage* > *Tools* > *ExampleElement* and *ExampleRelationship* elements.

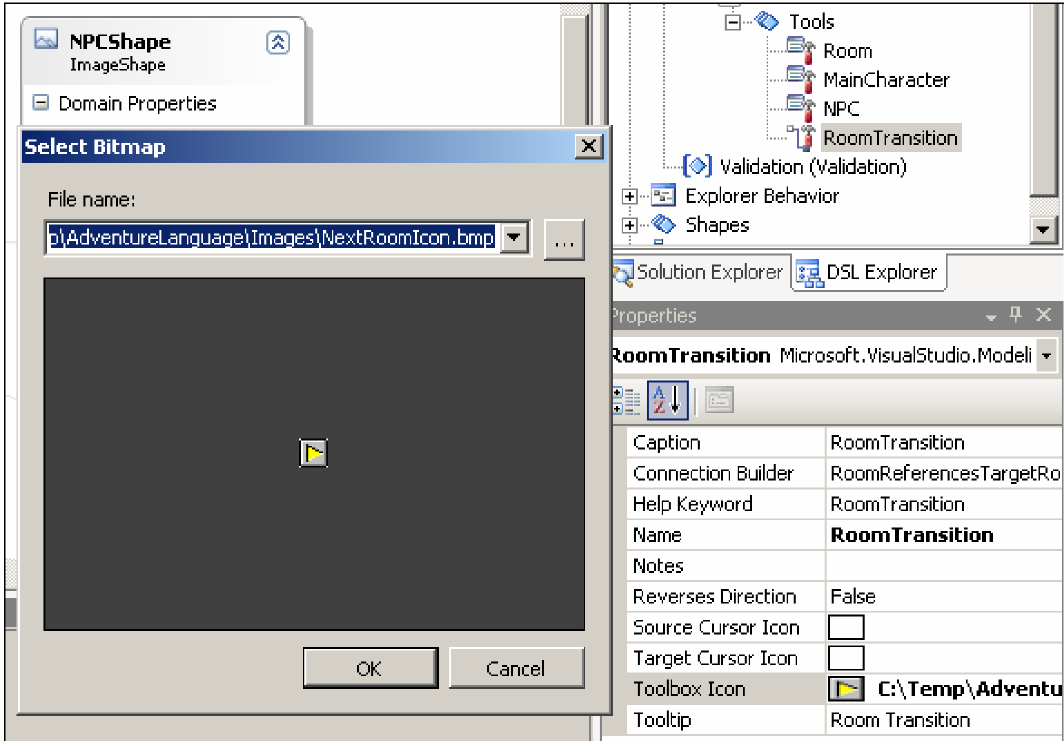
10.3. In the DSL Explorer window, right-click the element *Editor* > *Toolbox Tabs* > *AdventureLanguage* and then select *Add New Element tool*. Rename it from “ElementTool1” to “Room”. Set its *Class* property to *Room* and specify an image in the *Toolbox Icon* property.

10.4. Repeat the previous step to create other two Toolbox entries, one for the *MainCharacter* concept and the other for the *NPC* concept.

10.5. In the DSL Explorer window, right-click the element *Editor* > *Toolbox Tabs* > *AdventureLanguage* and then select *Add New Conection tool*. Rename it from “ConnectionTool1” to “RoomTransition”. Set its *Connection Builder* property to *RoomReferencesTargetRoomsBuilder* and specify an image in the *Toolbox Icon* property.

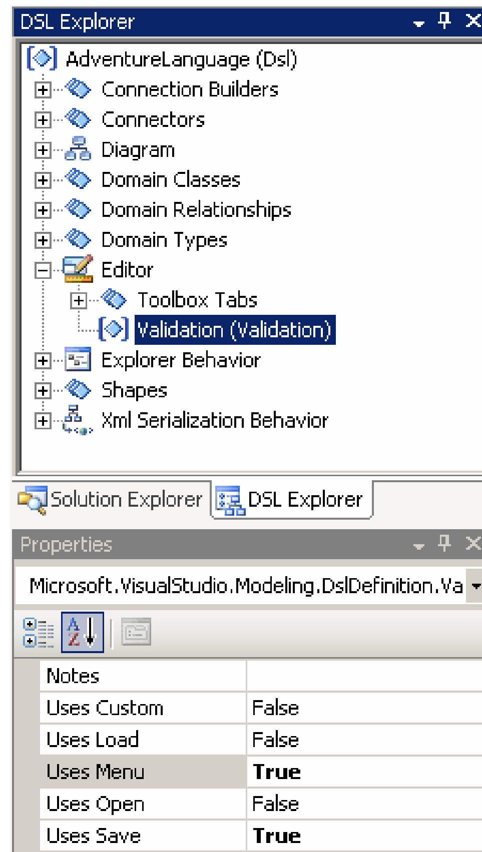
10.6. Repeat the previous step to create other a Toolbox entry named “EntityInclusion” for the relationship *RoomReferencesEntities*.


Editor
Toolbox Tabs
AdventureLanguage
Tools
Room
Validation (Validation)
Explorer Behavior
Select Bitmap
File name:
\Temp\AdventureLanguage\Images\RoomIcon.bmp
OK
Cancel
Solution Explorer
DSL Explorer
roperties
Room Microsoft.VisualStudio.Modeling.DslDefini
Caption
Room
Class
Room
Cursor Icon
Help Keyword
Room
Name
Room
Notes
Toolbox Icon
C:\Temp\Adventu
Tooltip
Room


NPCShape
ImageShape
Domain Properties
Tools
Room
MainCharacter
NPC
RoomTransition
Validation (Validation)
Explorer Behavior
Shapes
Select Bitmap
File name:
p\AdventureLanguage\Images\NextRoomIcon.bmp
OK
Cancel
Solution Explorer
DSL Explorer
Properties
RoomTransition Microsoft.VisualStudio.Modeli
Caption
RoomTransition
Connection Builder
RoomReferencesTargetRo
Help Keyword
RoomTransition
Name
RoomTransition
Notes
Reverses Direction
False
Source Cursor Icon
Target Cursor Icon
Toolbox Icon
C:\Temp\Adventu
Tooltip
Room Transition

11. Adding Semantic Validators

11.1. In the DSL Explorer window, select the *Editor > Validation* node and specify through the Properties window when semantic validation should be invoked (when saving the diagram file, when opening it, after clicking in a context menu command, etc.).



11.2. Add a class to the *Dsl* project by right-clicking the project name in the Solution Explorer window and then selecting *Add > Class*. The name of the class should be the name of the concept to which you want to add a semantic validator (for example, *Room*).

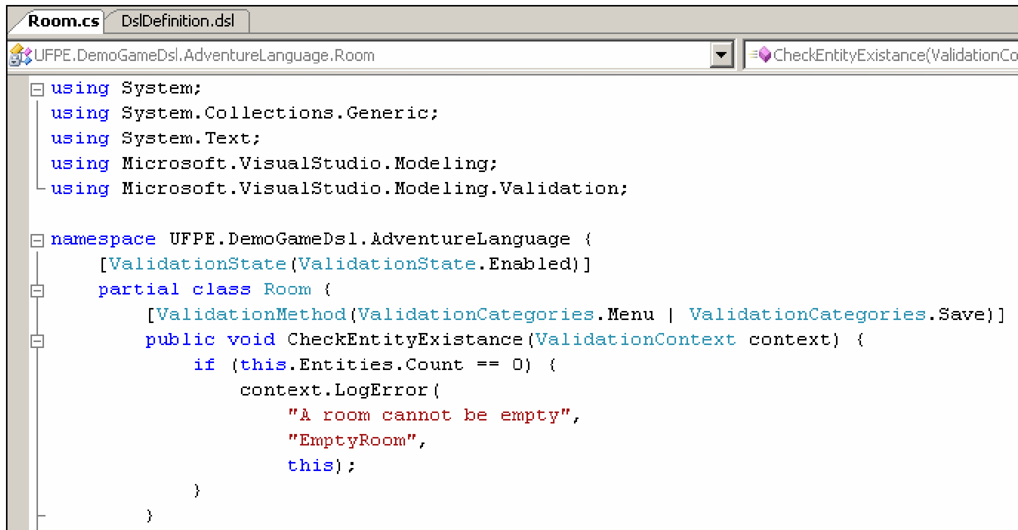
11.3. Add the *partial* modifier to the class definition and ensure that the class namespace is the same one you specified in the step 6.4.

11.4. Import the namespaces *Microsoft.VisualStudio.Modeling* and Microsoft.*VisualStudio.Modeling.Validation*.

11.5. Add the modifier *[ValidationState(ValidationState.Enabled)]* to the class.

11.6. Add to the class a method which will implement a semantic validator (for example, *CheckEntityExistance*). This method should receive a *ValidationContext* object as parameter. Add a *ValidationMethod* attribute the to the method, specifying through its constructor parameters when the validation should be invoked: *[ValidationMethod(ValidationCategories.Menu | ValidationCategories.Save)]*.

11.7. Implement business logic for the method, calling the *LogError* method of the *ValidationContext* object when the validation fails. Such a method receives as parameters the error text to be displayed in the Error List, an error code and the elements which will receive the focus when the user double-clicks the error. For example, the following code created a semantic validation to warn the user if a modeled Room contains no entities.

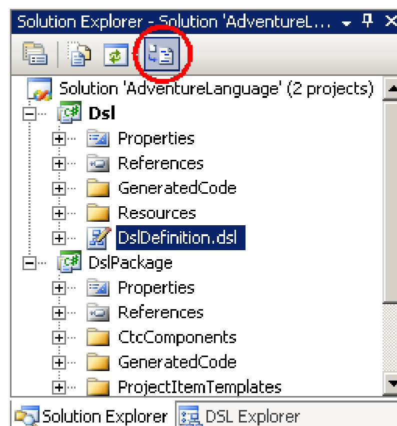


12. Implementing or Reusing a Domain Framework (Game Engine)

12.1. Developers using DSM can launch code generators, which receive modeled diagrams as input. The generated code consumes a domain framework which encapsulates domain knowledge as exposes it as a set of APIs (application program interfaces), that are reused by the many products generated from the modeling process. Considering the digital games development domain, such a domain framework is a game engine. Implement one or reuse an existing engine, probably adjusting it to the domain. The engine interface should be made easy to consume, in order to reduce the complexity required by the code generator.

13. Testing the DSL

13.1. In the Solution Explorer window, click in the *Transform All Templates* button.

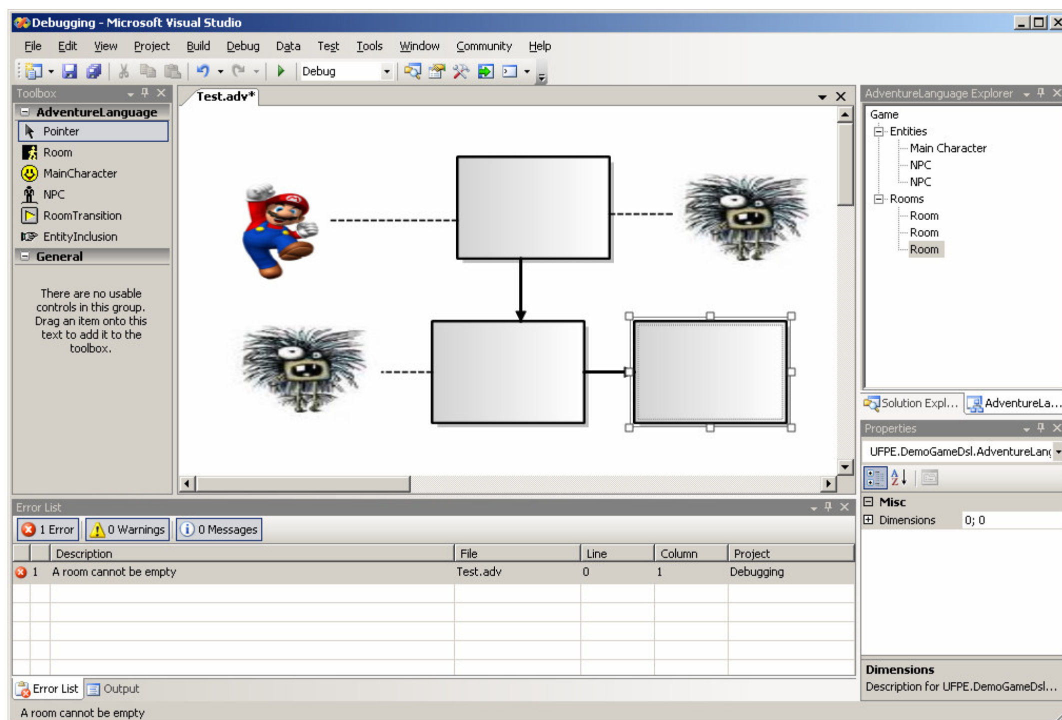


13.2. Build the solution by pressing *CTRL + SHIFT + B* (or menu *Build > Build Solution*).

13.3. Press *CTRL + F5* (or menu *Debug > Start Without Debugging*) to launch an experimental version of VS.NET which will contain the *AdventureLanguage* visual DSL already embedded.

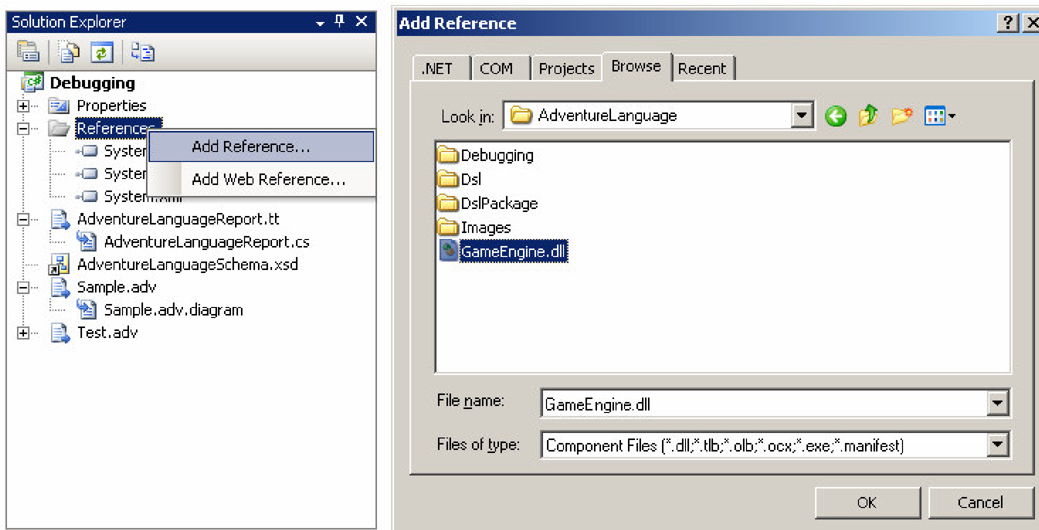
13.4. Open the *Test.adv* file and play around with it. Notice that:

- The Toolbox at the left contains domain concepts and relationships which can be dragged and dropped to the main designer.
- Concepts and relationships, once dragged to the main designer, show the visual representation previously defined.
- An *AdventreLanguage Explorer* window at the top right presents the modeled concepts hierarchically.
- Semantic validators raise errors to the Error List (at the bottom) when you try to save a file containing an empty room (i.e., a room with no entities).



14. Defining a Code Generator

14.1. Still in Visual Studio experimental instance (where the *AdventureLanguage* DSL is consumed) use the Solution Explorer window to add reference(s) to the .NET assembly(ies) of the domain framework (i.e., the game engine).



14.2. Delete the *AdventureLanguageReportVB.tt* file and open *AdventureLanguageReport.tt* ("tt" stands for "text template"). The DSL Tools includes a text template transformation toolkit that supports the processing of text templates. A text template is a file that contains a mixture of text blocks and control logic. When you transform a text template, the control logic combines the text blocks with the data in a model to produce an output file. You can use text templates to create text artifacts such as code files and HTML reports.

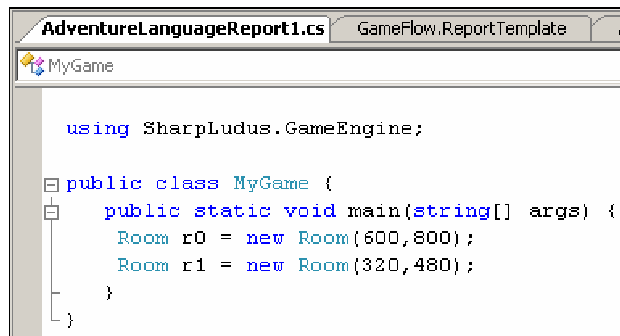
14.3. Since we'll be interested in creating C# source code, change the directive **<#@ output extension=".txt" #>** to **<#@ output extension=".cs" #>** (C# source code file).

14.4. Change the *requires* section of the *AdventureLanguage* directive (third line) to **requires="fileName='Test.adv'"**. This will make the code generator do receive the diagram modeled in the file *Test.adv* as input.

14.5. Implement your code generator by using C# programming language statements and expressions inside the **<# ... #>** tags. Text outside such tags will be directly copied to the output file. You can, however, use *if-then-else* branches, *for/foreach/while* loops and almost every C# statement to specify what will be outputted. To output the resulting value of a specific expression, use the <#= ... #> tags. For example, the following code generator script outputs a *MyGame* class containing a *Main* method where instances of the *Room* class (defined in the *GameEngine* assembly) are created. Notice that, since the generator script deals with the *Point* struct, which is defined in the *System.Drawing.dll* .NET assembly, it was necessary to add a **<#@ assembly name="System.Drawing.dll" #>** directive to the generator.

```
<#@ template inherits="Microsoft.VisualStudio.TextTemplating.VSHost.Mo
<#@ output extension=".cs" #>
<#@ AdventureLanguage processor="AdventureLanguageDirectiveProcessor"
<#@ assembly name="System.Drawing.dll" #>

using SharpLudus.GameEngine;

public class MyGame {
   public static void main(string[] args) {
<#

  int counter = 0;
  foreach (Room room in this.Game.Rooms)  {
    int width = room.Dimensions.X;
    int height = room.Dimensions.Y;

#>
    Room r<#= counter++ #> = new Room(<#= width #>,<#= height++ #>);
<#
  }
#>
   }
}
```

14.6. Save the *.tt* file and notice that a *AdventureLanguageReport1.cs* source code file is created, containing the output of the code generator. This file can be compiled with the remainder of the solution, including code that developers have added on their own.



14.7. Improve your code generator, making the generated code compliant with the game engine APIs. After this is done, modeled diagrams can automatically be transformed into the final code and the whole solution can be compiled, generating the final game.

15. Conclusions

This tutorial introduced the concepts of domain-specific modeling (DSM), domain-specific languages (DSLs) and the Visual Studio Team System DSL Tools. The required DSM steps in game development were covered, such as domain definition, concepts and relationship modeling, DSL visual syntax design and code generator implementation. A more complete DSL Tools usage example (the SharpLudus game software factory) can be found in http://www.cin.ufpe.br/~sharpludus.

As it can be noticed, taking advantage from domain-specific modeling does not come without a cost. Effort should be done to design a DSL, its visual representation, generators and domain framework. Therefore, the use of DSM only makes sense if a product line, and not only a single product, is intended to be developed.

While DSM seems to be a very interesting approach to provide a more intuitive game development experience, it is worth noticing that the presented proposal alone will not ensure the success of a game development. In fact, no technology is substitute for creativity and a good game design. Game industrialization, languages, frameworks and tools are means, not goals, targeted at the final purpose of making people have entertainment, fun and enjoy themselves. Players, not the game or its constituent technologies, should be the final focus of every new game development endeavor.

References

1. Entertainment Software Association, Essential Facts about the Computer and Video Game Industry, 2005;
2. Greenfield, J. et. al., Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley & Sons, 2004;
3. Zerbst, S., Duvel O., 3D Game Engine Programming, Course Technology PTR, 1st edition;
4. Domain-Specific Modeling (DSM) Forum, http://www.dsmforum.org;
5. Visual Studio 2005: Domain-Specific Language Tools, http://msdn.microsoft.com/vstudio/DSLTools/;
6. Deursen, A.; Klint, P.; Visser, J. Domain-Specific Languages: An Annotated Bibliography, http://homepages.cwi.nl/~arie/papers/dslbib/;
7. Fayad, M. E.; Schmidt, D. C. Object-oriented application frameworks, Communications of the ACM, 1997;
8. Johnson, R.; Foote, B. Designing reusable classes, Journal of Object-Oriented Programming, 1988;
9. Fowler, M. Language Workbenches: The Killer-App for Domain Specific Languages?, http://www.martinfowler.com/articles/languageWorkbench.html;
10. Ladd, D. A.; Ramming, J. C. Two application languages in software production, in USENIX Very High Level Languages Symposium Proceedings, 1994;
11. Kieburtz, R. B.; McKinney, L.; Bell, M.; Hook, J.; Kotov, A.; Lewis, J.; Oliva, D. P.; Sheard, T.; Smith, I.; Walton, L. A software engineering experiment in software component generation, in Proceedings of the 18th International Conference on Software Engineering, 1996;

12. van Deursen, A.; Klint, P. Little languages: Little maintenance?, Journal of Software Maintenance, 1998;
13. Herndon, R. M.; Berzins, V. A. The realizable benefits of a language prototyping language, IEEE Transactions on Software Engineering, 1988;
14. Menon, V.; Pingali, K. A case for source-level transformations in MATLAB, in Proceedings of the second USENIX Conference on Domain-Specific Languages, 1999;
15. Bruce, D. What makes a good domain-specific language? APOSTLE, and its approach to parallel discrete event simulation, in First ACM SIGPLAN Workshop on Domain-Specific Languages, 1997;
16. Basu, A.; Hayden, M.; Morrisett, G.; von Eicken, T. A language-based approach to protocol construction, in First ACM SIGPLAN Workshop on Domain-Specific Languages, 1997;
17. Sirer, E. G.; Bershad, B. N. Using production grammars in software testing, in Proceedings of the second USENIX Conference on Domain-Specific Languages, 1999;
18. Visual Studio Team System Developer Center: Software Factories, http://msdn.microsoft.com/architecture/overview/softwarefactories;
19. Krueger, C. W. Software reuse, ACM Computing Surveys, June 1992;
20. Visual Studio 2005 Team System: Overview, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvsent/html/vsts-over.asp;
21. Tolvanen, J.-P. Domain-specific Modeling: Welcome to the Next Generation of Software Modeling, http://www.devx.com/enterprise/Article/29619;
22. JetBrains Meta Programming System, http://www.jetbrains.com/mps;
23. Intentional Software, http://intentsoft.com;
24. C# Developer Center, http://msdn.microsoft.com/vcsharp/;
25. Crawford, C. A Taxonomy of Computer Games, http://www.vancouver.wsu.edu/fac/peabody/game-book/Chapter3.html;
26. Sawyer, B. The Getting Started Guide to Game Development FAQ, http://www.gamedev.net/reference/articles/article261.asp;
27. Wolf, M. Genre and the Video Game, in Wolf M.J.P, The Medium of the Video Game, University of Texas Press, 2002;