Automata-based Programming in Visual Studio 2005: State Machine Designer Tool

Evgeny O. Reshetnikov
Saint-Petersburg University of Informational Technologies, Mechanics and Optics
ereshetnikov@ rambler.ru

Abstract

This article introduces State Machine Designer tool for automata-based programming in Visual Studio 2005. State Machine Designer extends functionality of Visual Studio 2005 and gives developer more abilities for designing and realization of software products. This tool allows developer to create UML-like visual models of project, add automata behavior to any class in the project, and generate a part of source code on C# programming language.

1. Introduction

Currently it becomes obvious that there is no common way in developing of software products because of many different programming standards and techniques. Programming requires more and more specifications to make this process clear. Specification is a good practice but it is not enough because it does not reflect the code which developer will produce. Usage of visual models is a good way to bring source code concepts and specifications together. Modelling allows developer to see main structure of a project and main interactions between its components. The last problem is to make models and source code being related. *UML*-diagrams [1] may describe both of system behavior and project structure simultaneously. *State Machine Designer* allows usage of three *UML*-diagrams during project developing. Those are *Classes Diagram*, *Objects Diagram* and *State Machine Diagram*. By using these three models a part of source code could be generated automatically.

The following *UML*-diagrams are used in *State Machine Designer*:

Classes Diagram – describes classes, interfaces and its relations.

State Machine Diagram – describes behavior of the entity whose lifecycle could be represented as a state machine. This state machine consists of finite count of states and transitions between them. States represent some stable states of the system and transitions occur on the specified system events and if corresponding guard conditions are satisfied [2].

Objects Diagram – reflects instances of classes with initial values of its properties and aggregation relationships between these instances.

2. Automata-based programming

Automata-based programming is a programming technology where finite state machines are used for representing the whole program or some of its parts.

Finite state machines could be implemented using different methods such as state design pattern [3] or SWITCH-Technology [4] or any other well-known method. But the main idea remains the same: introduce finite number of states, provide transitions between these states on some system events and make specified actions on states entering, leaving and transitioning between them.

Automata-based programming is helpful in development of compilers, automation solutions and every application whose logic could be represented as finite state machines.

3. Implementation

State Machine Designer tool developed as a plug-in for Visual Studio 2005 [5] and based on Domain-Specific Language Tools [6] which is a part of Microsoft Visual Studio 2005 SDK.

Domain-Specific Language Tools are used for creation of custom visual editors. These editors serve for editing each of the three following models: *Classes Diagram*, *State Machine Diagram* and *Objects Diagram*. Obtained diagrams represent a part of a project. Source code for this part is generated automatically on *C#* programming language [7].

3.1. Classes Diagram

Classes Diagram is intended for visual creation of *classes*, *interfaces* of the project and setting of its relationships. *IDE Visual Studio 2005* has its own classes diagram which also allows creating classes, interfaces and so on. It has some disadvantages though: it is not always synchronized with a source code. For example, if we create any class on standard class diagram and then delete this diagram created class will stay in the project. And wise versa if we create a class independently of the diagram this class will appear on the diagram only after diagram's regeneration and saving. It is unacceptable to have a source code and diagrams which are different. Diagram and source code should always be synchronized to prevent misleading situations.

Classes Diagram introduced in this paper is completely synchronized with the code it reflects. This diagram also has possibilities for adding of automata behaviour for the classes on it. On adding automata behaviour from *Classes Diagram* to some class automatic transition to the *State Machine Diagram* is performed.

Classes Diagram also has validation mechanism which prevents creation of wrong constructions in meaning of C# language concepts.

3.2. State Machine Diagram

State Machine Diagram is intended for creation of visual models for the objects which lifecycle could be represented by finite count of states with guarded transitions between them.

State Machine Diagram has validation mechanism which allows guaranteeing the following rules:

1. State machine has only one initial state.

2. State machine has at least one final state.

3. Every transition has specified event on which this transition is performed.

4. Every state should be reachable from the initial state.

5. Transitions with the same event and same source state should have orthogonal guard conditions.

The fifth rule is very hard and has no accurate solution in this work.

3.3. Objects Diagram

Object Diagram is intended for visual creation of starting configuration of the application. For each object on the diagram instance of specified type is created and developer are able to set initial values of properties for each object, set relationships between different objects. And if there is object with automata behaviour on the diagram, developer can mark corresponding state machine as a start point of the whole application.

3.4. Code generation text templates

Code generation for specified model is possible with *DSL Tools.* Code generation is performed using text template transformation. Text templates are different for each of the three diagrams. On every model change new source code is generated. Thanks to this approach source code always completely represent visual model for any of the three diagrams.

Developer should never modify auto generated source code because it will be overwritten on the next saving or compilation of the project.

4. Usage

State Machine Designer could be used during development of a new project in *Visual Studio 2005* and during adding of new functionality to existing project as well. Usage of *State Machine Designer* is

useful in almost all cases when developer wants to add some automata behavior to a project.

When *State Machine Designer* plug-in is installed developer can add new entities in his project and edit diagrams using implemented editors. There are three items which are corresponding to diagrams: *ClassesLanguage*, *StateMachineLanguage* and ObjectsLanguage. Files with those extensions automatically use custom editors which are described below.

4.1. Classes Diagram editor

For editing of *Classes Diagram* developer adds new item to the project with special extension *ClassesLanguage*. When developer double-clicks on such an item special editor is appeared in new document window (Figure 1).

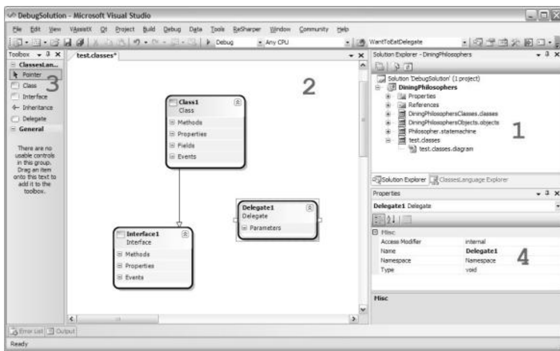

Figure 1. Classes Diagram editing in Visual Studio 2005.

Digits on the picture specify the main windows of *IDE* during classes editing:

1. *Solution Explorer* – tree list of modules and files in the project. Using *"Add New Item ..."* command developer can add new files to the project as well as files with *ClassesLanguage*, *StateMachineLanguage* and *ObjectsLanguage* extensions which represent supported by tool diagrams.

2. *Classes Diagram* editor area. Developer is able to add classes and interfaces to this area using extended toolbox.

3. Toolbox which contains items for adding classes, interfaces and inheritance relationships to *Classes Diagram*.

4. Property window for the diagram's active object.

Using this diagram adding of automata behaviour to any class is possible. After right click on any class figure on the diagram context menu is shown (Figure 2). Developer can choose *"Add/Edit automat behaviour"* item and active window will be automatically switched to the *State Machine Diagram* for the chosen class. If there is no such a diagram it will be automatically created.

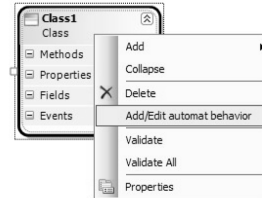

Figure 2. Context menu for the class on Classes Diagram.

All classes which have its own state machine are drawn with "A" icon in the top-left corner (Figure 3).

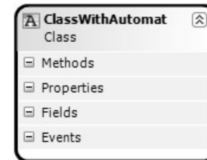

Figure 3. Class with automata behaviour.

Implementation for all methods for all classes from the diagram could be done using additional code editor which is shown after double click on any method (Figure 4).

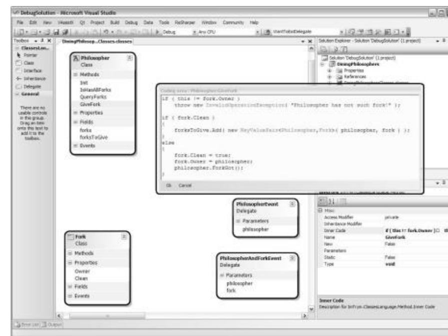

Figure 4. Additional source code editor.

Classes Diagram supports model validation. For example, it doesn't allow inheritance of interface from the class or class from two or more classes because such constructions are wrong in .NET languages. If some of the restrictions are not satisfied message with corresponding error is shown in error window.

4.2. State Machine Diagram editor

For editing of State Machine Diagram developer adds new item to the project with special extension *StateMachineLanguage*. Such items also have their own editor (Figure 5). *StateMachineLanguage* item could be created automatically when developer adds automata behavior to some class on the classes diagram.

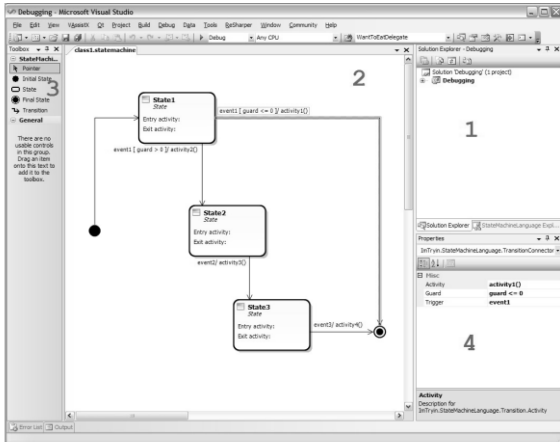

Figure 5. State Machine Diagram editing in Visual Studio 2005.

Digits on the picture mean the same areas as on Figure 1. Toolbox for state machine editor contains *"Initial state"*, *"State"*, *"Final state"* and *"Transition"* items. Using this items developer visually creates state machine. *State Machine Diagram* also supports validation. The following rules are always satisfied: there is only one initial state, there are no unreachable states and so on.

4.3. Objects Diagram editor

For editing of *Objects Diagram* developer adds new item to the project with special extension *ObjectsLanguage* (Figure 6).

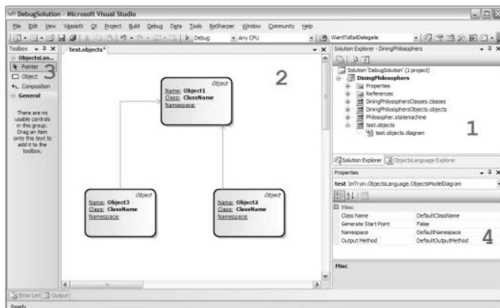

Figure 6. Objects Diagram editing in Visual Studio 2005.

This diagram is useful when application has fixed count of different objects. In this case objects configuration could be represented on the diagram. And those objects which have automata behaviour will be having "A" icon in the top-left corner of the shape (Figure 7).

Figure 7. Object with automata behaviour.

On *Objects Diagram d*eveloper could assign some of state machines to start when the whole application is started.

5. Conclusions

This paper describes a tool for *Microsoft Visual Studio 2005* which allows developer to model and develop application using three diagrams: *Classes Diagram*, *State Machine Diagram* and *Objects Diagram*. This tool extends abilities of *Microsoft Visual Studio 2005* in applications designing and development. With this approach a part of the source code is generated automatically that follows to decreasing of errors count. Thanks to visual models application becomes more clear and logical.

Developed tool could be used in development of any application but it is most helpful in cases of reactive systems [8]. Usage of the tool in development helps programmer to see static and dynamic models of the application simultaneously that also makes development process easier.

Currently there is no similar tool for *Visual Studio 2005* which will help developer to combine traditional programming techniques with automata-based programming.

6. References

[1] Dan Pilone, Neil Pitman, *UML 2.0 in a Nutshell*, O'Reilly, 2005.

[2] Tukkel N.I., Shalyto A.A., "State-based programming", *PC World*, 2001, #8, pp.116-121; #9, pp.132-138.

[3] Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns*, MA: Addison-Wesley Proffesional, 2001, p.395.

[4] Shalyto A.A., "SWITCH-Technology. Algorithmization and Programming of Logic Control Problems", St. Petersburg: Nauka, 1998.

[5] Microsoft Corporation, *Microsoft Visual Studio 2005*, http://msdn.microsoft.com/vstudio/.

[6] Microsoft Visual Studio Developer Center, *Domain-Specific Language Tools*, http://msdn.microsoft.com/vstudio/DSLTools/.

[7] Jesse Liberty, Brian MacDonald, *Learning C# 2005*, O'Reilly, 2006.

[8] Harel D. et al. "Statemate: A working environment for the development of complex reactive systems", *IEEE Software Eng.*, 1990, #4.