UML Modeling with Visio, Part 1

This is the first article in a two-part series whose aim is to explain the fundamentals of using Microsoft Visio to create UML diagrams for use with Visual Studio .NET. This first article explores some of the basics of Visio and UML diagrams, as well as how to create static diagrams within Visio. The second article details how to create dynamic diagrams in Visio and generate code from them in your language of choice.

Introduction

The Enterprise Architect version of Microsoft Visual Studio .NET 2003 includes a copy of Microsoft Visio for Enterprise Architects (VEA). For those of you new to the product, Microsoft Visio is a drawing package that allows you to select predrawn shapes from stencils to draw and design different kind of diagrams such as flowcharts, network diagrams, and software diagrams.

Software application developers can model the application's design and functionality with Visio and Unified Model Language (UML) 2.0 through Visio's UML Model Diagram template. Also, Visio can perform reverse engineering on an implemented system and transform existing code into a UML model.

This two-part article series explains how to model different UML diagrams with Visio and how to transform these diagrams into code in a .NET programming language, thus reducing the work necessary to implement the modeled solution from scratch.

In this first article, I start with a brief overview of several UML diagrams and examine their place in the development cycle. I analyze how to design a university system using UML diagrams. The system will allow students to enroll in seminars and take courses. Courses are taught by specific professors. The system will also store both students' and professors' addresses. This article covers the development of the static diagrams for the university system.

System Requirements

To work with the solution in this sample, you should have

- Visual Studio .NET 2003 Enterprise Architect (which includes VEA)

or

- Visual Studio .NET and Microsoft Visio 2000 or later

The Sample Code

All the diagrams developed in this article are available for download from the ASP Today site. The download contains the file *UniversityModel.vsd* , which contains all the diagrams in Microsoft Visio format.

UML Diagrams

The Unified Modeling Language (UML) was the result of the collaboration of three different modeling techniques that existed from 1980 to the mid-1990s, led by Grady Booch, James Rumbaugh, and Ivar Jacobson, who were pursuing a common modeling notation. As industry involvement grew in the project, Booch, Rumbaugh, and Jacobson (known as "The Three Amigos") worked with Object Management Group (OMG) to have UML adopted as the standard industry modeling language, which finally occurred in 1997. Since then, UML has evolved to address changing industry needs and will continue to do so in the future.

UML provides a common notation for specifying, constructing, and documenting systems that use object-oriented code, such as C# or Visual Basic .NET. UML defines a set of diagrams for Object-Oriented Analysis and Design (OOAD). These diagrams can be grouped into two distinct categories:

- **Static diagrams for UML analysis** represent the logical structure of information objects. They are used to map the conceptual problem domain. Their purpose is not to produce code, but to help developers understand the problem that needs solving. These diagrams include
- **Use case diagrams** : Used mainly for enterprise viewpoint modeling to discover the key functionalities of a system and how users (represented by actors) interact with it
- **Static structure diagrams** : Used to represent simplified or detailed class diagrams because they identify objects and define objects characteristics such as attributes and operations
- **Implementation (component and deployment) diagrams** : Describe the computational components in the system and how they are deployed
- **Dynamic diagrams for UML design** represent the behavior and activities of the information objects. They are used to map the physical problem domain. These diagrams include
- **Class diagrams** : Based on static structure diagrams, class diagrams are one of the most important dynamic diagram types in UML. Class diagrams produced at this point will be used to generate code.
- **Interaction diagrams (** such as sequence and communication diagrams): Model the behavior of use cases by describing the way groups of objects interact to complete a task.
- **State diagrams (such as activity diagrams**): Represent the different internal state and transitions of objects in the system.

The relation between the static and dynamic models is important. Static models represent concepts without much detail and cannot be accurate without associated dynamic models. Dynamic models, on the other hand, do not adequately represent considerations of structure and dependency management. Both types of diagrams need to be combined to model the best solution. Once you are working on dynamic diagrams, there is often too much emphasis on designing class diagrams rather than on other dynamic diagrams, such as interaction and communication diagrams. Interaction and communication diagrams, however, must not be undervalued. They are critical in the modeling process because they ultimately model the software behavior.

This article, Part 1, covers the static diagrams. I'll focus on the static structure diagram produced during the UML design to depict what is called the class diagram and the generation of code from it. Part II covers the dynamic diagrams in detail.

The entire UML specification and UML usage guidelines are available for viewing and download from the OMG site (see the Related Links section at the end of this article for the site's URL).

One of the biggest advantages of designing class diagrams with Visio is that they can automatically be transformed into code: C#, C++, or Visual Basic .NET - the decision is yours. This code can be placed into independent files or inside a Visual Studio .NET project. In addition to the code itself, class diagrams can generate documentation to be saved as a report in Microsoft Word format, as you'll see later. These two topics are covered in Part II.

Note: The best diagram to convert into code is the class diagram, because it represents the whole system in great detail. Static diagrams, as well as other dynamic diagrams such as interaction and communication diagrams, cannot be directly converted into code.

Visio UML Model Diagram Stencils

To work with UML diagrams, open Visio and choose *UML Model Diagram* under the *Software* category. Note that depending on the version of Visio you are using, you may be able to select UML Model diagrams in U.S. or metric units.

If you don't see the window displayed in Figure 1 when you open Visio, go to the menu bar and select *File --> New --> Software --> UML Model Diagram* .

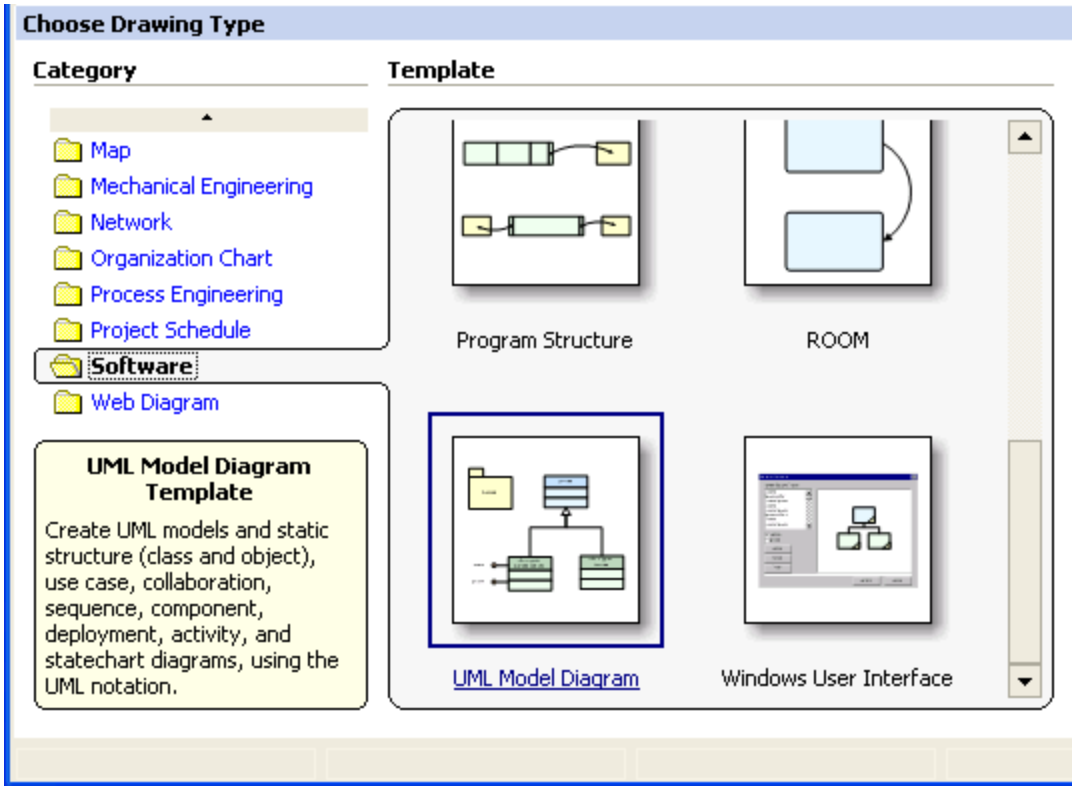

Figure 1. Visio Drawing Type window

This will create a new Visio document, as shown in Figure 2. On the left area of the screen is a window called *Shapes* that displays the different stencils available in the *UML Model Diagram* component. These stencils contain drag-and-drop elements you can place on the different UML diagrams described previously.

On the bottom area of the screen is a window called *Model Explorer* . This window will prove to be extremely useful when you are working with big models, as it allows you to see all the different objects and their properties, and all diagrams involved in a given project at a glance. You can also move, delete, or rename objects in Model Explorer, similar to the way you can in the *Class view* window in Visual Studio .NET. You can access the *Model Explorer* window at any time by going to the menu bar and selecting *UML --> View --> Model Explorer* .

You can set the programming language for the code generation by going to the menu bar, selecting *UML --> Code --> Preferences* , and choosing the appropriate language from the *Target language* drop-down menu.

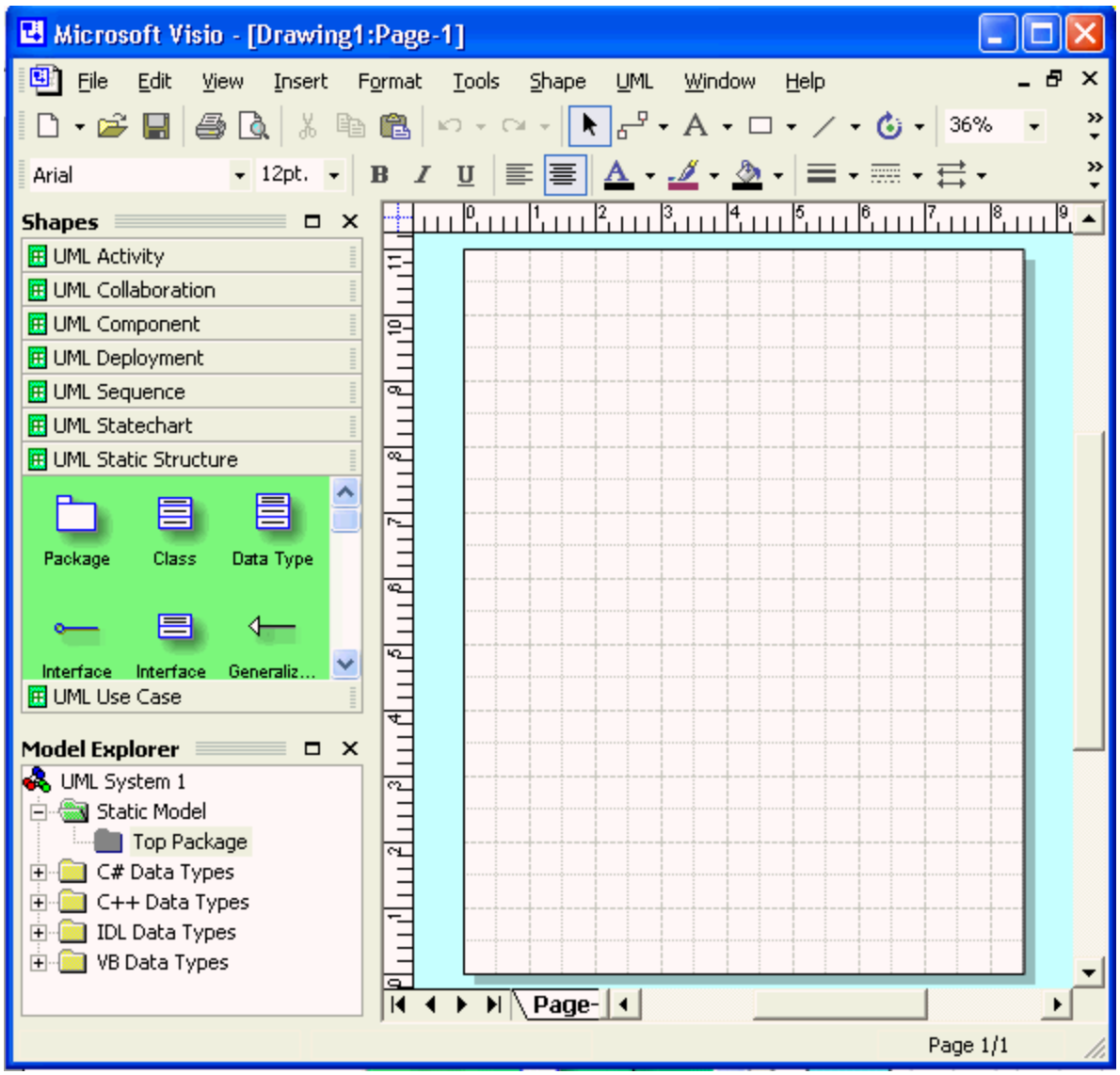

Figure 2. UML stencils, Model Explorer, and the Drawing area

UML Analysis: Static Diagrams

Before you start adding diagrams to the model, you need to stop and think about how you want to organize the different information in the model you are about to build into folders, and where you want to save different types of diagrams and objects in different packages. This point is very important because Visio will use the final model structure as namespaces when the code is generated.

To create a new package, go to the *Model Explorer* window, right-click *Top Package*, choose *Package*, and rename the new package *Conceptual Model*. All the different diagrams produced during the UML analysis phase can be saved in the *Conceptual Model* folder.

Note that you can rename the pre-existing packages in this window by simply right-clicking the name and choosing *Rename*. You can rename the *Static Model* package, for example, to something more meaningful to your business. Since the different examples in this article are based on a university system, I renamed it *University Model*. This means that the resultant namespace will be called *UniversityModel*.

Below the different packages included in the model, the *Model Explorer* window lists the most basic data types for some of the most popular .NET languages: C#, C++, and Visual Basic .NET. There may be occasions when you need to add new data types or interfaces to the language that you are working with. To add a new data type (or interface), just right-click the language where the data type (or interface) needs to be added and choose *New*. Then go ahead and enter the data type (or interface) name.

For some reason, the *C# Data Types* folder doesn't include a *DateTime* data type in Visio. To add it, follow these steps:

1. Right-click *C# Data Types* and choose *New* .
2. Enter the data type (or interface) name-in this case, it's *DateTime* , as shown in Figure 3.
3.

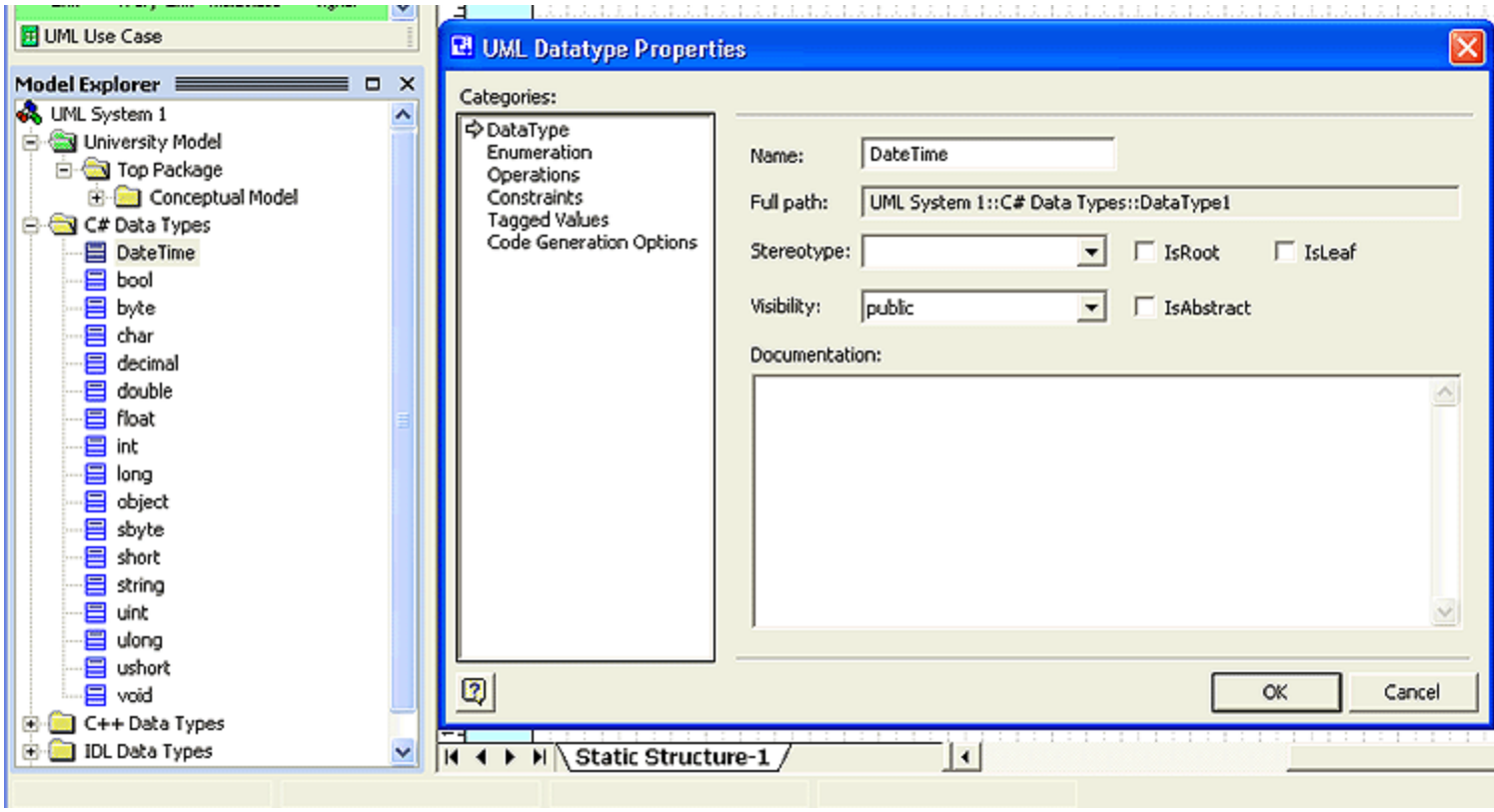

Figure 3. Datatype Properties window

Use Case Diagrams

Let's start with the first of the static UML diagrams, the use case diagram. In requirements analysis, uses cases describe the functionality of the system from the end user perspective, which identifies the system's boundaries and scope. They describe a sequence of actions initiated by external entities, either users or other systems called actors that provide some value to the project stakeholders or to other actor(s).

Basic elements in UML use case diagrams are as follows:

- **Use case** : This represents the use case itself, drawn as a horizontal ellipse.
- **Actors** : Actors are any external entity that makes use of the system being modeled, including any person, device, or external system that have access to, or make use of, the information and functions present in the system being modeled. They are drawn as stick figures.
- **Associations:** These occur between actors and use cases, and are used when an actor is involved with an interaction described in the use case. They are indicated by solid lines with an optional arrowhead on one end of the line. The arrowhead is often used to indicate the direction of the initial invocation of the relationship or to indicate the primary actor within the use case.

To create a use case diagram with Visio, click the *UML Use Case* diagram stencil in the *Shape* window to access the diagram's elements, and then click and drag elements to the drawing area as needed.

Continuing with the university system example, imagine I have a use case called *Student Enrolls in Seminar* . To model it with Visio, I would do the following:

1. In the *Model Explorer* window, right-click *Conceptual Model* , select *New -->Package* , and name the package *Use Cases* .

2. Right-click the *New --> Use Case Diagram* folder just created. Rename the diagram *Student Enrolls in Seminar*. Notice how the *Student Enrolls in Seminar* use case diagram also shows up in *Model Explorer*, as shown in Figure 4.
3. Click and drag an *Actor* from the stencil area to the drawing area. This actor is the person (or system) that starts the action in the use case (the student in this example). Double-click the figure to access its properties and rename it from *actor1* to *student*. Notice how the *student* actor also shows up in *Model Explorer*, as shown in Figure 4.
4. Click and drag a *Use Case* from the stencil to the drawing area. Double-click the ellipse that represents the use case to access its properties and rename it from *UseCase1* to *Enroll in Seminar*. Notice how the *Enroll in Seminar* use case also shows up in *Model Explorer*, as shown in Figure 4.
5. Connect the *Student* actor and the *Enroll in Seminar* use case by dragging an association *communicator* from the stencil.
6.

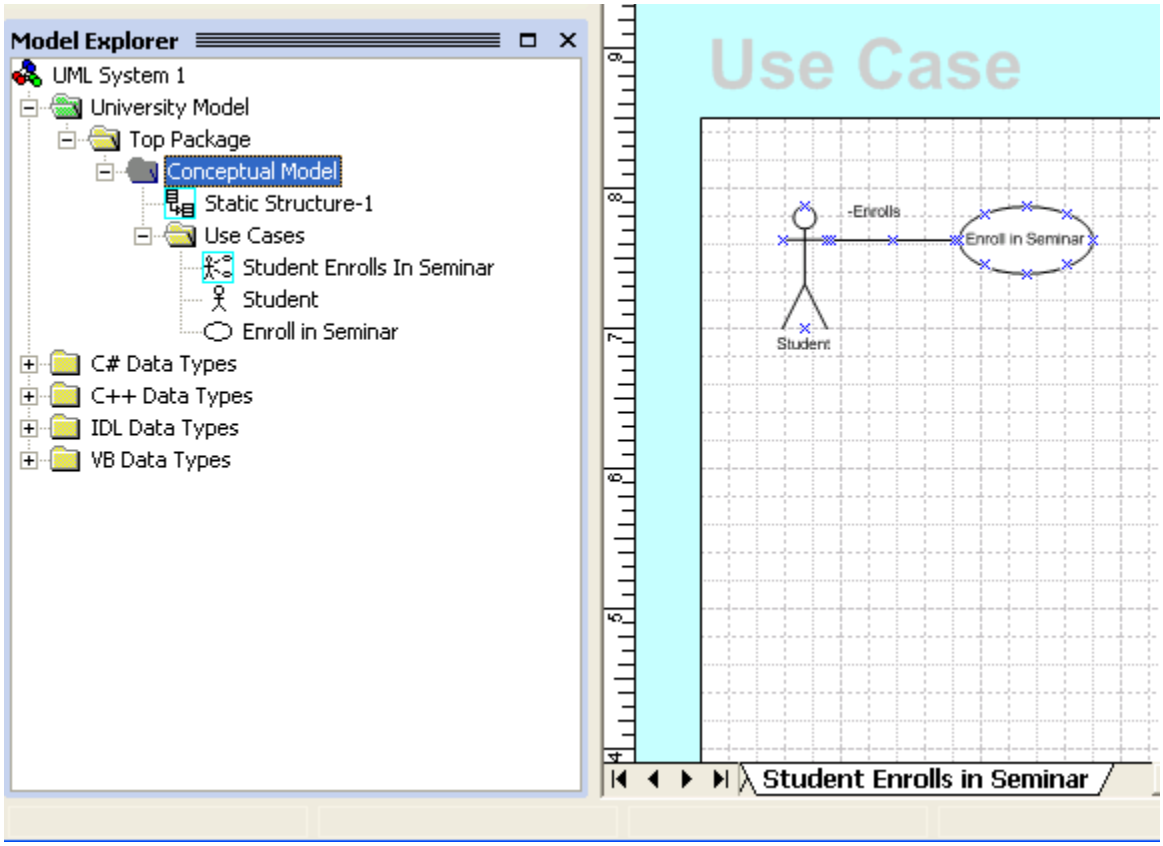

Figure 4. Use case diagram and Model Explorer window

Static Structure Diagrams: The Conceptual Model

Things start to get interesting here, since I'll use the most important diagram used for code generation, the static structure diagram, to represent the conceptual model diagram. Remember that the conceptual model won't include enough information to be transformed into code, but during the design process, you'll see how to work with static structure diagrams that you'll use when you're ready to work on the class diagram.

A conceptual model is a high-level static view of the objects and classes that make up the design/analysis space. It involves object-oriented programming (OOP) concepts such as classes, inheritance, and so on, which makes it the most (or one of the most) important UML diagram. It represents how the different objects in a model interact, without trying to define the objects' properties. The conceptual model only cares about understanding the problem that needs to be solved by identifying the main objects in the system and their relation. The conceptual model doesn't attempt to solve the problem, and therefore doesn't describe these objects and relations in detail.

Basic elements in UML static structure diagrams at the conceptual model level are as follows:

- **Classes:** Initially represented as boxes with three sections:
- The top section represents the name of the class.

- The middle section represents the attributes or properties of the class.
- The third section represents the methods of the class.

In the conceptual model, you aren't yet interested in attributes or methods, and classes will be represented by just a box. You'll see how this representation can be accomplished with Visio in a moment.

- **Associations:** Used to link different objects and represent static relationships between classes. Most associations in UML diagrams are plain lines between classes and/or objects. Every association can have a name and two ends to identify roles. Roles are used to describe the nature of the association or the way the two classes see each other. Each end's multiplicity can be set by a number or a range of numbers. You will see different types of associations in detail later in this article, such as
- **Generalization:** (Used to represent inheritance.) A generalization is represented as an inheritance link indicating one class is a superclass of the other, representing an "Is a" relationship. It is represented with a triangle pointing to the superclass.
- **Composition:** (Used for showing part-whole relationships.) Composition is an association in which one class belongs to a collection, representing a "Has a" relationship. It is depicted with a diamond end pointing to the part containing the whole.

Note: UML 2 (and Visio) no longer supports the concept of aggregation, a weaker form of composition, which was depicted in UML 1.x using a hollow diamond.

Visio creates by default an empty static structure diagram when you first start a UML model.

To create a conceptual model diagram in Visio, follow these steps:

1. Click *Static Structure* under *Conceptual Model* in the *Model Explorer* window, and rename it to something more meaningful, such as *Conceptual Model Diagram* .
2. Click and drag a *Class* element from the *UML Static Structure* to the drawing area and rename it *Student* . Note that a *Student* class appears now in the *Model Explorer* window.
3. In the conceptual model, you are interested only in concepts or objects, not their properties or attributes. To modify the appearance of a class, right-click the class and select *Shape Displays Options* . In the *Suppress* section, check *Attributes* and *Operations* . To make this change repeat automatically, also select the *Apply to subsequently dropped UML shapes of the same type in the current drawing window page* check box. The class now contains just the class name, like the *Student* class displayed in Figure 5.

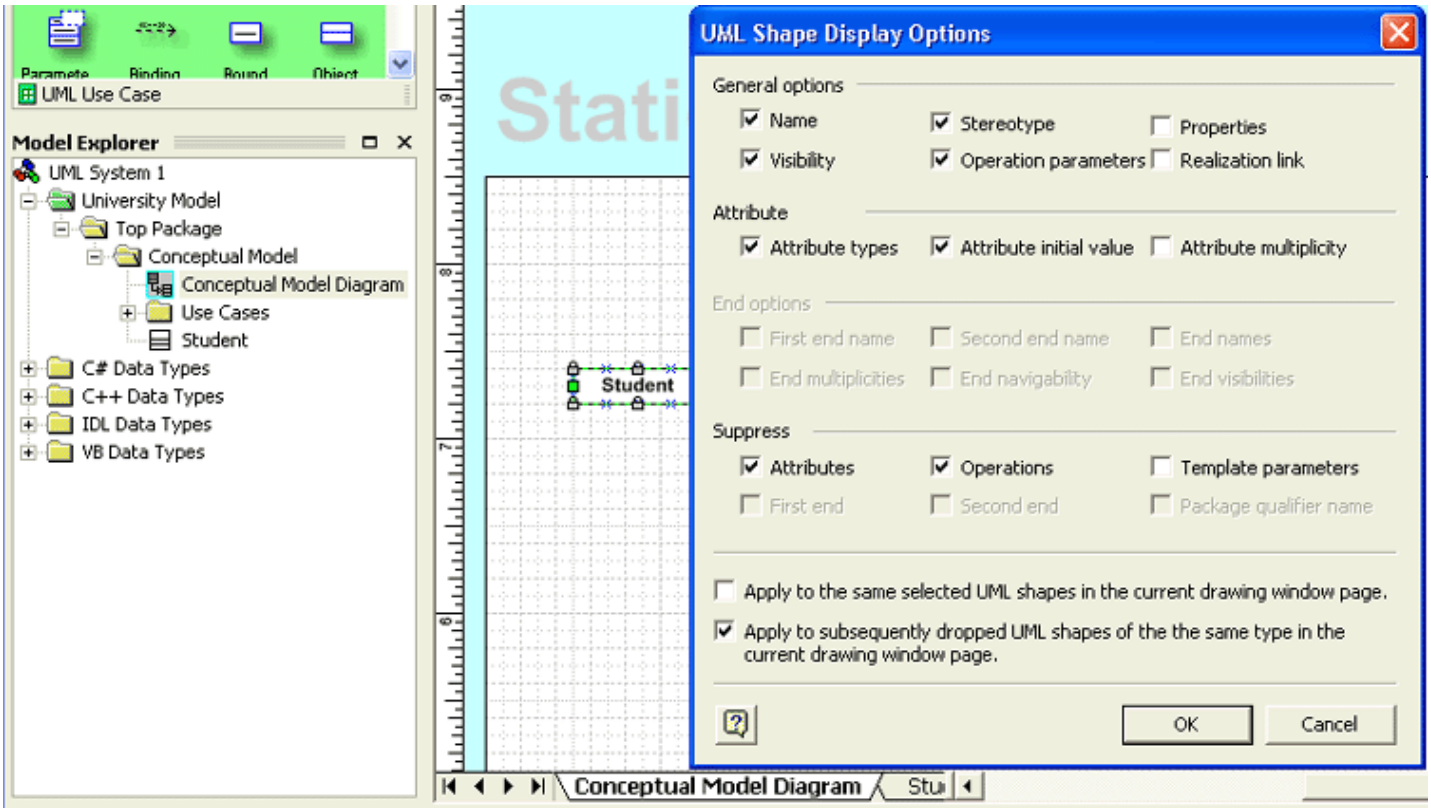

Figure 5. Student class Display Options window

4. Add the additional classes that will compose the model: *enrollment* , *seminar* , *course* , *professor* , and *address* .

Next, I have to identify how the different classes in the conceptual model relate to each other by establishing associations.

Note: When connecting two classes with an association, make sure that both ends of the association are connected to a class through a class port (one of those light blue triangles that surround a class). If an association is correctly established between two classes, it will be displayed in black. If there is an error in one of the ends, the association will automatically display in red.

You can edit the properties of an association, such as composition, multiplicity, visibility, and so on, by double-clicking the association itself, as shown in Figure 6.

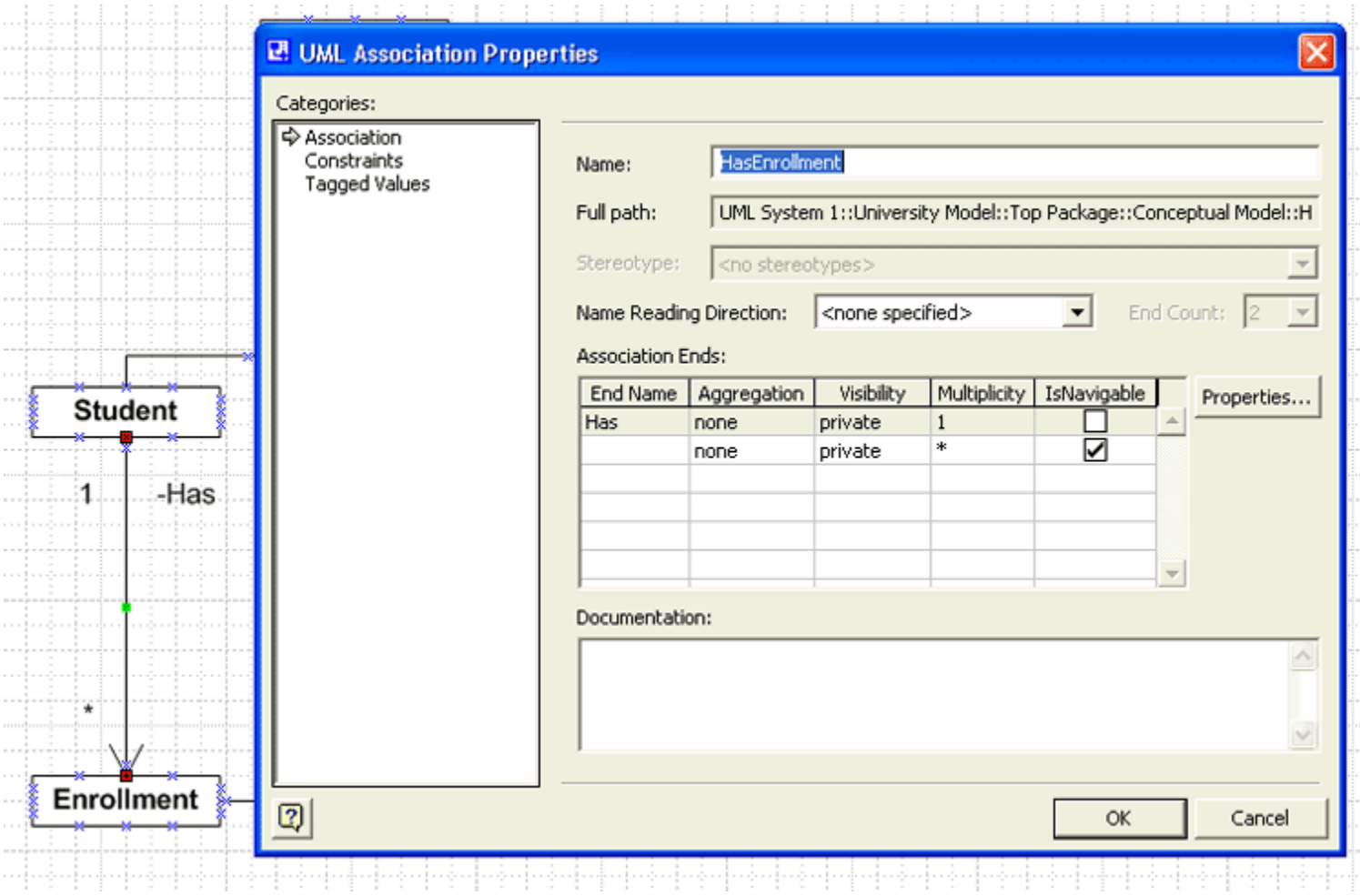

Figure 6. Association Properties window

Here is a brief explanation of the most important properties of an association:

- The *IsNavigable* check box defines whether to show the end of the association in code. The end will be shown if the check box is checked.
- The *Multiplicity* of an association end is the number of possible instances of the class associated with a single instance of the other end. Multiplicities are single numbers or ranges of numbers.
- 0..1 means zero or one instance.
- 0..* or * means there's no limit on the number of instances.
- 1 means exactly one instance.
- 1..* means at least one instance.

Once the different associations have been established, I need to analyze the model, searching for generalization and composition. In the university system example, there are students, seminars, classes, professors, and

addresses. Students and professors will share some properties. They both have an address, for example. Therefore, I design these classes as follows:

- Both `Student` and `Professor` are a generalization of a `Person` class.
- The `Person` class has an `Address` object.

Following the university system example, Figure 7 displays the conceptual model that depicts different objects that belong to the university system. Specifically, the model represents the students' enrollment in seminars that offer different courses taught by different professors.

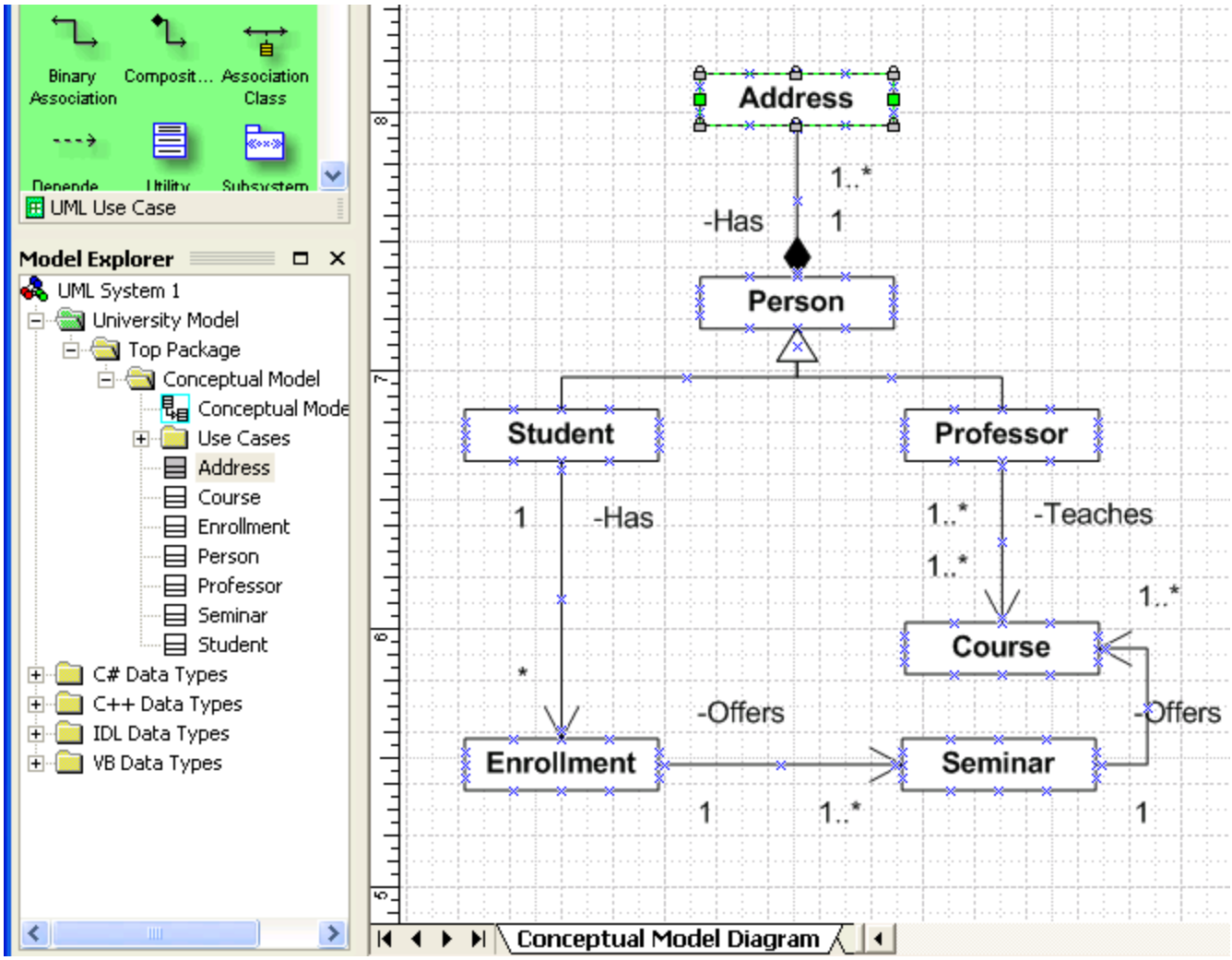

Figure 7. Conceptual model diagram

Note: Remember that the conceptual model is based on a static structure diagram and could be therefore transformed into code by Visio. Because the information contained in the conceptual model is minimal, and you are still trying to understand the problem rather than writing the final solution, the code generated from this model would only be a general blueprint of the real code, with no practical use.

Organizing the Objects in the Diagram Using Layers

When you work with big UML diagrams, it can be very useful to use layers to identify different areas in the diagram and assign which objects in the model belong to which layer. Layers are used to organize related objects on a diagram. By assigning shapes to different layers, you can selectively view, print, color, and lock different categories of shapes. In the university system example in this article, for example, it might be useful to define objects in the conceptual model in two layers:

- Grades layer: Contains classes with grade-related information
- Personal layer: Contains students' personal information

Doing this allows me to work with the grade-related objects in the diagram without having the personal information-related objects interfering.

In Visio, you create layers by going to the menu bar and selecting *Format --> Layer* . If no layers have been defined, a prompt will ask for the name of a new layer. If layers are already defined in the model, then you can create, edit, or delete layers by going to the menu bar and clicking *View --> Layer Properties* . In this window, you can also specify whether a layer is or is not visible. Hiding portions of the model can be useful to clear the drawing area and provide more space to work, or when you are sharing the diagram with someone who is involved in a just a portion of the model and does not need to see the whole thing.

Each layer is identified by a name and a color. To assign a class to a given layer, just right-click it and select *Format --> Layer* . Then select the layer that should contain the class.

Checking for Errors

There will be times when you are working on a given diagram, and suddenly an object turns red and bold which, as you might suspect, is not good. It means that there are one or more errors that relate to that object somewhere in the model or project.

When this happens, you can get information about the error by right-clicking the object that has turned red and selecting *Display Semantic Errors* . Visio then displays up to 20 errors found in the model and a brief description of each one in the *Errors* window on the bottom of the screen.

The description of these errors is very brief, and when you first start working with Visio, it is difficult to determine what the cause of the error is. However, once you have worked with Visio for a bit, you will have a pretty good idea of what is wrong, and these errors are usually easy to fix.

Note: Most times, the errors in a model refer to duplicated classes (classes with the same name) within the same package or to connections between classes that have not been correctly established. In a sense, Visio helps quite a bit to propagate these errors. To completely delete classes from a model, you must delete them from the Model Explorer window by right-clicking the class and selecting Delete , rather than deleting them from the diagram itself. A class deleted only from a diagram is not deleted from the model itself, and it will throw a duplication error if it's re-created again in the diagram with the same name. This problem occurs often when you are working to arrange classes on a diagram and you find yourself changing your mind about whether or not you need a class.

Another common problem that happens is an association giving an error because one or both of its ends are not correctly attached to the object. When connecting an association's end to an object, make sure that the end is pointing to one of the ports in the class (depicted by a pale blue triangle). When the association's end and the triangle are correctly aligned, a red square will appear, indicating that the connection on that end is correct.

Figure 8 shows an example of a class throwing an error, whose description is displayed in the *Output* window at the bottom of the screen and references the fact that a class name must be unique in a namespace. The *Model Explorer* window shows that there are two *Address* classes in the *Conceptual Model* package, which is the cause of the error. Also notice in the diagram the arrow point and the diamond, which represent composition (every person class has an address class) and generalization (students and professors are persons).

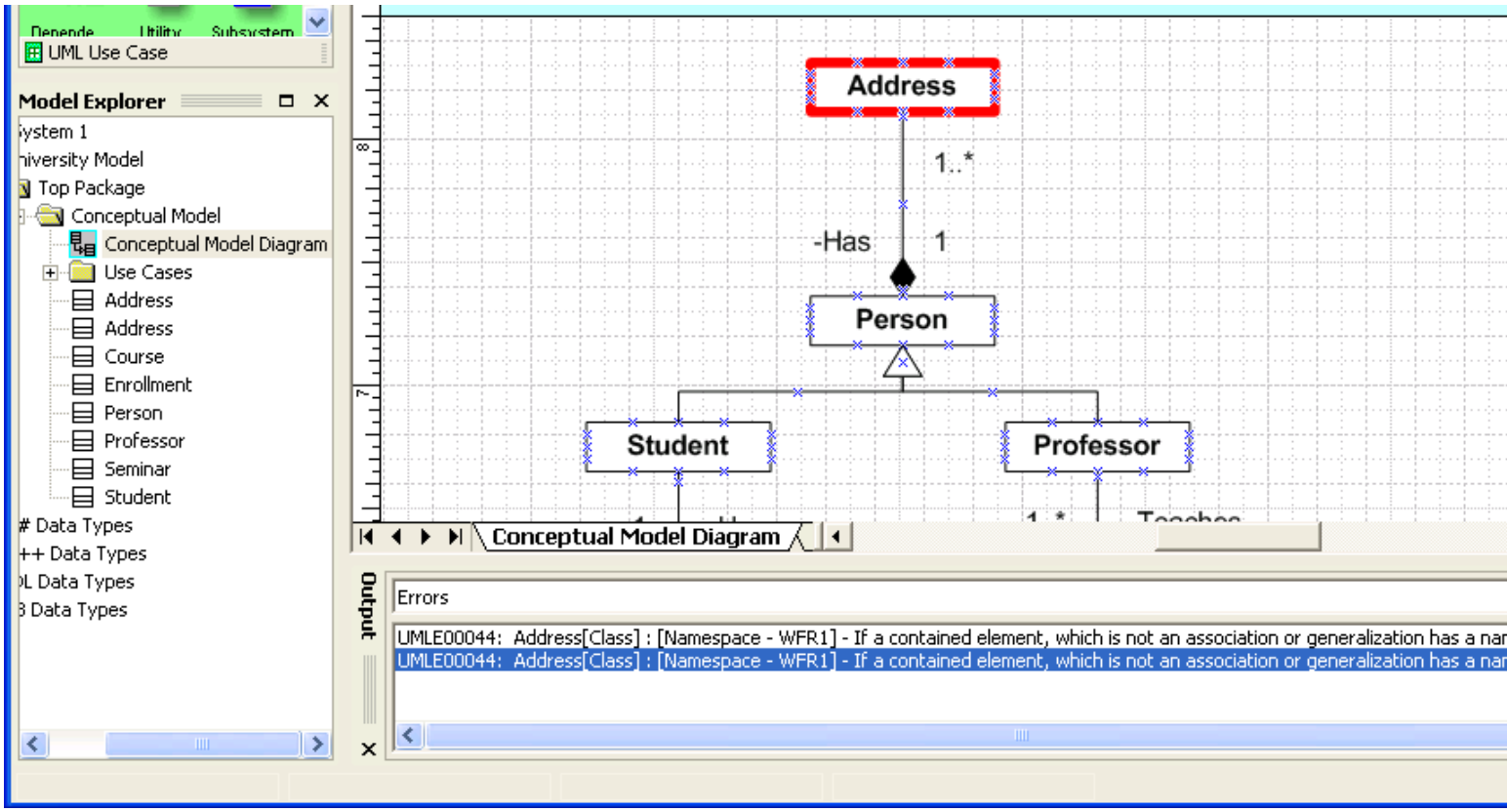

Figure 8. Class throwing an error in a conceptual model diagram

Implementation Diagrams

I'll cover the UML implementation diagrams next. As stated previously, these diagrams are extremely important for the design of the system I'm trying to build, but they cannot be converted into code. I'll cover their different elements briefly so I can identify them in the Visio's stencils, and I'll build simple diagrams so you know what implementation diagrams look like.

Component Diagrams

Component diagrams are physical analogs of static structure diagrams. They are used to describe the dependencies between software components-for example, the dependency between executable files and source files. Also, they often represent architecture-level information, to model either the business software architecture, the technical software architecture, or both. Physical architecture issues are better modeled via UML deployment diagrams, as I will show later.

The main elements in component diagrams are as follows:

- **Nodes** : These represent physical hardware.
- **Components** : These are physical building blocks of the system, represented as a rectangle with tabs. Each component belongs to a node. Examples of components are external files, databases, and applications.
- **Interfaces** : These describe a group of operations used or created by components.
- **Dependencies** : These are used to establish dependencies between components.

To create a component diagram with Visio in the current project, right-click the *Conceptual Model* folder in *Model Explorer* and select *New - - > Component Diagram* . Rename the diagram to something more meaningful.

Returning to the university system example, let's say I want to design a component diagram to model the different components involved when a student logs into the system to check the different seminars offered. Figure 9 displays what the diagram would look like. The diagram contains two components, the *Student Admissions Server* and the client's PC. The *Student Enrollment Application* runs on the *Student Admissions Server* . It allows students access to the StudentDB database through the `IstudentEnrollment` interface, which can be accessed from the client's PC.

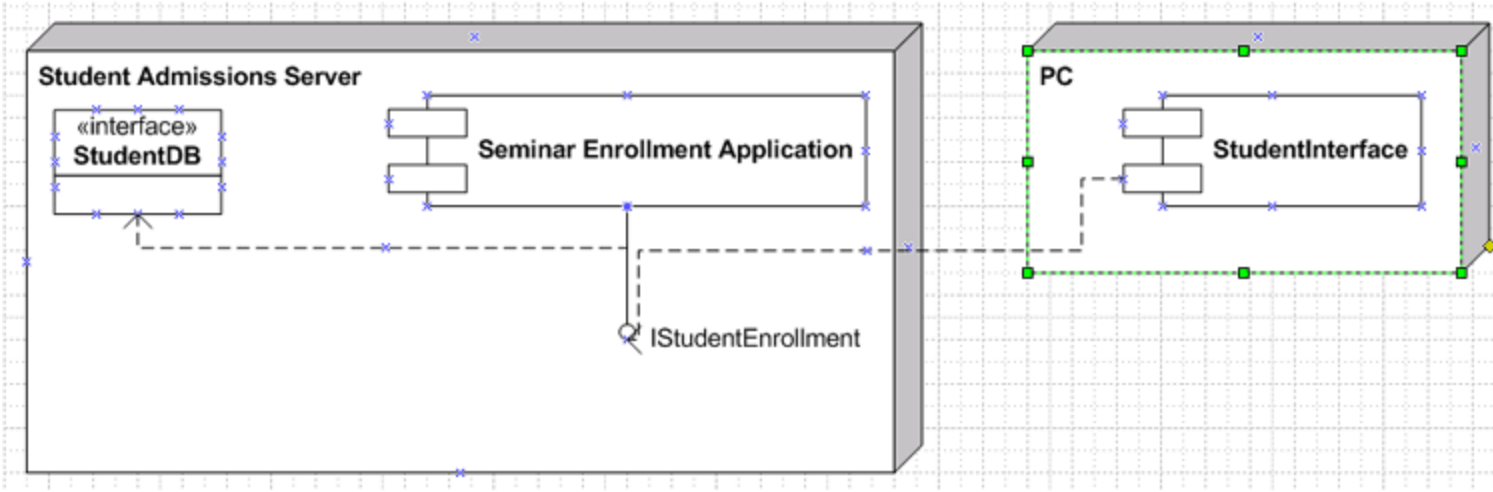

Figure 9. Component diagram to represent interaction between the Students Admissions Server, Seminar Enrollment Application, and client's PC

Deployment Diagrams

Deployment diagrams are a different type of implementation diagram. They show the physical configurations of hardware and software. They represent a static view of your system's hardware, the software that is installed on that hardware, and the middleware used to connect the machines. They are useful for applications that are deployed on several machines or to explore the architecture of an embedded system, by showing how the hardware and software components work together.

Similar to component diagrams, the main elements in deployment diagrams are basically components, interfaces, and dependencies, with the same definitions.

To create a deployment diagram with Visio in the current project, right-click the *Conceptual Model* folder in *Model Explorer* and select *New -- > Deployment Diagram* . Rename the diagram to something more meaningful.

Going back to the university system example, let's say I want to build a deployment diagram to model how the *Student Information Server* and the *Registrar's Server* relate. I would represent the applications running on the *Student Information Server* : the *Grades System* and the *Seminar System* . Then I would connect the *Grades System* to the *Registrar's DB* , to state that these two components need to communicate. The resultant deployment diagram is shown in Figure 10.

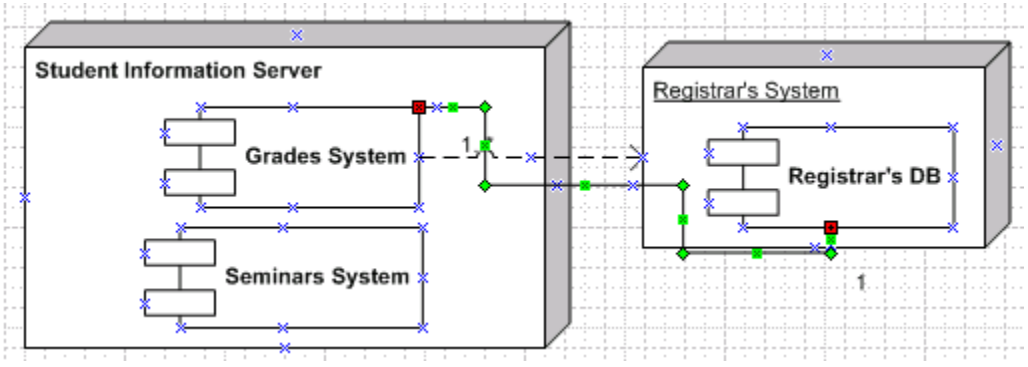

Figure 10. Deployment diagram of the Student Information Server

Conclusion

In this article, I introduced and discussed many of the UML diagrams available when designing an application. I concentrated on building the static diagrams for a sample application domain consisting of a university system. I have shown what a powerful tool Microsoft Visio is in creating these diagrams and have laid the foundation for Part II of this article. In Part II, I will move on to build the dynamic diagrams for the system, showing how to generate code and how to generate documentation from the UML diagrams.

UML Modeling with Visio, Part 2

This is the second article in a two-part series whose aim is to explain the fundamentals of using Microsoft Visio to create UML diagrams for use with Visual Studio .NET. The first article explored UML diagrams and concentrated on the creation of static diagrams. This second article concentrates on dynamic diagram creation, code generation, and documentation generation.

Introduction

The Enterprise Architect version of Microsoft Visual Studio .NET 2003 includes a copy of Microsoft Visio for Enterprise Architects (VEA). For those of you new to the product, Microsoft Visio is a drawing package that allows you to select predrawn shapes from stencils to draw and design different kinds of diagrams such as flowcharts, network diagrams, and software diagrams.

Software application developers can model the application's design and functionality with Visio and Unified Model Language (UML) 2.0 through Visio's UML Model Diagram template. Also, Visio can perform reverse engineering on an implemented system and transform existing code into a UML model.

This two-part article series explains how to model different UML diagrams with Visio and how to transform these diagrams into code in a .NET programming language, thus reducing the work necessary to implement the modeled solution from scratch.

In this second article, I start by exploring dynamic diagram creation within UML. I create several different types of dynamic diagrams, and then use these diagrams to show how to generate .NET code. Next, I generate some useful documentation from the diagrams, and finally, I cover code templates within Microsoft Visio.

This is the second article of the series. You can view the previous article at http://www.asptoday.com/Content.aspx?id=2332

System Requirements

To work with the solution in this sample, you should have

- Visual Studio .NET 2003 Enterprise Architect (which includes VEA)

or

- Visual Studio .NET and Microsoft Visio 2000 or later

Compiling and Installing The Sample Code

All the diagrams developed in this article are available for download from the ASP Today site. The download contains the file *UniversityModel.vsd* , which contains all the diagrams in Microsoft Visio format.

UML Design: Dynamic Diagrams

Let's start off by looking at a group of UML diagrams called the **UML design diagrams** . Diagrams in this group include the class diagram, which is based on static structure diagrams and can be used to generate a good portion of the code in a .NET project. The class diagram is similar to the conceptual model, but it contains a much higher level of detail. This level of detail converts the class diagram into one of the most complex UML diagrams.

When a class diagram is completed, Visio can transform it into code in the .NET language of your choice. This code can be used as the blueprint of a Visual Studio .NET project.

Let's look at the class diagram in detail.

The Class Diagram

I will create the class diagram in a new package, similarly to when we had the *Conceptual Model* package, in Part I of this article, to save the conceptual model in. In the university system example, I called this new package *Physical Model* .

Note: You need to give some thought to the way in which you want to organize the different diagrams and classes into packages. Remember that the names given to the packages in the Model Explorer window will have a direct impact on the code generated from the final model, as they will define namespaces names.

Similar to the conceptual model diagram, I first need to create a new static structure diagram to define the class diagram. I have two options regarding what objects I include in the class diagram:

- I can reuse the objects previously defined in the conceptual model diagram. I can just click and drag the objects defined from the conceptual model into the physical model class diagram and modify them accordingly.
- I can re-create the objects from the conceptual model, which will add new classes to the model rather than use the existing ones in the conceptual model.

The option you choose will depend on your preference and/or your situation regarding the development team and the size of the project you are modeling. I personally tend to prefer the second option (create new objects for the class diagram) because I think it is cleaner and because what I do in one model doesn't affect the other, although this can also backfire if you are not careful.

I will follow the second approach and re-create all the objects in the conceptual model in the class diagram. This approach doesn't throw errors, because the objects are placed in different packages. Next, I define the .NET language that the model should produce (for code generation), and I select the packages that will generate the code. On the Visio menu bar, go to *UML --> Code --> Preferences* and select the language to generate code in, as shown in Figure 1.

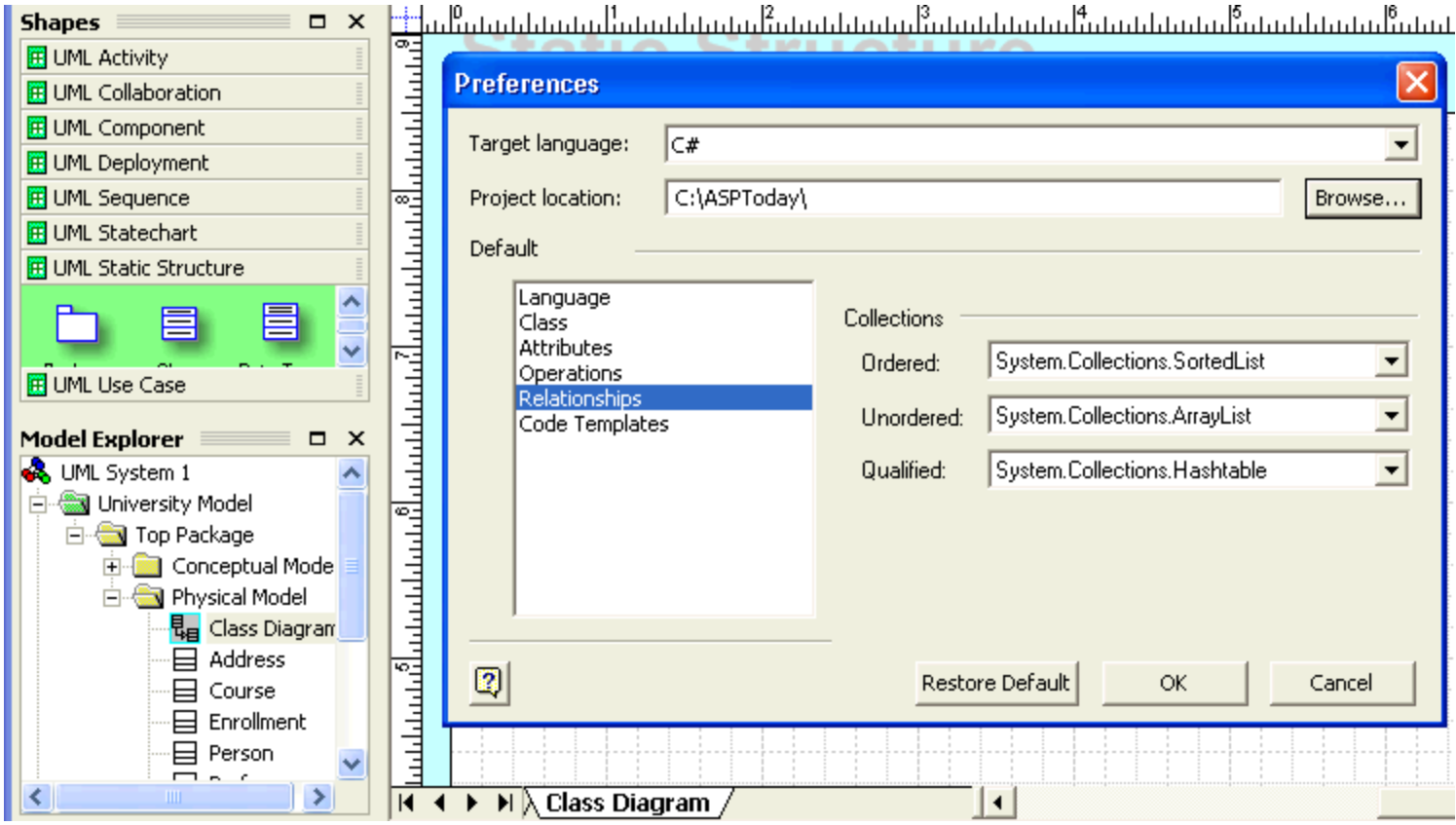

Figure 1. Visio code Preferences window

This *Preferences* window also gives you the possibility to customize other preferences, such as the type of collections to be generated or the code template to apply (discussed later in the article).

The language to generate the code is also accessible through the Visio menu bar by going to *UML --> Code --> Generate* . Here, you can also select the packages to transform into code.

Once you select the language to generate the code in, you won't need to work with other languages' data types. You can remove these data types from the drop-down menus by going to the menu bar and choosing *UML --> Options*. Select *UML Document* and uncheck all languages and packages that the code will not use or reference.

For the university system example, I chose *C#*, checked only the *Physical Model* package to generate code, and unchecked all data types and packages except *C#* and *Physical Model*.

Defining Classes

Objects in the class diagram have a lot of detail because they implement all attributes, operations, and methods necessary to implement in the real-world project. To add an attribute or operation to a class, double-click it to bring up the *UML Class Properties* window, as shown Figure 2.

Defining Attributes, Operations, and Their Properties

Attributes, operations, and their different properties (type, visibility, etc.) are defined in the *UML Class Properties* window. To add new attributes or operations, just click the *Attributes* or *Operations* node on the tree on the left and start entering them.

For both attributes and operations, data types under the *Types* drop-down menu are limited to the types selected earlier in the menu bar via *UML --> Options --> UML Document*. By default, and for obvious reasons, Visio sets the attributes' *Visibility* to `private` and the operations' *Visibility* to `public`, but you can change this if necessary just by clicking on it.

- `private` is depicted with a minus sign (-). It means that the attribute or operation is not accessible to anything outside the class.
- `protected` is depicted with a dollar sign ($). It means that the attribute or operation is accessible to classes that inherit from the class defined only.
- `public` is depicted with a plus sign (+). It means that the attribute or operation can be accessed by other classes.
-

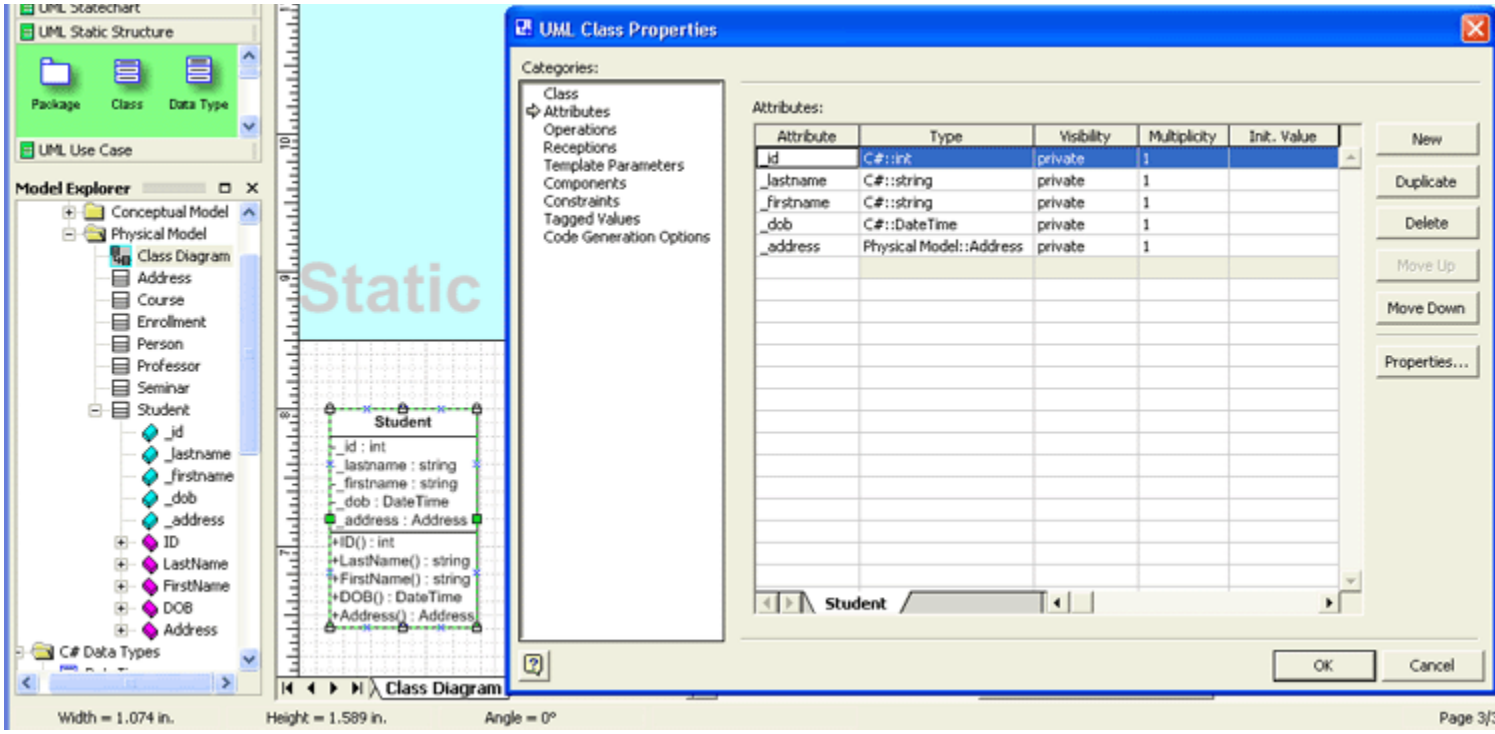

Figure 2. Student Class Properties window

Attributes

To access the properties of an attribute, simply select the attribute and click the *Properties...* button. This brings up the *UML Attributes Properties* window, as shown in Figure 3. Here, you can access and modify different attributes properties such as *Type*, *Visibility*, *Multiplicity*, *OwnerScope*, and *TargetScope*, and you can also set the attribute's initial value.

You have already seen what visibility and type mean in the attribute's definition. Here are some of the other properties:

- **Multiplicity:** Defines a single item or a collection.
- **Changeable:** Defines whether or not the attribute is a constant.
- **OwnerScope and TargetScope:** Allow the attribute to be declared as a classifier or an instance. In C#, an attribute ser as classifier will be declared as `static` .

The *Code Generation Options* option on the left menu bar allows you to preview the code generated by the attribute in different languages, as well as the application of code templates.

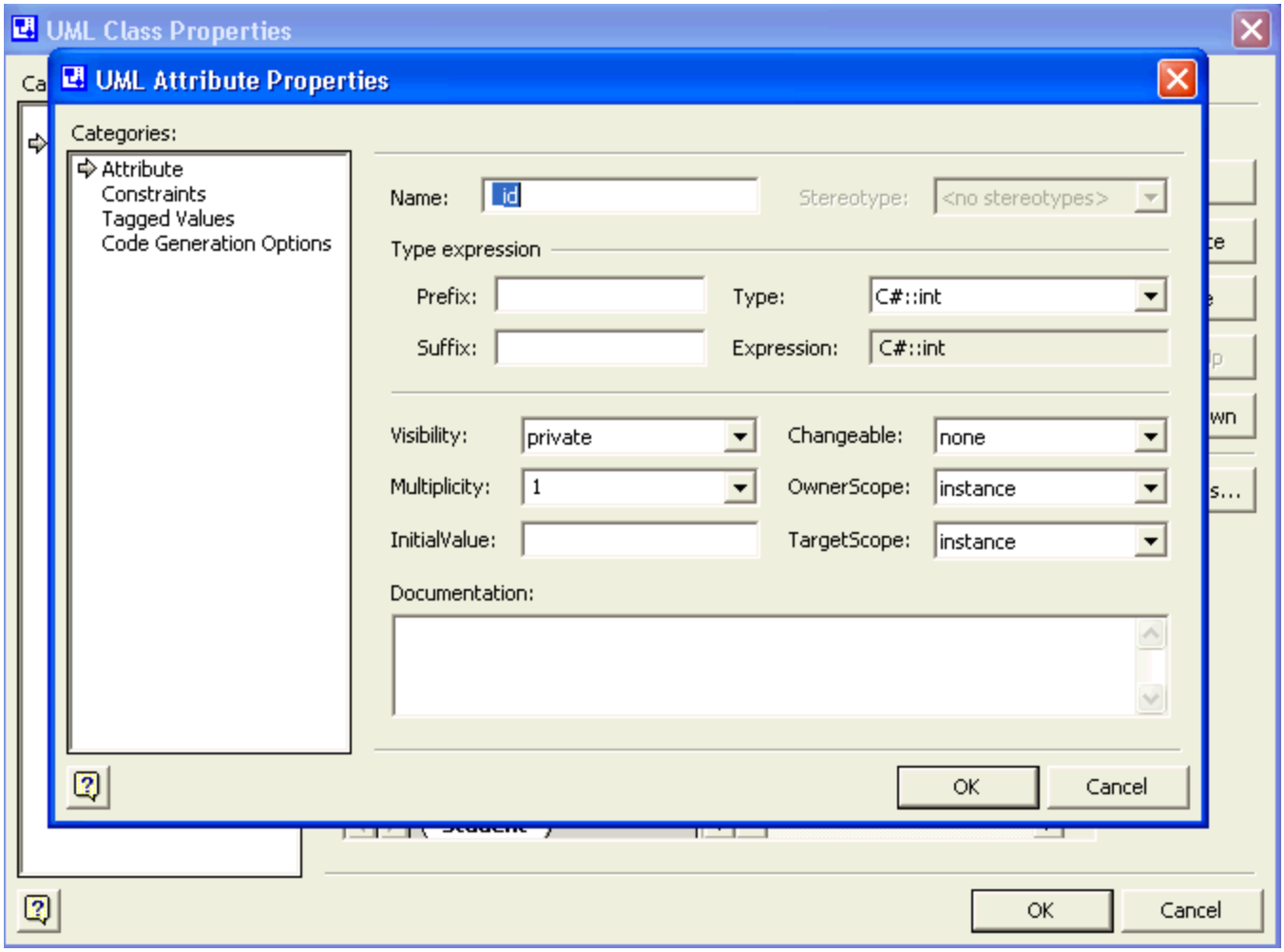

Figure 3. Class Attribute Properties window

Operations

- Classes in C# have the following different kinds of operations: `Constructor` , `Destructor` , `Event` , `Indexer` , `Operator` , `Procedure` , and `Property` .

Some notes regarding these types:

- **Constructor and Destructor:** If a procedure is defined as `Constructor` or `Destructor` , its name when the code is generated will match the class's name.
- **Event and Property:** When a procedure is defined as `Event` or `Property` , the *Create Get Method* and *Create Set Method* check boxes become active for adding get and set assessors.
- **Operator:** Defines an operator for operator overloading.

Note: Classes generated with VB .NET don't have the `Indexer` *and* `Operator` *operations.*

By default, Visio declares new operations as `Procedure`s, but we can change the type of an operation by selecting the operation and clicking *Properties* . This brings up the *UML Operations Properties* window, as shown in Figure 4.gif>, where you can do several things:

- You can set the operation's scope (classifier versus instance) and type by clicking *Operation* on the left menu bar.
- You can enter any code that the operation needs to execute by clicking *Method* on the left menu bar and entering the code in the *Method body* text box. Note that in order to type into this box, you must select the *Has Method* check box above it. The *Method body* text box is not exactly user-friendly; it is a rather simple text editor similar to Notepad that won't alert you about errors in the code you are entering. If the code you need to enter here is long or complex, you can type it first in Visual Studio, taking advantage of IntelliSense, and then copy and paste it into Visio. When you enter code in the *Method body* text box, the model becomes dependent on the programming language, since this code won't change if you finally decide to generate your model in a different language. Code added to method bodies will be inserted into generated code files without modification, regardless of the language chosen for generation. Doing this is strongly discouraged.
- If the operation being declared requires parameters, click *Parameter* on the left menu to define the operation's input and output parameters' names, type, kind (in, out, or inout), and default value.
- To change the operation type, click *Code Generation Options* on the left menu bar, and select the right option from the *Kind* drop-down menu, which displays *Procedure* by default.
- As with attributes, you can preview the code generated by an operation by selecting the language in the *Target language* drop-down list and clicking the *Preview code* button.
-

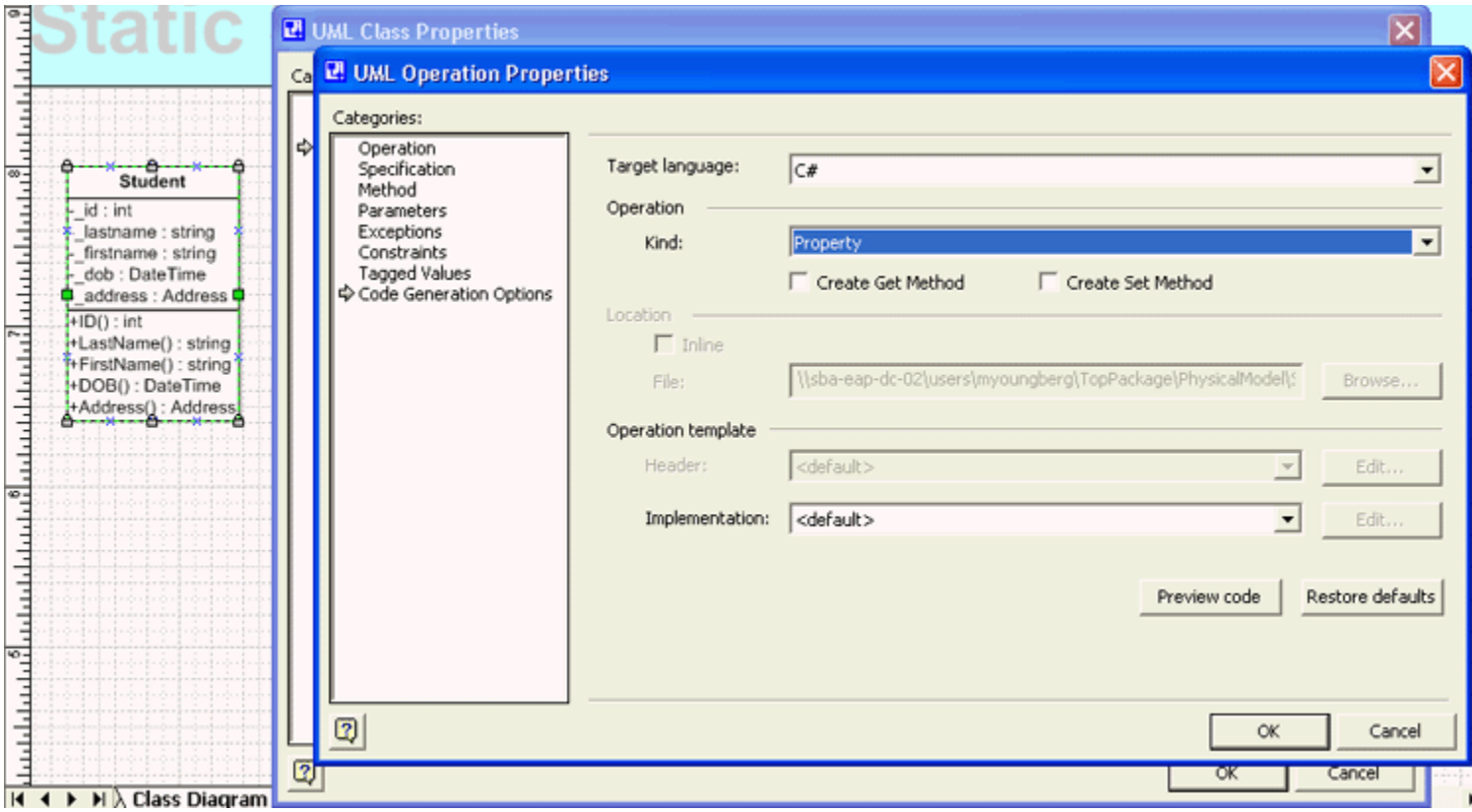

Figure 4. Class Operation Properties window

Inheritance, Abstract Classes, and Interfaces

Looking at the conceptual model of the university system that I designed in Part I of this article series, you can see that the `Student` and `Professor` classes have common information such as last name, first name, date of birth, and address. Based on the requirements, I could implement an interface or an abstract class called `Person`.

Defining Abstract Classes

To define an abstract class, I simply click and drag a class from the *UML Static Structure* stencil. I double-click it to access the *UML Class Properties* window, give it a name (in this example I call it *Person*), and check the *IsAbstract* check box. Because the class has been defined as abstract, its name will be displayed in italics in the class diagram, as shown Figure 5.

To establish the inheritance between `Student` and `Professor` and the `Person` class, I connect the classes with a *Generalization* element (the line with an open-ended arrow) from the *UML Static Structure* stencil. This is called a

generalization association , and it is used to show relationships (one-to-one, one-to-many, etc.) between model entities.

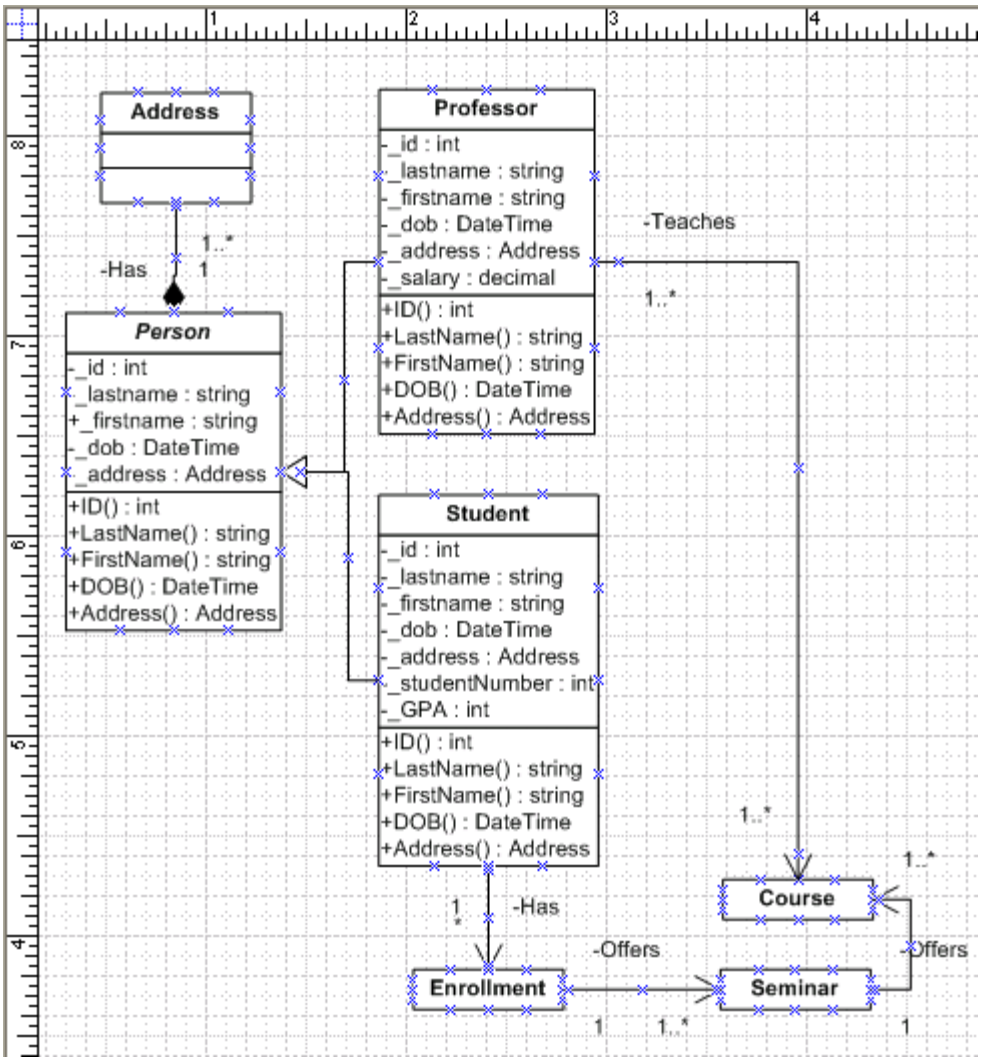

Figure 5. University system class diagram

Figure 6. Shows how the code preview of the `Student` class establishes that the `Student` class inherits from the `Person` class.

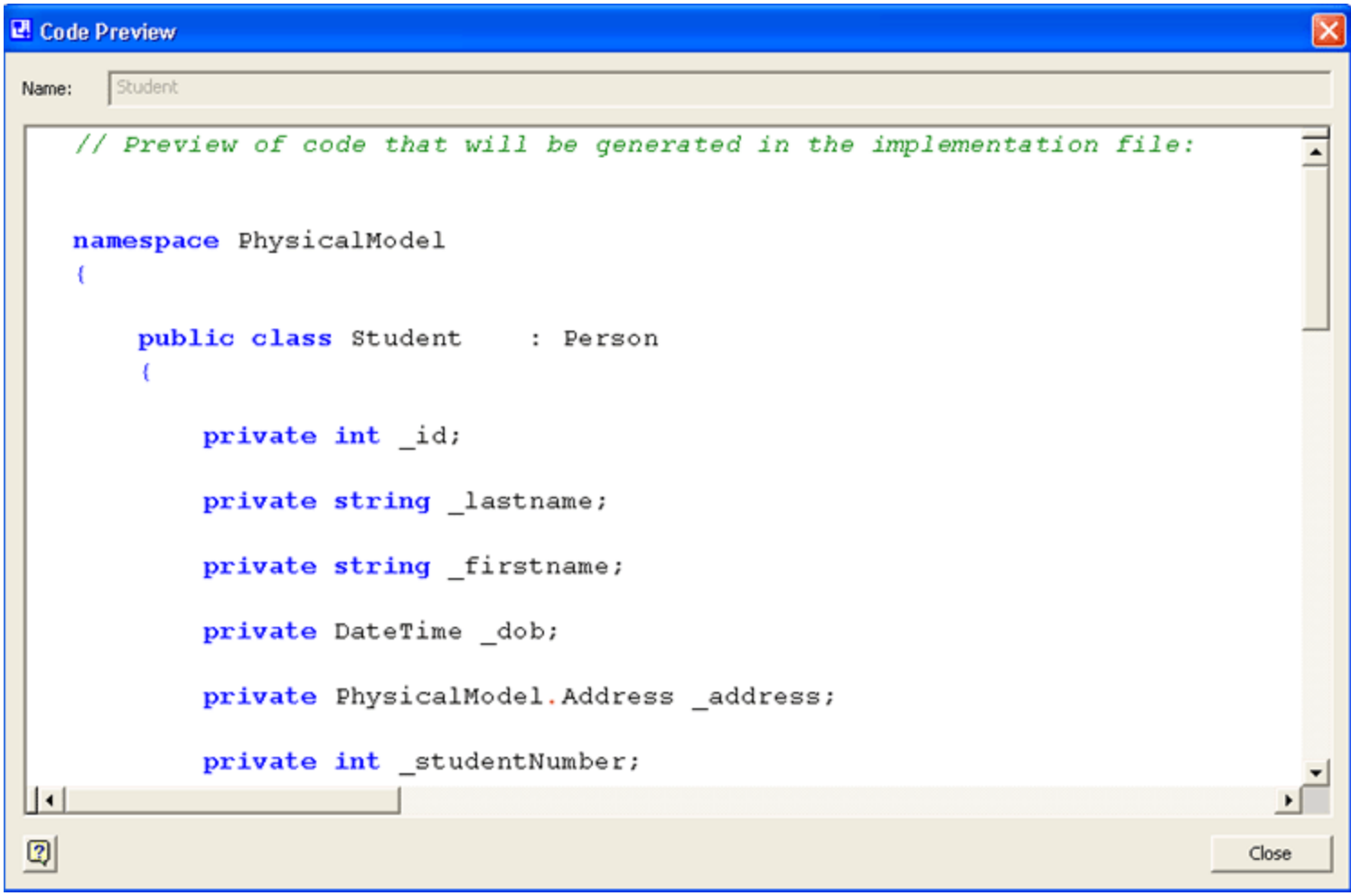

Figure 6. Class Diagram Code Preview window

Defining Interfaces

Another possibility when using inheritance is to declare an interface that different classes will inherit from. To define an interface, click and drag an *Interface* object from the *UML Static Structure* stencil. This stencil has two kinds of interfaces, as shown in Figure 7. The shape to use is completely up to you, although I personally like the one that looks like a class because I can see its different operations. The interface can be changed from the class shape to the lollipop shape by right-clicking it and selecting the shape to display. Another way to select the shape to use to display interfaces is by going to the menu bar, choosing *UML --> Options* , and selecting the desired option (i.e., lollipop versus classlike shape). The shape to use is up to you, although the classlike shape is more standard and strongly encouraged.

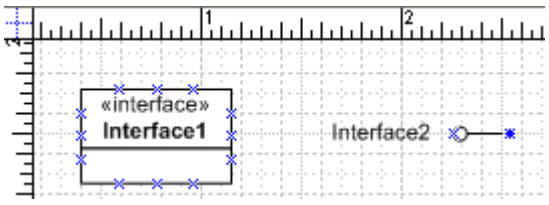

Figure 7. Different representations of interfaces

Because it is different from abstract classes, and because interfaces contain only the declaration of operations and no attributes, the interface in the class diagram allows for only the declaration of operations.

- Only operation names and data types can be changed. Operations are declared as `Public` and `Polymorphic` by default, and this cannot be changed.
- Because interfaces have no implementation, no method can be entered for these operations declared in an interface.

To establish inheritance between an interface and a class, you connect them with a *Dependency* arrow from the *UML Static Structure* stencil, as shown in Figure 8.

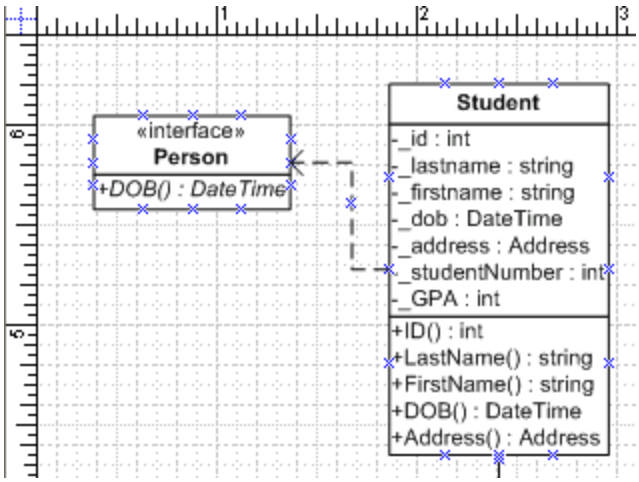

Figure 8. Inheritance established between the Person interface and the Student class

Events and Delegates

To define a delegate class, just right-click the class and select *delegate* in the *Stereotype* drop-down menu. Delegate classes implement a single operator.

Let's see an example of how to create events. Let's go back to the university model example in this article and perform the following steps:

1. Create a new delegate class called *UpdatedAddressDelegate* . To set the class as `delegate` , just right-click it and select *delegate* in the *Stereotype* drop-down menu.
2. Define a new operation called *Updated* .

Now, I need to create an event that links to the delegate class. To do this in the university model, follow these steps:

1. Right-click the *Address* class and add a new operation called *Updated* .
2. Set the return type of the *Updated* operation to *UpdatedAddressDelegate* .
3. Go to the operations properties and declare *Updated* as an *Event* .

A preview of the `Address` class code is shown in Figure 9.

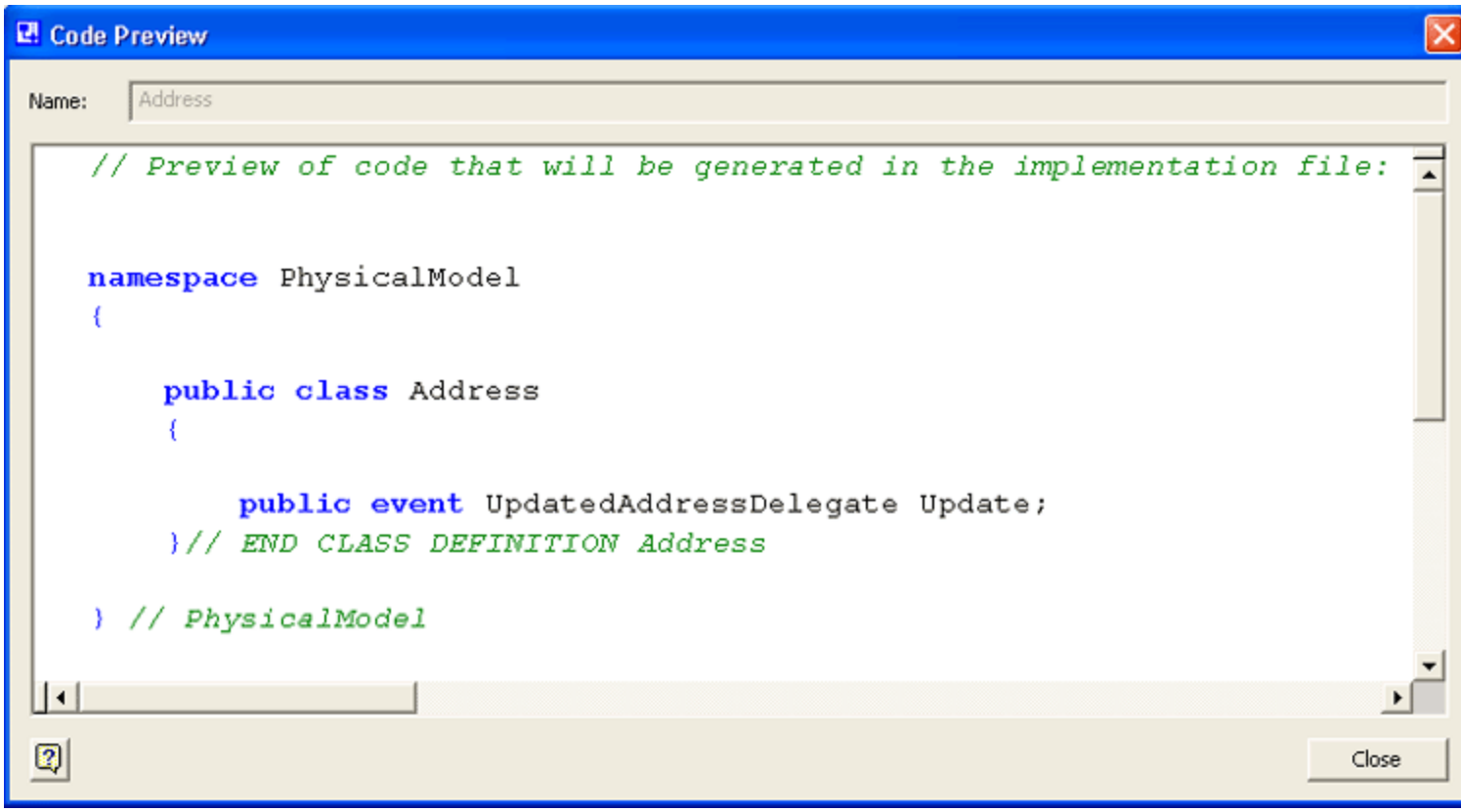

Figure 9. Code preview window of the Address class

Multiplicity and Collections

To include collections in the class diagram, I first need to add an `ArrayList` data type to the C# data types in *Model Explorer* (similar to when I added a `DateTime` data type). Next, I create a new class called `Students`, which will be a collection of `Student` objects that will have an indexer that returns a particular student given an ID number.

All collections have a key attribute and a key method: the collection itself and an indexer. Let's see how to implement them through Visio:

- Create a new attribute called *_students*. Modifying its multiplicity to *0..** will automatically generate a collection type member. Checking the code generated now through the *Code Generation Options* will display how this attribute will be automatically coded as an `ArrayList`: private System.Collections.ArrayList _students;
- To create an indexer, add a new operation and call it *Indexer*. From the *Parameters* category, add a new parameter of type C#::int called *index*. From the *Code Generation Options* for this operation, set *Kind* as *Indexer*. This will enable two check boxes, the *get* and *set* assessors. Check the assessors that you will be implementing. Checking the *Has Method* check box will allow you to enter specific code for the indexer.

Generating the Code

Once the class diagram is completed, Visio can transform it into code automatically. To generate the code, go to the menu bar and choose *UML --> Code --> Generate*. Figure 10.gif> shows that you can select the target language from the *Target language* drop-down menu and select the different modules, specific classes, or interfaces that you want to export. Any language choices made before at the class level will be overridden by the option chosen from the *UML --> Code --> Generate* menu.

Note: The language chosen in the Target Language field determines the language for each class, attribute, and operation involved in the model.

You also can specify whether the code will be saved to a file or to a Visual Studio .NET project by checking the *Add Classes to Visual Studio Project* check box. Let's look at the latter possibility in more detail in the following section.

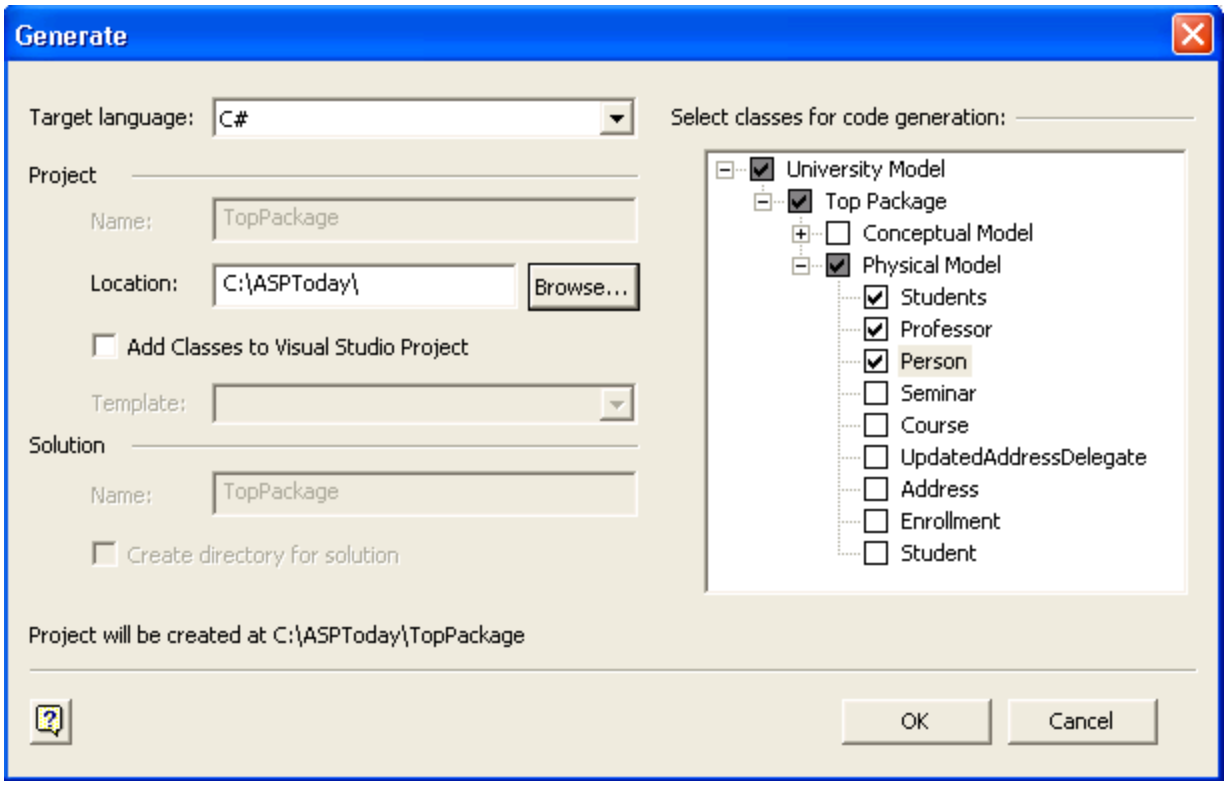

Figure 10. Generate code menu window

When you check the *Add Classes to Visual Studio Project* option, all possible types of Visual Studio .NET projects are listed in the *Template* drop-down menu. When working with web projects (Empty Web Application, ASP.NET Web Application, or ASP.NET Web Service), you have to specify a valid URL for the project rather than a file name, just as is required in Visual Studio .NET. Visio will generate code for each selected class and will place it into a separate file with the name of the class followed by the extension for that language (e.g., *Student.cs*).

You will usually generate your code, go back and refine your Visio model, and then generate the code again. When you do this, the existing files in your Visual Studio .NET project are not overwritten. Any existing files that would be overwritten are renamed to *<classname>~1* , *<classname>~2, ..., <classname>~n* followed by the language extension (e.g., *Student~1.cs*), and the new code is always saved as *<classname>* with the language extension (e.g., *Student.cs*).

Interaction Diagrams

There are two types of interaction diagrams: the sequence diagram and the communication diagram. The next sections cover these diagram types in detail.

The Sequence Diagram

Sequence diagrams describe how operations are carried out -- what messages are sent and when, while communication diagrams focus on object roles instead of the times that messages are sent. Sequence diagrams are organized according to time. The time progresses as you go down the page. The objects involved in the operation are listed from left to right according to when they take part in the message sequence. In sequence diagrams, object roles are the vertices and messages are the connecting links.

Along with static structure (class) diagrams, sequence diagrams are considered the most important models in application development. They are commonly used to model:

- Usage scenarios (potential ways the system is used)
- The logic of methods and/or services

Basic elements in sequence diagrams are:

- **Object Lifeline -** Represents an instance of a class previously defined in the Conceptual or Physical model and its lifespan in the scenario being modeled.
- **Activation -** Represents an active instance of the target class, indicating that processing is being performed by the target object/class to fulfill a message.
- **Message -** Messages are arrows that represent communication between objects. There are different types of messages:
- Asynchronous messages, represented with half-arrowed lines. They are sent from an object that will not wait for a response from the receiver before continuing its tasks.
- Synchronous messages, represented with full-arrowed lines. Initial messages are depicted with a solid line and return messages with a dotted one. Balking (messages created and returned by the same object) are represented as a circular message or a circular return message.

Figure 11. displays a sequence diagram that represents the process to update a student's address. It represents how an instance of the `Student` class (called `theStudent`) communicates with an instance of the `Address` class to retrieve the address associated with `theStudent` instance. If this address needs to be updated, the instance of the `Student` class sends a message to the `Address` class requesting the change along with the new value. Once the address has been updated, the `Address` class returns the new value for the `Address` instance.

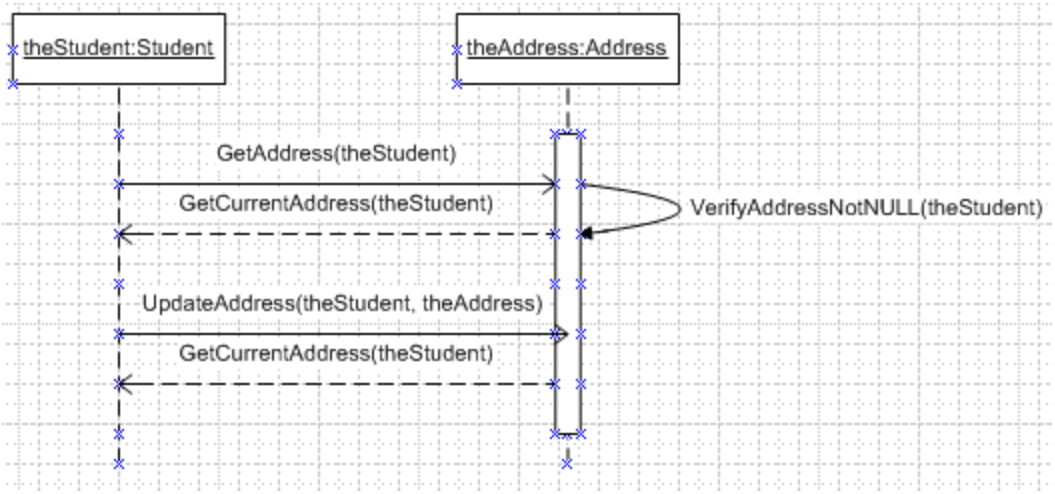

Figure 11. Sequence diagram

When dragging *Object Lifeline* elements into a Visio sequence diagram, you need to right-click it to assign it to a previously class defined. Previously declared classes are accessed at the *Classifier* drop-down menu.

In the example in Figure 11.gif>, *theStudent* object is an instance of the `Student` class defined in the physical model of the project, and *theAddress* object is an instance of the `Address` class defined in the physical model.

The Communication (Collaboration) Diagram

Communication diagrams represent the same information as sequence diagrams, but they focus on object roles instead of the times that messages are sent. Formerly known as collaboration diagrams in UML 1.x, communication diagrams show the message flow between objects in an object-oriented (OO) application and imply the basic associations between classes.

Note: Communication diagrams are still referenced in some versions of Visio as collaboration diagrams.

Communication diagrams show the message flow between objects and imply the basic associations between classes.

The basic elements in communication diagrams are as follows:

- **Classifier roles:** A classifier role is a specific role played by a participant in collaboration. It specifies a restricted view of a classifier, defined by what is required in the collaboration.
- **Multi-objects:** Represented as rectangles, multi-objects represent the various objects involved that make up the application.
- **Association roles:** Association roles represent the relationships between the objects. The same notation for classes and objects used on UML sequence diagrams is used on UML communication diagrams.

In Visio, accessing a classifier role's properties allows you to specify messages, which are depicted as a labeled arrow that indicates the direction, using a notation similar to that used on sequence diagrams. Multi-objects are based on classifiers (classes) defined in the model. When creating a new multi-object in the communication diagram, you need to right-click it to access the properties and assign the proper classifier. Similarly, association roles must be assigned to base associations. Otherwise, an error will be produced similar to the one in Figure 12.

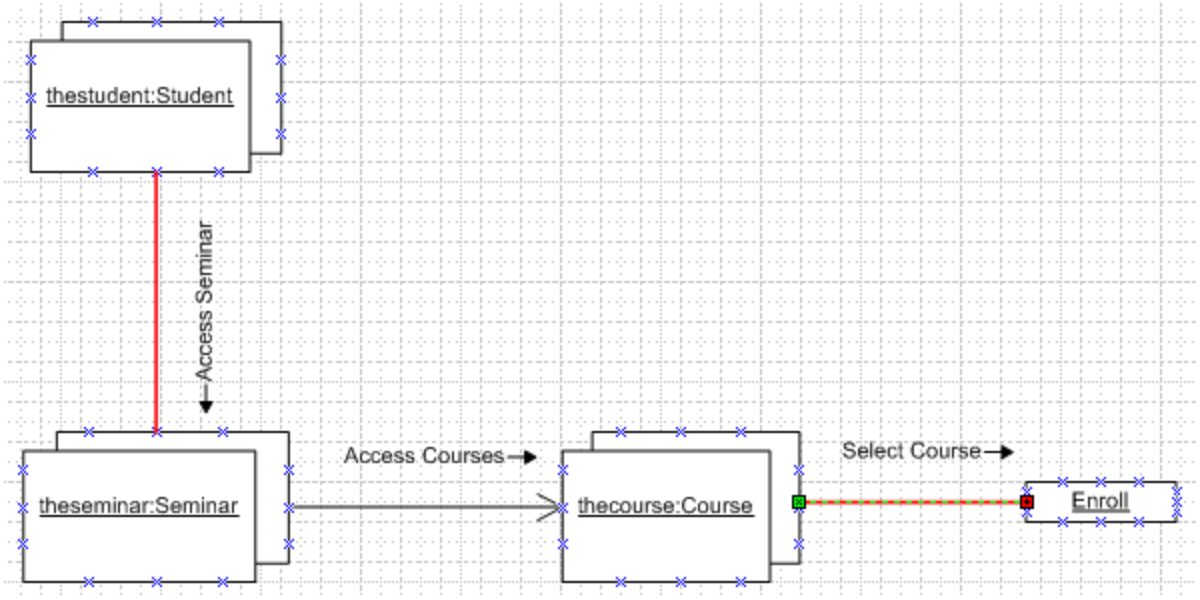

Figure 12. Communication diagram

State Diagrams

Objects have behaviors and state. The state of an object depends on its current activity or condition. Some objects are so complex that understanding them can be difficult. To better understand complex objects-especially those that act in different ways depending on their state-you should develop one or more UML state diagrams. State diagrams show the possible states of the object and the transitions that cause a change in state.

Note: Known as state diagrams in UML 2.0, these diagrams were referred to as state machine diagrams or statechart diagrams in previous versions of UML (UML 1.x).

Let's think of a state diagram in the university model to model the login function of an online student information system. Logging in consists of entering a valid username and password, and submitting the information for validation. Logging in can be factored into three states: getting the username and password, validating, and rejecting. The action that occurs as a result of an event or condition is expressed in the form action. While in its validating state, the object does not wait for an outside event to trigger a transition. Instead, it performs an activity. The result of that activity determines its subsequent state.

The basic elements in state diagrams are as follows:

- **Transitions:** Represented as arrows, transitions are progressions from one state to another and the result of the invocation of a method that causes a change in state. Understanding that not all method invocations will result in transitions is important. When declaring transitions, it is mandatory to indicate the event that causes the transition. The notation for the labels on transitions is in the format event [guard][method list].
- **Guards** : These are conditions that must be true for the transition to be triggered and are optionally indicated.
- **Method list** : This invocation of methods is also optional. The order in the listing implies the order in which the methods are invoked.
- **States:** Represented as rounded rectangles, states represent the values of the attributes of an object. The notation used is the same as that used on transitions, with only difference that the method list is mandatory and the event is optional. There are two special states:
- **Initial state:** Represented as a solid circle. It marks where the action starts.
- **Final State:** Represented as a solid circle surrounded by a line. It marks where the action terminates.

In Visio, to add a guard specification, just right-click the transition and click *Properties* . Guards can be entered in code, pseudo code, OCL, or text. The *Transition Properties* window is shown in Figure 13.

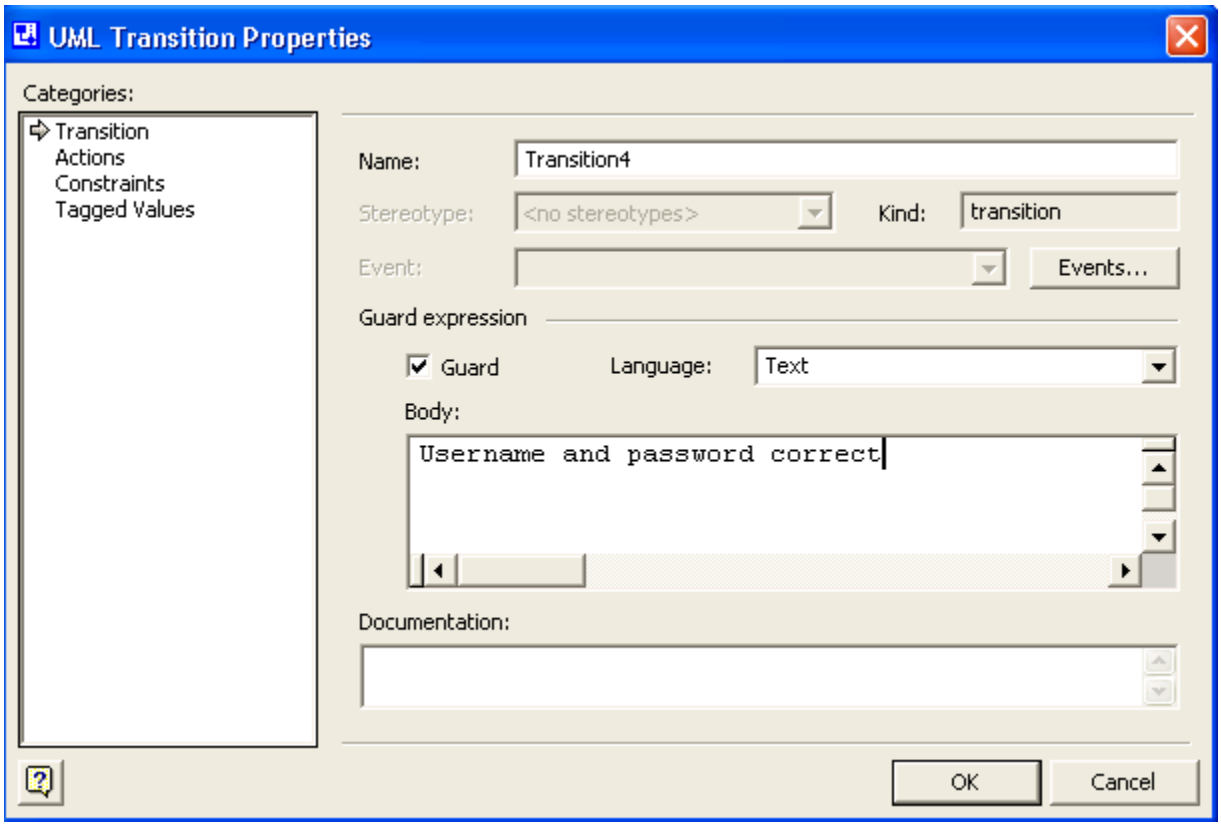

Figure 13. Transition Properties window

Figure 14. shows the resulting state diagram for the student system information login functionality.

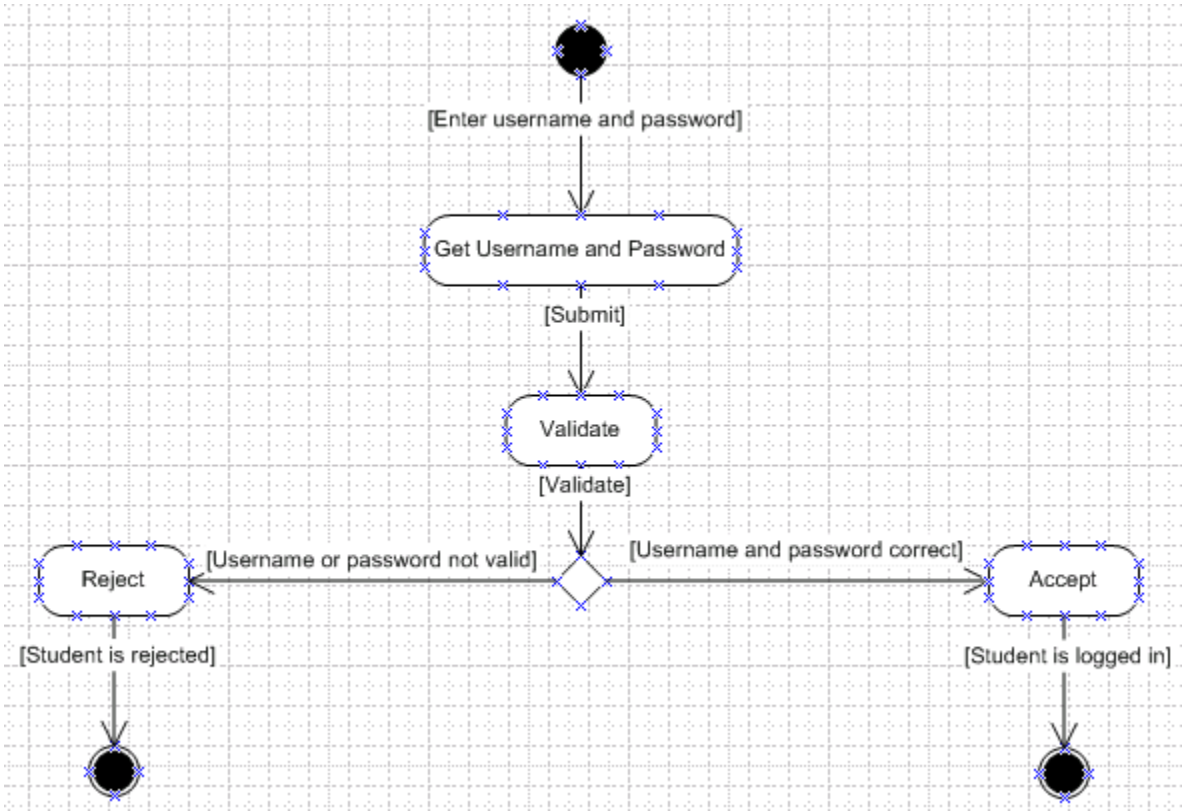

Figure 14. Transition diagram

Generating Documentation

Aside from code generation, Visio offers the possibility of documenting different diagrams in RTF (Rich Text Format) documents. To do this, go to the menu bar and select *UML --> Reports* , which will bring up the dialog box shown in Figure 15. On the *Report Types* tab, select the report to generate, which can then preview, export

into RTF format, or directly print. Clicking the *Customize...* button allows you to select the categories and operations that you want to document.

Other tabs such as *Title* and *Detail* let you customize the reports format and the amount of information to display.

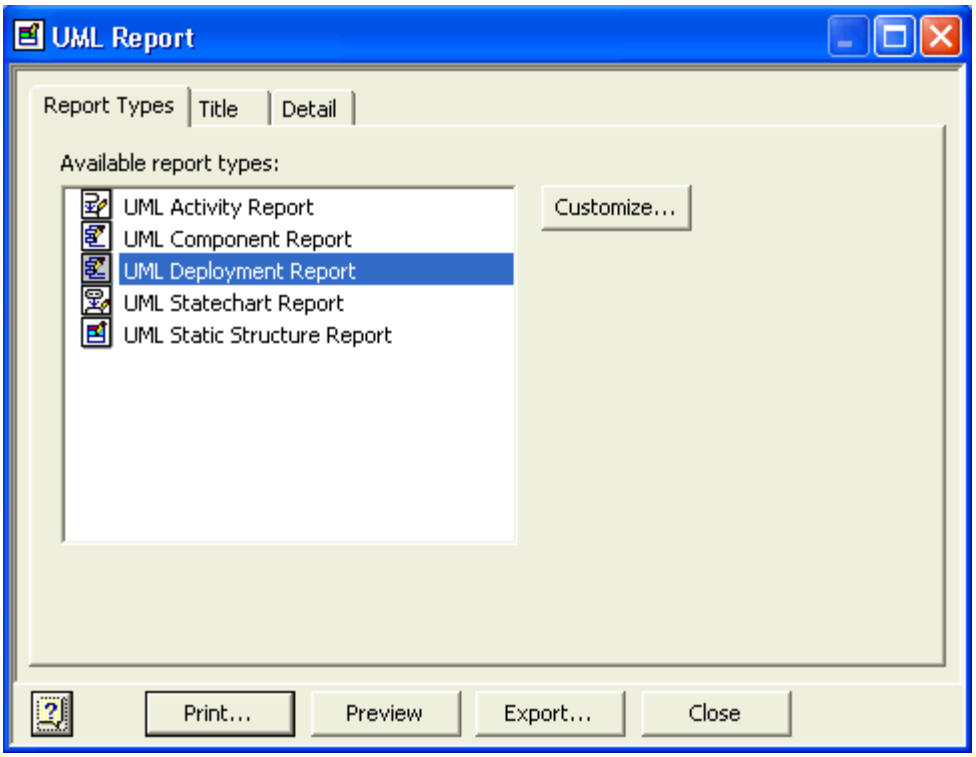

Figure 15. UML Report window

Figure 16.gif> shows the details of the *UML Deployment Report* generated from the university project I have been developing throughout this article.

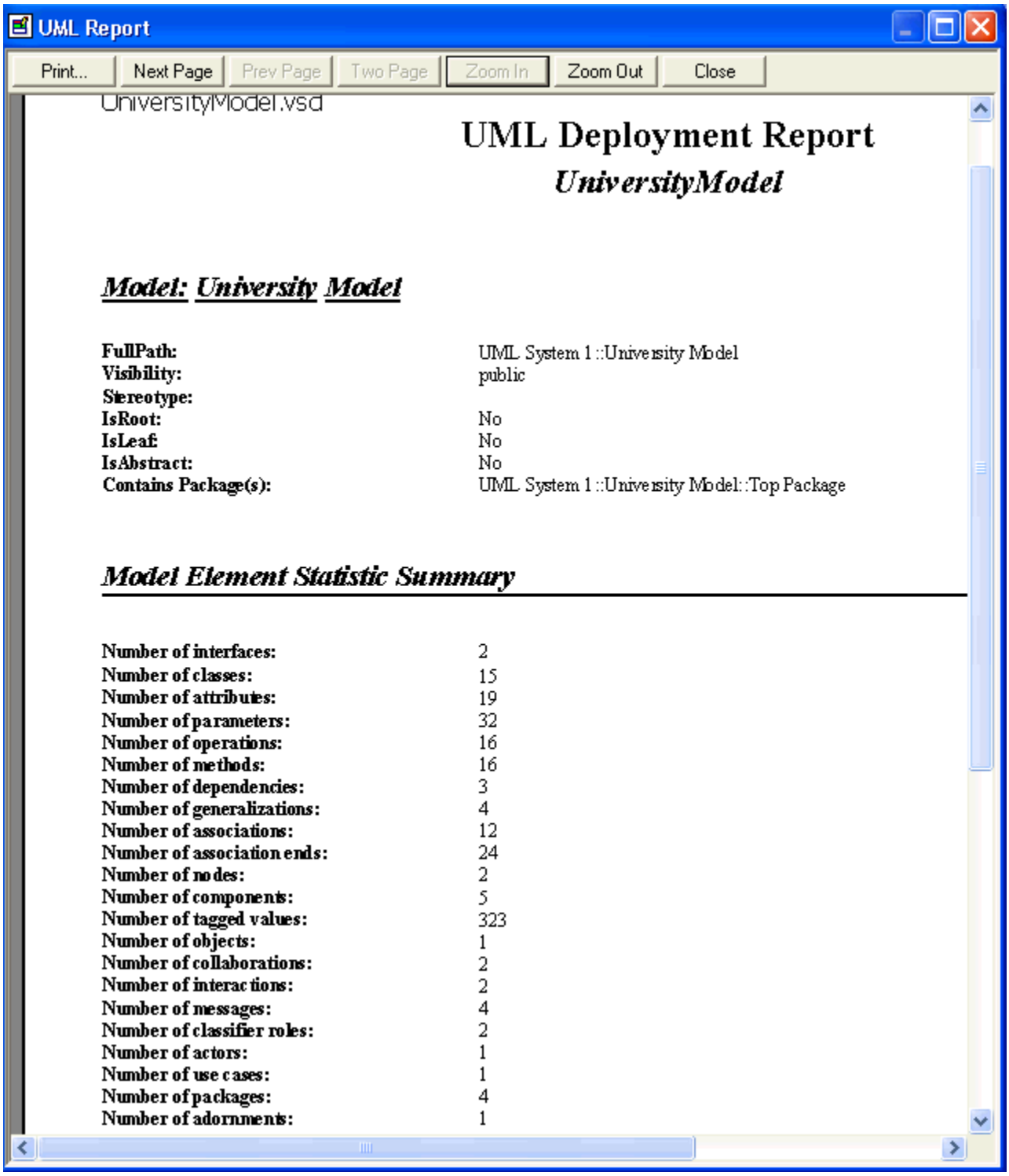

Figure 16. UML deployment Report preview window

Structuring the Code Using Templates

Visio generates code from the UML diagrams using **code templates**. Code templates give designers more control over the structure of the generated code, as they are able to specify the structure of the code.

By default, Visio uses a default template if one is not specified. When working in C#, you can define specific templates for classes, attributes, operations, and relationships. The biggest benefit of generating code using templates is standardization. With templates, you can enforce code guidelines to ensure that consistent code is developed throughout the system. This can be especially useful to enforce guidelines when you are working with big development teams. Also, if you need to make any changes in the template you are using (such as the system name or version), you need only update the central template file that Visio is using rather than all of the individual code files in Visual Studio.

Working with Existing Templates

To open a template in Visio, go to *Menu Bar --> Code --> Preferences*. In this window, if you click *Code Templates* on the menu on the left, you can see default templates defined to be applied to classes, attributes, properties, and relationships. In this window, you can edit, delete, duplicate, and create templates.

Templates contain **keywords** enclosed by percent symbols (%). These keywords act as placeholders where Visio will put the corresponding information that cannot be changed. The names given to these keywords are fairly explanatory of the code that Visio will replace them with: `class_components`, `visibility`, and so on.

By default when writing a class, Visio will place inner types first; then the public fields, public methods, and protected fields; and finally the private methods.

Figure 17. displays the default template used by Visio to write a class's implementation.

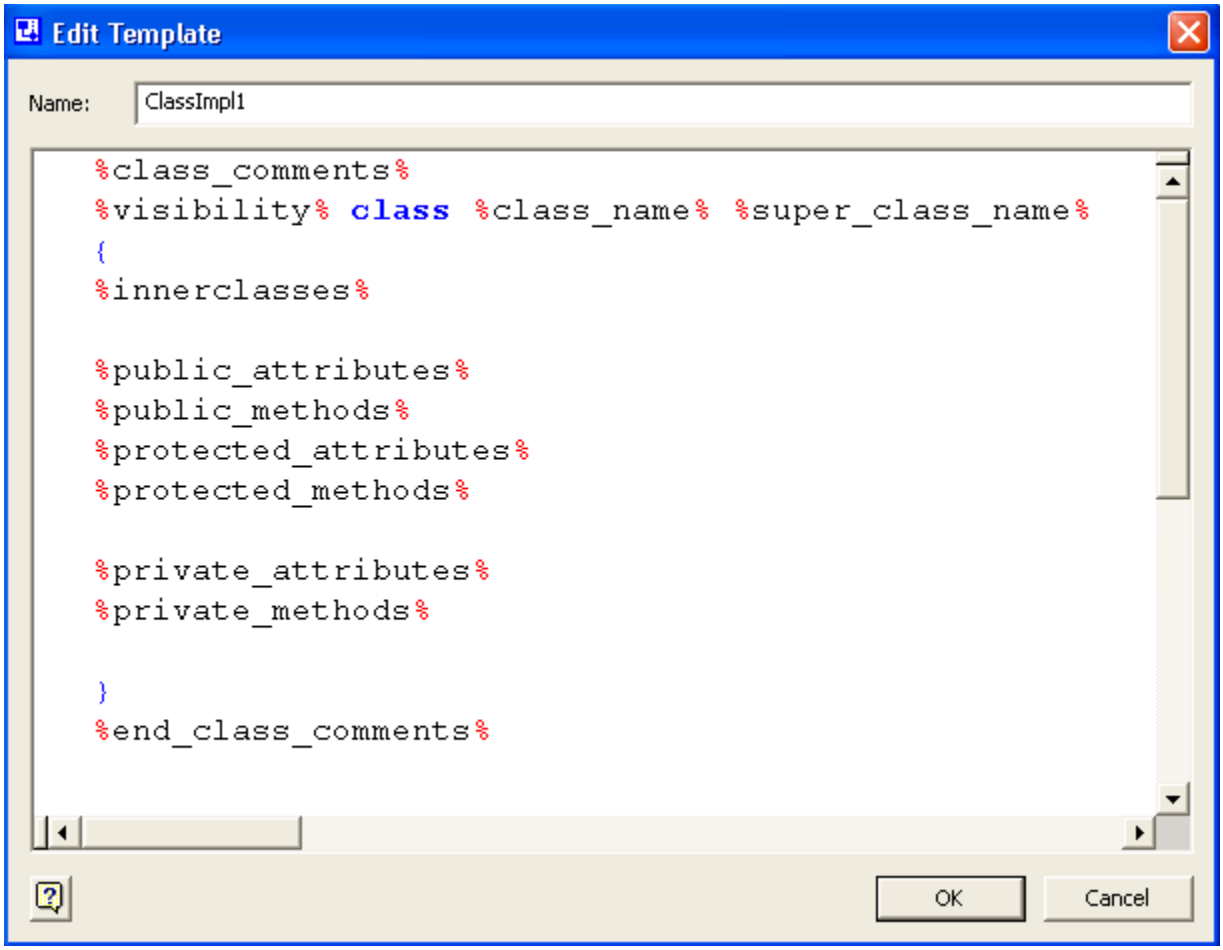

Figure 17.gif> Visio's default template

To edit an existing template, simply go to *Menu bar --> UML --> Preferences* and click *Code Templates* on the left menu. Then select the type of object and the template to edit, and click *Duplicate*. By doing this, you make sure that you don't modify the original template in case you want to undo your changes later. Clicking *Duplicate* opens a new window with a copy of the selected template and a new name for it at the top of the window.

Creating New Templates

To create a new template, go to *Menu bar --> UML --> Preferences* and click *Code Templates* on the left menu. Then select the type of object and the template to edit, and click *New*. A new window with an empty text box will open where you can enter the code for the new template and the template's name.

Conclusion

In this article, you've learned how you can use Microsoft Visio to model applications following the UML approach to create dynamic diagrams that helps you understand the problem that you are trying to solve (in this case, the university application). You've seen how the dynamic diagrams can be converted into actual .NET code that can

be directly plugged into your own projects, and you've also discovered how to generate documentation out of other diagrams, no matter which .NET language you are using. Finally, you've seen how you can define your own templates to give your code a consistent look and feel.