Objects have both behavior and state or, in other words, they do things and they know things. Some objects do and know more things, or at least more complicated things, than other objects. Some objects are incredibly complicated, so complex that developers can have difficulty understanding them. To understand complex classes better, particularly those that act in different manners depending on their state, you should develop one or more UML state machine diagrams, formerly called state chart diagrams in UML 1.x, describing how their instances work.

UML state machine diagrams depict the various states that an object may be in and the transitions between those states. In fact, in other modeling languages, it is common for this type of a diagram to be called a state-transition diagram or even simply a state diagram. A state represents a stage in the behavior pattern of an object, and like UML activity diagrams it is possible to have initial states and final states. An initial state, also called a creation state, is the one that an object is in when it is first created, whereas a final state is one in which no transitions lead out of. A transition is a progression from one state to another and will be triggered by an event that is either internal or external to the object.

Figure 1 presents an example state machine diagram for the Seminar class during registration. The rounded rectangles represent states: you see that instances of Seminar can be in the Proposed, Scheduled, Open For Enrollment, Full, and Closed to Enrollment states. An object starts in an initial state, represented by the closed circle, and can end up in a final state, represented by the bordered circle. This is the exact same notation used by UML activity diagrams, a perfect example of the consistency of the UML.

Figure 1. A seminar during registration.

The arrows in Figure 1 represent transitions, progressions from one state to another. For example, when a seminar is in the Scheduled state, it can either be opened for enrollment or cancelled. The notation for the labels on transitions is in the format event [guard]/[method list]. It is mandatory to indicate the event which causes the transition, such as open or cancelled. Guard, conditions that must be true for the transition to be triggered, are optionally indicated. The [not seat available] guard is shown on the student enrolled transition from the Open For Enrollment to the Closed To Enrollment state. Guards can be described in any manner, including both free form text or formal language – when I’m whiteboarding I’ll use free form text to ensure that it’s readable by everyone but with a sophisticated CASE tool I would consider using either a programming language such as Java or a modeling language such as Object.
Constraint Language (OCL) if the tool has the ability to actually process that information into something useful (such as executable code). The invocation of methods, such as `addToWaitingList()` can optionally be indicated on transitions. The order in the listing implying the order in which they are invoked.

States are represented by the values of the attributes of an object. For example, a seminar is in the Open For Enrollment state when it has been flagged as open and seats are available to be filled. It is possible to indicate the invocation of methods within a state, for example, upon entry into the Closed To Enrollment state the method `notifyInstructor()` is invoked. The notation used within the same as that used on transitions, the only difference being that the method list is mandatory and the event is optional. For example in the Full state the operations `addToWaitingList()` and `considerSplit()` are invoked whenever a student is enrolled. Had there been no event indicated those methods would be invoked continuously (in a loop) whenever the object is in that state. I indicate the methods to run during the state when I want to indicate a method is to be run continuously, perhaps a method that polls other objects for information or a method that implements the logic of an important business process. Methods to be invoked when the object enters the state are indicated by the keyword `entry`, as you see with both the Open For Enrollment and Closed To Enrollment states in Figure 1. Methods to be invoked as the object exits the state are indicated by the keyword `exit`. The capability to indicate method invocations when you enter and exit a state is useful because it enables you to avoid documenting the same method several times on each of the transitions that enter or exit the state, respectively.

Transitions are the result of the invocation of a method that causes an important change in state. Understanding that not all method invocations will result in transitions is important. For example, the invocation of a getter method likely wouldn’t cause a transition because it isn’t changing the state of the object (unless lazy initialization is being applied). Furthermore, Figure 1 indicates an attempt to enroll a student in a full seminar may not result in the object changing state, unless it is determined that the seminar should be split, even though the state of the object changes (another student is added to the waiting list). You can see that transitions are a reflection of your business rules. For example, you see that you can attempt to enroll a student in a course only when it is open for enrollment or full, and that a seminar may be split (presumably into two seminars) when the waiting list is long enough to justify the split. You can have recursive transitions, also called self transitions, that start and end in the same state. An example of which is the student dropped transition when the seminar is full.

For the sake of convention, we say an object is always in one and only one state, implying transitions are instantaneous. Although we know this is not completely true (every method is going to take some time to run), this makes life a lot easier for us to assume transitions take no time to complete.

Because the lifecycle of a seminar is so complex Figure 1 only depicts part of it. Figure 2 depicts the entire lifecycle, with Figure 1 shown as a substate of the Enrollment state. I could have included all of the details in Figure 2 but chose not to in order to keep the diagram simple – I prefer to follow the AM practices Depict Models Simply and Model in Small Increments. In fact, instead of creating a diagram such as Figure 2 I typically prefer something more along the lines of the high-level view of Figure 3 with detailed views such as Figure 1. This approach keeps the diagrams small and easy to understand.

Figure 2. The complete seminar lifecycle.
Figure 3. Top-level state machine diagram.

Figure 4 depicts a slightly different take on state machine diagrams, this time it is much closer to an analysis level diagram because it shows what is happening to the seminar while it is in this state from the point of view of the people involved. It is organized into two parallel swimlanes representing parallel substates – one from the point of view of the professor teaching the seminar and the other showing the actions of the teaching assistant responsible for keeping the seminar material up to date. Concurrent substates are common with hardware but very uncommon in business classes, hence the goofy example. Normally I would depict this sort of information using either a UML activity diagram or a UML timing diagram but I needed an example to show you extra notation.

Figure 4. The Being Taught state.
Figure 4 shows several ways to depict transitions. The Break Starts transition exiting from the Being Taught states is applicable to all of the substates, you know this because it exits from the superstate instead of an individual substate. The Work Submitted transition is potentially triggered by several sources, you know this because it is attached to the outside edge of the superstate, whereas the source of the Break ends transition is explicitly defined as the School Break state. The initial transition into this state is the Term Started transition, indicated through the use of an initial state symbol. I could also have modeled this state coming from an Enrollment state, either approach is fair.

The Term Started and Break Ends transitions are first merged, then they lead to a fork which in turn leads to one or the other set of concurrent substates. A history pseudo-state is shown, the circle with the $H$, indicating that if Seminar was previously in this state, left it, and the returns that it will go back to the substate it was originally in. The arrow leaving the history pseudo state indicates that the Deliver Course Material substate is the default the very first time that Seminar enters the Begin Taught superstate.

Drawing State Machine Diagrams

When drawing a state machine diagram the thing you want to do is to identify the creation state and whether any final states exist. After you have done this, ask yourself what other states or stages in the life of an object does it pass through? You can often find states by looking at the boundary values of your attributes. For example, when the number of students in a seminar reaches the maximum, it becomes full. Full is a valid state because different rules now apply: when a student tries to enroll, he is put on a waiting list and the seminar is a candidate to be split in two.

Once you have identified as many states as you can, start looking for transitions. For each state, ask yourself how the object can get out of it, if possible. This will give you a transition. Because all transitions lead from one state to another, ask yourself what new state the transition leads you to (don’t forget about recursive transitions that lead to the same state). You should also look at the methods you identified in your class diagram. Some of them will
correspond to a transition in your state diagram.

Identifying potential error conditions while you are state machine modeling is common because you are constantly asking “should this transition be allowed when the object is in this state?” When the answer is yes, you need to add the transition to your diagram. When the answer is no, you may need to document this potential issue so your programmers develop the proper error checking code, so the transition is not allowed to occur.

Although being able to inherit state diagrams would be nice, it is extremely unlikely this will happen. The definition of inheritance says that although the subclass is similar to the superclass, it is still different. The behavior of the subclass is usually different than that of the superclass. This means you need to reconsider the state diagram when you inherit from a class with one. The one good thing is many of the states and transitions are reusable. You will probably find you either add new states and transitions, or you will redefine some.

Remaining Agile

State machine modeling is a dynamic modeling technique, one that focuses on identifying the behavior within your system—in this case, behavior specific to the instances of a single class. My style is to draw one or more state machine diagrams when a class exhibits different behavior depending on its state. For example, the Address class is fairly simple, representing data you will display and manipulate in your system. Seminar objects, on the other hand, are fairly complex, and therefore it makes sense to create a state machine diagram for them.

In business applications it seems that a very small proportion of classes, perhaps 5% at most, are complex enough to warrant the creation of a state machine diagram. However, state machine diagrams are much more common in real-time systems (Douglass 1999).

Source

This artifact description is excerpted from Chapter 11 of The Object Primer 3rd Edition: Agile Model Driven Development with UML 2.

Suggested Reading

- Artifacts for Agile Modeling: The UML and Beyond
- Introduction to the Diagrams of UML 2
- Modeling Style Guidelines
- Pavel Hruby’s UML 2.0 Stencil for Visio
- Statechart Modeling Guidelines
- Why Extend the UML Beyond Object and Component Technology?

This is the "must have" book, IMHO, if you're doing real-time modeling with UML.
The Object Primer 3rd Edition: Agile Model Driven Development with UML 2 is an important reference book for agile modelers, describing how to develop 35 types of agile models including all 13 UML 2 diagrams. Furthermore, this book describes the techniques of the Full Lifecycle Object Oriented Testing (FLOOT) methodology to give you the fundamental testing skills which you require to succeed at agile software development. The book also shows how to move from your agile models to source code (Java examples are provided) as well as how to succeed at implementation techniques such as refactoring and test-driven development (TDD). The Object Primer also includes a chapter overviewing the critical database development techniques (database refactoring, object/relational mapping, legacy analysis, and database access coding) from my award-winning Agile Database Techniques book.

Agile Modeling: Effective Practices for Extreme Programming and the Unified Process is the seminal book describing how agile software developers approach modeling and documentation. It describes principles and practices which you can tailor into your existing software process, such as XP, the Rational Unified Process (RUP), or the Agile Unified Process (AUP), to streamline your modeling and documentation efforts. Modeling and documentation are important aspects of any software project, including agile projects, and this book describes in detail how to elicit requirements, architect, and then design your system in an agile manner.

The Elements of UML 2.0 Style describes a collection of standards, conventions, and guidelines for creating effective UML diagrams. They are based on sound, proven software engineering principles that lead to diagrams that are easier to understand and work with. These conventions exist as a collection of simple, concise guidelines that if applied consistently, represent an important first step in increasing your productivity as a modeler. This book is oriented towards intermediate to advanced UML modelers, although there are numerous examples throughout the book it would not be a good way to learn the UML (instead, consider The Object Primer). The book is a brief 188 pages long and is conveniently pocket-sized so it's easy to carry around.

Translations

- Japanese

Let Me Help

I actively work with clients around the world to improve their information technology (IT) practices as both a mentor/coach and trainer. A full description of what I do, and how to contact me, can be found here.

Copyright 2003-2006 Scott W. Ambler

Last updated: April 3, 2006
This site owned by Ambysoft Inc.