

Extraído de "UML DeMistified: A Self Teaching Guide"  
Paul Kimmel  
Editorial McGraw-Hill© 2005  
Capítulo 1, Págs 1-16

Pictures of little stick people represent the oldest recorded form of communication in human history. Some of these cave art have been dated to be as old as 75,000 years. Oddly enough, here we are at the turn of the twenty-first modern century, and we are still using little stick figures to convey information. That's right, a little stick man we'll call Esaw is a central character in one of the newest languages developed by humans (Figure 1-1).



Figure 1-1 Esaw, who is referred to as an actor in the UML.

The language I am talking about is called the *Unified Modeling Language*, or UML. The UML is a language just as sure as, Pascal, C# (C sharp), German, English, and Latin are languages. And the UML is probably one of the newest languages invented by humankind, invented around 1997.

As with other languages, the UML was invented out of necessity. Moreover, as with many languages, the UML uses symbols to convey meaning. However, unlike organic languages such as English or German that evolve over time from common use and adaptation, the UML was invented by scientists, which unfortunately is a problem. Scientists are very smart, but they often are not very good at explaining things to those less scientific. This is where I come in.

In this chapter we will look at the origin and evolution of the UML. We also will talk about how to create pictures using the UML, how many and what types of pictures to create, what those pictures should convey, and most important, when to stop drawing pictures and start writing code.

# Understanding Models

A *model* is a collection of pictures and text that represent something—for our purposes, software. (Models do not have to represent software, but we will narrow our scope to software models.) A model is to software what a blueprint is to a house.

Models are valuable for many specific reasons. Models are valuable because they consist of pictures to a large extent, and even simple pictures can convey more information than a lot of text, e.g., code. This is consistent with the somewhat modified old adage that a picture speaks a thousand lines of code. Models are valuable because it is easier to draw some simple pictures than it is to write code or even text that describes the same thing. Models are valuable because it is cheaper, faster, and it is easier to change models than it is to change code. The simple truth is that cheap, fast, easy, and flexible are what you want when you are solving problems.

Unfortunately, if everyone uses different pictures to mean the same thing, then the pictures add to the confusion rather than mitigate it. This is where the UML comes in.

## Understanding the UML

The UML is an official definition of a pictorial language where there are common symbols and relationships that have one common meaning. If every participant speaks UML, then the pictures mean the same thing to everyone looking at those pictures. Learning the UML, therefore, is essential to being able to use pictures to cheaply, flexibly, and quickly experiment with solutions.

It is important to reiterate here that it is faster, cheaper, and easier to solve problems with pictures than with code. The only barrier to benefiting from modeling is learning the language of modeling.

The UML is a language just like English or Afrikaans is a language. The UML comprises symbols and a grammar that defines how those symbols can be used.

Learn the symbols and grammar, and your pictures will be understandable by everyone else who recognizes those symbols and knows the grammar.

Why the UML, though? You could use any symbols and rules to create your own modeling language, but the trick would be to get others to use it too. If your aspirations are to invent a better modeling language, then it isn't up to me to stop you. You should know that the UML is considered a standard and that what the UML is and isn't is defined by a consortium of companies that make up the Object Management Group (OMG). The UML specification is defined and published by the OMG at [www.omg.org](http://www.omg.org).

## The Evolution of Software Design

If you feel that you are late to the UML party, don't fret—you are actually an early arrival. The truth is that the UML is late to the software development party. I work all over North America and talk with a lot of people at lots of very big software companies, and the UML and modeling are just starting to catch on. This is best exemplified by Bill Gates' own words after his famous "think week" in 2004, where Gates is reported to have talked about the increasing importance of formal analysis and design (read UML) in the future. This

sentiment is also supported by Microsoft's very recent purchase of Visio, which includes UML modeling capabilities.

The UML represents a formalization of analysis and design, and formalization always seems to arrive last. Consider car makers in the last century. Around the turn of the last century, every buggy maker in Flint, Michigan, was turning horse carriages into motorized carriages, i.e., cars. This occurred long before great universities such as Michigan State University (MSU) were turning out mechanical engineers trained to build cars and software tools such as computer-aided design (CAD) programs that are especially good at drawing complex items such as car parts. The evolution of formalized automobile engineering is consistent with the evolution of formalized software engineering.

About 5000 years ago, the Chinese created one of the first computers, the abacus. About 150 years ago, Charles Babbage invented a mechanical computing machine. In 1940, Alan Turing defined the Turing computing machine and Presper Eckert and John Mauchly invented Eniac. Following computing machines came punch cards and Grace Hopper's structured analysis and design to support Cobol development. In the 1960s, Smalltalk, an object-oriented language, was invented, and in 1986, Bjarne Stroustrup invented what is now known as C++. It wasn't until around this same time period—the 1980s—that very smart men like Ivar Jacobson, James Rumbaugh, and Grady Booch started defining elements of modern software analysis and design, what we now call the UML.

In the late 1980s and early 1990s, modeling notation wars were in full gear, with different factions supporting Jacobson, Rumbaugh, or Booch. Remember, it wasn't until 1980 that the average person could purchase and own—and do something useful with—a personal computer (PC). Jacobson, Rumbaugh, and Booch each used different symbols and rules to create their models. Finally, Rumbaugh and Booch began collaborating on elements of their respective modeling languages, and Jacobson joined them at Rational Software.

In the mid-1990s, the modeling elements of Rumbaugh [Object Modeling Technique (OMT)], Booch (Booch method), and Jacobson (Objectory and Use Cases)—Rumbaugh, Jacobson, and Rumbaugh are referred to as "the three amigos"—were merged together to form the *unified modeling process*. Shortly thereafter, *process* was removed from the modeling specification, and the UML was born. This occurred very recently, in just 1997. The UML 2.0 specification stabilized in October 2004; that's right, we are just now on version 2.

This begs the question: Just how many companies are using the UML and actually designing software with models? The answer is still very few. I work all over North America and personally know people in some very successful software companies, and when I ask them if they build software with the UML, the answer is almost always no.

### *If No One Is Modeling, Why Should You?*

A rational person might ask: Why then, if Bill Gates is making billions writing software without a significant emphasis on formal modeling, should I care about the UML? The answer is that almost 80 percent of all software projects fail. These projects exceed their budgets, don't provide the features customers need or desire, or worse, are never delivered.

The current trend is to outsource software development to developing or thirdworld nations. The basic idea is that if American software engineers are failing, then perhaps paying one-fifth for a Eurasian

software developer will permit companies to try five times as often to succeed. What are these outsourcing companies finding? They are discovering that the United States has some of the best talent and resources available and that cheap labor in far-away places only introduces additional problems and is no guarantee of success either. The real answer is that more time needs to be spent on software analysis and design, and this means models.

### ***Modeling and the Future of Software Development***

A growing emphasis on formal analysis and design does not mean the end of the software industry's growth. It does mean that the wild, wild west days of the 1980s and 1990s eventually will come to a close, but it is still the wild, wild hacking west out there in software land and will be for some time.

What an increasing emphasis on software analysis and design means right now is that trained UML practitioners have a unique opportunity to capitalize on this growing interest in the UML. It also means that gradually fewer projects will fail, software quality should improve, and more software engineers will be expected to learn the UML.

## **Modeling Tools**

Until very recently, modeling has been a captive in an ivory tower surrounded by an impenetrable garrison of scientists armed with metamodels and ridiculously expensive modeling tools. The cost of one license for a popular modeling tool was in the thousands of dollars; this meant that the average practitioner would have to spend as much on one application for modeling as he or she spent for an entire computer. This is ridiculous.

Modeling tools can be very useful, but it is possible to model on scraps of paper. Thankfully, you don't have to go that far. Love it or hate it, Microsoft is very good at driving down the cost of software. If you have a copy of MSDN, then you have a modeling tool that is almost free, Visio. Visio is a good tool, ably capable of producing high-quality UML models, and it won't break your budget<sup>1</sup>.

In keeping with the theme of this book—demystifying UML—instead of breaking the bank on Together or Rose, we are going to use the value-priced Visio. If you want to use Rose XDE, Together, or some other product, you are welcome to do so, but after reading this book, you will see that you can use Visio and create professional models and save yourself hundreds or even thousands of dollars.

---

<sup>1</sup> Microsoft has a new program that permits you to purchase MSDN Universal, which includes Visio, for \$375. This is an especially good value

## **Using Models**

Models consist of diagrams and pictures. The intent of models is that they are cheaper to produce and experiment with than code. However, if you labor over what models to draw, when to stop drawing and start coding, or whether your models are perfect or not, then you will slowly watch the cost and time value of models dwindle away.

You can use plain text to describe a system, but more information can be conveyed with pictures. You could follow the extreme Programming (XP) dictum and code away, refactoring as you go, but the details of lines of code are much more complex than pictures, and programmers get attached to code but not to pictures.

(I don't completely understand the psychology of this code attachment, but it really does exist. Just try to constructively criticize someone else's code, and watch the conversation deteriorate very quickly into name calling.) This means that once code is written, it is very hard to get buy-in from its coder or a manager to make modifications, especially if the code is perceived to work. Conversely, people will gladly tinker with models and accept suggestions.

Finally, because models use simple symbols, more stakeholders can participate in design of the system. Show an end user a hundred lines of code, and you can hear the crickets chirping; show such an end user an activity diagram, and that same person can tell you if you have captured the essence of how that task is performed correctly.

## **Creating Diagrams**

The first rule of creating models is that code and text are time-consuming, and we don't want to spend a lot of time creating text documents that no one wants to read. What we do want to do is to capture the important parts of the problem and a solution accurately. Unfortunately, this is not a prescription for the number or variety of diagrams we need to create, and it does not indicate how much detail we need to add to those diagrams.

Toward the end of this chapter, in the section "Finding the finish line.", I will talk more about how one knows that one has completed modelling. Right now, let's talk about the kinds of diagram we may want to create.

### ***Reviewing Kinds of Diagrams***

There are several kinds of diagrams that you can create. I will quickly review the kinds of diagrams you can create and the kinds of information each of these diagrams is intended to convey.

#### **Use Case Diagrams**

*Use case diagrams* are the equivalent of modern cave art. A use case's main symbols are the *actor* (our friend Esaw) and the *use case oval* (Figure 1-2).

Use case diagrams are responsible primarily for documenting the macro requirements of the system. Think of use case diagrams as the list of capabilities the system must provide.

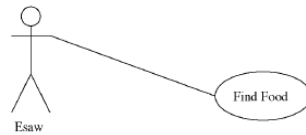


Figure 1-2 The "FindFood" use case.

### Activity Diagrams

An *activity diagram* is the UML version of a flowchart. Activity diagrams are used to analyze processes and, if necessary, perform process reengineering (Figure 1-3).

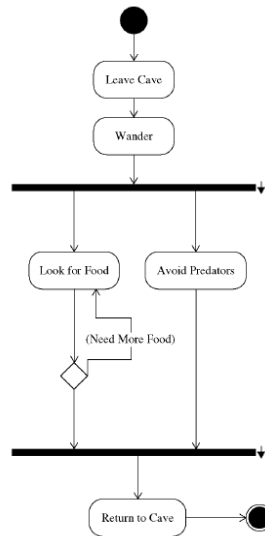


Figure 1-3 An activity diagram showing how Esaw goes about finding food.

An activity diagram is an excellent tool for analyzing problems that the system ultimately will have to solve. As an analysis tool, we don't want to start solving the problem at a technical level by assigning classes, but we can use activity diagrams to understand the problem and even refine the processes that comprise the problem.

### Class Diagrams

*Class diagrams* are used to show the classes in a system and the relationships between those classes (Figure 1-4). A single class can be shown in more than one class diagram, and it isn't necessary to

show all the classes in a single, monolithic class diagram. The greatest value is to show classes and their relationships from various perspectives in a way that will help convey the most useful understanding.

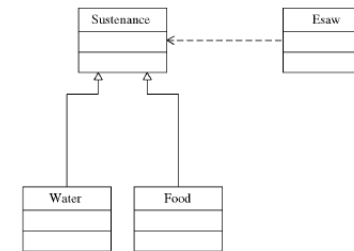


Figure 1-4 A single class diagram, perhaps one of many, that conveys a facet of the system being designed.

Class diagrams show a static view of the system. Class diagrams do not describe behaviors or how instances of the classes interact. To describe behaviors and interactions between objects in a system, we can turn to *interaction diagrams*.

### Interaction Diagrams

There are two kinds of interaction diagrams, the *sequence* and the *collaboration*. These diagrams convey the same information, employing a slightly different perspective. Sequence diagrams show the classes along the top and messages sent between those classes, modeling a single flow through the objects in the system.

Collaboration diagrams use the same classes and messages but are organized in a spatial display. Figure 1-5 shows a simple example of a sequence diagram, and Figure 1-6 conveys the same information using a collaboration diagram.

A sequence diagram implies a time ordering by following the sequence of messages from top left to bottom right. Because the collaboration diagram does not indicate a time ordering visually, we number the messages to indicate the order in which they occur.

Some tools will convert interaction diagrams between sequence and collaboration automatically, but it isn't necessary to create both kinds of diagrams. Generally, a sequence diagram is perceived to be easier to read and more common.

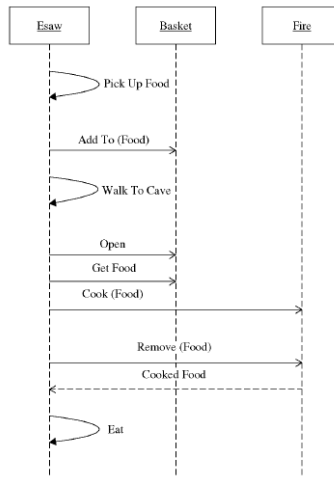


Figure 1-5 A single sequence diagram demonstrating how food is gathered and prepared.

## State Diagrams

Whereas interaction diagrams show objects and the messages passed between them, a *state diagram* shows the changing state of a single object as that object passes through a system. If we continue with our example, then we will focus on Esaw and how his state is changing as he forages for food, finds food, and consumes it (Figure 1-7).

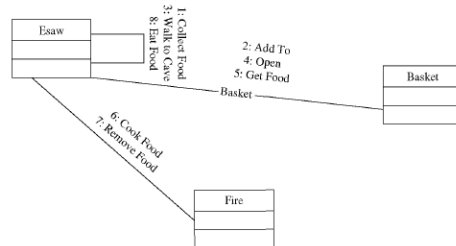


Figure 1-6 A collaboration diagram that conveys the same gathering and consuming

## Component Diagrams

The UML defines various kinds of models, including analysis, design, and implementation models. However, there is nothing forcing you to create or maintain three models for one application. An example of a diagram you might find in an implementation model is a component diagram. A *component diagram* shows the components—think subsystems—in the final product.

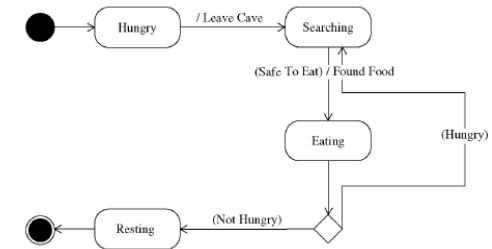


Figure 1-7 A state diagram (or statechart) showing the progressive state as Esaw forages and eats.

I'll cover deployment diagrams later in this book but defer citing an example for now. Generally, a component diagram is a bit like a class diagram with component symbols.

## Other Diagrams

There are other kinds or variations of diagrams we can create. For example, a *deployment topology diagram* will show you what your system will look like deployed. Such a diagram typically contains symbols representing things such as Web servers, database servers, and various and sundry devices and software that make up your solution. This kind of diagram is more common when you are building «-tiered distributed systems.

I will show you examples of some of these diagrams later in this book. Remember that the key to modeling is to modeling interesting aspects of your system that help to clarify elements that may not be obvious, as opposed to modeling everything.

## Finding the Finish Line

The hardest part of modeling is that it is so new that UML models are subjected to some of the same language wars object-oriented projects suffered from during the last decade. I encourage you to avoid these language wars as mostly unproductive academic exercises. If you find yourself getting hung up on whether something is or isn't good UML, then you are heading toward analysis (and design) paralysis.

The goal is to be as accurate as possible in a reasonable amount of time. Poorly designed software is bad enough, but no software is almost always worse. To determine if you are finished with a particular diagram or model, ask the question: *Does the diagram or model convey my understanding, meaning, and intent?* That is, is the diagram or model good enough? Accuracy is important because others need to read your models, and idiomatic mistakes mean that the models will be harder for others to read.

## How Many Diagrams Do I Create?

There is no specific answer. A better question is: *Do I have to create every kind of diagram?* The answer to this question is no. A refinement of this answer is that it is helpful to create diagrams that resolve persnickety analysis and design problems and diagrams that people actually will read.

### How Big Should a Diagram Be?

Determining how big a model needs to be is another good question to decide. If a given model is too big, then it may add to confusion. Try to create detailed models — but not too detailed. As with programming, creating UML models takes practice.

Solicit feedback from different constituencies. If the end users think that an analysis diagram adequately and correctly captures the problem, then move on. If the programmers can read a sequence and figure out how to implement that sequence, then move on. You can always add details if you must.

### How Much Text Should Supplement My Models?

A fundamental idea for using pictures for modeling instead of long-winded text is that pictures convey more meaning in a smaller space and are easier to manipulate. If you add too much text — constraints, notes, or long documents — then you are defeating the purpose of this more concise pictorial notation.

The best place for text is the use case. A good textual description in each use case can clarify precisely what feature that use case supports. I will demonstrate some good use case descriptions in Chapter 2.

You are welcome to add any clarifying text you need, but the general rule for text is analogous to the rule for comments in code: Only comment things that are reasonably subject to interpretation.

Finally, try to document everything in your modeling tool as opposed to a separate document. If you find that you need or the customer requires a written architectural overview, defer this until after the software has been produced.

### Get a Second Opinion

If you find yourself getting stuck on a particular diagram, get a second opinion. Often, putting a diagram aside for a couple of hours or getting a second opinion will help you to resolve issues about one model. You may find that the end user of that model will understand your meaning or provide more information that clears up the confusion, or a second set of eyes may yield a ready response. A critical element to all software development is to build some inertia and capture the macro, or big, concepts without getting stuck or keeping users waiting.

## Contrasting Modeling Languages with Process

The UML actually began life as the Unified Process. The inventors quickly realized that programming languages do not dictate process, and neither should modeling languages. Hence process and language were divided.

There are many books on process. I don't think one process represents the best fit for all projects, but perhaps one of the more flexible processes is the Rational Unified Process. My focus in this book is on the UML, not on any particular process.

I will be suggesting the kinds of models to create and what they tell you, but I encourage you to explore development processes for yourself. Consider exploring the Rational Unified Process (RUP), the Agile process, extreme Programming (XP), and even Microsoft's Services Oriented Architecture (SOA). (SOA is

more of an architectural approach using elements like XML Web Services, but it offers some good techniques.)

I am not an expert on every process, but here is a summary that will provide you with a starting point. The RUP is a buffet of activities centered on the UML that defines iterative, small waterfalls macro phases, including inception, elaboration, construction, and transition. XP is constructive hacking. The idea generally is based on building on your understanding, expecting things to change, and using techniques such as refactoring and pair programming to support changes as your understanding grows. Microsoft's SOA depends on technologies like COM+, Remoting, and XML Web Services and a separation of responsibilities by services.

Agile is a new methodology that I don't understand completely, but Dr. Boehm's book, *Balancing Agility and Discipline*, compares it with XP, and I suspect that conceptually it lives somewhere between RUP and XP.

It is important to keep in mind that many of the people or entities offering a process may be trying to sell you something, and some very good ideas have come from each of these parties.