Martin Fowler ≈ **fowler@acm.org**
Independent consultant in Boston, Massachusetts

What's a Model For?

There are many people who think the adoption of the UML is one of the most important developments in the last couple of years. While I agree that sorting out silly conflicts over notation is a big step forward, we still have a more fundamental set of questions: What exactly is the UML for? Why do we use it, and if we don't why should we? If you are using, or thinking of using, the UML, you ought to know the answers to these questions.

It's important to recognize that creating a UML diagram involves a cost, and a UML model is not something that fundamentally matters to a customer. Customers want software that works, not pictures, however pretty or standard. Thus, the value of modeling is entirely bound up with its effect on the software you are producing. If the model improves the quality or reduces the cost, then the model has value. But the model has no value on its own. It reminds me of a character in a (forgotten) novel: "He was like the 0 in 90, with his wife he was something, without her he was nothing."

The value that a model provides is fundamentally about giving the user a greater understanding of something than the software itself. This may be greater understanding of the software or of the domain that the software supports. It can do this either by a two-dimensional graphical presentation, or by highlighting important information.

A complicating factor is that all humans don't think alike. Some people understand things better in a textual form; others prefer pictures. People differ where they are on this scale, and some prefer pictures for some subjects and text for others. So beware of over-restrictive standards. Remember that individual needs differ, and that you need to find what the most effective blend is for the people you are working with.

I'll start by looking at the simple transformation from text to graphics, leaving all the details in. In this approach you take everything you can say in text and put it in the model. This takes a lot of work; doing this level of detail with model and source code takes too much time to be worthwhile, unless you have tools that automate the task. This is where many CASE tools come in: either round trip where tools convert from model to code and back again (e.g., Rational Rose); or tripless, where the code is the storage mechanism for the model (e.g., Together-J).

The cost of producing the model is lowered, although there is still a cost. However, when you have a model that detailed, you have to ask how much you're gaining. I think you can gain in structural relationships (such as data structures), but with control flow you often lose clarity.

Another important function of models is highlighting. It's hard to understand a complex system by diving into the details in all their gore. You can use a model to highlight just the important parts of the system, and you can highlight before you create software or after you've built all the code. The point is that you can understand the key structures that make the software work first, and then you're in a better position to investigate all the details.

Notice that I say, "highlight the important details," not "ignore the details." A high-level view that ignores all details is usually worthless because it doesn't tell you anything. Rather than a high-level view, I prefer to think of this style as a skeletal model. It shows you the bones of the system, which can be quite detailed.

To my mind, a skeletal model is better than a fully-fleshed model, where there's so much more information to wade through. When I'm trying to understand a model, I have to figure out where to start. Typically people do this by trying to find the important stuff, but since they don't know what's important, it's much easier to get the person who knows the model to do the selection. Similarly, if I'm creating a model before building the system, the key decisions are in those important details. Not all issues are equally important, and I have limited brain power. I need to expend that power on the important details, so I use a skeletal model.

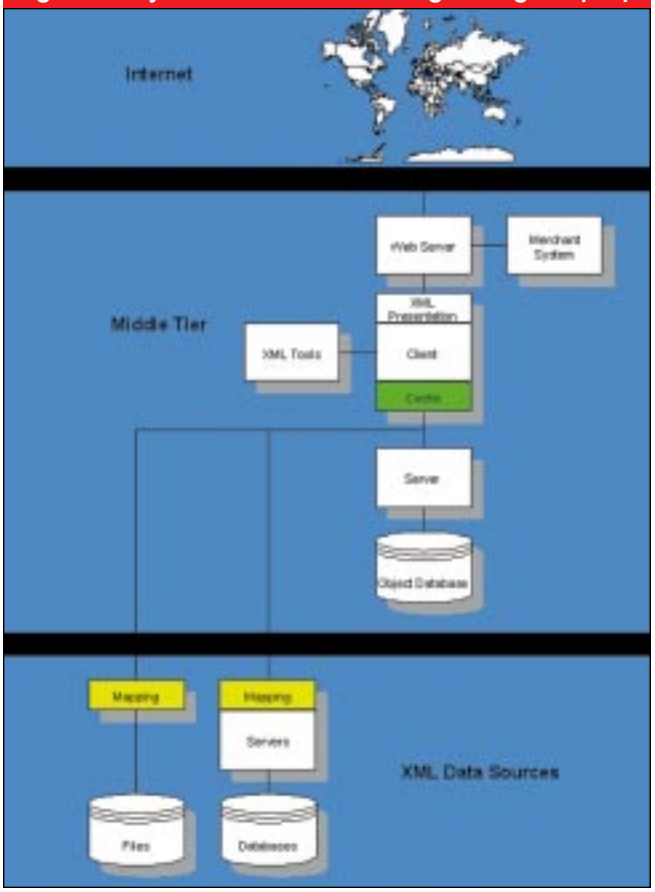
I've also found a skeletal model is easier to maintain than a fully-fleshed one. Important details change less often. And the skeletal model is more useful, so people are more inclined to keep it up to date. If people want details, they can explore the code once the skeleton has provided the overview. The nice thing about the code is that it's always in sync with itself.

The crux is that a model is either fully-fleshed or skeletal, it can't be both. You either include all details, or you decide to leave something out. As soon as you leave something out, you are going down the skeleton path, even though your skeleton may look less anorexic than mine. Your decision will be based on what you find most valuable in visualizing the system. People vary, so your choice will be different than other people's, but you do need to make an explicit choice.

This is a key problem with many CASE tools. The fashion these days is for the CASE tools to keep an automated link to the code. A tool, however, cannot tell important details from

continued on page 35

Figure 1. Object Structure for a Clothing Catalog Company



There are many database options for an XML data server. For this clothing company example, I am going to discuss using an ODBMS, which can be a good choice because XML data looks like a graph. (See Figure 1 for a fragment of the object structure for a clothing catalog company.) Searching for clothing information involves traversing this graph structure from product to item to size and to color swatch. Keep in mind that this object structure fragment has been simplified—it has objects such as color swatch that will be shared by other items in the catalog not shown in this figure.

If you have been reading my columns or visiting my Web site, you have probably read (many times) that ODBMSs are great when you have a business need for high performance on complex data. One sign of complex data is a graph structure coupled with access by traversal, such as this example. When you are selling products on the Internet and your Web site must respond quickly to people browsing or agent programs, your site must respond quickly, or the people/agents will just move on. This is a compelling business need.

Figure 2 shows a possible architecture that uses an ODBMS as an XML data server. It shows the mappings from the XML data sources discussed earlier as well as the tools for editing XML data. It also shows a connection to a merchant system. Note the caching in the middle tier that ODBMSs offer. This is another performance enhancement for providing complex data to web servers.

Several ODBMS vendors offer solutions similar to this. If you are interested, I suggest you check out the Web sites for Ardent Software **<www.ardentsoftware.com>**, Object Design **<www.objectdesign.com>**, and POET Software **<www. poet.com>**. These companies had XML data server offerings at the time I was writing this column. By the time it appears, there may be more companies with offerings. Come to my Web site to find if there are more.

We live in a world of many technical options. The key to finding good solutions often involves matching business needs for performance to the options available. In this column, an example of a good solution for a middle-tier database in an XML data server was outlined. In this case, an ODBMS is able to provide the high performance on complex data required. As we move through the world of increasing database diversity, an ODBMS is a serious option for an XML data server.

Editor's Note: Doug Barry has just published *XML Data Servers: An Infrastructure for Effectively Using XML in Electronic Commerce,* which looks at architectural options for XML data servers and provides analysis/features of each architecture (**www.xml-data-servers.com**).

METHODS IN PRACTICE

continued from page 33

side issues, so the result is always a fully-fleshed model. This is made worse by the fact that tools typically base their analysis on the data structure of the classes rather than the interfaces. The whole point of working with objects is that you see the interfaces, not the internal data structure, so the typical reverse-engineered diagram shows you the very things that are supposed to be hidden. CASE tool vendors need to put more time into thinking about how to show the user important things and how to hide things that should stay under wraps. Of course, since everyone's definition of important is different, this means there's a lot of tricky customization to do.

So, the unavoidable dilemma: In order to see the wood from the trees in design, you need a skeletal model. However, as soon as you do this, you lose the ability to have an automated connection to the details.

I don't think we'll see a solution from CASE vendors any time soon. If you want a model that really does communicate effectively, you need to make it a skeletal model, and that means you have to build it yourself. In the end the human brain counts.

As a last note, I'd like to thank Derek Coleman for helping me crystallize my thinking on this column with the assistance of some good wine and a paper napkin—two of the world's most effective (and enjoyable) design tools.