Editor: Martin Fowler ■ ThoughtWorks ■ fowler@acm.org

# Modeling with a Sense of Purpose

**John Daniels**

It's all a question of purpose. These days, practically everyone involved in developing software draws pictures that represent some aspect of the software or its requirements. They do this to improve their own understanding and, usually, to communicate that understanding to others. But all too often, the understanding is muddled and confused because the designer hasn't clearly established the picture's purpose or explained how others should interpret it. Surprisingly, this is true even when the designer uses an established modeling standard, such as the Unified Modeling Language (UML).

For many years, it has been common practice for database designers to create *logical models* and *physical models*. Typically, designers represent a logical model using some form of the entity-relationship (ER) diagram, whereas they ultimately represent a physical model using a schema held by the database engine. The important difference is the level of abstraction: a logical model ignores the constraints that the underlying database technology imposes and presents a simplified view.

Sometimes physical database designs are also drawn using ER diagrams but with explicit attributes for foreign keys that represent relationships. So, when presented with an ER diagram I've never seen before, I must establish its purpose before I can understand it. Fortunately, with an ER diagram, it's usually easy to see what kind of model this is, but in other circumstances—especially with UML diagrams—it isn't always so obvious.

## Implementation, specification, and conceptual models

I call a model built to explain how something is implemented an *implementation model* and a more abstract model that explains what should be implemented a *specification model*. Both are models of software systems, and if confusion often exists between these two models, then far greater confusion surrounds the relationship between models of software and models of real-world situations.

In software projects, we frequently need to find ways of gaining a better understanding of the real-world problem to be solved. Consequently, we often produce *information models* that depict the items of information in a particular business situation and their relationships, but two issues immediately arise.

First, what should be the scope of such models? As we saw with the Great Corporate Data Modeling Fiasco of the 1980s and 1990s, where large enterprises invested heavily in attempts to capture all the information used in an organization, these models easily become so large that they are impossible to keep up to date. The solution to this issue is, of course, to model only those parts of the organization needed to represent the current situation of interest—not to worry too much about the bigger picture.

Second, these models have no value unless we can directly apply their content to system-building projects. Unfortunately, for



0740-7459/02/$17.00 © 2002 IEEE

reasons that I will explain shortly, the approach claiming to offer the most assistance with this—object-oriented design—has been slow to deliver effective processes.

I'll refer to all models of situations in the world—of which information models are one example—as *conceptual models*; other terms that are used include essential models, domain models, business models, and even, most confusingly in my view, analysis models. So we have three kinds of models (see Figure 1) for quite different purposes:

- *Conceptual models* describe a situation of interest in the world, such as a business operation or factory process. They say nothing about how much of the model is implemented or supported by software.
- *Specification models* define what a software system must do, the information it must hold, and the behavior it must exhibit. They assume an ideal computing platform.
- *Implementation models* describe how the software is implemented, considering all the computing environment's constraints and limitations.

Most of the modeling I've done has been in the field of object and component systems, where opportunities for confusion between these three modeling perspectives are particularly large. This is because with object-oriented software, we are always striving to make the software elements that make up our program (classes, in most object-oriented languages) correspond closely to problem-domain concepts with similar names. So, if our conceptual model contains the concept of *customer*, our software will contain direct representations of customers, and our software customers will have similar attributes to their real-world counterparts.

We want this correspondence because it improves traceability between requirements and code, and because it makes the software easier to understand. In this respect, object-oriented programming has been a great success, but the yearning for this correspondence between the conceptual and implementation models has oversimplified the process of moving between them. One reason why this process is more complicated than we'd like is that, given the complexity of modern layered application architectures, we find different representations of the same concept in many different parts of the application.

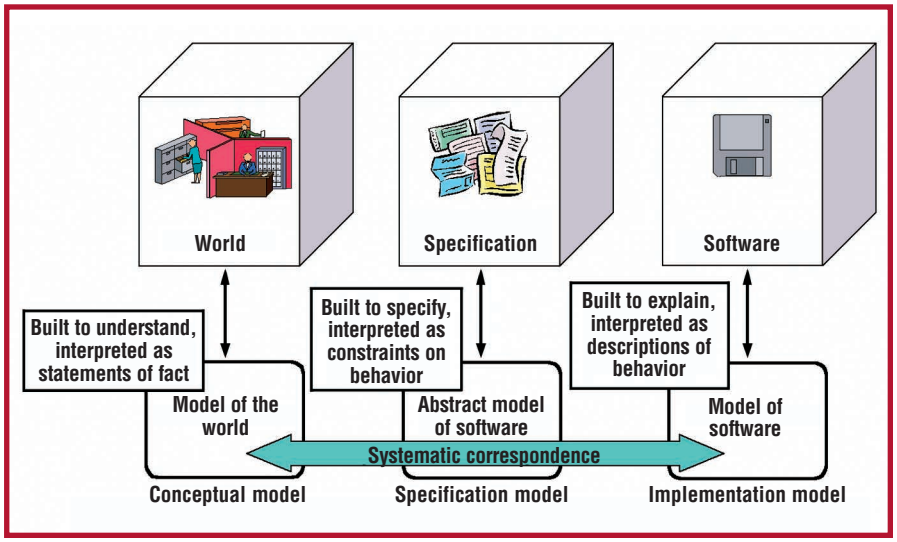


**Figure 1. Three modeling perspectives.**

## Notation overload

The examples we've looked at so far—information models and database models—are concerned with modeling the structure of things; life gets much more complicated when we attempt to model behavior. If we construct an implementation model of an object-oriented program—for example, one written in Java—we assume that software objects cooperate by sending each other synchronous messages. It might or might not be satisfactory to adopt the same paradigm when creating specification models of object systems, but that paradigm is certainly useless if we want to capture the world's concurrent and unpredictable nature in a conceptual model.

**Given the complexity of modern layered application architectures, we find different representations of the same concept in many different parts of the application.**

Despite these obvious problems, many methodologists devising ways of designing object-oriented systems during the late 1980s and 1990s persisted with the claim that conceptual models could be built using only the concepts of object-oriented programming. Furthermore, they claimed that the process of moving from conceptual model to code was one that simply involved elaboration rather than mapping or translation. Although we've since learned the hard way that this is too simplistic, that realization hasn't been manifested in the modeling notations we use or in the way people are often taught to use them.

Consider, for example, the facilities in the UML for describing software structure. The primary and ubiquitous notation for this is the class diagram,

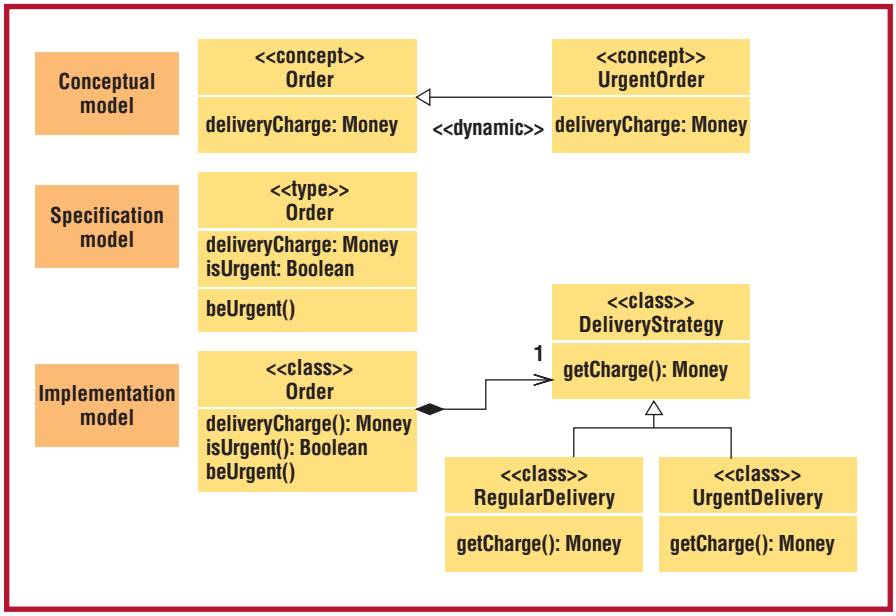


**Figure 2. One notation, three models.**

and we should be pleased that this relatively simple notation has proved useful for depicting everything from the structure of Web sites to the layout of XML documents. But its generality is part of the problem. Even within the limited scope of a single development project, you'll typically find class diagrams used for a range of purposes, including the structural aspects of all three kinds of models discussed earlier. Figure 2 illustrates this with a simple example. In the conceptual model, we are happy to use modeling features that aren't generally supported by programming environments, such as dynamic subtyping. In the implementation model, we use only features that our chosen language directly supports, and we show implementation choices such as the use of the strategy pattern here.

The same applies to all UML notations. Table 1 shows how I use the main UML notations within each kind of model. A blank cell indicates that I don't use that notation within that kind of model, but surely you'll be able to find someone who does. So it's vital that whenever you use one of these notations, you indicate clearly both what kind of model this is part of and precisely what you're trying to depict.

The UML would be much better if it had a built-in understanding of the three kinds of model and at least insisted that you state which one you are building. Unfortunately, it doesn't, so unless you are lucky enough to be working with UML tools that support profiles, the best you can do is use UML's stereotype feature to mark model elements appropriately.

So, if a class diagram depicting an information model is, despite appearances, saying something fundamentally different from a class diagram specifying the chosen object structures in our software, doesn't that just make life more difficult and confusing? No. By having these strong distinctions, we can model at different levels of abstraction at different times and separate the concerns that apply at the different levels. To get these benefits, though, modelers must be very clear in their minds about the different perspectives provided by the three kinds of models, and my experience is that even many experienced developers have yet to think clearly about all this. They need to get a sense of purpose.

**John Daniels** is a consultant at Syntropy Limited. Contact him at john@syntropy.co.uk.

## Table 1

### Using the main UML notations with conceptual, specification, and implementation models

| Diagrams | Conceptual model | Specification model | Implementation model |
|---|---|---|---|
| Use case | — | Software boundary interactions | — |
| Class | Information models | Object structures | Object structures |
| Sequence or collaboration | — | Required object interactions | Designed object interactions |
| Activity | Business processes | — | — |
| Statechart | Event-ordering constraints | Message-ordering constraints | Event or response definitions |