

Marc J. Balcer *Chief Scientist, ModelCompilers.com*

Scan any bookstore—online or real—and you’ll find that UML books tend to fall into two categories: books using UML diagrams to sketch requirements, and books using UML diagrams to design object-oriented software.

Requirements analysis uses UML to capture an external view of the problem. Software design uses UML as a graphical representation of the code to be written. The presence of a single notation used for both activities can lead to the presumption that UML enables as simple progression from requirements through to software design. Of course, in practice, we know that is not the case.

In this column, I’ll introduce the idea of a *solution model* that addresses the application issues independently of platform or implementation technology issues. I’ll introduce you to the concept of a Platform Independent Model (PIM), one of the key concepts of the Object Management Group’s (OMG) Model Driven Architecture (MDA).

### UML for Capturing Requirements

Requirements are intentionally informal collections of needs and wants. In requirements analysis, UML is often used informally as a “whiteboard notation.” Analysts sketch diagrams as “high-level” pictorial representations of the requirements. Use case diagrams identify the actors and use cases; activity diagrams put sequences use cases in context. Class diagrams summarize the major conceptual entities and the relationships between them.

Diagrams that capture requirements do so from an external or “black-box” perspective. These often make heavy use of textual notes and descriptions. Because they are sketches, they are not expected to be executable and can vary greatly in precision, level of detail, and completeness.

Consider the problem of deciding when and how an online bookstore processes back-orders. (A back-order is an order that is taken from a customer but that cannot be charged and shipped until the store has stock on hand.) A completely external view of the problem might

involve scenarios of the use cases “Order Merchandise” and “Receive Inventory” (figure). Alternatively, the requirements might be presented as a distinct use case “Order Back-Ordered Merchandise.” Each set of use cases may have their own textual description. However, this text is generally informal and can only be verified by reading and interpreting the text.

### UML for Describing Software

At the other end of the process are UML diagrams that describe the software to be (or that already has been) built. UML is all about objects, so it should only make sense that one can describe object-oriented software using UML diagrams. I sometimes refer to these types of diagrams as “graphical C++” or “graphical Java” because of their very tight coupling to the actual software implementation.

Class diagrams represent the software itself. In particular, these class diagrams may show how the specific code classes inherit from implementation technology base classes (e.g. `PublisherBean`, `ProductBean` inherit from `EJBEntityBean`). Sequence diagrams show how different software components interact (what calls what) to carry out some function. Deployment diagrams show how different tasks are allocated to different hardware, processes, and processors.

These sorts of diagrams serve two purposes: to plan the software before it is coded, and to document existing code. Of course, this presents the problem of having the same information in two different places (a topic I will cover in an upcoming column “One Fact in One Place”). Suffice it to say that there are several views on the value of maintaining diagrams that mirror code and the “reverse-engineering” tools that exist to support this work.

### What’s “In Between”

The informal sketches of the requirements do not contain sufficient information to be complete models of the application. Sketches can convey information, but models are testable. A testable model is one in which we

Copyright 2003 Marc J. Balcer and ModelCompilers.com. All Rights Reserved

Supplementary materials, including downloadable software and discussion groups, can be found online at <http://www.modelcompilers.com>.

can play through a scenario and verify that the model's behavior conforms to what is expected.

Once could argue that code does that. But MDA's concept of a platform-independent model is a model that can be understood completely in terms of the application subject matter. Executable PIMs can be tested and verified.

Why is this important? Presenting sketches such as those in (Figure) to the developers and expecting those developers to create the software from those sketches requires the developers to figure out both the software design issues as well as the application issues. Wouldn't it make more sense for the application experts to work out the application issues?<sup>1</sup>

## Solution Modeling

One might begin by characterizing requirements modeling as “external” and software modeling as “internal.” Alternatively, you might use the terms “problem” and “implementation.” The trouble is, this approach is a two-way partitioning of what is really a more complex problem.

To illustrate the differences between the types of models and to present the concept of a solution model, let's examine a simple problem. The requirements state,

If the merchandise is not in stock when the customer places his order, the order is “back-ordered” and will be shipped when available.

That requirement might be illustrated as use cases: (FIGURE)

But the use case diagram above is principally a “table of contents” document—the rules are totally not obvious. In fact, the one-sentence description above says more!

We can use an activity diagram to put the steps of the process into context. (Example)

Now at least the concept, “ship when stock is received” is obvious. A small class diagram can put the names of those obvious things into context. (example)

But all we've really done is to put the requirements descriptions into box-and-line drawings. These drawings do not at all address questions like these:

1. Can an order be split into multiple shipments?
2. Assuming there is more than one warehouse, what happens if there is not sufficient stock in one warehouse to fill an order, but there is suffi-

---

1. This argument is not taking sides in the debate over the merits of agile, incremental development approaches. Rather, the issues are how to formalize requirements and which group of people are best suited to do this job.

cient stock across several warehouses. Would the order be split? Can stock be sent between warehouses to balance orders?

3. When is the customer charged for merchandise on back-order?
4. How is the customer charged for the extra shipping costs of a split order?
5. Does the customer get a choice about splitting an order (to get more right away) or keeping it together (to minimize shipping costs)?

Note that these “detailed” questions differ from software design questions such as what persistence mechanisms to use, whether to use contrived primary keys in the database tables, and whether to make relationships uni- or bi-directional.

To get from requirements to software we need to create a *solution* that formalizes the application requirements in a testable model. This solution model is application-focused in that it focuses on the application problem free of software technology decisions. Yet even if you look at this merely as “formalizing” the requirements, this activity still requires creating abstractions—venting names for concepts or entirely new concepts. These abstractions may not always be obvious to users of the system (and they may not be explicitly stated in the requirements), but they are completely understandable by the business experts.

## Solution Models are Executable Models

For a model to be verifiable and testable, it must be capable of being executed like code. An executable UML [ref: Mellor/Balcer] model consists of three parts:

1. An information model in the form of a UML class diagram that identifies the classes, their attributes, and associations between the classes.
2. State models for each class in the form of UML statecharts that formalize the lifecycle of single instances of those classes,
3. Procedures for each state written in an action-semantics-compliant language. The procedure states what happens on entry to each state.

Executable UML uses other diagrams, but these are always derived from the basic three. For example, the event communication between state machines is summarized on a collaboration diagram and the specific sequence of events in a particular scenario is illustrated on a sequence diagram.

One of the hallmarks of Executable UML is that the models themselves must be complete. Informal sketches of the form in (figure) cannot be executed and are therefore not verifiable.

## Developing Solution Models

Simple elaborations of the project requirements do not lead to the most optimal software systems. The work to develop a platform-independent solution model requires abstraction skill and exposes detail about the problem. In upcoming columns, we will look at how the process of developing and testing a solution model helps to expose detail about the problem and ultimately leads to a more complete and correct software system.