AMERICAN PROGRAMMER®

UML: A Curse or a Blessing for the OO Community?

by Peter Hruschka

Most methodologies are short on "method" and long on "ology." When writing this article about the UML, I remembered this quote from the good old Yourdon days of structured methods. With UML, we now have an approach that is very short on method and long only on notation. This article discusses the advantages, disadvantages, experience, joy, and anger of and with the (still incomplete) UML and its predecessors.[1] The opinions I present are based on 20 years of experience in teaching and applying methods, from the first structured ideas onward to all the new object-oriented approaches. They also come from more than 15 years in the CASE tool business.

DOES THE WORLD NEED UML?

Depending on whom I talk to, I either curse or praise UML. In general, though, I am very positive about it. To understand why, we have to look back at the history of methods. In the '70s and '80s, we experienced the

[1]The points I make are based on all UML publications up to Version 0.91, as well as background material found in dozens of papers of our Rational amigos.

method wars between data modelers and function modelers. For a long time, people believed in either data flow diagrams *or* entity-relationship diagrams, but seldom both. Finally, toward the end of the '80s, people began talking to each other. Even the method gurus started to agree that using data models *and* function models *and* behavior models (or so-called real-time extensions) was a good idea for most projects. I call this approach "integrated structured methods," and it was (and is) good.

But just when we seemed to reach a consensus on the analysis and design approaches, some of the old — and many new — gurus started to preach the gospel of OO methods. The first reaction of the market and the CASE vendors was puzzlement, followed by chaos. It was not clear whom to follow. Dozens of notations, dozens of different explanations of what an object is and isn't, no stability, no trends visible.

It was only when Grady Booch and Jim Rumbaugh joined forces in late 1994 and announced a "unified method" that the average user saw light at the end of the OO tunnel. This unification effort is why I praise UML. The time is right to define strict semantics for the basic OO concepts and agree on definitions and notations. OO is no longer in its pioneer days. We have tried out basic principles long enough to have a deep understanding. A couple of years ago it was fine to experiment with different notations and different concepts, but now we need to settle on a few of them.

I sometimes curse UML, because in some aspects it seems to be a compromise between the interests of the methodologists and commercial interests. UML was not started from scratch but from competing methods, and every so often one can still see the compromises made in order to please the disciples (and the grand master) of one or the other school of thought.

THE BEST OF ALL POSSIBLE METHODS?

Will UML be a cure-all for every application area, the one and only approach to solving all our software problems? Will the other OO methods go away quickly? Certainly not! Nevertheless, UML will be one major notation. Two or three other methods will be available for a long time. I expect UML to be the COBOL of the OO methods, with an overwhelming market share. But as with programming languages, there is room for a few others.

UML FEATURES I LOVE

There are too many good features in UML to enumerate. So let me just mention some global concepts that I consider to be real steps forward:

★ The UML metamodel with clearly described semantical concepts and suggestions for standard notations. Let us hope that this will stop the invention of more and more undefined symbols and terms.

★ Class models (formerly "object models"); that is, classes with attributes and operations.

★ Associations between classes (finally with interpretable inscriptions, thanks to the little black arrowhead).

★ Special associations, such as inheritance and aggregation, denoted by special symbols. You really need them often, and therefore it is good to have symbols for them.

★ Readable multiplicity notation (numbers and wildcards) instead of coded symbols; rolenames, association classes, and many other good ideas from data modeling. The only minor detail that confuses many of my customers is the qualifiers, since they move good attributes of a class to the "wrong side" of the association (see "Navigation," below).

★ Explicit dynamic models: sometimes it is much easier to express parts of the requirements and design thoughts with such models.

★ Use cases as a starting point. Experience over the last few years has shown that customers are afraid of more formal models, but they love the informal use cases.

★ Explicit inclusion of distribution aspects in the deployment diagrams (formerly "platform diagrams"). This makes UML one of the few methods that take into account that our systems are more than structured pieces of code. It also opens up opportunities for extending the method for system engineering, including nodes that are not computers or devices but human operators, subsystems

using different technologies, and so on.

UML FEATURES THAT GIVE ME HEADACHES

Navigation

In the class model, I dislike one trend: the interpretation of associations. UML moves from seeing them as static, nondirectional relationships more and more toward (directional) navigation paths. By thinking too "directional" too early, you are losing some advantages that OO claims (extensibility, maintainability, etc.).

In Rumbaugh's OMT approach, they were still more symmetric. The more recent the UML document, the more directional they get, which is fine for an easy link with the final code in many cases. Still, it takes away some flexibility. A process for using UML should encourage analysts to achieve the higher level of abstraction with directionless associations, while only designers should add the navigability (including frequencies of usage, access rights, priorities, and the like). I have seen too many projects where missing the "backward direction" of an association in the analysis models led to expensive changes in the evolution or maintenance of systems.

A Surfeit of Diagrams

Dynamic models are a wonderful thing, as I mentioned above. But too many of them can make life hard. Think back to the days of real-time analysis. We had a process model (i.e., data flow diagrams) to express the functionality of the system and the causal dependencies (i.e., functions needing data as input from other functions and producing new data as output). Whenever this causality was not sufficient, we could add decision tables, process activation tables, or state diagrams to influence the execution of functions. That was sufficient to express behavior. Now we have got use cases, interaction diagrams in the form of either sequence diagrams or collaboration diagrams (formerly called event traces and object communication diagrams), state diagrams, and activity diagrams.

This means five diagrams to express dynamic behavior of the system instead of two. Of course these five diagrams have different uses: use cases are, among other things, a very informal starting point; both forms of interaction diagrams concentrate on special scenarios to model concrete sequences of messages between objects; state diagrams show the formal behavior of a single class; and activity diagrams show the detailed behavior of a single operation. Nevertheless, five seems too much, as I explain in the following.

Superfluous Activity Diagrams

The last ones, the activity diagrams, are needed as much as a goiter. When we were using assembly languages, it was helpful to describe a single algorithm with pseudocode or Nassi/Shneiderman charts. This was a higher level of abstraction than the language code. With the advent of higher languages, the usefulness of such detailed design notations declined.

When we decompose a system into proper objects with proper allocation of operations, it turns out that most operations are really simple and small, if not trivial. They do not require a graphical model to be understandable. More complex operations may be found for "control objects" (Ivar Jacobson's term), where sequencing, fork and join, alternatives, and loops could be needed.[2] A good process for UML would suggest starving the control object operations and delegating much of the work to entity objects, thus making operations of the control object simpler, too.

I can only imagine that the introduction of activity diagrams has been made to please Jim Odell or others, and I am afraid that the availability of these diagrams will lead to control-oriented models. Analysts and designers will be tempted to use them not for a single operation, but for the overall flow of the system. This brings us back to the days of flowcharts. I can already see the first extension suggested being the "goto," because some designers will surely miss that.

No Guidance for Allocating Messages to Objects

With the interaction diagrams, my experience shows a preference for sequence diagrams over collaboration diagrams. Most people still find it easier to think in terms of sequences than object communications.

The easy part of creating a sequence diagram is writing down

[2]The example used in Version 0.91 of preparing a beverage is such an operation involving multiple objects.

the special sequence of messages in a scenario; the hard part is assigning the messages to candidate objects. But that boils down to matching function view and data view, which was never easy. Beginners tend to introduce the "big master control object" or "user interface object" that handles most of the scenario, delegating little work to other objects. Good heuristics are needed in a process supplementing UML to help with the allocation of messages to objects.

Cluttered Diagrams, Blurred Distinctions

With the introduction of graphical elements for object creation, object deletion, (simple) alternatives, and recursion in these diagrams, it will be even more complicated to come up with readable interaction diagrams. My preference would have been to avoid some of these things in the graphics, leaving them for underlying textual message descriptions.[3]

Up to Version 0.8, one could teach that use cases are an abstraction of many related scenarios, while interaction diagrams precisely concentrate on one scenario with fixed decisions and fixed values in order to not miss the obvious cases. Now the scenarios allow for simple alternative paths, blurring the distinction between what should be modeled as a use case and what as a scenario.

I sincerely hope that the final definition of UML will be semantically precise on the relationship between sequence diagrams and collaboration diagrams, making these two just different views of the same thing, so that a CASE tool can automatically create one out of the other. I also hope it will specify what level of abstraction is appropriate for interaction diagrams to be used for analysis purposes and those to be used for design or coding purposes. Otherwise, it is left to methodologists designing a proper process for UML to tell users to avoid certain features at certain stages of a project.

Unnecessary Extensions to State Diagrams

Jim Rumbaugh has extended state-transition diagrams with many new concepts, and most of them seem to have made it into UML.[4] Some of the concepts (especially those taken from David Harel) are very useful, such as the nesting of states and concurrent substates, or the history construct. Some others are nice shortcuts but are not really necessary, such as the distinction between (long) activities and (short) actions. What is long and what is short? Or the entry and exit operations. Experience has shown that such extensions can create useless discussions instead of progress with the models. One should carefully examine where the line between necessary concepts and useful extension should be drawn. My feeling is that reducing the notation would be helpful.

No Structure for Use Case Descriptions

Use cases are a wonderful enrichment of the method. As soon as Ivar Jacobson's book was out [1], many users took up this idea. In addition, many competing methodologists adopted it when they found they could not kill it. Instead of saying that use cases are not object oriented, it was finally accepted that they are orthogonal to the class (or object) model. One criticism has remained: the way use cases are described is pretty informal. End users like this because it is their natural language and they can read and understand it (or even write it). Others miss certain style patterns that would distinguish the description of use cases from totally unstructured texts.

Before Jacobson joined the UML team, Jim Rumbaugh published some good papers suggesting more structure for use cases. Now we are back to, "Do what you like in a use case description." In Version 0.91, this reads like "can be . . . ," "several other means . . . ," "for example . . . ," instead of suggesting a way that it should be done. Furthermore, use cases are only superficially tied to the other modeling elements. A precise traceability definition is not yet given (see "Language vs. Method," below).

Experience in a large project (estimated to be around 1,000 person-years) has shown that the 500 use cases were much easier to write and check for quality once we introduced a description skeleton with headers such as: short de-

[3]I can hear Grady's reply to this: "Nobody forces you to use all these things. They are included in UML for those who want to show these aspects graphically."

[4]Note, however, that the metamodel does not yet include everything used in the examples in Version 0.8.

scription, starting event, preconditions, normal flow (with exceptions mentioned), exceptions, results, and postconditions. This did not reduce readability for normal mortals, but it improved the overall quality. It would be helpful for UML to strongly suggest such a description pattern, or even multiple patterns for the different applications of use cases over the life cycle.

UNDERESTIMATED AND DANGEROUS FEATURES OF UML

Packages

Packages were introduced in Version 0.91 as a means for grouping all the things you ever wanted to group for any reason. They are now replacing many other terms such as categories, subsystems, and the like. Version 0.91 recommends using stereotypes to indicate the different usages of the package construct (e.g., packaging classes, processors, nodes, or use cases). Since I consider the various reasons for packaging to be very different (e.g., packaging a group of use cases to allow for better project management, or packaging classes for logical reasons, or packaging modules for deployment), it may not be effective to make the distinction solely through stereotypes without suggesting symbols for them. I like the idea of packages, but I think this is still a weak point in UML.

The semantics of packages includes grouping, naming, and configuration control. UML also includes other forms of clusters, such as nodes, tasks (which are processes and threads), composite objects for distribution units, and so on. I think it will take a while to sort out all these terms for packaging, add more semantics to them, and clearly state which should be used for what purpose. This area was a weak point in Version 0.8; it has improved in Version 0.91, but we are not yet finished. While we had different terms for different concepts in the past, packages now try to unify many of them. I believe that we need some distinctions between packaging concepts, and these should clearly be extended into the notation.

Stereotypes

Stereotypes are a wonderful concept for making the semantic definition easier, but when overused as they are for packages, they may be confusing. The purpose of UML is to standardize something. With the introduction of stereotypes, this goal is somewhat endangered. The authors of UML already show an example in which CASE tool vendors can show an actor as (1) a normal class symbol of stereotype <actor> (as Rumbaugh would draw it), (2) a class symbol with a stick figure inside, or (3) only a stick figure, similar to Ivar Jacobson's notation. Now imagine creative users being allowed to design their own stereotype symbols for predefined concepts. This limits the portability of the notation. Why is it possible for an electrical engineer in China to interpret a plan drawn in Brazil? Because electrical engineers are not allowed to change the symbols for a transistor or a resistor ad lib.

Let's take another example from programming languages: Why was PL/I criticized a lot? Because the language allowed programmers to express identical things in many different ways, thus making it difficult to settle on one style and to exchange programs between programmers. Stereotypes with new symbols are acceptable for those situations in which one wants to extend the concepts beyond the predefined ones. For the predefined concepts, however, the notation should not be so liberal.[5]

LANGUAGE VS. METHOD

Having discussed some pros and cons of concepts and notational elements in UML, let us focus on the missing process part of UML. For many years, I have taught in my courses that a method is much more than just the notation you see. It includes guidelines and rules specifying how to think and work with the notation, and it includes management practices (e.g., guidelines concerning when to do what, when to stop, when to switch to another part of the approach, and so forth). I also teach that a good method needs time to ripen; just inventing a new notation is the easy part of the job.

When the UML project started, the goal was to come up with a "real, complete method" in the sense I have just described it. When you can't reach a goal in time, you have two possibilities: extend the schedule (UML has already extended its schedule and is still extending it) and/or lower the expectations about the result. Well, between Versions

[5]Even if this means that some of the amigos' older books would not be quite in agreement with UML anymore.

0.8 and 0.91, the latter also happened. Now we only get a notation, with more or less semantical underpinning. For applying the notation, we are to follow the motto, "As you like it."

I simply consider this chickenhearted. Of course, we have learned that developing a system is more complicated than most of the methods in the '70s and '80s supposed. Development is not as simple as the waterfall model suggested. With concurrent engineering, component use and reuse, standardized interfaces for parts of the system architecture, integration of legacy systems, and iterative and evolutionary life cycles, it is much harder to define processes nowadays. But certain rules, patterns, guidelines, and heuristics of processes can and should be defined.

Well, UML has deliberately left the process open. Nevertheless, the suggested standard could be more precise on the relationship between various parts of the notation, since they can be statically defined without looking at processes. Whenever it comes to mappings and traceability, the current version of UML flees into vague terms like "could be . . . ," "may be . . . ," or "sometimes it is useful . . . ." I hope the final version will be clear, for example, on the relationships between:

- ★ Use cases and scenarios
- ★ Scenarios and state diagrams
- ★ Logical class models and the various forms of packaging
- ★ Component diagrams and deployment diagrams

Either balancing rules or traceability rules should be established instead of blaming everything on a yet-undefined process.

CONCLUSION

UML has unified some of the existing OO notations, thus creating a single point of reference for many important concepts. While the OMT approach and the Jacobson method were simple in their notations, the Booch method could be called semantically rich or overly complex — depending on whether you liked it or not. If you watch the evolution of UML, you will notice that it becomes more complex from month to month, with more and more concepts being added to the graphical notation. As long as new and needed concepts are added this is OK, but we also see many alternative notations for the same concepts. This is understandable, since UML tries to please as many methodologists as possible. It is, however, a dangerous trend, as UML runs the risk of becoming another PL/I.

ACKNOWLEDGMENT

I personally have learned a lot from the work of Grady Booch, Ivar Jacobson, and Jim Rumbaugh, and I want to thank them for their courage in bringing their ideas together. So even if I dislike some of the detailed decisions, UML is a big step forward. I wish them (and all of us users of methods) luck and success.

REFERENCE

1 Jacobson, Ivar, et al. *Object-Oriented Software Engineering: A Use Case–Driven Approach.* Reading, MA: Addison-Wesley, 1992.

Peter Hruschka is a specialist in technology transfer for software and systems engineering. His work ranges from surveys to determine the capabilities of organizations, constructing strategic plans, and seminars and workshops for both structured and object-oriented methods, to coaching, consulting, and project reviews.

Dr. Hruschka received his masters degree and Ph.D. in computer science from the Technical University of Vienna. He started his career at GEI, a leading German systems and software house, in 1976. His first project dealt with the standardization of the German real-time language PEARL. From 1979 to 1982, he was head of the training department, and from his seminars came the idea to automate some of the methods. As a result of one of his papers, the successful CASE environment PROMOD was published in 1981.

Dr. Hruschka has published several books and many articles. These include My Way to CASE and CASE in Industrial Applications *(Carl Hanser-Verlag [in German]) and* Real-Time Systems — State of Practice *(John Wiley), as well as the translation of McMenamin and Palmer's* Essential Systems Analysis *and DeMarco and Lister's* Peopleware. *For many years, Dr. Hruschka served on the editorial board of the Springer magazine* Informatik — Forschung und Entwicklung *(Research and Development in Computer Science). Currently he is a member of the editorial board of the German* Objekt Spektrum, *where he writes a regular column*

on object-oriented analysis, design, and management.

When he is not working, he is usually found with his wife in some of the most scenic parts of world, trying to hit little white balls into holes that are far too small.

Dr. Hruschka can be reached at Atlantic Systems Guild, Langenbruchweg 71, D-52080 AACHEN, Germany (+49 241 16 56 70; fax +49 241 96 21 50; e-mail: 100064.2760@compuserve.com). ★

This article originally appeared in Vol. 10, no. 3 of ***AMERICAN PROGRAMMER***®.

The monthly ***AMERICAN PROGRAMMER***® Newsletter is edited by Ed Yourdon and published by:

CUTTER INFORMATION CORP.
37 Broadway, Suite 1, Arlington, MA 02174-5552, USA. Phone: (617) 648-8702 or (800) 964-8702,
Fax: (617) 648-1950, E-mail: lovering@cutter.com, Web site: http://www.cutter.com