Foreword to the Third Edition

Bruce Douglass' book has only improved with time. The main change, of course, is that it now caters to the new version of the the language, UML 2.0. As with the first edition, this edition too is one of the clearest and most valuable texts for engineers who want to model and specify systems using the UML, especially reactive and real-time ones. Hence, I applaud Bruce for updating the text and presenting to the public another valuable product of his prolific pen (keyboard? . . .).

Still, I should say a few words about the UML itself, especially relating to the following two passages from the earlier foreword—one a prediction and one an opinion:

> *The recent wave of popularity that the UML is enjoying will bring with it not only the official UML books written by Rational Corporation authors, but a true flood of books, papers, reports, seminars, and tools, describing, utilizing, and elaborating upon the UML, or purporting to do so. Readers will have to be extra careful in finding the really worthy trees in this messy forest. I have no doubt that Bruce's book will remain one of those. . . .*
>
> *Despite all of this, one must remember that right now UML is a little too massive. We understand well only parts of it; the definition of other parts has yet to be carried out in sufficient depth to make crystal clear their relationships with the constructive core of UML (the class diagrams and the statecharts). . . .*

As to the first of these quotes, it wasn't too hard to predict the flood, which has materialized above all expectations. Here is one small statistic: A search at amazon.com for books with "UML" in the title results in 213 items, and the same search limited to 1998 and on yields 198 items. That is, there were 15 UML books when the first edition of this book was published, and there are some 200 more now! Nevertheless, I maintain that Bruce's book indeed remains one of the few really worthy ones.

As to the second remark, about the UML being a little too massive, things have not really improved much. With version 2.0 almost ready to be launched, a fact that is doubtless a milestone in the development of the UML, we may ask ourselves whether it has become leaner and meaner, or larger and messier. Many people hoped that a new version of of something that was so multifaceted and complex, but which had been adopted as a standard to be used worldwide, would concentrate on its most important aspects. It would improve and sharpen them and narrow down or discard those things that turned out to be inessential or less well-defined. This could have resulted in a language that was easier to learn, easier to use, easier to implement responsibly, and thus would carry a lot more "punch." While UML 2.0 contains several exciting new features, especially for the realm relevant to this book—real-time and reactive systems—the new version of UML is larger and more complex.

As mentioned in the 1997 foreword, object-orientation is here to stay and so is the UML, probably in a big way. Let us thus hope that version 3.0 of the language will remove, intergrate, clarify, and solidify more than it adds. In any case, good books about a language are almost as important as the language itself, and in this respect the present book is one of only a handful that can be heartily recommended.

David Harel
The Weizmann Institute of Science
Rehovot, Israel
November 2003

Foreword to the Previous Editions

Embedded computerized systems are here to stay. Reactive and real-time systems likewise. As this book aptly points out, one can see embedded systems everywhere; there are more computers hidden in the guts of things than there are conventional desktops or laptops.

Wherever there are computers and computerized systems, there has to be software to drive them—and software doesn't just happen. People have to write it, people have to understand and analyze it, people have to use it, and people have to maintain and update it for future versions. It is this human aspect of programming that calls for modeling complex systems on levels of abstraction that are higher than that of "normal" programming languages. From this also comes the need for methodologies to guide software engineers and programmers in coping with the modeling process itself.

There is broad agreement that one of the things to strive for in devising a high-level modeling approach is good diagrammatics. All other things being equal, pictures are usually better understood than text or symbols. But we are not interested just in pictures or diagrams, since constructing complex software is not an exclusively human activity. We are interested in *languages* of diagrams, and these languages require computerized support for validation and analysis. Just as high-level programming languages require not only editors and version control utilities but also—and predominantly!—compilers and debugging tools, so do modeling languages require not only pretty graphics, document generation utilities, and project management aids, but also means for executing models and for synthesizing code. This means that

we need *visual formalisms* that come complete with a syntax to determine what is allowed and semantics to determine what the allowed things mean. Such formalisms should be as visual as possible (obviously, some things do not lend themselves to natural visualization) with the main emphasis placed on topological relationships between diagrammatic entities, and then, as next-best options, geometry, metrics, and perhaps iconics, too.

Over the years, the main approaches to high-level modeling have been *structured analysis* (SA), and *object orientation* (OO). The two are about a decade apart in initial conception and evolution. SA started in the late 1970s by DeMarco, Yourdon, and others, and is based on "lifting" classical, procedural programming concepts up to the modeling level. The result calls for modeling system structure by functional decomposition and flow of information, depicted by (hierarchical) data-flow diagrams. As to system behavior, the early- and mid-1980s saw several methodology teams (such as Ward/Mellor, Hatley/Pirbhai, and the STATEMATE team from i-Logix) making detailed recommendations that enriched the basic SA model with means for capturing behavior based on state diagrams or the richer language of statecharts. Carefully defined behavioral modeling is especially crucial for embedded, reactive, and real-time systems.

OO modeling started in the late 1980s, and, in a way, its history is very similar. The basic idea for system structure was to "lift" concepts from object-oriented programming up to the modeling level. Thus, the basic structural model for objects in Booch's method, in the OMT and ROOM methods, and in many others, deals with classes and instances, relationships and roles, operations and events, and aggregation and inheritance. Visuality is achieved by basing this model on an embellished and enriched form of entity-relationship diagrams. As to system behavior, most OO modeling approaches adopted the statecharts language for this (I cannot claim to be too upset about that decision). A statechart is associated with each class, and its role is to describe the behavior of the instance objects. The subtle and complicated connections between structure and behavior—that is, between object models and statecharts—were treated by OO methodologists in a broad spectrum of degrees of detail—from vastly insufficient to adequate. The test, of course, is whether the languages for structure and behavior and their interlinks are defined sufficiently well to allow the "interpretation" and "compilation" of high-level models—full model execution

and code synthesis. This has been achieved only in a couple of cases, namely in the ObjecTime tool (based on the ROOM method of Selic, Gullekson, and Ward), and the Rhapsody® tool (from i-Logix, based on the Executable Object Modeling method of Gery and the undersigned).

In a remarkable departure from the similarity in evolution between the SA and OO paradigms for system modeling, the last two or three years have seen OO methodologists working together. They have compared notes, debated the issues, and finally cooperated in formulating a general Unified Modeling Language (UML) in the hope of bringing together the best of the various OO modeling approaches. This sweeping effort, which in its teamwork is reminiscent of the Algol60 and Ada efforts, is taking place under the auspices of Rational Corporation, spearheaded by G. Booch (of the Booch method), J. Rumbaugh (co-developer of the OMT method), and I. Jacobson (czar of use cases). Version 0.8 of the UML was released in 1996 and was rather open-ended, vague, and not nearly as well defined as some expected. For about a year, the UML team went into overdrive, with a lot of help from methodologists and language designers from outside Rational Corporation (the undersigned contributing his 10 cents worth, too), and version 1.1, whose defining documents were released in early 1997, is much tighter and more solid. The UML has very recently been adopted as a standard by the object management group (OMG), and with more work there is a good chance that it will become not just an officially approved, if somewhat dryly documented, standard, but the main modeling mechanism for the software that is constructed according to the object-oriented doctrine. And this is no small matter, as more software engineers are now claiming that more kinds of software are best developed in an OO fashion.

For capturing system structure, the UML indeed adopts an entity-relationship-like diagrammatic language for classes and objects. For early-stage behavioral thinking it recommends use cases and utilizes sequence diagrams (often called message sequence charts or MSCs). For the full constructive specification of behavior it adopts statecharts.

In this book, Bruce Douglass does an excellent job of dishing out engineering wisdom to people who have to construct complex software—especially real-time, embedded, reactive software. Moreover, it does this with UML as the main underlying vehicle, a fact which, given the recent standardization of the UML and its fast-spreading usage, makes the book valuable to anyone whose daily worry is the expeditious and

smooth development of such systems. Moreover, Bruce's book is clear and very well written, and it gives the reader the confidence boost that stems from the fact that the author is not writing from the ivy-clouded heights of an academic institution or the religiously-tainted vantage point of a professional methodologist, but that he has extensive experience in engineering the very kinds of systems the book discusses.

The recent wave of popularity that the UML is enjoying will bring with it not only the official UML books written by Rational Corporation authors, but a true flood of books, papers, reports, seminars, and tools, describing, utilizing, and elaborating upon the UML, or purporting to do so. Readers will have to be extra careful in finding the really worthy trees in this messy forest. I have no doubt that Bruce's book will remain one of those.

Despite all of this, one must remember that right now UML is a little too massive. We understand well only parts of it; the definition of other parts has yet to be carried out in sufficient depth to make crystal clear their relationships with the constructive core of UML (the class diagrams and the statecharts). For example, use cases and their associated sequence and collaboration diagrams are invaluable to users and requirements engineers trying to work out the system's desired behavior in terms of scenarios. In the use case world we describe a single scenario (or a single cluster of closely related scenarios) for all relevant objects—we might call it *interobject behavior.* In contrast, a statechart describes all the behavior for a single object—*intraobject behavior.* I would like to term this stark difference as *the grand duality of system behavior.* We are far from having a good algorithmic understanding of this duality. We don't know yet how to derive one view from the other, or even how to efficiently test whether descriptions presented in the two are mutually consistent.

Other serious challenges remain, for which only the surface has been scratched. Examples include true formal verification of object-oriented software modeled using the high-level means afforded by the UML, automatic eye-pleasing and structure-enhancing layout of UML diagrams, satisfactory ways of dealing with hybrid systems that involve discrete as well as continuous parts, and much more.

As a general means for dealing with complex software, OO is also here to stay, and hence, so is the UML. OO is a powerful and wise way to think about systems and to program them, and will for a long time to

come be part and parcel of the body of knowledge required by any self-respecting software engineer. This book will greatly help in that. On the other hand, OO doesn't solve *all* problems, and hence, neither does the UML. There is still much work to be done. In fact, it is probably no great exaggeration to say that there is a lot more that we *don't* know and *can't* achieve yet in this business than what we do and can. Still, what we have is tremendously more than we would have hoped for five years ago, and for this we should be thankful and humble.

David Harel
The Weizmann Institute of Science
Rehovot, Israel
October 1997