1

Introduction

Organizations want systems. They don't want processes, meetings, models, documents, or even code.[1] They want systems that work—as quickly as possible, as cheaply as possible, and as easy to change as possible. Organizations don't want long software development lead-times and high costs; they just want to reduce systems development hassles to the absolute minimum.

But systems development is a complicated business. It demands distillation of overlapping and contradictory requirements; invention of good abstractions from those requirements; fabrication of an efficient, cost-effective implementation; and clever solutions to isolated coding and abstraction problems. And we need to manage all this work to a successful conclusion, all at the lowest possible cost in time and money.

None of this is new. Over thirty years ago, the U.S. Department of Defense warned of a "software crisis" and predicted that to meet the burgeoning need for software by the end of the century, everyone in the country would have to become a programmer. In many ways this prediction has come true, as anyone who has checked on the progress of a flight or made a stock trade using the Internet can tell you. Nowadays, we all write our own

[1] Robert Block began his book *The Politics of Projects*[1] in a similar manner.

programs by filling in forms—at the level of abstraction of the application, not the software.

1.1 Raising the Level of Abstraction

The history of software development is a history of raising the level of abstraction. Our industry used to build systems by soldering wires together to form hard-wired programs. Machine code allowed us to store programs by manipulating switches to enter each instruction. Data was stored on drums whose rotation time had to be taken into account so that the head would be able to read the next instruction at exactly the right time. Later, assemblers took on the tedious task of generating sequences of ones and zeroes from a set of mnemonics designed for each hardware platform.

Later, programming languages, such as FORTRAN, were born and "formula translation" became a reality. Standards for COBOL and C enabled portability between hardware platforms, and the profession developed techniques for structuring programs so that they were easier to write, understand, and maintain. We now have languages such as Smalltalk, C++, Eiffel, and Java, each with the notion of object-orientation, an approach for structuring data and behavior together into classes and objects.

As we moved from one language to another, generally we increased the level of abstraction at which the developer operates, requiring the developer to learn a new higher-level language that may then be mapped into lower-level ones, from C++ to C to assembly code to machine code and the hardware. At first, each higher layer of abstraction was introduced only as a concept. The first assembly languages were no doubt invented without the benefit of an (automated) assembler to turn the mnemonics into bits, and developers were grouping functions together with the data they encapsulated long before there was any automatic enforcement of the concept. Similarly, the concepts of structured programming were taught before there were structured programming languages in widespread industrial use (*pace*, Pascal).

Layers of Abstraction and the Market

The manner in which each higher layer of abstraction reached the market follows a pattern. The typical response to the introduction of the next layer of abstraction goes something like this: "Formula translation is a neat trick, but even if you can demonstrate it with an example, it couldn't possibly work on a problem as complex and intricate as mine."

As the tools became more useful and their value became more obvious, a whole new set of hurdles presented themselves as technical folk tried to acquire the wherewithal to purchase the tools. Now managers wanted to know what would happen if they came to rely on these new tools. How many vendors are there? Are other people doing this? Why should we take the risk in being first? What happens if the compiler builder runs out of business? Are we becoming too dependent on a single vendor? Are there standards? Is there interchange?

Initially, it must be said, compilers generated inefficient code. The development environment, as one would expect, comprised a few, barely production-level tools. These were generally difficult to use, in part because the producers of the tools focused first on bringing the technology to market to hook early adopters, and later on prettier user interfaces to broaden that market. The tools did not necessarily integrate with one another. When programs went wrong, no supporting tools were available: No symbolic debuggers, no performance profiling tools, no help, really, other than looking at the generated code, which surely defeated the whole purpose.

Executable UML and the tooling necessary to compile and debug an executable UML model are only now passing from this stage, so expect some resistance today and much better tools tomorrow.

But over time the new layers of abstraction became formalized, and tools such as assemblers, preprocessors, and compilers were constructed to support the concepts. This has the effect of hiding the details of the lower layers so that only a few experts (compiler writers, for example) need concern themselves with the details of how that layer works. In turn, this raises concerns about the loss of control induced by, for example, eliminating the GOTO statement or writing in a high-level language at a distance from the "real machine." Indeed, sometimes the next level of abstraction has been too big a reach for the profession as a whole, of interest to academics and purists, and the concepts did not take a large enough mindshare to survive. (ALGOL-68 springs to mind. So does Eiffel, but it has too many living supporters to be a safe choice of example.)

Object Method History

Object methods have a complex history because they derive from two very different sources.

One source is the programming world, whence object-oriented programming came. Generalizing shamelessly, object-oriented programmers with an interest in methods were frustrated with the extremely process-oriented perspective of "structured methods" of the time. These methods, Structured Analysis and Structured Design, took functions as their primary view of the system, and viewed data as a subsidiary, slightly annoying, poor relation. Even the "real-time" methods at most just added state machines to the mix to control processing, and didn't encapsulate at all. There was a separate "Information Modeling" movement that was less prominent and which viewed data as all, and processing as a nuisance to be tolerated in the form of CRUD++. Either way, both of these camps completely missed the object-oriented boat. To add insult to injury, one motivation for objects—the notion that an object modeled the real world, and then seamlessly became the software object—was prominently violated by the emphasis in transforming from one (analysis) notation, data flow diagrams, to another (design) notation, the structure chart.

Be that as it may, the search was on for a higher level of abstraction than the programming language, even though some claimed that common third-generation programming languages such as Smalltalk had already raised the level of abstraction far enough.

The other source was more centered in analysis. These approaches focused on modeling the concepts in the problem, but in an object-oriented way. Classes could be viewed as combinations of data, state, and behavior at a conceptual level only. In addition to the model, reorganization of "analysis" classes into "design" classes, and re-allocation of functionality were expected. There was no need to model the specific features used from a programming language because the programmer was to fill in these details. Perhaps the purest proponents of this point of view were Shlaer and Mellor. They asserted classes with attributes clearly visible on the class icon seemingly violating encapsulation, with the full expectation that object-oriented programming schemes would select an appropriate private data structure with the necessary operations.

These two sources met in the middle to yield a plethora of methods, each with its own notation (at least 30 published), each trying to some extent to meet the needs of both camps. Thus began the Method Wars, though Notation Wars might be more accurate.

UML is the product of the Method Wars. It uses notations and ideas from many of the methods extant in the early nineties, sometimes at different levels of abstraction and detail.

As the profession has raised the level of abstraction at which developers work, we have developed tools to map from one layer to the next automatically. Developers now write in a high-level language that can be mapped to a lower-level language automatically, instead of writing in the lower-level language that can be mapped to assembly language, just as our predecessors wrote in assembly language and translated that automatically into machine language.

Clearly, this forms a pattern: We formalize our knowledge of an application in as high a level language as we can. Over time, we learn how to use this language and apply a set of conventions for its use. These conventions become formalized and a higher-level language is born that is mapped automatically into the lower-level language. In turn, this next-higher-level language is perceived as low level, and we develop a set of conventions for its use. These newer conventions are then formalized and mapped into the next level down, and so on.

1.2 Executable UML

Executable UML is at the next higher layer of abstraction, abstracting away both specific programming languages and decisions about the organization of the software so that a specification built in Executable UML can be deployed in various software environments without change.

Physically, an Executable UML specification comprises a set of models represented as diagrams that describe and define the conceptualization and behavior of the real or hypothetical world under study. The set of models, taken together, comprise a single specification that we can examine from several points of view. There are three fundamental projections on the specification, though we may choose to build any number of UML diagrams to examine the specification in particular ways.

The first model identifies, classifies, and abstracts the real or hypothetical world under study, and it organizes the information into a formal structure. Similar "things," or *objects*, in the subject matter under study are identified and abstracted as *classes*; characteristics of these objects are abstracted as *attributes*; and reliable associations between the objects are abstracted as *relationships*.

Concept	Called	Modeled As	Expressed As
the world is full of things	data	classes attributes associations constraints	UML class diagram
things have lifecycles	control	states events transitions procedures	UML statechart diagram
things do things at each stage	algorithm	actions	action language

Figure 1.1 *Concepts in an Executable UML Model*

Operations do not appear explicitly as entries in Figure 1.1 because Executable UML derives operations from actions on state machines.

Invoked actions may be shown as operations on classes, but their existence is normally dependent on the invocation that occurs in a state machine.

We express this first model using a *UML class diagram*. The abstraction process requires that each object be subject to and conform to the well-defined and explicitly stated rules or policies of the subject matter under study, that attributes be abstractions of characteristics of things in the subject matter under study, and that relationships similarly model associations in the subject matter.

Next, the objects (the instances of the classes) may have *lifecycles* (behaviors over time) that are abstracted as state machines. These state machines are defined for classes, and expressed using a *UML statechart diagram*. The abstraction process requires that each object be subject to and conform to the well-defined and explicitly stated rules or policies of the world under study, so each object is known to exhibit the same pattern of behavior.

The behavior of the system is driven by objects moving from one stage in their lifecycles to another in response to events. When an object changes

Executable UML is a single language in the UML family, designed for a single purpose: to define the semantics of subject matters precisely. Executable UML is a particular usage, or *profile*, the formal manner in which we specify a set of rules for how particular elements in UML fit together for a particular purpose.

This book, then, describes a profile of UML for execution.

state, something must happen to make this new state be so. Each state machine has a set of *procedures,* one of which is executed when the object changes state, thus establishing the new state.

Each procedure comprises a set of *actions*. Actions carry out the fundamental computation in the system, and each action is a primitive unit of computation, such as a data access, a selection, or a loop. The UML only recently defined a semantics for actions, and it currently has no standard notation or syntax, though several (near-)conforming languages are available.

These three models—the class model, the state machines for the classes, and the states' procedures—form a complete definition of the subject matter under study. Figure 1.1 describes the concepts in an Executable UML model.

In this book we will informally make use of other UML diagrams, such as use case and collaboration diagrams, that support the construction of executable UML models or can be derived from them. We encourage using any modeling technique, UML-based or otherwise, that helps build the system.

1.3 Making UML Executable

Earlier versions of UML were not executable; they provided for an extremely limited set of actions (sending a signal, creating an object, destroying an object, as well as our personal favorite, "uninterpreted string"). In late 2001, the UML was extended by a semantics for actions. The action semantics provides a complete set of actions at a high level of abstraction. For example, actions are defined for manipulating collections

Executable UML isn't just a good idea, it's real. There are several Executable UML vendors, and the models in this book have been executed to ensure they are correct. The case study models and the toolset are downloadable.

For the latest information on executable UML, go to http://www.executableumlbook.com.

of objects directly, thus avoiding the need for explicit programming of loops and iterators. Executable UML relies on these new actions to be complete.

For UML to be executable, we must have rules that define the dynamic semantics of the specification. Dynamically, each object is thought of as executing concurrently, asynchronously with respect to all others. Each object may be executing a procedure or waiting for something to happen to cause it to execute. Sequence is defined for each object separately; there is no global time and any required synchronization between objects must be modeled explicitly.

The existence of a defined dynamic semantics makes the three models computationally complete. A specification can therefore be executed, verified, and translated into implementation.

Executable UML is designed to produce a comprehensive and comprehensible model of a solution without making decisions about the organization of the software implementation. It is a highly abstract thinking tool to aid in the formalization of knowledge, a way of thinking about and describing the concepts that make up an abstract solution to a client problem.

Executable UML helps us work out how we want to think about a solution: the terms we need to define, the assumptions we make in selecting those terms, and the consistency of our definitions and assumptions. In addition, executable UML models are separate from any implementation, yet can readily be executed to test for completeness and correctness.

Most important of all, together with a model compiler, they are executable.

1.4 Model Compilers

At some level, it is fair to say that any language that can be executed is necessarily a programming language; it's just a matter of the level of abstraction. So, is executable UML yet another (graphical) programming language?

An executable UML model completely specifies the semantics of a single subject matter, and in that sense, it is indeed a "program" for that subject matter. There is no magic. Yet an executable UML model does not specify many of the elements we normally associate with programming today. For example, an executable UML model does not specify distribution; it does not specify the number and allocation of separate threads; it does not specify the organization of data; it does not even require implementation as classes and objects. All of these matters are considered decisions that relate to hardware and software organization, and they have no place in a model concerned with, say, the purchase of books online.

Decisions about the organization of the hardware and software are abstracted away in an executable UML model, just as decisions about register allocation and stack/heap organization are abstracted away in the typical compiler. And, just as a typical language compiler makes decisions about register allocation and the like for a specific machine environment, so does an executable UML model compiler make decisions about a particular hardware and software environment, deciding, for example, to use a distributed Internet model with separate threads for each user window, HTML for the user interface displays, and so on.

An executable UML *model compiler* turns an executable UML model into an implementation using a set of decisions about the target hardware and software environment.

There are many possible executable UML model compilers for different system architectures. Each architecture makes its own decisions about the organization of hardware and software, including even the programming language. Each model compiler can compile any executable UML model into an implementation.

The notion of so many different model compilers for such different software architecture designs is a far cry from the one-size-fits-all visual modeling tools of the past.

Here are some examples of possible model compilers:

1. Multi-tasking C++ optimized for embedded systems, targeting Windows, Solaris, and various real-time operating systems. [3]
2. Multi-processing C++ with transaction safety and rollback. [2]
3. Fault-tolerant, multi-processing C++ with persistence supporting three processor types and two operating systems.
4. C straight on to an embedded system, with no operating system. [3]
5. C++, widely distributed discrete-event simulation, Windows, and UNIX.
6. Java byte code for single-tasking Java with EJB session beans and XML interfaces.
7. Handel-C and C++ for system-level hardware/software development.
8. A directly executing executable UML virtual machine.

A single model compiler may employ several languages or approaches to problems such as persistence and multi-tasking. Then, however, the several approaches must be shown to fit together into a single, coherent whole.

Of these, some are commercially available, as indicated by the references provided above, and some are proprietary, built specifically to optimize a property found in related systems produced by a company, such as the fault-tolerant multi-processing model compiler. Some are still just prototypes or twinkles in our eyes, such as the last three.

As a developer, you will build an executable UML model that captures your solution for the subject matter under study, purchase a model compiler that meets the performance properties and system characteristics you require, and give directives to the compiler for the particular application. Hence, a system that must control a small robot would select the small footprint C model compiler or one like it, and a system executing

financial transactions would prefer one with transaction safety and roll-back.

The performance of the model compiler may depend on the allocations of application model elements, and a model compiler may not know enough to be able to allocate a particular class to that task or processor for the best performance. Similarly, a model compiler that provides persistence may not know enough about your subject matter to determine what to make persistent. Consequently, you will also need to provide model compiler–specific configuration information. Each feature provided by the model-compiler that does not have a direct analog in executable UML will require directives to determine which feature to use.

These choices will affect performance of the model compiler. One particularly performance-sensitive feature is static allocation to tasks and processors. Allocating two classes that communicate heavily with different processors could cause significant degradation of network performance and of your system. If this is so, of course, it's a simple matter to re-allocate the elements of the model and recompile. This is why executable UML is so powerful—by separating the model of the subject matter from its software structure, the two aspects can be changed independently, making it easier to modify one without adversely affecting the other. This extends the Java notion of the "write once, run anywhere" concept; as we raise the level of abstraction, we also make our programs more portable. It also enables a number of interesting possibilities for hardware-software co-design.

1.5 Model-Driven Architecture

Executable UML is one pillar supporting the Model-Driven Architecture (MDA) initiative announced by the Object Management Group (OMG) in early 2001, the purpose of which is to enable specification of systems using models.

Model-driven architecture depends on the notion of a Platform-Independent Model (PIM), a model of a solution to a problem that does not rely on any implementation technologies. A PIM is independent of its platform(s).

A model of a online bookstore, for example, is independent of the user interface and messaging services it employs.

A PIM can be built using an executable UML.

Some proponents of MDA hold that a specification of the interface in a language such as the OMG's Interface Description Language (IDL), plus some constraints, is sufficient to specify without overspecifying. The views of these two camps are not contradictory, but complementary. There is no technical reason why a PIM specified using an executable UML cannot be bridged to one specified in terms of interfaces and constraints. One is just a more complete version of the other.

It is because an executable model is required as a way to specify PIMs completely that we view an executable UML as a foundation of model-driven architectures.

MDA also defines the concept of a Platform-Specific Model (PSM): a model that contains within it the details of the implementation, enough that code can be generated from it. A PSM is produced by weaving together the application model and the platforms on which it relies. The PSM contains information about software structure, enough information, possibly, to be able to generate code. Executable UML views the PSM as an intermediate graphical form of the code that is dispensable in the case of complete code generation.

At the time of writing, MDA is still being defined. However, some variation of the concepts of executable UML will, in our opinion, be required to support MDA. We offer our view on executable UML concepts here. Describing and defining MDA is another project and another book.

1.6 References

[1] Block, Robert: *The Politics of Projects*. Yourdon Press, New York, NY, 1983.

[2] Kabira Technologies URL: *www.kabira.com*

[3] Project Technology, Inc. URL: *www.projtech.com*