Models to Code

With No Mysterious Gaps

Leon Starr
Andrew Mangogna
Stephen Mellor

Apress®

Models to Code

With No Mysterious Gaps

Leon Starr
Andrew Mangogna
Stephen Mellor

Apress®

Models to Code

Leon Starr
San Francisco, California, USA

Andrew Mangogna
Nipomo, California, USA

Stephen Mellor
San Francisco, California, USA

ISBN-13 (pbk): 978-1-4842-2216-4 ISBN-13 (electronic): 978-1-4842-2217-1
DOI 10.1007/978-1-4842-2217-1

Library of Congress Control Number: 2017944371

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Louise Corrigan
Development Editor: James Markham
Coordinating Editor: Nancy Chen
Copy Editor: Sharon Wilkey
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484222164. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper.

To Kristina and Marjorie

Contents at a Glance

Contents

About the Authors

Leon Starr has been developing real-time distributed and embedded software with object-oriented, executable models since 1984. His models have been used in fighter jets, factory material transport control systems, ultrasound diagnostic and cardiac pacing systems, gas chromatography and semiconductor wafer inspection systems, video postproduction systems, and networked military battle simulators. He has taught numerous courses on executable systems and data modeling to systems engineers and software developers worldwide through his company Model Integration, LLC (`modelint.com`) based in San Francisco, California. He is the author of *How to Build Shlaer-Mellor Object Models*, *How to Build Class Models*, *Executable UML: A Case Study*, and assorted papers at `uml.org` and `modeling-languages.com`. He regularly assists project teams who model complex requirements and generate code from those models for challenging hardware and software platforms.

Andrew Mangogna has developed a number of open source tools, including pycca, which we use in this book to specify Executable UML models and translate them into code. He has also worked extensively in the medical device community. For more than 30 years, Andrew has been a hands-on builder of embedded software systems. He has worked in application areas ranging from laboratory instrumentation, remote data collection, and video special effects, to implantable medical devices. Andrew has always had a special interest in applying more formal techniques to the challenge of engineering software to create systems in a cost-effective manner with demonstrable quality. Trained in the basics of modeling by Stephen Mellor himself, he has successfully applied executable modeling techniques and model translation to many projects and has written several tools to help automate the translation process. With a keen interest in technology and a practical realization of the benefits of modeling, he has a mastery of mapping models to appropriate implementation technology to obtain high-quality software systems.

Stephen Mellor is the primary author of *Executable UML: A Foundation for Model-Driven Architecture,* as well as several other books on model-driven software. He is a frequent speaker and a key contributor to many modeling language standards established by the Object Management Group (OMG).

Acknowledgments

Any substantial project always depends on the generous help of others. Friends, colleagues, and a host of experiences have provided valuable service to *Models to Code*.

We all wish to thank Ed Seidewicz, our technical reviewer, for his many insightful and detailed comments throughout the text. His eye for detail rooted out quite a few potential and devilishly subtle errors. He passed many of our "tests" to see whether he was actually reading this stuff. We especially appreciate his patience with us, as we are well aware that our opinions and experience are often at odds with current practice in the greater UML community.

I want to express my heartfelt thanks to friends and colleagues who expressed early enthusiasm for this project. This includes many of my colleagues at SAAB Aerospace here in Sweden. At the Por Que No? taqueria in Portland, Oregon in March of 2016, Dan George and I had some amazing tacos, and I was encouraged by his infectious comments and enthusiasm. Back here in Linköping, Sweden, my present home base, I enjoyed many beers, laughs, and technical discussions with Nils Paulsson of SAAB at The Bishops Arms pub. He also contributed helpful comments on early drafts (in both the writing and beer sense of the word). Thanks also to my Scandinavian "agent" and good friend, Christer Andersson, for equal measures of encouragement and pestering me to get this project over with so I could get back to doing "real" work. My longtime friend and business partner, Michael M. Lee, has served many a quality margarita on my recurring visits back to the Bay Area in California. I would like to thank the staff at Babbettes Kafferie in Linköping for letting me use a corner of their cafe as my second office and keeping me well supplied with coffee, morning buns, and warm conversation during the dark and cold Swedish winter days. Most important, though, I owe a massive debt of gratitude to my wife, Kristina, who set me up with a beautiful home office and forbade me from getting a new Xbox until the book was complete and for putting up with my writing throughout our summer wedding plans. Along with Andrew's superb project management skills, you can thank her the most for this book making it to the shelves in 2017.

—Leon Starr

The many discussions over coffee and beer of modeling and its foundations with Paul Higham were fundamental to a broader understanding of how all the pieces fit together. There were many people, such as Melanie Gurunathan, Cary Campbell, and Thomas Brennan-Marquez, who actually used pycca to deliver real products and whose feedback was essential. Finally, without the patience and support of my wife, Marjorie Lane-Mangogna, it is hard to envision how this book could ever have happened. Few would have the patience to deal with the blah, blah, blah chatter that surrounds someone immersed in writing on a technical subject.

—Andrew Mangogna

Foreword

I published my first book on requirements analysis and modeling back in 1997. It was based on my experience working with teams to build complex, real-time, model-driven software. Like so many other books on software modeling, I left the process of transforming the models into code as an exercise for the student. Ha!

In fact, on actual projects, we were able to make this transformation, but it certainly wasn't easy. Worthwhile, yes. The models proved themselves again and again as being worth the effort. But the code translation problem has never been trivial. We always required a small team dedicated to the translation task running in parallel with the modeling effort. Note that this was not a waterfall approach, as modeling and design were being accomplished simultaneously.

Over time, the tools and technology to bridge the path from models to code improved. Unfortunately, they were locked up in proprietary solutions and expensive licensed products. This put me in an awkward position when extolling the benefits of modeling to younger engineers. Getting them interested in the benefits of modeling and a more promising style of model execution and code generation was easy. I would then be confronted with the inevitable question, "That's sounds great; how do I get started with this stuff?" I then had to concede that, while you might toy around with the ideas on your own, you really needed to get attached to a project that could fund the required tooling. Consequently, I didn't want to write another book without giving the reader a real path, any path, that took the models into running code without requiring the purchase of any expensive tools.

Andrew and I have worked together on and off on various projects over the years. One of his early translation systems extracted models from a tool called BridgePoint at the time (now xtUML) and transformed those models directly into assembly language. Because of the specialized CPU, no sufficiently optimal C compiler was available. It was a proprietary solution for a particular embedded device company, but it was a compelling proof of concept. Abstract models could be compacted and reorganized directly to the machine instruction level and run on specialized and limited resource hardware to yield the precise behavior specified by the models.

Andrew has since moved on to build a number of open source translation systems of which one, pycca, is used in this book. I eventually realized that these open source tools were the key to explaining how to bridge the gap from models to code while providing an accessible solution to any interested party.

Thus began our latest book project. After coaxing Andrew to take part, and enticing the primary author of *Executable UML* and Agile signatory, Stephen Mellor, to join in, we had the perfect mix of expertise.

As you read along, you may find that this is not an easy subject, and may require some hard work to make it through. The reward is the ability to take executable models that solve real problems through to an efficient implementation without destroying the models in the process. I want you to know that you are not alone in this journey, and we're more than happy to help if you have questions or run into obstacles. Feel free to use our contact page at the end of the book. Happy reading!

—Leon Starr
Linköping, Sweden
March 15, 2017

In conversations with colleagues who share a similar outlook about software development, the topic invariably turns to why the practice of software development using models and translation is not more widespread. Many reasons are offered, but usually the conversation trails off into consternation over an industry that claims to want well-engineered software juxtaposed to the repeated practices that do not deliver.

For my part, I have concluded that modeling and translation are hard because abstraction is hard and requires practice to master. One need only look at the facial contortions of a young teenager when initially exposed to the idea of variables in beginning algebra to understand that we are creatures first of the immediate and concrete. Gone are the comforts of explicit numbers, and suddenly a new set of rules is in place to cope with the idea that an arbitrary symbol, which doesn't look anything like a numeral, can now range over many values. The look is similar to that of a beginning programming student struggling to fix firmly the difference between a variable and the value contained in the variable when the same symbol is used for both and only context provides the distinction. Even though I have successfully produced software systems using modeling and translation for many years, there are times when looking at pages of boxes and arrows that it is hard to see how the gap between symbols on a page and running software will ever be filled. It looks like a jump across the Grand Canyon to be attempted only by an extreme stunt rider equipped with a magic, rocket-propelled vehicle.

The young algebra student, by working through many specific problems, does come to grasp the underlying abstractions. Reasoning, and sometimes wrestling, with the concrete yields up the abstract. In this book, we undertake to illustrate general principles of translating models into code by examining specific examples. We practice a specific way to model requirements and translate to a specific type of target platform using a specific programming language. We hold tightly to our tagline of "no mysterious gaps" in our attempt to show, at each step along the way, exactly how the translated software is produced and runs. We show lots of models and lots of code, confident in the awesome power of an individual to grasp the abstractions embedded in the specifics.

There is a danger that some readers will look at the details of the platform or translation technique and conclude that, because the specifics don't match their interests or preferences, the idea of translating models into programs is itself not generally applicable. In that case, we will have failed as authors to provide the vision of model translation as a tangible engineering approach. But I am confident that there are readers who will realize that symbols on a diagram can be made tangible as a running program (as tangible as software ever is) and that obtaining code by translating models looks more like driving over a bridge than jumping the Colorado river.

—Andrew Mangogna
Nipomo, California
March 15, 2017

CHAPTER 1

The Modeling Landscape

We could begin this book with a litany of the failings and sins of software development. We won't bother with that. If you are a practicing software developer, you know the problems and have lived through them. Instead, we'll get straight to the point.

We believe that modeling requirements and translating those models directly to code can solve many of the failings of software development. Unfortunately, a huge gap exists between creating requirement models and a running program. How to bridge that gap is not obvious. If it were smooth, easy, and productive, we would expect modeling to be everywhere. But it's not. Reports vary, but only about 10 percent of development efforts use models. Clearly, models are not delivering enough value to take the software development world by storm. We believe this is caused primarily by the difficulty in bridging the gap between models and implementation.

In this book, we're going to take you on a detailed journey from a platform-independent model to a very platform-specific file of C code. We focus on a specific modeling language, translation process, implementation language, and class of hardware in an endeavor to demonstrate by example how to turn a model into code. The study of concrete examples is essential to grasping the general principles of obtaining code from models. This means that occasionally you may see concepts introduced that are not fully described until later. It may be necessary—in fact, it is encouraged—to take multiple reading passes through some of the chapters.

Prerequisites

We assume that you have some interest and background in modeling and UML. Although we won't be teaching modeling here, we explain key aspects of the Executable UML (xUML) modeling language we use, provide external references, and expand on our modeling approach as we go.

There will be code. So we assume you have some familiarity with the C programming language. We'll be walking through a lot of it as we proceed. We don't make any deep dives into the dark recesses of C, so a passing knowledge should be sufficient for these purposes. And for whatever we can't squeeze into the book, we'll provide online references.

No Magic

Software modeling tools and techniques have been around for many years, yet the industry seems to be waiting for some future development to set us on the right path and make the whole process work like magic. But this book isn't about the future, unicorns, or fairies. It is about what you can do today, on a real project in a real organization to produce deliverable code.

L. Starr et al., *Models to Code*, DOI 10.1007/978-1-4842-2217-1_1

Elaboration: The Easy Path to Failure

To appreciate our approach, it is helpful it to compare it to a completely different and common approach called *elaboration* that has repeatedly failed to deliver results. Elaboration was initially popular because of its intuitive appeal and low bar of admission when it comes to learning anything new about analysis and modeling. Knowledge of notation and skill in object-oriented programming were the only engineering prerequisites. As it turns out, this practice generally leads to disaster. In fact, elaboration has caused many a project to abandon the whole idea of model-driven software, stigmatizing the practice of modeling in the process.

Elaboration has you start out creating a "system" or "architectural" design by building a high-level model. The concept of *high level* has no definition, so it is supposedly intuitive. That fact alone should be a sign that you are headed for trouble. This period is often concerned less with determining how the components of the system will work together than with partitioning the work among the developers on the project.

Over time, classes become less high level and more Java-ish or C++-ish. It all seems so natural. Implementation boundaries cause model elements to be grouped along task, thread, or CPU boundaries. Private and public methods and inheritance, in whatever particular form is supported by the target language, are sculpted in and around the modeled classes.

Sooner or later in the modeling process, you will trip up on some vexing problem. After banging your head a while trying to solve it at your oxygen-deprived modeling altitude, someone says, "Oh, that can be fixed in implementation; don't worry about it." Then you smooth over the whole nastiness with an aggregation diamond or whatever and keep going. That is where the lack of a definition of *high level* makes it easy to avoid solving problems while modeling.

More modeling goes along until you realize that it's hard to know when to stop modeling. The usual guideline is to stop when all that remains is "implementation detail." Sadly, there is no good definition of implementation detail, either; there are usually lots of discussions and disagreements over what constitutes a detail. This is another warning flag of trouble ahead.

You rinse and repeat until either gradual or abrupt failure sets in.

Elaboration–Gradual Failure

Here the models are elaborated until code results. Attributes are added, states are added, and models are reorganized as necessary. Eventually, each modeled class corresponds directly to a Java or other implementation language class. Each attribute corresponds to a property of an implementation class. Each package corresponds to a thread, task, or other software execution unit. And you get code. As code is produced, there are even tools that will "round trip" the code base with the model diagrams to keep them in sync.

How is this a failure? Two reasons. First, the "high-level" models were destroyed. Adding detail progressively obscures requirements, and the models morph into pictures of the code. These pictures may be useful for navigating the code base, but any statement of the fundamental application requirements is soon obscured.

So what was the point of building models you were going to destroy? They'll have no life beyond the project. They'll be of little value moving to other platforms or software architectures. They're not substantially easier to understand than the code base itself and will be completely foreign to anyone in the organization outside the software development staff.

Second, the modeling process wasn't of much value because it wasn't focused on capturing the requirements of the problem. The difficult problems got swept under the rug and then later solved while adding code, when there is no choice but to do something, anything, however right or wrong that may be. So, the preliminaries are rightly perceived, in retrospect, as an additional, time-consuming, and not necessarily helpful phase.

Elaboration–Abrupt Failure

In the abrupt case, pressure to deliver increases to the point where everyone freaks out, casts the models aside, and starts coding. This start of coding is deemed when the "real work" begins, and any further attempts to understand the problem being solved are abandoned. The models may be referenced later somewhat, but attention inexorably shifts to the mounting code base because that's now the only thing that matters.

Model Destruction

In either failure scenario, the models are rendered useless. In the gradual case, the essential application logic is obscured, and the models become little more than pictures of the code. In the abrupt case, the models may survive for a short time as advisory documentation, but soon lose their relevance. After the project, in either case, the value of having built the models in the first place is rightfully brought into question. With results like these, it is no wonder that modeling never became widely accepted as an essential software development activity.

The Value of a Good Model

Plenty of other books detail the many good reasons to model the requirements of your application. Here we have to assume that you think models have the potential for redeeming value even if that value has appeared tenuous in your experience. Let's instead try to boil down whether you are getting tangible benefits from your modeling efforts with a hardware engineering analogy.

Consider a typical workflow of our electrical engineering cousins when designing a printed circuit board (PCB). They first capture the *application logic* of the circuits in a schematic diagram. They define the components that are required and the way they are connected. The schematic defines exactly how the board works.

Only after schematic capture is consideration given to PCB layout. Layout is about the physicality of the board: component sizes, trace widths, ground planes, signal distances, and many other elements must be specified. In many organizations, the person doing the layout is different from the one who did the schematic capture, because the skills for the two tasks are that different. When the board is fabricated, the electrical circuits *must* function as dictated by the schematic diagram—layout does not change functionality.

To an electrical engineer, a schematic diagram is a model of the application logic. What is excluded from the schematic diagram is the physical arrangement of the components on a PCB and all the practical considerations of electrical connectivity. That exclusion is then introduced by the board layout.

After the board is fabricated, if testing indicates an error, then not only is the board patched, but the schematic diagram is scrupulously updated to reflect the fix. Amazingly, the entire workflow of schematic diagram, board layout, and schematic updates is never a subject of contention among electrical engineers. The value of a schematic diagram and the absolute necessity that it accurately describes the way a board works is part of the shared outlook for electrical engineers.

If you ask an electrical engineer a question about how a board works, immediately he or she will pull out a thick wad of paper containing the schematic diagram drawings for the board. They flip to the relevant page and proceed to point and trace through the diagram to answer the question. They do this with absolute confidence that the answer obtained from the schematic diagram matches the way the board works on the lab bench.

Now, if someone asks you a question about how a software feature works and your first instinct to answer the question is to browse through code files, to read and execute, in your head, a bunch of code, then whatever models may exist for that software are not providing enough value to be worth producing. On the other hand, your models truly capture the requirements of the system when someone asks, "How does the gronkolator masticator know to turn itself off?" and you reach, without hesitation, for a thick wad of paper (or load up a model file on your giant display) containing the software model diagrams and start pointing and tracing to answer the question.

A Better Way Forward: Translation

We subscribe to an entirely different approach to getting code from models called *translation*. This translation approach is based on two fundamental tenets:

- The models are *detailed* expressions of application logic. They should capture and formalize requirements of a problem.
- Code is *derived from* the models. The models are not destroyed in the process of translation.

The models are detailed in two key respects: executability and platform independence. The vague concepts of *high* and *low* level are not meaningful or relevant here.

Executable Models

The models are fully executable without code. By *executable*, we mean that there must be a clear, complete, unambiguous set of rules for running the models.

Each set of models is adequately detailed so that it can be executed and tested without the need for any programming language code. This is made possible by a small set of well-defined, platform-independent execution rules and descriptions of processing that operate exclusively on model elements. The execution rules have been designed so that they can be implemented on a broad array of platforms such as a single microcontroller or spread out across a distributed system with many parallel processing units.

If you are curious, you can get a taste of the execution rules by looking at how platform-independent synchronization is managed in xUML by visiting www.modelint.com/MBSE or www.executableuml.org.

Platform-Independent Models

The models must *not* contain implementation technology. This is for two reasons.

First, the logic of an application stands apart from the means and methods of its implementation. We have long been defining logic and processing without using any programming language. Second, as many developers have discovered, implementation features are more flexibly defined in code rather than in the constrained context of model formalisms. Any implementation concepts assumed in the models hampers the choices that can be made downstream. It becomes more difficult to optimize and adjust to platform-specific requirements when implementation choices are forced by the models. It's the age-old principle of using the right tool for the right job. Programming languages are great for devising implementations and not so good at expressing application logic distinctly. Models of the type that we build are great at expressing logic and required data and synchronization, but are a terrible place to design code.

Deriving Code from Models

Assuming that the models are executable and platform independent, we get code by mapping them onto the implementation technology. All implementation decisions and artifacts are folded in downstream from the modeling process. Consequently, all the work and intellectual property that goes into the models is maintained, distinct from any particular implementation.

We never mix code anywhere back into the models themselves. We need a modeling language that supports these features and, to bridge the gap to code, we need a way to take those models as input and add the necessary implementation technology to transform them into running programs on a hardware platform. Let's start with the language.

xUML: Same Notation, Different Attitude

We use Executable UML (xUML) as our modeling language. Like most modeling languages, it has a graphical notation and underlying semantics. Semantics is just a short way of saying, "what the notation actually means." xUML uses a subset of the UML notation on top of platform-independent execution semantics. This is not the same as the object-oriented semantics employed by the greater UML community. Both *Executable UML: A Foundation for Model-Driven Architecture* by Stephen Mellor and Marc Balcer (Addison-Wesley, 2002) and *Model Driven Architecture with Executable UML* by Chris Raistrick et al. (Cambridge University Press, 2004) provide complete descriptions of xUML modeling.

Let's take the class symbol, for example. Most UML folks will look at it and imagine a "high level" Java-ish or C++-ish class. This brings with it all of the attendant object-oriented programming concepts such as object references, public/private methods, inheritance, and so forth in whatever object-oriented programming language you assume is most likely targeted.

In xUML, the semantics are entirely different. First of all, there is no implicit presumption that we are targeting an object-oriented programming language. Instead, xUML is built on a set of mathematical formalisms. Now, we do *not* intend to write lots of Greek letters or upside-down Latin characters when we describe a model. Being based on a formalism is not the same as mimicking a formalism. We also don't want to imply that the modeling formalism is somehow associated with proving the correctness of the resulting program. We'll leave formal program proofs to the computer scientists and simply strive to be better software engineers. It is not necessary to understand the underlying formalisms to use xUML, because they are expressed as a set of modeling rules. However, understanding the fundamentals will make you a better modeler. The formalisms remove any platform bias, and the mathematical basis provides assurance that we can translate to any target platform required to meet our specific engineering needs, object oriented or otherwise. A good introduction to the mathematics of logic and its use in describing data can be found in *Applied Mathematics for Database Professionals* by Lex de Hann and Toon Koppelaars (Apress, 2011).

The *x* in xUML

As programmers, we read a lot of code. Reading code leads to understanding a program's behavior, because you can run the code in your head. In this sense, C code is executable because there are well-documented execution rules to accurately predict what the code will do. Consider, for example, the following function, written in C, which computes the y-coordinate of a line, given the x-coordinate, slope, and intercept:

```
int
linearValue (
    int x,
    int m,
    int b)
{
    return (m * x) + b ; /* yes, the parentheses are unnecessary */
}

int y = linearValue(5, 10, -2) ;
```

Any two C programmers can mentally execute this code line by line and conclude that the value contained in the y variable will be 48. It is not subject to opinion, mood, or perspective, because each programmer envisions the same execution model that defines how the program runs. (And if the answers diverge, it means that one programmer or both are misunderstanding that model!) Yet we also know that real computers do not behave in strict accordance with the execution model of the C language. The C compiler lays out memory for variables, selects the correct integer instructions, and generates the necessary

instructions to enforce the conventions for passing arguments. But it *must* yield the same result as our execution model. You don't need to envision the compiler's complicated layout scheme to get the correct result if the compiler is doing its job correctly.

The same principle applies to an executable model. You must be able to read a model and accurately predict an outcome, though the model may be implemented entirely differently from the way we execute it in our heads. Nothing is open to subjective interpretation. UML alone does not provide this level of certainty, but the xUML we use in this book does. Figure 1-1 illustrates this principle.

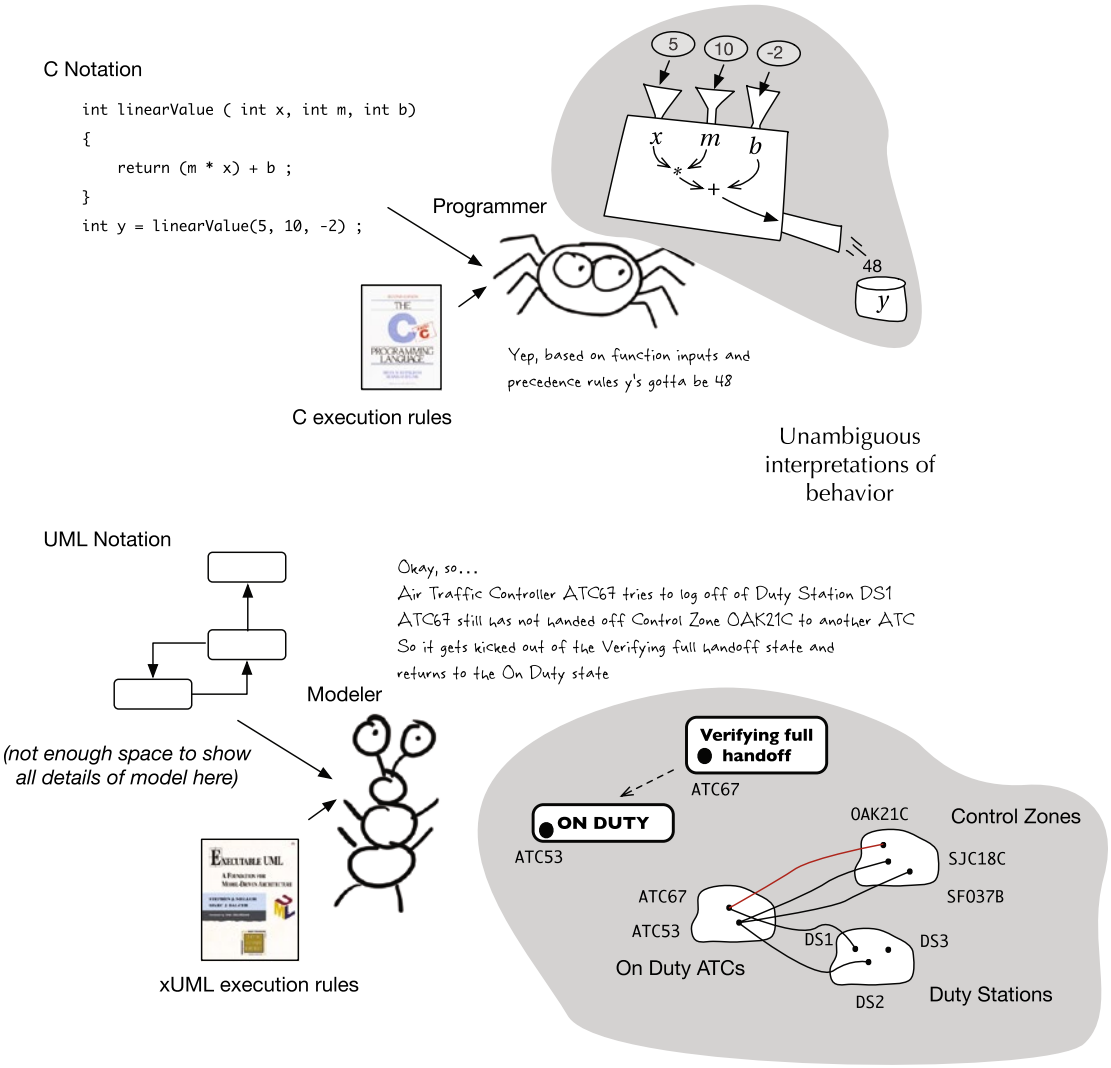

Figure 1-1. *Mentally executing code and models*

The syntax of any executable language associates keywords and symbols with corresponding executable meaning or semantics. Assembly language, for example, operates with the registers and memory structure of a particular processor. C, on the other hand, lives in the world of variables, structs, arrays, loops, and functions. Our executable modeling language takes the level of abstraction up a few notches and operates in a world of sets, relationships, life cycles, and data flows. This is ideal for capturing application logic without distraction.

With an executable model, you can model applications unambiguously and in sufficient detail that they can be executed and verified just as you commonly do with code, without actually *being* code. This forces you to think about what the application is doing. Decisions must be made. Inadequate subject-matter knowledge must be filled in. You cannot defer difficult problems with statements like, "We'll sort that out when we write the code." When we model the requirements for an air traffic control system, for example, we must decide exactly what is to happen for an on-duty controller to go off duty. We must define what data is required, what computations must be performed on it, and how the computations must be sequenced or otherwise synchronized. Model-level executable semantics let us specify all of this without assuming any particular target programming language, design patterns, processor distribution, framework, library, or any other implementation technology.

Translation

Executable models are not the same as code. Executable models describe application logic. To get to code, an executable application model must be translated onto implementation technology.

Our insistence on a separation of logic and technology has created a larger gap to fill than if we had insinuated implementation concepts into our modeling language. If your modeling language looks a lot like an object-oriented programming language, the step to get object-oriented code is not very big. In our case, we must fold in all the required technology. This can run the entire gamut of programming technology such as a target programming language, design patterns, processor distribution, framework, runtime library, and any other mechanisms of implementation we need to create a running program.

It is important to emphasize that translation adds nothing to the logic or behavior of the system. It adds computing technology only. Whereas earlier we ignored implementation technology to capture concisely the application logic, translation shifts the emphasis from application logic to technology. Because we are concerned only with how computing technology implements the application logic, the details of how the application logic meets the requirements of the system can be ignored. When we translate, we don't try to second-guess the application logic, just as we didn't try to specify the implementation mechanism when we were modeling.

Our Target Technology

The landscape of modern computing technology is vast, so many decisions need to be made in determining the type of computing technology that should be applied to a given problem. Translation is directed, then, at a particular target technology and environment, and we must be specific about the details of how and where that program will run.

Although the approach we describe can be used in multiple contexts (web applications, software-oriented architecture, batch programs, and distributed systems, to name just a few), it would be distracting and counterproductive to keep switching back and forth between implementation environments. Instead, we pick one and consider the consequences all the way through to the code. This book shows one example of how to apply specific technology to translation. It is not a comprehensive treatise of translation theory. Still, we expect you to come away with a greater understanding of how translation is accomplished.

We demonstrate translation assuming our target computing environment has the following characteristics:

- The environment is based on an embedded microcontroller. These typically have less than 512 KB of memory, and frequently as little as 32 KB. We will not consider the very small end of this scale. Microcontrollers with less than 16 KB of read-only memory or less than 4 KB of read/write memory are outside the design scope of our target environment.
- The implementation language is C. C is common in this realm, and good C compilers are available.
- The class of applications is reactive in nature. The software responds to unsolicited external stimuli and interacts with its environment via a set of hardware peripherals. This is in contrast to applications that are primarily transforming in nature, such as rendering a graphical image.

WHY NOT C++?

Some readers may wonder why C was chosen rather than C++. Despite their syntactic similarities, C and C++ are very different languages. C++ provides powerful features, not all of which are supportable in a microcontroller environment. Some embedded projects use subsets of C++ to avoid these difficulties. But programming language features make it easier for human programmers to accomplish programming tasks. The same features are not as useful in a translation environment. The role of an implementation language in translation is different than when you are directly coding a program. A translation program doesn't care whether a particular concept is not well supported in the implementation language. Finally, several, obscure incompatibilities exist between C and C++, and our translator program needs a construct in C that is illegal in C++.

This target technology would not work well for a web application. But that does not mean the translation approach fails for a web application; it simply means that models for web applications must target a different technology.

Our Translation Environment

There is a broad spectrum of ways to accomplish the translation of a model into code. Any mechanism that faithfully maps the logic of the application onto its code equivalent without destroying the model is a candidate.

At one end of the spectrum, translating a model starts with software tools that capture a graphical representation of the model. Dozens of UML drawing tools are available. The model's meaning is captured in a database by using sophisticated tools. The database is then traversed in potentially complicated ways to produce code. The main advantage of this approach is power and generality. The main disadvantage is that it appears to be magic. This perception is heightened by the number of elements you must understand before you can make anything happen and by the amount of unseen processing that goes on to get between the model and the code.

Our approach starts at the other end of the spectrum. We have defined the details of the target platform and fixed the way model execution rules are implemented within the technology of that target platform. A set of platform-specific rules determine the choices for how model elements are mapped onto the implementation constructs. We encode the mapping of model elements into implementation constructs by using a text-based domain-specific language (DSL). Then, a relatively simple tool, named pycca, generates the output C code. The main advantage is that we are assured of obtaining a program that matches our target platform technology. The main disadvantage is the lack of integration to front-end model development.

Here is our basic workflow:

1. Create an xUML model of the subject matter. Use your favorite model-drawing tool. It does not matter which one.
2. Analyze the model to determine the translation characteristics.
3. Using the drawing tool artifacts, encode the structural model elements, such as classes and state models, into a text file by using the pycca DSL.
4. Translate the actions into C code that is also placed directly into the pycca file.
5. Generate a code file by running pycca.
6. Compile and link the resulting code.

What pycca lacks in front-end integration, it makes up for in these important characteristics: platform specificity, transparency, and availability.

Platform Specificity

Every target platform has its engineering challenges. If you have the luxury of commonly available hardware and software technology to satisfy the needs of your application, you are indeed fortunate and can focus your attention on model logic. However, experience shows that there are many expectations for the characteristics of an implementation. For our example target, small memories and slow processors mean that techniques that would be acceptable on a conventional desktop computer simply do not work in a microcontroller environment. Unlike many code generation schemes, pycca lets you handcraft your own algorithms to implement modeled activities. This capability is crucial when you grapple with the idiosyncrasies of legacy code or other peculiar aspects of your target platform.

Transparency

The correspondence between the model, the pycca source, and the generated C code is clear and direct. You can understand the role that each part plays in your program. The particulars of what constructs are supported by a drawing tool or how a drawing tool stores model content does not affect the translation to code.

Availability

Pycca is freely available, and all it takes to get this scheme to run is a little brain power and a C compiler.

A Final Word About UML and Standards

We use the UML notation as a lingua franca for presenting model diagrams in this book. When it comes to models, however, we are mostly concerned with the stuff underneath the notation: model execution semantics and platform independence. For this, we turn to xUML, which provides us with exactly what we need: a strong mathematical foundation and platform-independent rules for running and testing models.

Occasionally, the UML notation is at odds with the xUML semantics. This is largely because UML is biased in the direction of an object-oriented programming paradigm, whereas xUML is platform neutral. As we consider the notation the less important factor, when push comes to shove, the execution semantics will always prevail in our work. After all, it is easier to bend the interpretation of a graphical symbol than to overcome the surprise of a whole new notation. Like the English language, the consequence of UML being a lingua franca is that everyone is allowed to give the language its own regional flavor (sometimes to the annoyance of the native speakers!).

You may also notice that we don't claim to adhere to any particular UML standards. We think the best standards in software are those that codify existing practice or attempt to ensure interoperability. Existing practice in UML is targeted in many directions, and interoperability is demonstrated only by explicit testing. With its dizzying selection of diagram types, UML is used for activities ranging from cocktail-napkin sketches to, as we present here, formalized statements of software requirements. We find it difficult to pin down existing industry practice. Usually, published standards result in nonstandard implementations. From C compilers to SQL query languages, standards compliance does not seem to inhibit extending functionality and limiting interoperability, because implementation necessities always win out in the end. Noncompliance to a certain standard is just another engineering trade-off that a project team must evaluate. We are not averse to standards that contribute value to a project team, and if it is important to your project team to have UML diagrams that adhere to a particular version of the UML standard, then you should pursue that goal. We still hold to the proposition that it is not the *shape* of the boxes and arrows that matter; rather, it is the *meaning* attributed to them that determines true value.

What's Next?

Our goal in this book is to teach the key principles of how models are translated into code by using detailed examples. Model translation is not magic and need not be shrouded in mystery. We do our best to be grounded firmly in the engineering realities of producing running software. Though we present completed models and explain what they mean, it is not our intent to teach modeling here. While we demonstrate how pycca is used to accomplish a translation into code, this is certainly not a pycca manual. (The documentation for that is available online.) We show you C code and assume that you can read and understand it. All the details of models that don't fit here are readily available from `www.modelstocode.com`. There you find all the examples completely worked out.

To get you to code as soon as possible, we're going start by introducing a small example air traffic controller model in Chapter 2. This model will also serve as a nice introduction (or review, depending on your background) of xUML basics. We'll show how the model captures platform-independent application requirements and how we can walk through a model execution scenario.

We'll then review our example executing model in Chapter 3 with a tour of the types of design decisions that must be made to influence the translation process. Pycca will be introduced as a language for specifying these decisions. The first glimmer of code structures to be shaped will appear in this chapter.

Things get serious in Chapter 4, where we write pycca statements to define our full translation to C. Throughout, we will emphasize the tight correspondence between the input model, the pycca statements, and the generated C code.

In Chapter 5, we pop open the hood of the supplied model execution runtime code that will be linked with your generated code. The focus here is on how state machines are executed and events are queued, dispatched and, where necessary, delayed. We will show how real-world interactions via interrupts can be serviced in conjunction with model execution.

Having successfully translated our small model, we'll expand on how real systems are put together with a new and more challenging example. The automatic lubrication system for vehicles and machinery will be introduced, featuring multiple modeled components called *domains*. Chapter 6 introduces a domain concerned with user application logic, and Chapter 7 presents a domain called Signal I/O (SIO) that makes solid contact with physical sensors and actuators.

We'll pull it all together in Chapter 8, integrating the domains with a concept called *bridging*. This will allow us to knit the domains together into a complete functioning system.

In Chapter 9, we'll take a look at how polymorphic events are defined, signaled, and dispatched in generalization relationships.