Assessing a C# Text

Bertrand Meyer, ETH

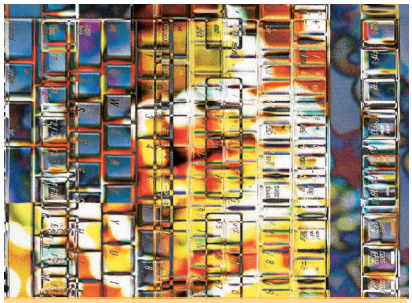
***Programming C#,* 2nd ed., Jesse Liberty; O'Reilly, Cambridge, Mass.; http://www.oreilly.com; ISBN 0-596-00309-9; 658 pp.; $39.95.**

Jesse Liberty obviously understands that it's not particularly exciting to devote a book to C# as a language, even if that hasn't deterred publishers from filling bookstore shelves with C# books. You can't admit this with a title such as *Programming C#*, so he takes care to define C# in the preface as a language that "builds on the lessons learned from C (high performance), C++ (object-oriented structure), Java (security), and Visual Basic (rapid development)."

This must be meant tongue in cheek: C# is less close to C than C++ is, so it's not clear where the concern for high performance lies. Nor can object-oriented structure be C++'s key characteristic when, in response to "Is C++ an OO language?" the language's designer describes it as "multiparadigm." C# retains far more than security from Java, and Visual Basic's rapid development support comes from its environment rather than the language, with the exception of features such as loose typing, which C# doesn't retain. More accurately, C# can be defined as Java plus:

- a consistent type system—as in Smalltalk and Eiffel, all types are based on classes;
- a few extensions, including delegates;
- a type-safe way to remedy Java's lack of function pointers;
- properties, as in Delphi and Visual Basic; and
- attributes, program annotations that compilers retain for component-based development.

I am simplifying, but to cut to the



essentials, C# is Java as Microsoft has always wanted it to be since Visual J++, its Java implementation whose language extensions attracted Sun's ire. This doesn't diminish the book's value, but it should be explained in a presentation intended for programmers who weren't born yesterday. It would also facilitate Liberty's job if he more often described C# constructs in terms of their Java counterparts, especially as he does make comparisons to C++.

.NET GEMS

If C# seems too narrow a topic for an entire book, the good news is that Liberty broadened his scope to cover the .NET framework as well. The first C# books concentrated on the language: Christopher Wille's *Presenting C#* (Sams Publishing, Indianapolis, Ind., 2000) debuted early and served as a short, general introduction; Eric Gunnerson (*A Programmer's Introduction to C#*, 2nd ed., Apress, Berkeley, Calif., 2001) gave the designing team's perspective; and Tom Archer (*Inside C# Architectural Reference*, Microsoft Press, Redmond, Wash., 2001) provided an in-depth presentation of the language's concepts.

What I found most interesting in Liberty's book is what goes beyond the language. Part 2, which covers using .NET to build graphical applications with Windows Forms, Web applications with Web Forms, Web Services with ASP.NET, and database applications with ADO.NET, provides an excellent overview of the .NET framework's central tools, part of .NET's main attraction to developers.

Part 3 goes further, explaining programming techniques for the Common Language Runtime. These include versioning, which lets a module specify when it accepts upgrades of the modules it relies on, and reflection and attributes. Unfortunately, the author does not provide a full explanation of the innovative .NET concept of metadata and its role in getting rid of COM's IDL.

Liberty also covers techniques for concurrent and distributed programming: threads, synchronization, remoting, and asynchronous I/O. He uses examples from the Visual Studio.NET environment, which deserves a chapter of its own.

Parts 2 and 3 thus form a useful introduction for programmers writing in any of the many languages available on .NET. That introduction holds its own against many of the current books describing .NET as a whole. So it's when Liberty seemingly goes off-topic that his contributions appear most interesting. I learned a few things about .NET that I hadn't seen elsewhere.

PUZZLING LAPSES

I hope I can trust what I read, however. Some of Liberty's pronouncements about general software topics cause me to wonder. Early on, Chapter 3, "C# Language Fundamentals," tells us that

> A stack is a data structure used to store items on a last-in first-out basis (like a stack of dishes at the buffet line in a restaurant). The stack refers to an area of memory supported by the processor, on which the local variables are stored.

> In C#, value types (e.g. integers) are allocated on the stack—an area of memory is set aside for their value, and this area is referred to by the name of the variable.
>
> Reference types (e.g. objects) are allocated on the heap. When an object is allocated on the heap its address is returned, and that address is assigned to a reference.

This explanation confuses four notions:

- a variable, an element of the program text that denotes possible run-time values;
- the values themselves, including both simple ones such as integers and references to objects;
- such objects themselves; and
- the types of values and objects.

In a typed OO language, programmers declare every variable with a type, restricting the kind of values that it can take at runtime. This declaration can be an "expanded" type in Eiffel terminology, or a value type in .NET, meaning that the variable will directly denote objects or other values. Or it can be a reference type so that the variable denotes references to objects of certain types compatible with its declaration.

The preceding description, however, commingles everything. The stack doesn't store the variables, it stores their values. Value types are not allocated on the stack, their instances are. An instance of a type is a value or object described by that type. In both cases, the "e.g." is wrong: An integer is not an example of a value type—it's an instance of a sample value type, int. Nor is an object an example of a reference type—it's twice remote: the value of a reference type's variable is a reference that, if not void, is attached to an object.

It's meaningless to "allocate a value type on the stack" or to "allocate a reference type on the heap." You don't allocate types, you allocate values corresponding to variables declared to be of that type, or objects attached to the corresponding references. True, the .NET terminology doesn't help by talking of value versus reference types. Given that all types have values, this makes it clumsy to explain that the possible "values" of a value type are "values" while the "values" of a reference type are references. But it is possible to present these concepts simply and correctly.

Apart from being confusing, the explanation is wrong. It's not true that values of expanded types are allocated on the stack and objects are attached to reference types on the heap. Any first-year computer science student who has played with an OO language—or with C or Pascal—knows that you can find many simple values, such as integers, in the heap.

When Liberty talks about .NET's specific mechanisms, he is cogent and seemingly accurate.

Yet the comment is repeated throughout, as in Chapter 4: "The primitive C# types (int, char etc.) are value types, and created on the stack; objects, however, are reference types and are created on the heap." This introduces a further confusion, also occurring several times: an object is not a type.

We can assume that Liberty knows the difference, and I did find one hesitation, confined to a parenthetical remark: "A value type holds its actual value in memory allocated on the stack (or it is allocated as part of a larger reference type object)." But I wonder about the effect of these constantly repeated mistakes on the reader, especially the kind of reader who needs to be told what a stack is.

When Liberty talks about .NET's specific mechanisms, he is cogent and seemingly accurate. But when he introduces "the *this* keyword" he writes that it "refers to the current instance of an object." Earlier, he used types when he meant their instances; now it's the other way around. Liberty should simply have said that *this* denotes a reference to the current object.

In Chapter 8, we learn that "your Document class can be stored and it also can be compressed," but this has nothing to do with zipping the class text to send it to your friends. Rather, it's supposed to mean that you can store and compress Document objects: instances of the class. Or take this explanation, from the same box that helpfully told us what a stack is: Heap objects are garbage-collected "sometime after the final reference to them is destroyed." You don't need a garbage collector to get rid of values on the stack. The system simply deallocates these values from the stack on routine exit, as in any block-structured language since Algol 60.

Overall, these confusions and errors reflect on the publisher as well as the author. Liberty must know the difference between a type and an instance or between stack-managed and garbage-collected values. He just doesn't seem to know how to explain these things, feeling more at ease with .NET's intricacies. Given the list of people who, according to the preface, the author and publisher "enlisted" to "ensure that *Programming C#* is accurate, complete and targeted at the needs and interests of professional programmers," it's hard to accept the repeated misstatements of elementary notions. As usual with O'Reilly books, the text shows signs of having been checked for style, although it retains things like "there are three ways in which this reference is typically used." Why wasn't similar care applied to basic concepts?

This brings up the more general issue of the quality of computer books, and what this column can do about it.

THE GREAT DIVIDE

In the computer section of a bookstore today, you'll find shelf after shelf of *trade books*: hands-on, learn-as-you-go titles meant to give the reader immediate proficiency in the technology du jour. In many bookstores, they're all there is to see. Something about pub-

lishers' discount schedules plays a role here, although I don't claim to understand it.

In the better stores, you'll find a shelf or two stocked with technical books on more highbrow topics. Even there, however, only a few titles deal with real concepts.

The two categories seem doomed to their clichés. The highbrow books are accurate, boring, and don't sell. The trade books are targeted to a specific market, have a time-limited value, hope to sell well, and are put together in a rush to catch their audience before someone else does. As a result, their authors don't care that much about the solidity of the concepts or even technical accuracy. Often, they don't bother to tell you that developers used anything else before the appearance of the specific technology they describe, especially if the previous product came from another persuasion: Java Server Pages books won't acknowledge Microsoft's Active Server Pages, for example, while C# books pretend that Java never existed.

Once in a while, a book does have the best of both sides: It talks to the practicing programmer, provides immediately applicable material, and is conceptually sound. *Design Patterns* by Erich Gamma and colleagues (Addison-Wesley, Reading, Mass., 1995) is an example. But usually we must choose between a book that's useful and unreliable or one that's solid and inapplicable—or, in the words of David Parnas, between "gadgets without methods and theories without applications."

CALL TO ARMS

It doesn't have to be that way, and I hope this column can play a small part in raising the standards on both sides of the aisle. I intend to review both academic and practitioner-oriented books, examining the former to see that they are useful and the latter to see that they are accurate. Who says that an author who knows about C# and other cool new stuff shouldn't exercise the same care in distinguishing types from instances?

I refuse the rift. What I like most about *Computer* is that it doesn't shy away from articles on either theory or practice, as long as they're good. I intend to follow the same open outlook. A good book should have a certain set of basic qualities: It should tell a story and tell it well, distinguish facts from opinion, provide a broad enough background, treat its reader as a grown-up, and offer useful information.

Trade books require particular focus because the tendency to rush to market often leads to cutting conceptual corners. One publisher told me that this tradeoff occurs because "the first decent book on a topic captures the market," so while it must be both the first *and* decent, it doesn't need to be better than decent.

While it may be critical at times, this column won't be cowardly.

As consumers, we need not accept this situation forever. By submitting books to serious review, applying to them the same criteria as to more ambitious intellectual endeavors, we can increase the rewards of careful research, writing, editing, and publishing.

Books are important. Many professionals I know have had their careers shaped by the first books they read. I can certainly remember the four or five books I read as a student that changed my outlook forever. When hiring software development candidates, I have found that an effective interview question, first suggested to me by Ada designer Jean Ichbiah, is to ask candidates to name a few technical books they have found particularly useful. Web pages help, and magazines like *Computer* play an invaluable role, but nothing will replace the good computer book with its wealth of carefully distilled wisdom.

GROUND RULES

While it may be critical at times, this column won't be cowardly. As a book author myself, I have left so many thousands of my own printed pages behind—with more yet to come—that anyone who wants to retaliate to my critiques can choose from many targets.

I've set a few ground rules for this column. First, I will only review books that I would have read anyway for my own sake—books from which I expect to learn something. This may limit this column's scope, but it means that I will always have a personal interest in the books reviewed. I may occasionally ask guest reviewers to cover areas beyond my purview.

Second, I will not review garbage. There's a fair amount of it around, and it would be easy to have a good laugh at the expense of some poor author who put together an incompetent account of some technology to make a quick buck. But such a review would offer little of interest to *Computer*'s readers, as I assume you have your own BIMs—Baloney Identification Mechanisms—firmly in place.

The target of this first review is typical: What's frustrating about this book's mangling of basic OO mechanisms is that, in the end, I can recommend it as a useful source of information about C#. A third edition correcting the conceptual blemishes would be truly excellent.

So I intend to select interesting books that cover interesting topics—ranging from practical descriptions of specific technologies to the purely theoretical—and submit them all to the same criteria. I will challenge the lowbrow on their conceptual soundness and the highbrow on their relevance.

I hope these reviews will benefit all of us—readers, publishers, writers—and that they will help us to recognize and promote quality in these ever more necessary tools of our trade: technical books. ■

***Bertrand Meyer** is professor of software engineering at ETH in Zürich, http://se.inf.ethz.ch, and scientific advisor of ISE, Santa Barbara, Calif., http://www.eiffel.com. Contact him at book_review@eiffel.com.*

However, this default designer doesn't include an implementation for the `BaseLine` snap line, which is used to align controls via their contained text. This article shows how to create a custom designer to allow your controls to easily include this alignment option.Â One of the nice things about the Visual Studio WinForms designers are the guidelines it draws onto design surfaces, aiding you in perfectly positioning your controls. These guidelines are known internally as snap lines, and by default each visual component inheriting from Control gets four of these, representing the values of the control's Margin property.