This is a pre-copy-editing version of a review published in IEEE *Computer*, vol. 35, no. 4, April 2002, pages 86-88. It includes some material that had to be removed from the published version because of space constraints.

BOOKS OF NOTE

by Bertrand Meyer

Jesse Liberty: *Programming C#*, O'Reilly, 2nd edition 2002, 658+xix pages.

(Sidebar)

Other books mentioned in this article

Tom Archer: *Inside C# - Architectural reference*, Microsoft Press, 2001.

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns*, Addison-Wesley, 1995.

Eric Gunnerson: *A Programmer's introduction to C#*, second edition, Apress, Berkeley (CA), 2001.

Christopher Wille: *Presenting C#*, Sams Publishing, Indianapolis (IN), 2000.

Something Jesse Liberty obviously understands is that it's not particularly exciting to devote a book to C# as a language, even if that hasn't deterred publishers from starting to fill the bookstore shelves with C# books. You can't admit this when your title is *Programming C#*, so he takes care to define C# on the first page of 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 new concern for "high performance" lies; "object-oriented structure" is not the key characteristic of C++ (its designer describes it in his home page, in response to "*Is C++ an O-O language?*", as "multi-paradigm"); C# retains far more than security from Java; and the rapid development support of VB comes from its environment rather than the language (with the exception of such features as loose typing which C#, in fact, doesn't retain). A more accurate definition of C# is that it's Java plus a consistent type system (like in Smalltalk and Eiffel, all basic types are based on classes), a few extensions ─ delegates as a type-safe way to remedy Java's lack of function pointers, and in the sense of Delphi and VB —, and the notion of "attributes", program annotations retained by compilers for component-based development. I am simplifying (I almost forgot the presence of *ifdef* to please C/C++ programmers, and you will write *bool* rather than *boolean*), but C# is Java the way Microsoft always wanted it to be since Visual J++, its Java implementation whose language extensions attracted Sun's ire. This doesn't diminish its value but 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. He does make comparisons to C++, so it's hard to understand what stops him here.

If C# is not enough of a topic for a book, the good news is that Liberty broadened his scope to cover not just C# but the .NET framework. The first C# books have concentrated on the language: Wille was available early and served as a short, general introduction; Gunnerson gives the perspective of the designing team; and Archer an in-depth presentation of the language 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 these central tools of the .NET framework, part of its main attraction to developers. Part 3 goes further by explaining programming techniques for the Common Language Runtime: versioning, enabling a module to specify when it accepts upgrades of the modules it relies on; reflection and attributes (not explaining well enough to my taste, however, the innovative .NET concept of metadata and its role in getting rid of COM's IDL), as well as techniques for concurrent and distributed programming: threads, synchronization, remoting, asynchronous I/O. The book also provides glimpses of Visual Studio.NET, even if I was surprised that after stating early on that "*although you can develop software using Visual Studio .NET, the IDE provides enormous advantages*" it doesn't devote a full chapter to it.

The result of parts 2 and 3 is a general introduction to .NET, useful to programmers writing in any one of the many languages, besides C#, available on .NET. That introduction holds its own against many of the current books describing .NET as a whole. So it's where Liberty seemingly goes off-topic that his contribution is the most interesting. I learned a few things about .NET that I hadn't seen elsewhere.

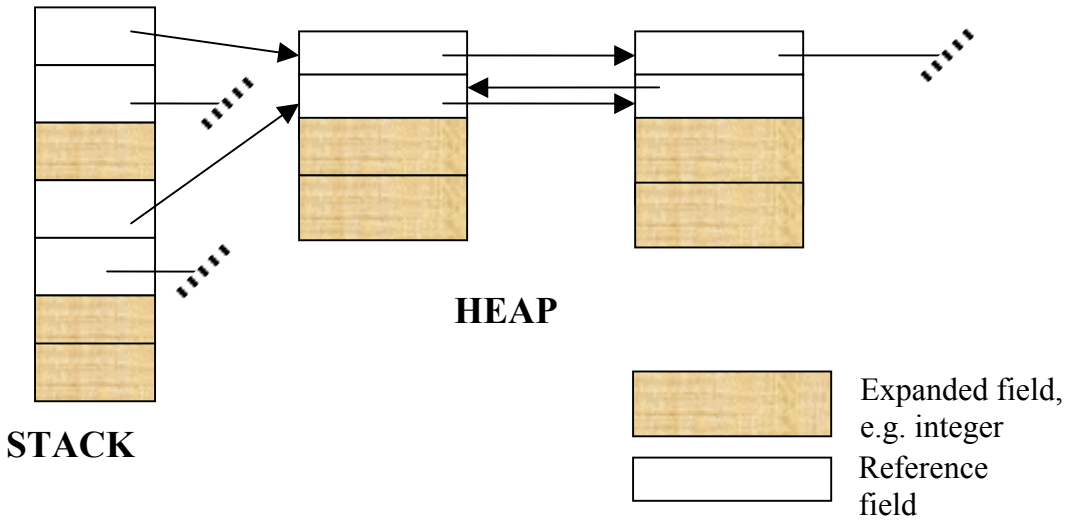
But then I hope I can trust what I read. Some of Liberty's pronouncements about general software topics are enough to make one wonder. They start early, in chapter 3, "C# Language Fundamentals", telling us that

> *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 puzzling explanation confuses four notions: a *variable*, an element of the program text that denotes possible run-time values; these 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 O-O language every variable is declared with a type, restricting the kind of values that it can take at run time; this can be an "expanded" type in Eiffel terminology — *value type* in .NET — meaning that the variable will directly denote objects or other values; or a *reference type* so that the variable denotes references to objects of certain types compatible with its declaration. But in the above description everything is commingled. 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.) Both "*e.g.*"s are wrong: an integer is not an example of value type (it's an instance of an example value type, *int*) and an object is not an example of a reference type. It's in fact twice remote: the value of a variable of a reference type is a reference which, if not void (null), 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 of that type, or objects attached to the corresponding references. When "*an object is allocated on the heap*" its address is indeed returned somewhere, but to whom? What should have been explained here is that the corresponding variable gets, as its value, a reference attached to the new object. True, the .NET terminology doesn't help by talking of "value" vs. reference types: since all types have values, this makes it clumsy to explain that the possible "values" of a value type are "values" and the "values" of a reference type are references. That's why Eiffel uses "expanded type" to denote types whose values are objects, and "reference type" for those whose values are references. But it's possible to present these concepts simply and correctly.

The explanation cited is not only confusing but wrong. It's not true that values of expanded types are allocated on the stack and objects attached to reference types on the heap. Any Computer Science 1 student having played with an O-O language — or just with C or Pascal — knows that you can find many simple values, such as integers, in the heap part of the structure:



Yet the author clings to this mantra, 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*". (Note the confusion of types and instances, occurring through the book — an *object* is not a *type*!). This is very disturbing. One may assume that the author does know 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.

There are many other examples. Whenever Liberty talks about the specific mechanisms of .NET he is cogent and, one hopes, accurate. But when he introduces the "**this** keyword" he writes that it "*refers to the current instance of an object*". Earlier we saw types used when their instances were meant; now it's the other way around. Objects don't have instances, "current" or not; they themselves are instances of types. Only types (or classes) have instances. So this should simply have said that **this** "denotes a reference to the current object". We learn in chapter 8 about an example class that "*your* Document *class can be stored and it also can be compressed*", but of course this has nothing to do with zipping the class text to send it to your friends, it's supposed to mean that one may store and compress *Document* objects — instances of the class. Or take this explanation, from the same box that helpfully told us what a stack was: we now learn that heap objects are garbage-collected "*sometime after the final reference to them is destroyed*" (an approximate statement anyway, since references are not "destroyed" but get reattached to other objects), whereas

> *the garbage collector destroys stack objects sometime after the stack frame they are declared within ends.*

Rather wrong, even disregarding the phrase "*stack object*" not used in previous discussions (which told us variously that the stack contained "*local variables*" and "*value types (e.g. integers)*" and that "*reference types (e.g. objects) are allocated on the heap*"). To get rid of values on the stack, you don't need a GC; this is simply done on routine exit, as in any block-structured language since Algol 60. It's the beauty of the "last-in first-out" structure of stacks that it takes care of allocating and de-allocating routine data without the need for a more elaborate scheme. The GC provides that elaborate scheme, but for heap objects; it keeps its hands off the stack.

Such confusions and errors reflect on the publisher as well as on the author. Mr. 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 the intricacies of .NET. But readers don't care that he understands the concepts if he can't explain them. Given the list of people who, according to the preface, were "*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 general issues, and the goal that I'd like to pursue for this column: the quality of computer books. In the computer section of a bookstore today, you'll find shelf after shelf of so-called "trade books": hands-on, learn-as-you go titles meant to give you immediate proficiency in the technology du jour. Java Beans in Seventy-Two Hours, XML While You Sleep, ActiveX Controls For The Intellectually Challenged, Scare Away Your Enemies With Your Macromedia Shockwave Skills, Learn XP Administration During Happy Hour. In many bookstores this is the only kind of book you'll find. In the better stores you'll find a shelf or two stocked with books on more highbrow topics, although even there once you have put aside the inevitable UML volumes not much is left that deals with 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, hope to sell well for a limited time, and are put together hastily to catch their audience before someone else does; often they don't care that much about solidity of the concepts or even accuracy. They usually don't

bother to tell you anything else existed before the technology they describe, especially if it came from another persuasion: Java Server Pages books won't acknowledge Microsoft's Active Server Pages, C# books will ignore Java.

Once in a while, you get a book that has the best of both: it talks to the practicing programmer, provides immediately applicable material, and is conceptually sound. The Patterns book by Gamma et al. is an example. But most of the time it seems that we are stuck with having to choose a book that's useful and unreliable or one that's solid and inapplicable. In David Parnas's words: "gadgets without methods or theories without applications".

It doesn't have to be that way and I hope this column can help raise the standards on both sides. I intend to review books from both sides of the aisle, some academic, some utilitarian, questioning whether the former are useful and the latter accurate. Who says that because an author knows about C# and other cool new stuff he shouldn't be able to tell a type from an instance?

I refuse the rift. What I like about *Computer* is that it doesn't shy away from either theory or practice as long as they're good. I intend to follow the same open outlook. A good book — whether about lambda calculus or version 6 of FrameMaker — should have certain basic qualities: it should tell a story and tell it well, distinguish facts from opinion, provide a broad background, treat its reader as a grown-up, bring useful information. Trade books require particular focus, because the rush to market often leads to cutting conceptual corners. One publisher told me that the reason is that "the first decent book on a topic captures the market". Perhaps. But we, the consumers, don't have to accept this situation forever. By submitting trade books to scrutiny — applying the same criteria as for intellectually more ambitious endeavors — we can increase the rewards of careful research, writing, editing and publishing.

Books are important. Many professionals' careers have been shaped by the first books they read. I can certainly remember the four or five which I read as a student and which changed my outlook forever. When interviewing candidate developers, I have found that an effective interview question — first suggested to me by Jean Ichbiah, the designer of Ada — is to ask them to name a few books they found useful. Web pages and magazines help, but nothing will replace the good computer book with its wealth of carefully distilled wisdom.

While it may be critical at times, this column won't be cowardly: as a book author myself, I have left behind enough thousands of printed pages, not counting those yet to come, to provide enough targets for anyone who would want to take me to task. (About the present review I must add, for disclosure, that on the topic of .NET I have published a CD-based training course, am writing a book, and edit a book series.)

I've set a few ground rules. 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 the scope but means that I will have a personal interest in the books reviewed. I may 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 take a laugh at the expense of some unfortunate author who put together an incompetent account of some technology to make a buck; but the interest to readers would be limited, as you undoubtedly have your own BIMs (Baloney Identification Mechanisms) in place. The target of this first review is typical: what's frustrating about its mangling of basic O-O mechanisms is that it's a useful book, which I can in the end recommend as information about C# and .NET. A third edition correcting the conceptual blemishes would be truly excellent. (The second edition merely fixes errata.)

I intend to select interesting books covering interesting topics, ranging from theoretical issues to descriptions of specific technologies, and submit all to the same criteria, challenging 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 recognize and promote quality in technical books, these ever more necessary tools of our trade.