Why your next project should use Eiffel

By Bertrand Meyer

Over its 10-year life Eiffel has evolved into one of the most usable software development environments available today. Other articles in this special section summarize its theoretical contributions; in this essay I will address a more mundane subject: how practical software projects can benefit, today, from the power of Eiffel. In so doing I will largely rely on published assessments from both Eiffel users and book authors. In fact, a quotation from one of the best-known books in the object-oriented (O-O) field—OBJECT-ORIENTED MODELING AND DESIGN by James Rumbaugh and colleagues, the text that introduced the Object Modeling Technique object-oriented analysis method—provides a good start:

> Eiffel is arguably the best commercial object-oriented language available today.[1]

WHAT IS EIFFEL? First we should define what the word *Eiffel* means. If you are thinking "a programming language," you are not wrong (and the preceding quotation shows that you are in good company), since the programming language is indeed the most visible part; but it is only a reflection of something broader: a comprehensive approach to the production of quality software. As Richard Wiener wrote:

> Eiffel is more than a language; it is a framework for thinking about, designing and implementing object-oriented software.[2]

The Eiffel approach includes a method (a "methodology," if you prefer) based on a number of pervasive ideas such as design by contract, seamlessness, reversibility, rigorous architectural rules, systematic use of single and multiple inheritance, static type checking, and several others. Besides a method and a language, Eiffel also means powerful graphical development environments, such as ISE Eiffel, available across a wide number of industry-standard platforms and supporting analysis and design as well as implementation, maintenance, and evolution.

The language itself, indeed (which Wiener calls "an elegant and powerful language for object-oriented problem solving"), is not just a *programming* language but extends to the phases of system construction that both precede and follow implementation. This is sometimes hard to accept if you have been raised in the view that software development must involve a sequence of separate steps; that one should initially use an analysis method and then at some point switch to a programming language, with perhaps a design method in-between. This view is detrimental to the software process and to the quality of the resulting product, as it does not support the inevitable back-and-forth hesitations that characterize real software development.

Wisdom sometimes blooms late in the season. However careful you may have been at the analysis stage, some great ideas will hit you—or your implementers—past the point at which you thought you had all the specifications right. Why renounce the benefit of such belated but valuable ideas? Eiffel and the associated Business Object Notation approach to analysis and design accommodate them naturally, by providing a single conceptual framework from the beginning to the end of the process.

Here Eiffel does not have much competition. The most bright-eyed Smalltalk or C++ enthusiast would not seriously claim that one can do design, let alone analysis, in his or her language of choice. And users of any of the popular O-O analysis notations know that at some stage they must stop working on their model and move on to the implementation in some programming language. Eiffel is unique in helping you for all of these tasks, without ever introducing the *impedance mismatches* that characterize other approaches.

As a recent reviewer wrote:

> As a design language, Eiffel continues to be a better model for object-oriented programming than Ada. It is even better than the new Ada 9X standard.[3]

THE COMMERCIAL AND POLITICAL CONTEXT

In the next few sections I will try to give you a glimpse of the technical contributions of Eiffel or, more precisely, of what other people have written about them. But of course the best technology in the world requires infrastructure and support to succeed.

Eiffel has plenty of these. It has been around for 10 years (the first compiler was available from ISE at the end of 1986, but was started about a year earlier, so that this special section is right on track for the anniversary). Compilers exist from three commercial sources—two in the U.S. and one in Europe—with more to come. Free compilers exist from ISE (you can download it directly from http://www.eiffel.com) and SiG (see the SimTel archive). Full graphical environments start at $69.95 (ISE's Personal Eiffel for Windows). The number of licenses sold is in the tens of thousands. Reusable library classes are in the thousands.

The platforms covered range from Unix (all of Unix, the famous and the arcane) and Linux to VMS, OS/2, Windows 3.1, Windows NT, Windows 95, and the Macintosh. The last major platform not yet addressed, MVS, will be added for the greatest benefit of large financial and corporate users; IBM has just announced a partnership with ISE to bring MVS to Eiffel in 1996.

Particularly impressive is the growth of Eiffel usage in education. Eiffel is quickly becoming the language of choice for teaching modern software technology, including, increasingly, introductory programming. A dozen excellent textbooks are now available from Prentice Hall, Addison-Wesley, Macmillan, and others, with about as many announced just for the coming months. (Someone was remarking recently that there seems to be more Eiffel textbooks than Smalltalk textbooks, even though Smalltalk has been around for so much longer.) Addison-Wesley even has an entire book series devoted to Eiffel: EIFFEL IN PRACTICE.

It is not just the professors who like the approach. Here is just one typical comment on student reaction, from an institution (Rochester Institute of Technology) having adopted Eiffel as its first-year introductory language on a massive scale:

> We were pleased to discover many of our more skeptical students turning around and admitting that Eiffel was a "fun" language in which to work.[4]

A recent COMPUTERWORLD confirmed the need for Eiffel in training the high-powered software professionals of tomorrow. Quoting Amy Cody-Quinn from Management Recruiters International, the journalist writes:

> There is a big problem with people who say they know C++—but they don't really know how to do objects. If they have Eiffel on their résumé, then we know they really have the proper understanding of what they are doing.[5]

But it would be a mistake to think of Eiffel as just an academic tool. A little-known fact is that some of the biggest O-O projects ever undertaken (at least the successful ones—other O-O languages have had their share of large-scale failures) are being done in Eiffel. The hot areas at the moment are banking and the financial industry (in particular some very large derivative trading systems), telecommunications, and health care. These are all areas in which all that counts in the end is quality and time to market, so that project developers need to select the best technology available. Quoting from an article by Philippe Stephan, the system architect of such a project (Rainbow, a major derivative trading system built with ISE Eiffel):

> We evaluated three major object-oriented languages for the project—Smalltalk, C++, and Eiffel—and chose Eiffel. . . . Rainbow currently comprises over 400,000 lines of code, for a total of approximately 3,000 classes. . . . [Current figures are way over these mid-1995 counts.] The developers feel very productive. This was confirmed when Rainbow's financial backers brought in object professionals to audit the project. . . . The auditors evaluated the project during July 1994 and were impressed with the productivity of the Rainbow development group.[6]

The development group in question is remarkable because only a third of its members are software engineers. The others are professionals from other disciplines (such as trading and financial analysis) who, Stephan writes, "can express business concepts in Eiffel because they can focus on design and implementation, rather than struggling with memory management problems and debugging."[6]

The result has received lavish praise from such publications as COMPUTERWORLD and analysts:

> Industry experts briefed on Rainbow said they were impressed with the results. CALFP is "progressive" in . . . committing the organization's mission-critical systems development efforts to this architecture, said Richard Crone, senior manager of financial services at KPMG Peat Marwick in Los Angeles. "What's unique here is that [CALFP is] delivering this system end-to-end using object-oriented technologies," said Henry Morris, a research analyst at International Data Corporation (IDC) in Framingham, Mass.[7]

Along with these Eiffel megaprojects, you will also find myriad smaller endeavors. Many consultants, in particular, have found for themselves the key competitive advantage that they can gain from Eiffel's excellence. In ensuring this spread of Eiffel throughout the industry, the benefit of cheap yet complete environments such as ISE Eiffel for Linux has been immeasurable.

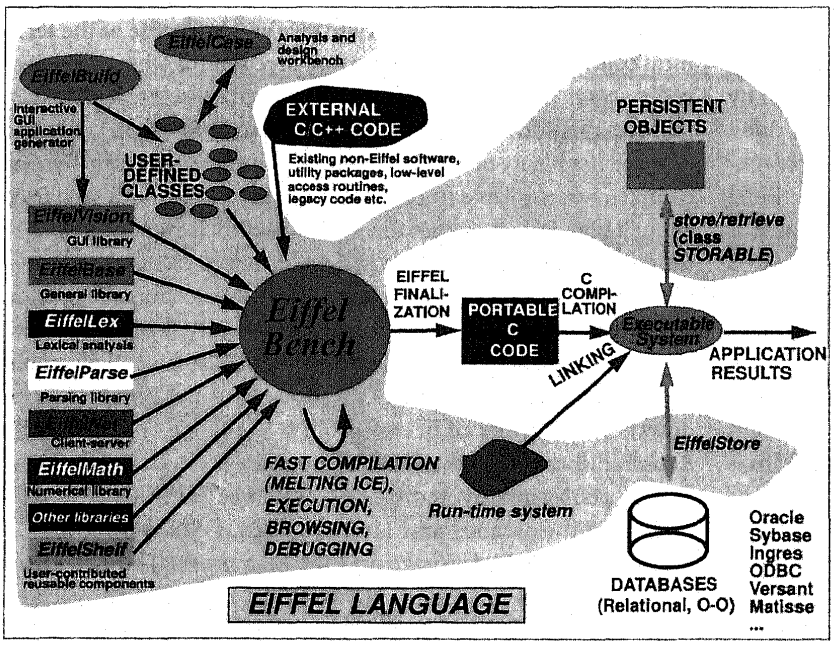

Figure 1.

Also crucial to the development of Eiffel has been the neutral status of its definition, now controlled by a consortium of vendors and users, the Nonprofit International Consortium for Eiffel (NICE). NICE has already produced a library standard and expects in 1996 to produce the language standard, which should shortly thereafter enjoy a smooth ride through ANSI and other international standards bodies.

The pace of Eiffel history has been accelerating in the past few months. This has been picked up by many journalists. As Dan Wilder wrote:

> With an open specification for both the language and the kernel libraries, and support from multiple vendors, Eiffel now stands poised to take off.[8]

THE CRITERIA Eiffel—the method, the language, the environment—is based on a small set of goals, addressing the crucial needs of software quality and pro-

ductivity. Quoting from a recent review of Tower Eiffel in BYTE magazine:

> Developers who want an object-oriented language that adheres to the keystone principles of software engineering need look no further than Eiffel.[9]

Or, as Steve Bilow wrote in a review of ISE's Melting Ice compiling technology (which he calls "an outstanding marriage between portability and development speed"):

> Eiffel was designed precisely for the purpose of enabling software developers to deliver high quality, reliable, efficient, extensible, reusable code.[10]

RELIABILITY The first goal is reliability. No other approach available today has made the effort to give developers all the tools that they need to produce correct and robust software—software that will run without bugs the first time around. Crucial in this effort is the presence of static typing (*real* static typing, not "a little bit typed" as in those languages that still keep C-like type conversions); assertions and the whole mechanism of design by contract, about which more than one Eiffel developer has said "this has changed my life" by enabling him or her to specify precisely what the software should do, and to track at runtime that it does it; disciplined exception handling; automatic garbage collection, which eliminates a source of horrendous bugs in C-based environments (and a large part of the code); a clean approach to inheritance; the use of dynamic binding as the default policy, meaning the guarantee that all calls will use the right version of each operation; and the simplicity of the language design, which enables Eiffel developers to know *all* of Eiffel and feel in control.

The role of assertions and design by contract is particularly important here. According to a recent article in the JOURNAL OF OBJECT-ORIENTED PROGRAMMING:

> The contribution of Eiffel is significant: it shows how invariants, preconditions, and postconditions can be incorporated into a practical developer's view of a class. Wider use of Eiffel . . . will encourage a greater use of simple but powerful mathematics during development.[11]

REUSABILITY The second goal is reusability. This has become a catchword, but Eiffel is the only approach that has taken this requirement and its consequences all the way to the end. Quoting Roland Racko in SOFTWARE DEVELOPMENT:

> Everything about [Eiffel] is single-mindedly, unambiguously, gloriously focused on reusability—right down to the choice of reserved words and punctuation and right up to the compile time environment.[12]

Eiffel benefits here from being a simple ("but not simplistic," writes Racko) and consistent design, not a transposition from older, unrelated technology. Beyond the language and the environment facilities (such as precompilation), the crucial help to reusability is of course the presence of thousands of high-quality library classes, such as, in ISE Eiffel (see Fig. 1), EiffelBase (a "Linnaean approach to the reconstruction of software fundamentals"), EiffelNet for client/server communication, EiffelStore for relational and O-O database manipulations, EiffelLex and EiffelParse for lexical analysis and parsing, EiffelMath for object-oriented numerical computation, EiffelVision for portable graphics, the Windows Eiffel Library for Windows-specific graphics, and many others. Various suppliers have their own libraries, such as the Booch components for Tower Eiffel and GRAPE for Eiffel/S. Not even mentioning quality, the result is probably the biggest repository of O-O components available anywhere. The care that has been applied to the production of these libraries also has considerable pedagogical benefits: the way people learn Eiffel is by learning the libraries—first to use them, then to adapt them if necessary, then to write their own software.

Part of the single-mindedness mentioned by Racko is the emphasis on abstraction. In contrast with, say, Smalltalk, you do not read the source code of a class when you want to use it. This may be fine for a couple dozen classes, but not for a large, powerful library. Eiffel introduces the notion of *short form*: an abstract version of the class, keeping only the interface information, including assertions. This is an ideal tool for documenting classes but also for discussing designs and presenting them to outsiders—managers or customers—who need to know what is going on without getting bogged down in the details.

Let me mention just one of the unique reusability-supporting features of Eiffel, without which it is, in my experience, impossible to have a long-term reuse effort. Racko again:

> The language's designer . . . recognized that no reusable library is ever perfect and, thus, that libraries are always in flux. So he built a kind of version-control system into the language. Specifically, there are language elements to demarcate obsolete code that is, however, still being supported. When these elements are referenced by someone unaware of such code's obsolescence, the compiler will issue a warning at compile time about the impending doom that awaits persons who continue the referencing.[12]

It is this kind of detail that can make or break the success of reuse in a company.

EXTENDIBILITY Next comes extendibility. With Eiffel, modifying software is part of the normal process. As Philippe Stephan writes of the external audit of his project: "The auditors rated the responsiveness of the development team as very high."[6]

Chief among the method's support for extendibility is the careful design of the inheritance mechanism. Unlike Smalltalk, which is fatally limited by the absence of multiple inheritance, the Eiffel approach fundamentally relies on multiple inheritance to combine various abstractions into one. As Dan Wilder notes:

> Most object-oriented languages do not attempt multiple-inheritance. The literature is full of elaborate explanations why. This is sad. Eiffel demonstrates that multiple inheritance need not be difficult or complex, and it can also yield some quite practical results.[8]

The approach also enforces a strict form of information hiding, which means that a module (a *client* in Eiffel design-by-contract terminology) that uses another's facilities (its *supplier*) is protected against many of the changes that can be made later on to these facilities. This is essential in preserving the coherent evolution of a large system—and the sanity of its developers.

EFFICIENCY Performance is almost as much an obsession

in Eiffel as reusability. The software field is still, and will remain for a long time, largely driven by performance considerations. (Do not believe anyone who says that speed does not matter. If we get faster computers, it is to do things faster and especially to do more things—not to use more CPU cycles to run the same old applications at the same old visible speed.)

There is no reason whatsoever to leave the mantle of efficiency to the proponents of machine-oriented languages such as C/C++, or to follow the path of Smalltalk, which sacrifices performance to object orientation. With Eiffel, to use Steve Tynor's favorite phrase, you can "have your cake and eat it." Thanks to a performance-obsessed language design and 10 years of research and competition on compiling algorithms, the speed of Eiffel-generated code (in such modes as what is known as "finalization" in ISE Eiffel) is as good as that of hand-produced C code, or better.

Software producers should stand up to their ideas. That is what we do at ISE: apart from the runtime engine (a few thousand lines of C), all of our software—thousands of classes, hundreds of thousands of lines—is written in Eiffel, and it runs fast. Typical of the situation is a recent incident with the EiffelLex library: it still had a few C elements, remnants of an earlier design. We rewrote them in Eiffel—for a 30% performance gain.

Why these gains? The answer is simple. The C/C++ approach of doing everything by hand, under tight programmer control, works well for small programs. Similarly, a good secretary has no equivalent for keeping one person's records. But in the same way that no humans can match the performance of a computer for managing, say, the records of a bank or a city, no programmer can beat a sophisticated Eiffel compiler for optimizing a large program. Against the automatic application of inlining, static binding, memory management, and other optimizations, the human does not stand a chance.

To have one's cake and eat it also means not to have to choose between runtime and compilation-time performance. For programmers used to the contrast between a Smalltalk-like style of rapid turnaround and the interminable edit-compile-link cycle of most compiled environments, the following comments by Dan Wilder will be shocking:

> ISE Ebench uses "melting ice technology," which allows incremental changes to run in an interpreted mode. Only modified classes are recompiled. Changing one class and clicking the Melt button caused only a few seconds of compilation. . . . My test application took 20 seconds to compile from scratch in "melt" mode.[13]

Steve Bilow provides further explanations:

> Based on the observation that software development is an iterative process which is usually focused on constructing systems from code modifications, the folks at ISE have developed something that they call "Melting Ice Technology." Essentially, this means that when you make a [change] and you want to try it out, you simply "melt" it into the system. You don't need to regenerate a bunch of C code, so your changes are integrated into the system proportionally to the amount of code changed. Even in C and C++, 'make' still has to relink.[10]

What this also indicates in passing is the technology choice made by ISE Eiffel and all current implementations: using C as the portable implementation vehicle. By going through C, the compilers gain efficiency and portability. This also makes Eiffel one of the most open environments around; in contrast to the self-centered view that predominates in Smalltalk, Eiffel software is born with a sociable attitude, ready to interface with all kinds of other software written in C or other languages. This, needless to say, is a key to the success of realistic applications.

WITH US, EVERYTHING'S THE FACE A good way to think about Eiffel—the seamlessness of it, the insistence on getting everything right, the conviction that software should be beautiful in and out, specification and implementation—is this little anecdote that I steal from Roman Jakobson's essays on general linguistics:

> In a far-away country, a missionary was scolding the natives. "You should not go around naked, showing your body like this!" One day a young girl spoke back, pointing at him: "But you, Father, you are

continued on page 82