Principles of language design and evolution

Bertrand Meyer

Interactive Software Engineering
ISE Building, 356 Storke Road, Goleta, CA 93117 USA http://www.eiffel.com

Heeded or not, Tony Hoare's *Hints on Programming Language Design* [1] remains, more than 25 years after publication, the principal source of wisdom on how to produce sound programming languages. I will try to expand on Hoare's principles by presenting some of what my own experience has taught me, through my work not only on Eiffel but also on numerous "little languages" as well formal specification languages such as Jean-Raymond Abrial's Z [2], and through a lifetime passion for critical observation of languages of all kinds, from JCL, Fortran, troff, csh and awk to Miranda, Java, Perl, and XML.

The topic is not just language *design* but the often neglected case of language *evolution*. In the same way that a software engineering curriculum misses its target if it confines itself to initial program construction and fails to address the successive mutations that in the end account for most of the work on a real program, a discussion of language design must encompass the successive revisions that mark the life of a language — especially a *successful* language — and constantly threaten to annul whatever qualities its original version may have had.

A good part of the discussion will be drawn from the appendix on language design of the first edition of "*Eiffel*: *The Language*" [3], the reference on Eiffel.

1 THE BONZAI AND THE BAOBAB

One view of design holds that good languages should be small. For many years the best way to discredit any proposed design was to hint at similarity with PL/I. Just uttering that name from the back of the room was guaranteed to bring laughter to the audience and ridicule to the presenter. But many successful languages are large and complex; C++ is the most obvious example, but Java is just as typical; a look at the description of Java initialization semantics at *http://www.javaworld.com/javaworld/jw-03-1998/jw-03-initialization.html* should be enough to dispel any suspicion of simplicity.

Oversize has many damaging consequences: making it harder to learn the language; causing surprises even to experienced users, since they often will master only a subset, and may involuntarily use properties they don't know; increasing the likelihood that compilers will be buggy, bloated, and late.

Citation reference: Bertrand Meyer, *Principles of Language Design and Evolution*, in *Millenial Perspectives in Computer Science* (Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare), eds. Jim Davies, Bill Roscoe and Jim Woodcok, Cornerstones of Computing, Palgrave, 2000, pages 229-246. The present version, pre-copy-editing, reflects the author's intent.

But languages should not be too simple, and the language designer should not resist useful additions on principle. One can conjecture that Pascal could have had a much more significant industrial role if a few extensions (such as variable-length array access and an elementary module facility) had been included in the standard in the late nineteen-seventies or early eighties. They were not, and Pascal was largely displaced by C, certainly a regrettable development for software engineering.

So the truth has to be somewhere between the monsters of complexity and the zen-like masterpieces of ascetism — between the bonzai and the baobab.

To complicate the discussion, there is no single definition of size. The Eiffel language book occupies 594 pages, and the ongoing third edition [4] will probably reach into the 800s, which would seem to suggest that Eiffel is complex. But then if you read the book you will realize that most of these pages are devoted to comments and explanations, and it is possible to talk about pure Lisp (or for that matter about love, another seemingly simple concept) over many more pages. Then if you consider that the syntax diagrams occupy only four pages, Eiffel is very simple. From yet another viewpoint, the language properties that enable a beginner to start writing useful software may be defined in the 20 pages of chapter 1; that is pretty short too. A “reference only” extract of the book, retaining only the formal rules (syntax, validity, semantics) interspersed throughout the text, would occupy about 40 pages.

We could paraphrase a famous quote and state that a language should be as small as possible but no smaller. That doesn’t help much. More interesting is the answer Jean Ichbiah gave to the journalist (for the bulletin of INRIA) who, at the time of Ada’s original publication, asked him what he had to say to those who criticized the language as too big and complex: “Small languages”, he retorted, “solve small problems”.

This comment is relevant because Ada, although undoubtedly a “big language”, differs from others in that category by clearly showing (even to its critics) that it was *designed* and has little gratuitous featurism. As with other serious languages, the whole design is driven by a few powerful ideas, and every feature has a rational justification. You may disagree with some of these ideas, contest some of the justifications, and dislike some of the features, but it would be unfair to deny the consistency of the edifice. Consistency is indeed the key here: size, however defined, is a measure, but consistency is the goal.

2 CONSISTENCY

Consistency means having a goal: never departing from a small number of powerful ideas, taking them to their full realization, and not bothering with anything that does not fit with the overall picture. Transposed to human affairs this may lead to fanaticism, but for language design no other way exists: unless you apply this principle you will never obtain an elegant, teachable and convincing result.

Note the importance for the selected ideas to possess both of the properties mentioned: each idea should be *powerful*, and there should be a *small number* of them. Eiffel may be defined by something like twenty key concepts. Here, as an illustration, are a few of them:

- Software architectures should be based on elements communicating through clearly defined *contracts*, expressed through formal preconditions, postconditions and invariants.
- *Classes* (abstract data types) should serve as both modules and types, and the modular and typing systems should entirely be based on classes. (Two immediate consequences are that no routine may exist except as part of a class defining its target type, and that Eiffel systems do not have a main program.)
- Classes should be *parameterizable* by types to support the construction of reusable software components.
- *Inheritance* is both a module extension facility and a subtyping mechanism. Attempts to restrict the mechanism to only one of these aspects, in the name of some misdirected attempt at purity, only serve to trouble the programmer with irrelevant questions. Attempt to portray *multiple* inheritance as evil only stem from clearly inadequate uses, or badly conceived language mechanisms.
- The only way to perform an actual computation is to *call* a (dynamically bound) feature on an object.
- Whenever possible, software systems should *avoid explicit discrimination* between a fixed list of cases, and instead rely on automatic selection at run time through dynamic binding.
- Client uses of classes should only rely on the official *interface*.
- A strong distinction should be maintained between *commands* (procedures) and *queries* (functions and attributes).
- A *contract violation* (exception) should lead to either organized failure or an attempt to achieve the contract through another strategy.
- It should be possible for a static tool to determine the type consistency of every operation by examining the software text, before execution (*static typing*).
- It should be possible to build sophisticated *run-time object structures*, modeling the often complex relations that exist in the external systems being modeled, and to let the supporting implementations take care of *garbage collection* to reclaim unused space automatically.

Eiffel is nothing else than these ideas and their companions taken to their full consequences.

Why is consistency so important? One obvious reason is that it determines your ability to teach the language: someone who understands the twenty or so basic ideas will have no trouble mastering the details, and from then on will remember most of them without having to go back all the time to the manual.

Another justification of the consistency principle is that with more than a few basic ideas the language design becomes simply unmanageable. Language constructs have a way of interacting with each other which can drive the most careful designers crazy. This is why the idea of orthogonality, popularized by Algol 68, does not live up to its promises: apparently unrelated aspects will produce strange combinations, which the language specification must cover explicitly.

An extreme example in Eiffel is the combination of the *obsolete* and *join* mechanisms, two seemingly unrelated facilities. A class may declare a feature as obsolete to prepare for its eventual removal without destroying existing software; this is a fundamental tool for library design and evolution. In the inheritance mechanism, a class may merge ("join") features inherited from different parents. No two mechanisms seem at first sight more "orthogonal" with each other. Yet they raise a specific question: the Join rule must give all the properties of the feature that results from joining a few inherited features, in terms of the properties of the inherited versions; but then one of these features may be obsolete. Not the most fascinating use of language facilities; but there is no reason to disallow it. (This would require an explicit constraint anyway, and simplicity would not be the winner.) Now does this make the joined version obsolete? The language specification must give an answer. (The answer is no.)

Such cases should suffice to indicate how crucial it is to eliminate anything that is not essential. Many extensions, which might seem reasonable at first, would raise endless questions because of their possible interactions with others.

Another interesting example of interference is the absence of garbage collection in most C++ implementation. Although often justified *ex post facto* in the name of the C philosophy of putting the programmer in control of every detail, this limitation is in reality a consequence of the language's design: the presence of C-style casts makes it possible to disguise a pointer into something else, thus fooling a garbage collector and leading to serious potential errors. Many programmers do not realize how a seemingly remote property of the type system exerts such a direct influence on the very practical issue of memory management.

3 UNIQUENESS

Taken to its full consequences, the principle of Consistency implies the principle of Uniqueness, which states that the language design should provide one good way to express every operation of interest; it should avoid providing two.

This idea explains, for example, why Eiffel, almost alone among general-purpose languages, supports only one form of loop. Why offer five or six variants (test at the beginning, the end or the middle, direct or reverse condition, "for" loop offering automatic transition to the next element etc.) while a single, general one will be easy to learn and remember, and everything else may be programmed from it?

The loop example deserves further attention. A well-written Eiffel application will have few loops: a loop is an iteration mechanism on a data structure (such as a file or list); it should be written as a general-purpose routine in a reusable class, and then

adapted to specific contexts through the techniques illustrated in the discussion of iterators. (Such pre-programmed iteration mechanisms are indeed available from libraries, and made more attractive by the recent addition of an *agent* mechanism to the language.) Then having to write *i* := *i* + *1* manually for the equivalent of a For loop is not a problem.

This observation, which would not necessarily transpose to another language, illustrates an important aspect of the Eiffel method, which makes almost all "*X* considered harmful" observations, for arbitrary *X*, obsolete. The resolution of this apparent paradox comes from the power of object-oriented abstraction. As soon as you recognize some pattern *X* as useful, this immediately makes it harmful, by suggesting that you should not from then on reproduce *X*-like patterns in your software texts, but instead hide *X* in a reusable software component and then reuse that component directly.

Loops are harmful, then, not because they pose a danger by themselves (as may be argued of goto instructions), but because their very usefulness as a common pattern of data structure traversal suggests packaging them in reusable components describing higher-level, more abstract forms of these patterns. The only danger here would be long-term — not taking advantage of potential reuse.

The principle of Uniqueness is a particularly useful guide for language evolution, after initial design. It is natural for users of a language to request new facilities that simplify their job. Most of the time, it was possible to do this job before, which suggests that the principle requires rejecting these extensions. But that's not necessarily a correct interpretation, since the principle requires providing one *good* way of addressing each need. The question then becomes whether the previous way is good enough.

Creation expressions provide a good example. Until recently, Eiffel had a creation instruction (to create and initialize an object) but no creation expressions. The language design discussion in [1] explains the rationale in detail, stating, however, that creation expressions might have a role in the future. That future has come. Along with a creation instruction

create *x*.*make* (...) [A]

which creates an object of the appropriate type, attaches it to *x*, and initializes it with the given procedure and arguments, you may also write

x := **create** {*TYPE*}.*make* (...) [B]

where *TYPE* is the type of *x*[1]. Is this a violation of the principle of Uniqueness? As presented, yes. But in practice no good programmer will ever use form [B] in the case given, because there is a better way: form [A], which avoids the need to specify the type. Why specify *TYPE* since (the language being strongly typed) it follows from the declaration of *x*? There is no good reason. Creation expressions, however, are useful in another case: creating an object whose only use is to be passed as an argument to a routine. Then you can write

1. In both cases some variations and simplifications are available, especially for omitting the creation procedure if it is the *default_create* associated with a class and guaranteed to preserve the class invariant.

```
some_routine (..., create {TYPE}.make (...), ...)
```

where the phrasing would be far more cumbersome if we only had creation instructions:

```
new_object: TYPE        -- Declare local variable just for this purpose
...
create new_object.make (...)
some_routine (..., new_object, ...)
```

Extensive, experienced users found that such schemes occurred frequently and caused useless effort and distraction. It's not a matter of keystrokes, as a longer form is *preferable* when it adds relevant information; it's a matter of not wasting one's time in repetitive schemes that bring nothing new and obscure the truly relevant parts of the software.

So the two mechanisms, creation instructions and creation expressions, are both useful because they cover complementary needs.

A similar example is "Unique" values and "Inspect" instructions. Because of Eiffel's emphasis on avoiding explicit discrimination and relying on dynamic binding instead, all in the name of modular, extensible, reusable architectures, the language did not initially (until 1989) include multi-branch mechanisms. As experience grew, it became clear that such mechanisms were still needed in some cases, where they did not conflict with object-oriented principles. Hence the introduction of Unique values (integer constants whose values do not have to be specified by the programmer, being instead chosen by the compiler) and Inspect instruction (a kind of **case** ... **of** discriminating on integers or characters). It is significant that the original solution erred on the side of caution: only when extensive experience clarified the conditions under which explicit discrimination was still legitimate did we go for the corresponding extensions. Better be restrictive at first, and loosen the strings later when you fully understand what's truly needed and what would be mere featurism.

4 TOLERANCE AND DISCIPLINE

Using the word "restrictive" reminds us of the somewhat disciplinarian attitude that is not infrequent in the software community. One commonly hears such phrases as "preventing the programmers from doing their dirty tricks". It is as if language designers were invested with a moral mission, and languages served as ramparts against the threat of the developers' natural uncleanliness.

I disagree with this view. (This will seem surprising to those who have heard Eiffel being categorized, I believe quite wrongly, as a language of the restrictive school.) Programming language designers are not in the chastity belt business. Their role, to repeat a comment which I first heard many years ago — at a Marktoberdorff summer school — from C.H.A. Koster, is not to prevent developers from writing bad software (a hopeless endeavor anyway), but to enable them to write good software; and perhaps to make the task pleasurable as well.

This must be applied together with the principle of Uniqueness. If you exclude a certain facility, be it the goto or function pointers, it is not to save humanity from some abomination (although you may also be doing that) but because you are providing elsewhere a better way to achieve the goals which the excluded constructs purported to

address. Loops and conditionals are better than gotos, and dynamic binding under the control of static typing is better than function pointers or explicit discrimination.

In other words, if a design is defined as much by what it leaves out as by what it includes, one cannot justify the exclusions without knowing the inclusions.

These ideas pervade Eiffel. The language's ambition is to support an elegant and powerful method for analysis, design, implementation and reuse, and to help competent developers produce high-quality software. The method is precisely defined, and the language does not attempt to promote any other way of developing software; but it also does not attempt to prevent its users from applying their creativity.

The details of the inheritance mechanism provide a clear example of these principles. The relation between inheritance and information hiding is a controversial topic; Eiffel takes the view that descendants should be entirely free to define the export status of inherited features, without being constrained by their ancestors' choice. Nothing really forces everyone to agree: a project leader may take a more restrictive approach and, for example, prohibit the hiding of a feature exported by a parent. It is not difficult to write a tool that will check adherence to this rule. Had the language specification taken the restrictive stand, it would have been impossible for a project leader to enforce the inverse policy.

In summary: language designers should not exclude "bad" constructs out of a desire to punish or restrict the users of the language; that is not their job. The exclusions are justified only by the inclusions: the designer should focus on the constructs that he deems essential, and his responsibility is then to remove everything else, lest he produce a monster of complexity.

5 METHODOLOGY

In a bad language design, the programmer is presented with a wealth of facilities, and left to figure out when to use each, when not, and which to choose when more than one appears applicable.

In a good design, each language facility goes with a precise theory — presumably explained in the accompanying book or books — of the purpose it serves: when it is desirable, when it is not.

The example of Unique values and Inspect constants in Eiffel, discussed earlier, provides a typical example. Few constructs are bad by themselves (one might even argue for a goto instruction in an assembly language). What makes a construct desirable or detrimental is the software development methodology that the language reflects. It was prudent to refrain at first from including explicit discrimination constructs, since they seem to fly in the face of the object-oriented method (i.e. data abstraction and Design by Contract principles). When, later on, we started to get a better appreciation for the role such constructs may retain in a development process that flawlessly applies these principles, it became safe to reintroduce a carefully designed variant which fits precisely in the method.

A counter-example of the Methodology principle is the use of argument overloading in C++ and Java, a somewhat extreme case of a flexibility that has zero advantages and more than a few disadvantages. In an object-oriented language, you get a powerful form of *dynamic* overloading: the guarantee that $x.f$ will trigger the exactly appropriate version of *f*, determined anew for each execution of the call depending on what kind of object *x* happens to denote. The semantics of every such variant of *f* must be compatible with the basic specification expressed by the *contract* of *f* in the common originating class. So you use the same name to denote *different variants* of the *same basic semantics* determined at *execution time*. The mechanism provides a notable increase in expressive power, since it offloads some of the programmer's manual work (choosing between variants of an operation) to an automatic mechanism provided by the computer. As an added bonus, the effect on the flexibility of software architectures, the extendibility of the resulting programs, and the reusability of their components are profound and beneficial.

"Static" overloading of the kind provided by the languages cited does nothing of the sort (see [5] for more details). all that overloading provides is the ability to use the same name to denote *different semantics* determined at *compilation time*, also known in less technical parlance as "shooting yourself in the foot". For different operations, one should choose different names; using the same name just creates confusion and errors. The mechanism adds not an ounce of expressive power: there is nothing you can do with overloading that you couldn't do just as well (better, in fact, since the text will be more clear) without.

In addition, the criterion used to disambiguate overloaded routines is wrong. Competing routines must have different argument signatures, which will enable the compiler (and the poor human reader) to determine which is desired in every case. The very first example of a "constructor" used in almost every textbook disproves the wisdom of that criterion. If you have a *POINT* or *COMPLEX_NUMBER* class, it will most likely have a constructor that takes cartesian coordinates, and one that takes polar. Unfortunately, both take the same arguments: two real numbers. Missed.

Even more fundamentally, overloading destroys the fundamental simplicity and beauty of the object-oriented model, where the basic construct, the class, represents a mapping from operation names to operations. (With overloading you may have any number of operations associated with an operation name for a given class.). This makes it impossible to reason about programs and programming models, destroying the whole semantic edifice for the benefit of a dubious syntactical convenience.

As if this were not enough, static overloading leads to more confusion and complex semantic rules when used in *conjunction* with polymorphism and dynamic binding. This is (at least in an object-oriented language) a rather stark example of a language trait that contradicts the basic methodology of writing good programs.

6 MEA CULPA, MEA MAXIMA CULPA

The surest sign of a problematic design is the presence, in a language manual, of comments stating that some constructs should never be used. A typical example in the C++ and Java literature is the (justified) advice to avoid direct assignments to fields of objects, as in *x.a := b*, which indeed violate all the principles of information hiding and object technology. (It is possible, as in Delphi, to provide *x.a := b* as a syntactical abbreviation for the procedure call *x.set_a (b)*, but this is not what the C++ and Java mechanisms provide; they are direct field assignments, incompatible with modern principles of data abstraction and information hiding.)

The natural question — especially for such a recent design as Java, which does not have the excuse of being constrained by the requirement of full compatibility with C — is how one can justify producing a programming language and immediately starting to warn users against certain facilities. If the designer truly thinks (asks the naïve observer) that a certain construct is harmful, could he perhaps not have refrained from including it in the first place? Is the designer not the one who decides what goes in and what stays out?

Loving your language means never having to say you're sorry.

7 THE LANGUAGE AND THE LIBRARIES

In a method supporting reusability, it is often possible and desirable to provide a new feature through a library facility rather than through a language change.

Like some other languages, Eiffel uses libraries for mechanisms such as input and output, rather than defining language constructs. The inheritance mechanism also provides a class *ANY*, inherited by all classes and offering them a number of crucial general-purpose features: *copy*, *clone*, *deep_clone* (producing recursive copies of arbitrarily large and complex object structures), equality, *out* (which produces a printable image of any value or object).

Other powerful library mechanisms include the *STORABLE* class, providing a straightforward way to store an object structure — again, arbitrarily large and complex — into a file, or to transmit it across a network, in a machine-independent format if desired.

A cynic might question the benefit of extending the libraries to keep the language simple. Indeed, tough problems of consistency and simplicity do arise for libraries. There is an important difference, however: one of level. The library as well as any user application are defined with respect to the basis provided by the language. Because everything else relies on it, this basis must be kept simple at all costs. Complexity should be avoided in libraries too, of course, but the consequences are less grave.

Mathematical theories provide the appropriate comparison. Adding a language construct is like adding an axiom, certainly not a decision to be taken lightly. Adding a library class or routine is simply like adding another theorem, inferred from the current axioms.

The interaction of libraries and language in Eiffel is sometimes intricate. The basic exception mechanism is very simple; class *EXCEPTIONS* provides further tuning, for example to handle various kinds of exception differently, or to ignore certain signals. Similarly, *MEMORY* allows for fine control over the operation of the garbage collector. *INTERNAL* gives access to the internal structure of objects, useful to write system-level tools or interfaces to databases.

Arrays are not a language construct but come from a library class *ARRAY*, since an array can be described as an abstractly specified object, in the same way as a list or a stack; this greatly simplifies the language and makes programs more consistent and readable. The notion of *TUPLE* is handled in a similar way. In both cases, there is a language connection through special syntax for manifest arrays or tuples.

Similarly, all basic types, from *INTEGER* to *BOOLEAN* and *STRING* are formally treated as classes. Unlike the solution of C++ and Java (which separates the basic types from the rest of the type system) this makes it possible to have container structures — lists, trees, hash tables, arrays — that may contain integers, characters and strings as well as instances of programmer-defined classes. To the programmer, the basic types are indeed normal classes, which can be browsed through the normal tools, and have their proper place in the inheritance hierarchy (*INTEGER*, for example, inherits from *NUMERIC*, describing number-like objects belonging to a set with the structure of a ring, and *COMPARABLE*, describing objects belonging to a set equipped with a total order relation). The compiler, however, cheats since it knows about these classes and can generate better code for them. This is an attempt to combine the best of both worlds: the consistency, simplicity and elegance resulting from a uniform type system; and the efficiency resulting from special knowledge.

8 ON SYNTAX

One of the most amusing characteristics of the software development community, from a language designer's viewpoint, is the discrepancy between professed beliefs and real opinions on the subject of programming language syntax. The official consensus is that syntax, especially "concrete" syntax (governing the textual appearance of software texts) does not matter. All that counts is structure and semantics.

Believe this and be prepared for a few surprises. You replace a parenthesis by a square bracket in the syntax of some construct, and the next day a million people march on Parliament to demand hanging of the traitors.

Of the pretense (syntax is irrelevant) and the actual reaction (syntax matters), the one to be believed is the actual reaction. Not that haggling over parentheses is very productive, but unsatisfactory syntax usually reflects deeper problems, often semantic ones: form betrays contents.

Once a certain notation has made its way into the language, it will be used thousands of times by thousands of people: by readers to discover and understand software texts; by writers to express their ideas. If its esthetically wrong, it cannot be successful.

There is no recipe for esthetic success, but here again consistency is key. To take just one example, Eiffel follows Ada in making sure that any construct that requires an instruction (such as the body of a Loop, the body of a Routine or a branch of a Conditional) actually takes a sequence of instructions, or Compound. This is one of the simple and universal conventions which make the language easy to remember.

For syntax, some pragmatism does not hurt. A modern version of the struggle between big-endians and little-endians provides a good example. The programming language world is unevenly divided between partisans of the semicolon (or equivalent) as terminator and the Algol camp of semicolon-as-delimiter. Although the accepted wisdom nowadays is heavily in favor of the first approach, I belong to the second school. But in practice what matters is not anyone's taste but convenience for software developers: adding or forgetting a semicolon should not result in any unpleasant consequences.

In the syntax of Eiffel, the semicolon is theoretically a delimiter (between instructions, declarations, Index_terms clauses, Parent parts); but the syntax was so designed as to make the semicolon syntactically redundant, useful only to improve readability; so in most contexts it is optional.

This tolerance is made possible by two syntactical properties: an empty construct is always legal; and the use of proper construct terminators (often **end**) ensures that no new component of a text may be mistaken for the continuation of the previous construct. For example in

x := *y*

a := *b*

there is no syntactic ambiguity, even without a semicolon, since no construct may involve two adjacent identifiers such as *y* and *b*.

It is interesting to note here that the study often invoked to justify the C-Java-Ada style of semicolon as terminator (Gannon and Horning, IEEE *TSE*, June 1975) actually used subjects that were trained in PL/I and a test "separator" language that (apparently) treated successive semicolons as an error, a completely unrealistic assumption. This seems to invalidate the piece of conventional wisdom that asserts separators are better than terminators. The experience of Eiffel since semicolons were made optional massively suggests that semicolons are in most cases a mere nuisance. (See the detailed analysis in reference [5].)

Another example of the importance of syntax is the dominant practice, in the C-C+-Java-Perl etc. world, of the equality symbol = as assignment operator, going against centuries of mathematical tradition. Experienced programmers, so the argument goes, will never make the error. In fact they make it often. A recent review of the BSD operating system source, performed over one week-end, identified three cases of if (x = y) — a typo for if (x == y) which, unfortunately, is legal in C and C++ although it leads to unexpected results. (In Java, at least, the first form is invalid so the error will have no catastrophic consequence.) Syntax matters.

9 THE INVENTOR AND THE ASSEMBLER

One of the most original comments in Hoare's *Hints* (inspired, it seems, by the experience of Algol W) is the suggestion that the two main tasks of language design are best handled by different people: one proposes constructs, the other refrains from invention but assembles other people's suggestions into a coherent engineering construction.

The design of Eiffel has tried to disprove this rule. Eiffel embodies a significant number of inventions. Although many have been contributed by other people, a number of the concepts were devised and integrated in a single process. They include such ideas as once routines for shared objects and decentralized initialization, the multiple inheritance mechanism, object-oriented contracts and their relation with inheritance, renaming, and many others. I hope the result shows that the roles of construct inventor and system assembler are in fact compatible.

10 FROM THE INITIAL DESIGN TO THE ASYMPTOTE

Although the programming literature contains a few references on language design, less attention has been devoted to the subject of evolution after initial design. Yet successful languages live and change; none of the major languages in use today still adheres to the letter of its original definition. How do the design principles governing the childhood of a language carry over to adolescence and adulthood?

Software developers are inordinately opinionated people, especially on the subject of languages. Inevitably, they will come up with requests for change and extensions. Add to this tremendous and constant source of ideas the contribution of co-workers, users, course participants, colleagues in panels at conferences, and you get a constant influx of new ideas.

In the current state of technology a new element, exciting and sometimes frightening, complements these traditional sources of input: the net. Electronic mail, Usenet forums and specific discussion groups (such as available through Talkitover.com, used for current Eiffel discussions [4]) mean that thousands of people can learn in a few hours about the latest announcements, ideas, proposals, opinions and suggestions — and react to them. For Eiffel this has been a tremendous benefit. The number of people who have sent public or private comments is incomparably greater than what it would have been just a few years earlier. Even Ada, probably the language most widely and thoroughly debated before its final design, was born before network access became available on a grand scale, and did not benefit from the unique combination of breadth, depth and timeliness made possible by today's technology.

It is striking to see how many of these ideas are in fact excellent; but this does not mean that they should all be included!

First they may raise subtle or major incompatibilities with other language features; but even if this is not the case they will make the language more complex. The designer must weigh the evidence: is the purported benefit really worth the increase in complexity? In nine out of ten cases the answer is no. Again this usually is no reflection

on the quality of the idea. But the designer's primary responsibility is to keep in mind the elegance of the overall picture.

What can one do in such a context? The best tactics is to say "no", explain that you are on your way to Vladivostok, and emerge some time later to see if there is still anyone around. This is the basic policy: do not change anything unless you cannot find any more arguments for the status quo.

But saying "no" most of the time is not an excuse for not listening. Almost any single criticism or suggestion contains something useful for the language designer. This includes comments by novices as well as expert users. Most of the time, however, you must go beyond what the comment says. Usually, what you get is presented as a solution; you must see through it and discover the *problem* that it obscures. Users and critics understand many things that designers do not; the users, in particular, are the ones who have to live with the language day in and day out. But design is the job of the designers; you cannot expect users to do it for you. (Sometimes, of course, they will: someone comes up with just the right suggestion. This happened several times in the history of Eiffel, with some recent examples visible on the Web in the ongoing discussion group [4]. Then you can be really grateful.)

So there are deep and shallow comments but almost no useless ones. Sometimes the solution simply resides in better documentation. Often it lies in a tool, not in any language change. Even more often, as discussed above, the problem may be handled by library facilities: after all, this is the aim of an object-oriented language — not to solve all problems, but to provide the basic mechanisms for solving highly diverse problems.

Once in a while, however, none of this will work. You realize that some facility is missing, or inadequately addressed. When this happens — and only as a last resort — the tough conservative temporarily softens his stance. There are two cases, truly different: an extension, or a change.

11 EXTENSIONS

Extensions are the language designer's secret vice — the dieter's chocolate mousse on his birthday. After much remonstrance and lobbying you finally realize what many users of the language had known for a long time: that some useful type of computation is harder to express than it should be. You know it is extension season.

There is one and only one kind of acceptable language extension: the one that dawns on you with the sudden self-evidence of morning mist. It must provide a complete solution to a real problem, but usually that is not enough: almost all good extensions solve *several* potential problems at once, through a simple addition. It must be straightforward, elegant, explainable to any competent user of the language in a minute or two. (If it takes three, forget it.) It must fit perfectly within the spirit and letter of the rest of the language. It must not have any dark sides or raise any unanswerable questions. And because software engineering is engineering, and unimplemented ideas are worth little more than the whiteboard marker with serves to sketch them, you must see the implementation technique. The implementors' group in the corner of the room is

grumbling, of course — what good would a nongrumbling implementor be? — but you and they see that they can do it.

When this happens, then there is only one thing to do: go home and forget about it all until the next morning. For in most cases it will be a false alarm. If it still looks good after a whole night, then the current month may not altogether have been lost.

12 CHANGES

What happens if you realize that some existing language feature, which may be used by thousands of applications out in the field, could have been designed better?

The most common answer is that one should forget about it. This is also the path of least resistance: listening to the Devil of Eternal Compatibility with the Horrors of the Past, whose constant advice is to preserve at all costs the tranquillity of current users. The long-term price, however, is languages that forever keep remnants from another age. For a glimpse of the consequences, it suffices to look at recent versions of Fortran, still retaining (although they are meant for the most powerful parallel computers of tomorrow) some constructs reflecting the idiosyncrasies of the IBM 701's 1951 architecture, or at more recent "object-oriented" extensions of C, faithfully reproducing all the flaws of their parent, compounded by extra levels of complexity.

The other policy is harder to sustain, but it is also safer for the long term: if something can indeed be done better, and the difference matters, then change the construct. Such cases should of course be rare and far between — otherwise one can doubt the very soundness of the original design. They should meet two conditions:

1 There must be wide agreement that the new solution is significantly better than the original one. It must not entail any negative consequence other than its incompatibility.

2 The implementors must provide a conversion mechanism for existing software.

If these conditions are met, then I believe one should cut one's losses and go ahead with the change. To act otherwise is to act arrogantly (pretending that something is perfect when it is not), or to sacrifice long-term quality for short-term tranquillity.

All the issues discussed above arose in the transition between successive versions of Eiffel. It is only for the language users to judge whether the changes and extensions were justified, and whether they followed the principles discussed here. More striking than the changes has been the stability of Eiffel: the language's key properties, especially its semantics, are essentially identical to what was described in the very first publication. But the maintainers of Eiffel have not refrained from making changes, including incompatible ones. It is surprising to see both the intellectual cowardice of many people in language committees, and the positive reaction of actual users. If a change is for the better, it is clearly explained, carefully prepared, and well organized (avoiding pointing a gun to them: change *now* or die!), they will go for it.

13 THE POLITICS OF LANGUAGE EVOLUTION

The mention of committees brings in the final observation of this overview, addressing not the technology of language evolution but its politics. A number of models are possible:

- The Town Hall model.

  Everyone votes, and the majority wins.

- The Venetian model.

  The Doges haggle it out between themselves.

- The Tammany Hall model.

  Everyone votes, and the bosses haggle it out between themselves.

- The Dog Pack model.

  He who shouts the loudest wins.

- The Usenet model.

  He who shouts the longest wins.

- The dictatorship model.

  The dictator wins (*until he is toppled*).

- The engineering project model.

  The chief engineer wins, but only if he can convince the other engineers most of the time.

- The CEO model

  Like the engineering project model, but the board must approve major decisions.

Without reference to the management of society, where different criteria apply, I have through my experience come to the conclusion that the appropriate model for language evolution is one of the last two. Democracy is admirable for the government of humans, but a language is before all an engineering project, and someone should be in charge. As in a company, many checks and balances should be provided, and the chief engineer should very seldom be permitted to pass his views just because he is the chief engineer: a technical leader who has to govern by fiat — as opposed to convincing the troops on the sheer strength of technical arguments — will not remain a leader for very long. Once in a while, of course, he gets to make a choice simply because someone needs to choose (concrete syntax details are the most common example); but these should remain rare cases. After all, if the chief engineer deserves the position, his ideas, or more commonly his ability to sort out the good ideas from the bad, regardless of who originated them, should be better than everyone else's, so he should win on the merits.

It is remarkable to see how the people who produce Internet standards, some of the most successful ever, have reached similar conclusions. Although not every transposes literally, the similarities are striking enough to justify citing a substantial extract of Christian Huitema's description of the process in a recent book[2]:

2. From section 2.5.3 of reference [6]. Reproduced with the permission of the author and publisher.

> *The IETF* [Internet standards group] *is not the only organization that produces networking standards, but it has a distinctive flavor. As Dave Clark explained it during the 1992 discussions, "We reject kings, presidents, and voting; we believe in rough consensus and running code".*
>
> *The IETF working groups generally don't vote. The goal of the processis to obtain a consensus of all participants; if your technical contribution cannot withstand technical cricism by your peers, i1t will simply be rejected. Voting occurs occasionally when there is a choice between two equally valid proposals, such as on the position of fileds within a packet, but this is more a rational way to flip a coin than anything else. In fact, voting supposes counting voting rights, which supposes some formal membership — for example, one vote per country or one vote per member organization. This is almost contradictory with the open membership nature of the working groups which any interested individual can join. By avoiding votes, the IETF also avoids the "horse trading" that occurs in the last days preceding a vote in other organizations (*"I will vote for your checksum algorithm if you vote for my address format"*). Voting leads to compromises and overblown specifications that include every member's desires. This kind of compromise would not fly in the IETF. One has to get consensus through the excellence of the technical work. The search for consensus leads to technical elegance.*
>
> *Consensus does not mean unanimity. There will always be some unconvinced minority. This is why the consensus is expected to be "rough". In fact, it can be so rough that two irreconcilable camps opt for two very different solutions. Treying to merge them would yield what is often derided as "a camel— a horse designed by a committee", where the bumps on the back reflect the diverse options. There are two famous examples of this situation, one in the management area (CMOT, derived from ISO's CMP versus SNMP) and one in the routing area (OSPF versus the adaptation of ISO's IS-IS). In both cases the problem was solved by letting two independent working groups pursue the two different options, producing two incompatible but self-consistent standards, and letting tyhe market choose.*
>
> *This is why the last part of the equation, running code, is important. An IETF specification can only progress from proposed standard to draft and then full standaard if it is adopted by the market (if it is implemented and sold within networking productgs). A perfect specification that could not be implemented would just be forgotten; the document would be reclassified as "historical".*

For Eiffel as for other languages, the temptation periodically arises of deciding language features through a vote. The IETF model — rough consensus and running code — seems far preferable.

REFERENCES

[1] C.A.R. Hoare: *Hints on Programming Language Design*, Stanford University Artificial Intelligence memo AIM-224/STAN-CS-73-403, 1973; reprinted in *Essays in Computing Science*, ed. C.B Jones, Prentice Hall International, 1989, pp. 193-214.

[2] Jean-Raymond Abrial: *The Specification Language Z*: *Syntax and* "*Semantics*", Oxford University Computing Laboratory, Programming Research Group Technical Report, Oxford, April 1980.

[3] Bertrand Meyer: *Eiffel*: *The Language*, Prentice Hall, 1991.

[4] *Eiffel*: *The Language*, ongoing work at http://eiffel.com/private/meyer/etlnew, with discussion group at http://talkitover.com/etl. User name: Talkitover; password: etl3.

[5] Bertrand Meyer: *Object-Oriented Software Construction*, *second edition*, Prentice Hall, 1997.

[6] Christian Huitema: *Routing on the Internet*, 2nd edition, Prentice Hall, 1999.

ACKNOWLEDGMENT

The prefaces to almost every one of my books acknowledge the debt I have to Tony Hoare. His work, and his personal example, have inspired and influenced my entire career. In addition to his scientific contributions, he showed that technical writing, however technical, is also writing. I am deeply grateful to him for setting the standard so high.