From the Net... Sather's Design

Some of the recent discussion in the comp.lang.eiffel group on the net has revolved around design decisions made in Sather that differ from those in Eiffel. Dr. Stephen Omohundro, the chief Sather designer, replied with this article discussing the motivations behind some of these design decisions.

The design of a language is necessarily a balancing act between conflicting goals and priorities that will result in different designs. The Sather design has undergone many changes and the final decisions made were not made lightly. I will try to explain some of the reasons for the differences between Sather and Eiffel.

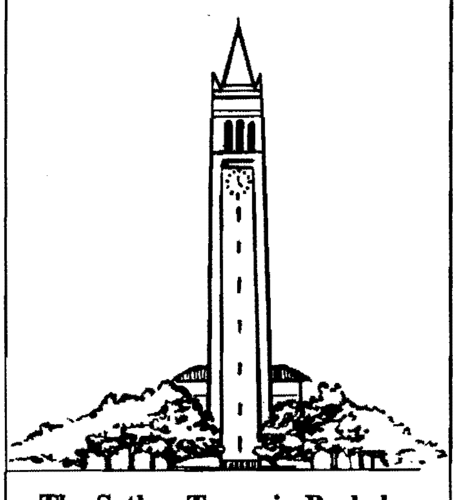
Sather was developed because several research projects (particularly a general-purpose connectionist simulator and a high-level vision system) here at the International Computer Science Institute required both high efficiency and a modular design with complex data structures. In addition, we needed a clean, non-proprietary object-oriented platform on which to build parallel languages for experimental hardware. Initial experience with SmallTalk, C++, Objective-C, CLOS, and Self eventually led to our using Eiffel for about a year and a half to construct a system of a couple of hundred classes. This experience convinced us of many of Eiffel's strengths but also showed us several places where it was not suitable for our needs. The primary problem was efficiency, but some of the language's complexities were also a factor.

It would have been nice to maintain compatibility with Eiffel, but we had to make several semantic changes that prevented it. Once we went that far, we decided to simplify several other aspects as well. Heinz Schmidt has added features to the Sather emacs editing mode to convert between Sather and Eiffel syntax. This should take care of the syntactic differences, though not the semantic ones. In practice, we find it's not too difficult to convert classes between the two languages.

To design Sather, we identified the features of Eiffel that we actually used in our code. We also identified several features that felt clumsy and some features that needed to be added for efficiency. One of the design's guiding principles was simplicity. There is tremendous pressure toward "creeping featurism" in language design (witness the ever-growing size of Common Lisp). One of the great potential virtues of object-oriented design is keeping the language simple and putting new complexity and features into well-encapsulated classes. There were (and still are) many suggestions for additions to Sather. We've tried to be extremely careful by adding only absolutely critical features. In fact, several features in the original design were eliminated. This has led to a design that is easy to remember and use. The syntax description fits on a page and is very regular (I sometimes still forget the syntax rules for C!). Ease of implementation was not an overriding factor, except to the extent that a feature that is easier to implement is often easier to understand and use in practice (since you have a good idea of what the compiler is going to generate).

An awkward aspect of Eiffel arises when the programmer must keep a list of items consistent with code that is textually in a different location. For debugging purposes, I almost always found myself exporting almost all of the classes attributes. Eiffel currently requires putting these items in the "export" list at the top of the class construct. Most of my programming errors were a result of forgetting to put something in this list or changing the name of a routine without altering the list.Jumping back and forth between this list and the routine was a frustration during editing. In Sather, we make features exported by default and private by special case. The "private" declaration is made at the routine or attribute definition as in "private foo:INT;". This is textually near the item so marked and doesn't require keeping two copies of a name consistent.

Another example of this problem in Eiffel is having to declare all local variables at the beginning of a routine. You end up with a big list of items far separated from the point of use. It is easy to forget to declare something, or to give it the wrong declaration, or to forget to remove a declaration when it is no longer used. In Sather, we followed the lead of C++ by allowing local variable declaration anywhere a statement is allowed. This also permits a consistent style for attribute declaration and initialization, e.g., "foo:INT;" or "foo:INT:=6".


The Sather Tower in Berkeley

Simplified Syntax

Once we eliminated these essential lists, it became clear that the syntax of classes and routines could be simplified. The Eiffel class definition syntax with the portions: "class FOO export ... inherit ... rename ... redefine ... feature ... invariant ... end" was hard for me to remember. In Sather it is just: "class FOO is ... end". Inheritance is specified by including a class in the feature list. Similarly, the construct for defining routines of the form: "foo is require ... local ... do ... ensure ... end;" was hard to remember. We made it similar to class declarations: "foo is ... end;". The "do" in particular seems redundant. We made assertions ordinary statements and allowed them to be individually named. At compile time, individual assertions can be turned on or off. This is critically important for "debug" statements when they are used to turn on different kinds of monitoring, as is quite common. Once assertions can be named, we use the convention "assert (pre) ... end;" and "assert (post) ... end" for pre-conditions and post-conditions rather than having separate language constructs.

We found it very important to separate class names from UNIX file names (to eliminate size constraints, etc.) and to allow multiple classes in a single file. This allows classes to be grouped naturally (e.g., adding test classes to a classes file, keeping all the little non-terminal syntax classes in a parser together).

Explicit Type Specification

The most important semantic change we made was the introduction of the ability to explicitly specify types. While dispatching is essential to object-oriented programming, we found that in practice, only a very small percentage of a system's references actually referred to more than one class of object. Unfortunately, in many cases the Eiffel compiler was not able to discover this fact (in many cases, it would not be possible to discover it). We also discovered that many non-intuitive aspects of the type system arose from the fact that any variable could potentially hold any descendant. In most cases the programmer knows the type (e.g., "a:INT"), so we made the unmarked case be that the variable holds the type specified. To indicate its slightly higher cost, a dollar sign is used to indicate that a variable might hold any descendant object (e.g.,. "b:$FOO"). In addition to allowing the programmer to directly specify potentially more efficient code, the stronger specifications allow the compiler to perform stronger type checking. In this sense, it is a push toward even stronger type checking than in Eiffel.

The introduction of the new type specification flexibility also cleaned up a number of semantic issues. Situations often occur where you want child classes that do not have all of the features of their parents. One example discussed on the net was that of class SQUARE which didn't want the routine "add_vertex" defined in parent class POLYGON. Because we can specify the difference between variables that support dispatching and those that do not, we can distinguish between routines that are used in a dispatched fashion from those that are not. In Sather, we only require a descendant to be consistent with its ancestors on those features which are applied to dispatched variables. If no code does "a.add_vertex" to a variable "a:$POLYGON" then SQUARE is not required to define this routine. Calls to "b.add_vertex" are fine if "b:POLYGON". This allowed us to introduce the declaration "UNDEFINE", which allows you to delete features defined in ancestors.

The ability to explicitly specify dispatched variables also allowed us to use the natural contravariant rule for routine arguments in inherited routines. This choice was not made for implementation reasons, as suggested by some. In fact, the early versions of the compiler were covariant. Heinz Schmidt noticed that the reasons for this choice, which were ably defended by Bertrand Meyer for Eiffel, no longer applied to the Sather type system. The reason contravariance is more natural is that one wants any call that is legal on an ancestor object to still be legal on a descendant object.

Let FOO define the routine "baz(x:A)" and its child BAR define the routine "baz(x:B)". If "a" is declared to be of type FOO, then consider a call "a.baz(x)" where "x" is of type "A". This is clearly legal if "a" holds a FOO object. If "a" holds a BAR object then it is legal only if A is a descendant of B. This is the contravariant rule. With the covariant rule (that B must be a descendant of A) this type of call may not be legal. I believe that Eiffel inserts runtime checks to catch this kind of error. The reason for having to use the less type-safe rule is that it's a common to have an argument of the same type as the class it is defined in. Thus we want "baz(x:BAR)" in BAR to replace "baz(x:FOO)" in FOO. In Eiffel, this forces the dangerous covariant rule. In Sather, we have no such problem because we only need to ensure conformance if a call is ever made in a dispatched way. The compiler will complain if we do "a.baz(x)" if "a:$FOO" and "x:BAR", but it should since the code might use features not defined for "x". If we never do this kind of dispatched call, then there is no restriction. That is why we were able to change to the contravariant convention. (By the way, as someone noted on the net, it is a trivial change to the compiler.)

Miscellaneous Issues

We found "once" functions awkward to use in practice and decided to go with class variables (shareds). As in C++, we allow direct access to routines in classes (e.g., "FOO::baz"). This is only possible because we allow routine calls on void objects when the type is specified at compile time. This is only possible because of the "$" convention.

Efficient arrays are critically important in our work, so we added them directly to the language (objects can have a variably sized array portion after their attributes with direct access to elements). We wanted to use the standard notations: "a[5]", "b[3,2]", "a[5]:=9", etc., so we had to use curly brackets instead of square brackets for parameterized classes.

There are several other interesting design issues, but this should give you an idea of our decision process. Our goal was to develop a tight, fast vehicle for developing efficient, reusable code. The real challenge is to build powerful libraries on this base. Sather is primarily needed by small groups carrying out scientific research. Our hope is that the tools and libraries will be useful to other such groups and that a large collection of reusable classes in a wide variety of areas will be developed. Eiffel addresses different needs. We think that the formation of the Eiffel Consortium and recent developments in the language are important.

Stephen M. Omohundro, Ph.D.
International Computer Science Institute
1947 Center Street, Suite 600
Berkeley, CA 94704
Telephone: 415-643-9153
Fax: 415-643-7684
Email: om@icsi.berkeley.edu