REQUIREMENTS FOR A FIRST YEAR OBJECT-ORIENTED TEACHING LANGUAGE

Michael Kölling, Bett Koch and John Rosenberg
Basser Department of Computer Science
University of Sydney, Australia
{mik,bett,johnr}@cs.su.oz.au

Proceedings SIGCSE'95, Nashville, Tennessee, ACM Press

ABSTRACT

Interest in teaching object-oriented programming in first year computer science courses has increased substantially over the last few years. While the theoretical advantages are clear, it is not obvious that the available object-oriented languages are suitable for this purpose. None of the existing languages is appropriate for *teaching* object-oriented principles. In this paper we discuss the requirements for an object-oriented teaching language and draw attention to the deficiencies of existing languages. In particular, the paper examines C++, Smalltalk, Eiffel and Sather. Finally we outline characteristics of a new language, specifically designed for teaching purposes.

1 INTRODUCTION

Over the last few years the object-oriented approach to system design has become widely accepted in industry as a valuable paradigm and has been adopted by many large companies. Partly as a result of this, and the advantages from a software engineering point of view, the use of object-oriented technology and languages is now taught at some point in most computer science courses.

Initially object-oriented systems were seen as an advanced topic and were taught in higher year courses. However, it is now being realised that object-oriented concepts are, more generally, a good basis upon which to teach fundamental programming skills, the usual aim of a first year computer science course. The possibility of using an object-oriented language for first year teaching is being seriously considered in many institutions.

Several arguments strongly support the use of an object-oriented language in first year:

- Object-orientation encourages well structured programming, which is one of the most important lessons we try to convey to first year students.
- Re-using existing code can be taught in addition to the development of new code, leading to a more realistic perception of the tasks expected of a programmer.
- The ability for students to make use of ready-made objects in their applications opens a wide range of possibilities for real-world and interesting examples and exercises.
- Important software development concepts, such as evolution and re-use, can be introduced and experienced through object-oriented techniques at an early stage.
- Problems with the paradigm shift in moving between object-oriented and non-object-oriented environments seem to be reduced. It has been found that many students whose first programming language is a procedural language, such as Pascal, experience problems in adjusting to the object-oriented paradigm [3]. On the other hand, switching from an object-oriented language to a non-object-oriented one is not anticipated to cause as much difficulty (provided the syntax is not too different) [2].

All of these arguments support the idea of using the object-oriented paradigm in teaching programming in the first year of a computer science course.

Unfortunately when one examines the object-oriented languages which are available, they all have major deficiencies which make them inappropriate as a first year teaching language. We would contend that there is a major need for a new object-oriented programming language *specifically* designed for teaching. Such a language would serve a similar purpose to that of Pascal in the 1980s.

This paper is organised as follows. In the next section we identify the requirements of an object-oriented teaching language. We use these requirements to evaluate a number of existing object-oriented languages which may be candidates for a teaching language. We conclude from this evaluation that none of these is suitable and in the subsequent section we outline some of the features that may be included in a new language.

Proceedings of 26th SIGCSE Technical Symposium on Computer Science Education, Nashville, Tennessee, U.S.A., SIGCSE Bulletin 27, 1, March 1995, pp 173-177.

2 REQUIREMENTS FOR A NEW LANGUAGE

The requirements for an object-oriented language are reasonably well understood and there is no need for us to elaborate these here. However, there are several specific requirements for a first year teaching language as follows.

1. The language should support clean, simple and well-defined concepts. This applies especially to the type system, which will have a major influence on the structure of the language. The basic concepts of object-oriented programming, such as information hiding, inheritance, type parameterisation and dynamic dispatch, should be supported in a consistent and easily understandable manner.
2. The language should exhibit "pure" object-orientation in the sense that object-oriented constructs are not an additional option amongst other possible structures, but are the basic abstraction used in programming.
3. It should avoid concepts that are likely to result in erroneous programs. In particular it should have a safe, statically checked (as far as possible) type system, no explicit pointers and no undetectable uninitialised variables.
4. The language should not include constructs that concern machine internals and have no semantic value. This includes, most importantly, dynamic storage allocation. As a result the system must provide automatic garbage collection.
5. It should have a well defined, easily understandable execution model.
6. The language should have an easily readable, consistent syntax. Consistency and understandability are enhanced by ensuring that the same syntax is used for semantically similar constructs and that different syntax is used for other constructs.
7. The language itself should be small, clear and avoid redundancy in language constructs.
8. It should, as far as possible, ease the transition to other widely used languages, such as C.
9. It should provide support for correctness assurance, such as assertions, debug instructions, and pre and post conditions.
10. Finally, the language should have an easy-to-use development environment, including a debugger, so that the students can concentrate on learning programming concepts rather than the environment itself.

Note that some issues, such as efficiency, which are often considered extremely important for production programming languages, are of little significance for a teaching language. We believe that it is only required that the language be able to be supported in a teaching environment with reasonable response time. Similarly, it is not important that the language be flexible enough to develop real-world applications (e.g. by the inclusion of operations such as arbitrary bit manipulation) — it will never be used for this purpose.

The aim is not to replace all other object-oriented languages — industry will always favour languages which have clear run-time efficiency advantages. Rather, our aim is to educate students in such a way that they understand the underlying concepts and are then able to write good programs in any language, even if it doesn't have a good structure.

3 SURVEY

In this section we evaluate several object-oriented programming languages which have been used or proposed as first year teaching languages. The languages considered are C++ [14], Smalltalk [4], Eiffel [9] and Sather [11] [6].

C++

Although C++ is fast becoming an industry standard and the most popular object-oriented language, it is one of the worst candidates for our needs. First of all, it is a hybrid language that supports object-oriented programming as well as non-object-oriented programming, leading to the temptation to develop solutions that are not really object-oriented. This is particularly a problem for those already familiar with C, which is the case for an increasing number of students entering computer science courses. Learning object-oriented concepts becomes more difficult because the language does not encourage their use.

Partly as a result of the need to keep compatibility with C, C++ has many redundant features. In many cases several different language constructs exist for the same semantic concept and these sometimes differ in subtle ways. This makes reading as well as writing C++ an unnecessarily difficult task.

This problem can be illustrated by an examination of the object creation mechanisms in C++. There are three different methods of object creation — automatic, explicit, or by assignment — which all differ in small aspects. In the first case objects are destroyed automatically, including an automatic call to the destructor function; in the second case objects are not automatically destroyed; in the third case the objects are created without a call to the constructor function and calling the destructor can be a problem (although it may be called automatically, anyway). The subtleties of this are extremely difficult to explain to students.

The explicit dynamic storage allocation in C++, in combination with the lack of garbage collection, forces the programmer to think at an unnecessarily low level and greatly increases the risk of errors. Often C programs which have been tested and used for some time have "memory leaks" and other bugs that are caused by improper storage handling. In C++ this problem becomes more serious, since deallocation is often associated with a function call. A missing deallocation omits this function call and this can cause a much greater variety of unwanted effects than just disappearing memory.

Another serious criticism is the handling of dynamic dispatch in C++. Dynamically dispatched functions must be explicitly declared as such in the parent class. This seriously restricts code re-use, as programmers must anticipate all later descendants and the routines they may

want to redefine. It also leads to a complicated execution model.

The confusion is further increased by the ability to declare a function in a derived class with the same name as a non-dynamic function in the parent class. It is left to the reader to determine the different possibilities and problems resulting from such an action.

Although the problems discussed above are not the only deficiencies of C++, they are serious enough to disqualify C++ as a candidate for a teaching language.

SMALLTALK

Smalltalk is an absolutely pure object-oriented language, that supports the underlying concepts of object-orientation in a clean and consistent manner [19]. It enforces the development of code in an object-oriented style and, consequently, the programmer must adopt an object-oriented way of thinking about problems and solutions. Smalltalk usually comes with an integrated, graphical programming environment.

However, there are some problems with Smalltalk as a candidate for a first year teaching language.

Smalltalk is not statically typed and, as a result, type errors will not be detected until run-time. In the worst case, errors may not be detected at all, if, for example, the program is not thoroughly tested and not regularly used (as is often the case with students' assignments). Dynamic type checking also increases the difficulty of locating the cause of an error.

Another problem is the size of the system. Smalltalk usually offers a huge class library. Since everything in Smalltalk is an object (including, for example, control structures), extensive use must be made of the library from the very beginning. Experiences with teaching Smalltalk have shown that the Smalltalk system environment is considered by most students to be confusing, hard to understand and not helpful enough [15]. It usually takes considerable time to learn to use the environment before a beginner can start to concentrate on the language itself.

Another drawback is the syntax. While the unification of all operations and control structures as method invocations has a theoretically nice appeal, it unfortunately makes the syntax more obscure. For example:

```
if (x > y) then
begin
   max := x; index := index + 1;
end
```

becomes in Smalltalk:

```
x > y ifTrue: [max <- x.
               index <- index + 1]
```

This style of syntax is not familiar to students who have already seen a procedural language and is not a good preparation for a later switch to a more commonly used language.

EIFFEL

Eiffel is probably the language that comes closest to fulfilling our requirements. It supports object-oriented concepts in a very clean way, avoids redundancy and has a clear, easily readable syntax.

The problems with Eiffel are similar to those discussed for Smalltalk. The system is considered overwhelming, and the programming environment offers insufficient support for the programmer. Considerable time is needed to understand how to handle the programming system. Lutz [7] stated that he had to abandon Eiffel because of these problems.

While people are working on the development of better programming environments, the language itself is being revamped and is becoming less suitable for our purposes. After having a very small and clean type system in early versions, current versions of Eiffel feature explicit pointers and an implementation alternative for classes that is visible in the class interface definition ("expanded classes"). This complicates the execution model and forces the programmer to think about implementation details during class design. The changes are a result of the use of Eiffel in more and more commercial applications and the associated need for efficiency.

SATHER

Sather can be seen as a cross between C++ and Eiffel. Unfortunately, it adopts some of the characteristics that we mentioned as problematic in the discussion of C++.

These are:

- the inclusion of untyped objects and a reduction in the level of static type checking. Some type checks are static while others are dynamic.
- the need to explicitly identify dynamic dispatch. This is done in Sather on the basis of a variable holding an object reference. This causes problems which are different from those with the C++ mechanism, but still unacceptable in our context.

Overall Sather operates at too low a level for good conceptual development. The reason for this approach is the high efficiency of the resulting code, which is not a major concern for us.

4 DISCUSSION

Object-oriented languages are currently taught at the first year level at several institutions and some of these have reported their experiences. A common theme emerges: teaching the object-oriented paradigm in general is seen as having a very positive effect on the enthusiasm and the progress of the students [5, 12, 15]. It is perceived that students gain a better grasp of the fundamentals of programming since it is possible to teach concepts before syntax [13].

All publications, however, include a list of difficulties with the language that was used. Comments about C++

regularly stated that students had difficulties with the C++ syntax [8, 10, 16], and the authors suggested that it should be replaced by a language that is easier to use [1].

Institutions adopting Smalltalk reported difficulties in the usage of its environment. In particular, students had difficulties using the debugger and inspector and were overwhelmed by the size of the class library [13].

Experience with Eiffel includes statements that compilation and linking was too slow, students had difficulty writing the System Description File, and had problems navigating through the library [8].

Most of the studies reported difficulty in switching to the object-oriented paradigm from the procedural approach. A survey by D. Mazaitis of different courses teaching object-orientation in the undergraduate curriculum reaches the conclusion that they suffer from common problems: "difficulties with the language chosen, inadequacies in existing support tools, and the amount of time students need to become proficient with a new paradigm, environment, language and set of tools" [8].

A comparison of the languages examined with our requirements listed in section 2 suggests the following deficiencies.

Sather does not meet the requirements 3, 4, and 5 (error prone constructs, low level constructs, simple execution model).

Eiffel, due to its size, does not meet requirement 7 (language should be small). The featuring of “expanded” and indirect classes violates requirement 5 (simple execution model).

Smalltalk does not meet requirement 3, since it is not statically typed, or requirement 8, because of its syntax.

C++ violates almost every requirement.

In addition, all languages have major deficiencies in their programming environment, violating requirement 10. C++, Eiffel and Sather, do not provide an adequate environment, and Smalltalk, requires too much of its extensive environment to be used to write simple programs.

The solution seems obvious: what is needed is an object-oriented teaching language that avoids the above mentioned problems, meets the listed requirements, and is embedded in an easy-to-use development environment.

If such a system could be developed, it would strongly increase the interest in object-oriented teaching. We suspect that many institutions are generally interested in teaching object-oriented concepts, but hesitate in the realisation because no suitable language is available.

5 A NEW LANGUAGE?

In this section we briefly outline our ideas for a new object-oriented language specifically designed for teaching first year students.

Our first principle is that the language should be a pure object-oriented language. This means that every program is written as one or more objects, and that objects are the fundamental construct for building systems. This immediately excludes the development of a system as an upwards compatible extension of an existing, procedural language.

Although we strongly believe in "purity" in the sense discussed above, we feel that special support should be given to fundamental language elements. Constructs such as control structures should not be viewed as objects, but rather as basic language building blocks; basic data types such as integers, characters and strings should have built-in language support by having specialised syntax for accessing their operations.

The inclusion in the language of syntax for control structures and basic data types enables the use of a syntactic structure very similar to that in common procedural languages, avoids the startup difficulties that for example Smalltalk has, and simplifies the switch to other languages.

While the syntactic *structure* of C and Pascal are very similar, their actual syntax is not. We consider it important to maintain similarity with that structure, while avoiding the problems associated with the concrete syntax, especially in the case of C.

One of the main issues in this respect is that we favour the use of textual keywords over symbols. C was developed in the spirit of having as few keywords as possible, leading to a cryptic syntax, overloading of the same keyword for different purposes, and confusing constructs. C++, which has taken the same approach, illustrates how this can inhibit understandability of the language.

As an example, consider the definition of deferred functions (functions declared but not defined in a class, and which must subsequently be defined in a descendant class). These are called "pure virtual functions" in C++. The syntax in C++ is:

```
virtual void f () = 0;
```

By employing a syntax similar to variable assignment, the meaning of the construct is hidden and the declaration becomes unreadable to a non-C++-programmer. Even experienced programmers in other object-oriented languages would have difficulties understand the meaning of this construct. The simple replacement of "= 0" with a keyword such as "deferred" or "abstract", would result in a reader familiar with object-oriented concepts understanding the construct.

For beginners the keywords have the additional advantage that they can be looked up in the index of a good textbook. It is much harder to use a textbook if the language prevents the book from having a convenient index referring to different language constructs.

Another important characteristic of a new language is that no implementation concerns should be visible in the language. Constructs such as "packed" in Pascal or "register" in C are exclusively optimisation considerations and only serve to confuse students.

A similar argument can be applied to the use of explicit pointers. Having two different mechanisms for accessing an object, directly or via a reference, tends to be confusing to beginners. While object references might have pointer semantics, these references should be the only construct to access the object, and pointers should not be explicit in the

language. Objects should be created in a uniform manner and must be automatically garbage collected.

The language should be strongly typed with static type checking wherever possible. This allows errors to be detected early and avoids difficulties in locating the source of an error.

The issue of strong typing is connected to the question of multiple inheritance. Strong typing is usually too restrictive for certain applications if no mechanism is supplied to treat an object as being of different types in different contexts. One way to achieve this is a type casting mechanism such as that in C++. Such a mechanism basically breaks the type system and can be abused. It is therefore not a desirable solution.

Another possibility in an object-oriented language is the use of multiple inheritance. We consider multiple inheritance itself to be too confusing for beginners. These thoughts lead to the conclusion that the language should either

- support only single inheritance. It is hoped that the resulting restrictions in combination with the type system will not be a serious a problem since the students would not be building large scale applications in their first year. If this turns out to be a severe restriction then
- support multiple inheritance, but provide it in such a way that it does not complicate the syntax as long as it is not used. This would allow the lecturer to first introduce single inheritance only, without having to mention the possibility of multiple inheritance.

The last important group of characteristics is concerned with ensuring a smooth introduction to programming during the early stages of the course. The language and its environment should be intuitive and simple so that students can concentrate on the concepts.

This requirement clearly justifies the special treatment of basic types in the language. The students should not be required to use a significant number of libraries to write a first, simple program.

It is also essential that there is a well designed environment in which the students develop their applications. Where there is a conflict, simplicity in usage should be given priority over sophisticated functionality.

6 CONCLUSION

There is currently a strong interest in teaching object-oriented concepts at the first year level. None of the existing languages can be considered appropriate for teaching object-orientation to beginners. A new teaching language is required to meet the needs for teaching the object-oriented paradigm. This language does not have to be a real world production language and thus can avoid the compromises in conceptual cleanness for efficiency that cause many of the problems with existing languages.

We have listed requirements for an object-oriented teaching language in this paper and are currently in the process of designing such a language and an associated development environment.

REFERENCES

[1] Rick Decker in *Using C++ in CS1/CS2*, ACM, SIGCSE 1994, Vol 1.

[2] R. Decker, St. Hirshfield: *Top-Down Teaching: Object-Oriented Programming in CS 1*, Dept. of Mathematics and Computer Science, Hamilton College, Clinton, NY, 1994 ACM, SIGCSE 1993, Vol 1.

[3] R. Decker, St. Hirshfield: *The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught in CS 1*, Dept. of Mathematics and Computer Science, Hamilton College, Clinton, NY, 1994 ACM, SIGCSE 1994, Vol 1.

[4] A. Goldberg and D. Robson: *Smalltalk-80 - The Language*, Addison-Wesley, 1989.

[5] R.C. Holt: *Introducing Undergraduates to Object Orientation Using the Turing Language*, Dept. of Computer Science, University of Toronto, 1994, ACM, SIGCSE Bulletin, Sept. 1993, Vol 25, No 3.

[6] Chu-Cheow Lim and Andreas Stolcke: *Sather Language Design and Performance Evaluation*, ICSI Technical Report TR-91-034, 1991.

[7] Michael J. Lutz: *Experiences With an Undergraduate Seminar on Object-Oriented Concepts*, Proc SOOPPA 1990.

[8] D. Mazaitis: *The Object-Oriented Paradigm in the Undergraduate Curriculum: A Survey of Implementations and Issues*, St. Josephs College, West Hardford, Ct, 1993, ACM, SIGCSE Bulletin, Sept. 1993, Vol 25, No 3.

[9] Bertrand Meyer: *Eiffel - The Language*, Prentice Hall 1992.

[10] Dung Nguyen in *Using C++ in CS1/CS2*, ACM, SIGCSE 1994, Vol 1.

[11] S.M. Omohundro: *The Sather Language*, ICSI, 1991 Part of the Sather system distribution.

[12] R.J. Reid: *The Object-Oriented Paradigm in CS1*, Computer Science Dept., Michigan State University, 1993 ACM, SIGCSE 1993, Vol 1.

[13] S. Skublics, P. White: *Teaching Smalltalk as a First Programming Language*, School of Computer Science, Carlton University, Ottawa, Ontario, Canada, 1991 ACM, SIGCSE 1991, Vol 1.

[14] B. Stroustrup: *The C++ Programming Language*, 2nd edition, Addison-Wesley, 1991.

[15] M.C. Temte: *Let's Begin Introducing the Object-Oriented Paradigm*, Dept. of Computer Science, Indiana University - Purdue, University at Fort Wayne, IN, 1991 ACM, SIGCSE 1991, Vol 1.

[16] Eugene Wallingford in *Using C++ in CS1/CS2*, ACM, SIGCSE 1994, Vol 1.