BLUE – A LANGUAGE FOR TEACHING OBJECT-ORIENTED PROGRAMMING

Michael Kölling and John Rosenberg
University of Sydney, Australia
{mik,johnr}@cs.usyd.edu.au

ABSTRACT

Teaching object-oriented programming has clearly become an important part of computer science education. We agree with many others that the best place to teach it is in the CS1 introductory course. Many problems with this have been reported in the literature. These mainly result from inadequate languages and environments. Blue is a new language and integrated programming environment, currently under development explicitly for object-oriented teaching. We expect clear advantages from the use of Blue for first year teaching compared to using other available languages. This paper describes the design principles on which the language was based and the most important aspects of the language itself.

1 INTRODUCTION

Object-oriented languages are becoming increasingly widely used in software projects. Their importance for state-of-the-art software development is now generally accepted, and they have achieved popularity with academics and practitioners alike. As this trend has become clear, many tertiary institutions have included the teaching of an object-oriented language somewhere in their curriculum. Often, this has led to a wide variety of problems. Many different approaches have been taken to the teaching of object-oriented concepts and to address the problems related to it. The main questions include:

- when to teach the first object-oriented language
- what set of concepts to include or exclude from the course, and
- what programming language to use.

We have argued earlier [4] (as have many others before us [1,2,8]) that introducing an object-oriented language as the first programming language in the first course has many benefits and can greatly improve ease of learning (mainly by avoiding a "paradigm switch").

Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, U.S.A., SIGCSE Bulletin 28,1, March 1996, pp 190-194.

Several results of attempts of teaching an object-oriented language in an introductory course have been published [3,6,7,8,10,11,12]. While the use of object-orientation was generally seen as a clear improvement, many problems with the specific languages and environments used have been reported. This prompted us to examine available languages for their appropriateness for teaching, and led us to the conclusion (in [4]) that a new programming language and environment is needed.

We propose the development of a language that is designed specifically for teaching. Often it has been argued that universities and colleges cannot afford to teach "toy" languages, since they have an obligation to provide their students with real-world skills for real-world jobs. This has led many institutions to adopt C++ as their teaching language.

We agree that a graduate must be a competent programmer in C++ or a similar widely used language. It is our firm belief, however, that experience with one year of a good teaching language and one year of C++ produces better C++ programmers than two years of C++.

This paper will describe Blue, the language we have developed, in some detail. It will focus on the language design and leave the description of the environment, also an integral part of the project, to [5]. We will discuss design principles, and the reasons for the various design decisions. When we give examples of problems with existing languages we will mainly use C++. This does not mean that other languages do not have these or similar problems. We argued about other languages in [4] and use C++ here because it seems to be the most widely known object-oriented language and it is the main competitor for every new language considered for CS1.

2 PRINCIPLES

The design of Blue was guided by a set of basic principles that were used to make decisions about individual language issues. These principles are based mainly on an educational viewpoint, having the goal of the development of a teaching language clearly in mind. We are not trying to compete with real-world production languages, but we believe that we can make significant improvements to the quality of programming languages as an educational tool.

We will first discuss these design principles and then, in the next section, talk about individual language issues and the application of the principles.

Principle 1: No Conceptual Redundancy

Maybe the most confusing aspect of many existing languages is their ability to achieve the same thing in a variety of different ways. What can be flexibility for the expert is usually confusion to the beginner.

The most fundamental example of this is the object model in many object-oriented languages. In C++, for instance, an object can exist on the stack, or on the heap, it might be created explicitly, implicitly or by assignment, its constructor might be executed or not, it might be deleted automatically or not, all depending on details of the declaration and use of the object.

From a formal point of view, all this is an unnecessary complication. It has its basis only in the requirement to develop highly optimised code, an issue completely irrelevant for introductory courses. At the same time it hides the issues we really want to teach behind a mountain of language details.

Our principle states that there should be one well defined mechanism to express each concept that we want to teach.

Principle 2: Clean Concepts

The principle of "clean concepts" states that the concepts we want to teach should be represented in the language in a way that directly reflects the theoretical model and is not compromised by secondary issues. An example of a violation of this principle in C++ is the dynamic dispatch mechanism.

We consider dynamic dispatch to be one of the fundamental characteristics of object-oriented languages, yet in C++ the dynamic dispatch of a function must be explicitly defined for individual functions (called "virtual functions" in C++). This introduces a wide variety of possible problems for beginners (and for many more advanced programmers as well) in understanding and using this mechanism.

Again, the reason for the particular implementation of this construct in C++ is efficiency.

Principle 3: Readability

The readability of a language is significant in several aspects.

Firstly, learning by example is one of the strongest learning mechanisms in programming. Who has not experienced or at least seen many others flip through a book about a programming language, reading nothing but the example programs?

This is not a character flaw of the student, but a natural and valid way of learning that should be encouraged (in conjunction with other techniques). Having a programming language that actually hints at its semantics with its syntax is a great help in doing so. Such support of readability can be achieved by favouring expressive keywords over abstract symbols. A further advantage of the use of keywords rather than symbols is that they can be looked up in the index of a good text book, thus supporting independent learning.

The second aspect of readability is that it helps students to understand their own programs. It is possible to avoid certain errors that are only introduced because of the poor syntax of a language. C is the most infamous example of a language that supports obscure programs, and unfortunately C++ has inherited its difficulties.

Principle 4: Software Engineering Support

We do not want our students to write just any programs, we want them to write good programs. Software engineering as a discipline has developed a number of mechanisms and guidelines that support good program development. While many of those mechanisms (such as assertions and pre and post conditions) can be included in programs written in just about any language, they are not part of the actual definition of many languages. This often leads to the result that they are either not taught properly, taught only later in the curriculum, or not taken seriously by the students.

In addition to the above principles, an overall guiding philosophy of our language design was not to invent many new features. It is tempting for a language designer to develop completely new syntax and constructs. Our approach has been evolutionary rather than revolutionary. There is a considerable danger that a language which is revolutionary fails (or at least fails to be implemented). Ada is an example, where it took many years to produce acceptable compilers (although many very competent people worked on Ada), because the language designers could not resist adding every useful construct they could think of to the language.

It is more promising to develop a teaching language by extracting the good aspects from existing languages, and avoiding techniques that have been recognised to cause difficulties. This also assures that the techniques which students learn with Blue are relevant later when other languages are introduced.

3 SPECIFIC LANGUAGE ISSUES

Blue is a pure object-oriented language (it does not support development of non-class-based code). It supports strong static typing, single inheritance, automatic dynamic dispatch, generic classes, garbage collection, and a powerful interactive development environment. All of the object-oriented concepts are represented in the language in a clean and consistent way.

This section will discuss some language issues we consider interesting from the educational viewpoint. Space prohibits us from discussing all aspects of Blue or delving into minute detail. We believe, however, that this description provides the reader with an overall impression of the language and its characteristics.

3.1 General Language Constructs

Class Structure

Every class is defined in a single file. There is no distinction between an interface file and an implementation file. This avoids code duplication and inconsistencies. There is, however, a distinction between the interface and implementation *view*. Tools exist to look at the interface or the implementation of a class.

```
class classname is superclass
        -- here is the comment describing the
        -- class in general
    uses other_class1, other_class2
    internal
        var
            count : integer
            name : string
        routines
            internal routines here
    interface
        creation (parameters)
            body of creation routine here
        routines
            interface routines here
end class
```

Figure 1: Structure of a Blue class

Figure 1 shows an overview of the general structure of a class. The locations of the definitions of different parts of a class (such as constants, variables, internal and interface routines) are fixed and always appear in the same order. General comments to assist with understanding the code may be placed anywhere. The different parts of a class are:

- the header containing the class name and an optional super class from which it inherits
- the class comment, describing the class's functionality. (A standard format for this comment will be defined and a class browser will be able to use it for searching and display of a class library.)
- a list of other classes used by this class
- internal entities:
  - instance constants and variables
  - internal routines
- interface entities:
  - the creation routine. (There is only one creation routine and it is always defined at the beginning of the interface.)
  - interface routines

There are no variables on the interface.

This structure may seem restrictive to C++ programmers, but it greatly increases the ease with which a class can be read and understood (principle 3). It is also an application of principle 1. Allowing, for example, variable definition anywhere in the code may seem convenient, but for beginners the price they pay in terms of confusion seems clearly higher than the gain in convenience. As an important side effect, this structure also eases the development of tools for the programming environment.

Routines

Routine declarations always start with the routine name. They then list the routine parameters and the return values. Figure 2 shows an example.

```
set (n: integer; s: string) -> (ok: boolean; os: state)
```

Figure 2: Example of a routine header

A clear distinction is made between those parameters which are passed into a function and those which are returned from it. Note that a function can return more than one value.

This definition avoids some of the type problems that can occur in connection with reference parameters and inheritance.

Having the name at the beginning of the declaration simplifies the lookup of a routine in a class for a human reader. It makes it easy to scan down the list of names of routines in a class interface. Preceding the names with types and keywords moves the name towards the middle of the header and makes it harder to find.

The body of the routine is strictly structured as well (figure 3).

```
routinename (parameters) -> (return values) is
        -- routine comment
    precondition
        pre conditions
    var
        variable declarations
    do
        instructions
    postcondition
        post conditions
    end routinename
```

Figure 3: Structure of a routine definition

Pre and post conditions are optional. If they are present they are part of the interface of the function. They are always checked at runtime. Variables have to be declared in the *var* section. This structure serves the same purpose as the strictness in the class structure: it makes definitions easy to find and supports readability and understandability of the code. It is based on principles 1, 2 and 4.

Variables

Variables may be initialised at declaration (figure 4). Any valid expression (including function calls) may be used for their initialisation.

```
var
    num : integer
    count : integer := 0
    name : string := get_name (fname)
```

Figure 4: Variable declarations

Apart from its value, each variable has a state associated with it. The three possible states are *uninitialised*, *undefined* and *defined*. Each variable that is not initialised at declaration is in the state *uninitialised*. Using an uninitialised variable in an expression results in a runtime error. Once a variable is assigned a value, it enters the state *defined*. It never returns to the uninitialised state.

The *undefined* state can be used in programs to explicitly pass or return undefined parameters and results. Variables can be set to be undefined and checks for that state are available.

This mechanism is also used to ensure return of proper values from functions. Initially, each return value of a function is *uninitialised*. Returning from a function with an uninitialised value results in a runtime error, thus ensuring that every result value was explicitly assigned. (The result may be *undefined*, if the function wants to express explicitly that it can not return a meaningful value.)

This mechanism detects a group of common errors frequently made by beginners (principle 4).

Predefined Types

Some types are predefined in the language[1]. They are integer, real, boolean, string and array. No character type exists. A character is handled as a string of length one.

Strings are fully dynamic. No space has to be allocated explicitly by the programmer, and the string can grow and shrink without defined restrictions during program execution.

Arrays are dynamic as well. The array bounds are not part of the type definition and can be changed dynamically (the problem of space allocation is handled by the Blue runtime system). This avoids tedious repetitive code concerned with dynamic space allocation in other languages. It therefore avoids code duplication and a whole group of common errors (principle 4). In addition, by removing low-level problems from the language, it allows the teacher/programmer to concentrate on high-level issues, thus supporting principle 2.

Control Structures

Three control structures exist: an if-statement, a multi-branch case-statement and a loop. The if and case-statements are similar to those supported by other languages and do not need further explanation. There is only one loop structure in Blue. The loop construct can be used to achieve the semantics of while, repeat and for loops as well as more general definitions (figure 5).

[1]We do not distingish between a *class* and a *type* in Blue – they are synonymous in our context.

```
loop
    statement-list
    exit on condition1
    statement-list
    [ exit on condition2
    statement-list ]
end loop
```

Figure 5: Structure of a loop

Each statement list can be empty. A loop may contain one or more exit statements. By replacing several loop constructs with a single one, we apply principle 1, thus easing teaching and learning of the language, without losing readability. It has also been argued (e.g. in [9]) that internal loop exits make it easier to implement some common algorithms and should be supported in an introductory language.

3.2 Object-Oriented Constructs

This section describes some details of the object-oriented constructs in Blue. These are the parts of language definition with which we were particularly dissatisfied in existing languages.

The Object Model

All objects are treated in a uniform manner. All variables contain references to objects[2]. Assignments always pass references to objects. Objects are never created at runtime without an explicit creation instruction. Blue distinguishes two different kinds of classes: manifest classes and dynamic classes. Manifest classes are those whose objects all exist automatically. Objects of a manifest class are never created at runtime. Examples of manifest classes are integer, boolean and enumeration classes.

Dynamic classes are those whose objects come into existence only by explicit creation. Examples are arrays and arbitrary user defined classes.

It is important to understand the difference between this *logical* distinction and the distinction of storage modes in other languages. In Blue it can never happen that different storage models exist for different classes. It is also guaranteed that two objects of the same class can always be treated in the same way (which is not true in most other languages).

This model avoids all questions about whether or not references to objects can be passed around, whether objects are deleted automatically, lifetime questions, etc. It is based on principles 1 and 2.

Some syntactic sugar (called aliases) is provided so that the natural syntax for objects such as integers can be utilised.

[2]At least from the logical view of the language. This does not mean that a particular implementation has to actually implement all variables as references. A sensible implementation would probably store some values, such as integers, directly.

Information Hiding

Blue supports strict information hiding that cannot be broken. Variables cannot be made visible on the interface and no references to internal variables can be passed out. This not only *enables*, but *guarantees* proper information hiding (principles 2 and 4).

Inheritance

Inheritance is intended only to model pure *is-a* relationships. Using inheritance to avoid an indirection when re-using code (modelling a usage relationship with inheritance) or just to avoid writing an object identifier before a function call (syntactical laziness) is not recommended. Consequently, Blue does not allow hiding or interface redefinition of inherited routines[3]. The implementation of routines may be redefined, since this does not affect the inheritance relationship (the pre and post conditions still have to be met, though, enabling some semantic restrictions to be expressed).

Interfaces

Interface routine calls are always dynamically dispatched. No construct exists in the language to influence the semantics of routine call instructions.

It is not possible to overload routine names to call different routines of the same class when called with different parameter type combinations. Although examples can be found where this technique is sensibly applied, we have come across more examples where it leads to errors in the code. In addition, a side effect of this mechanism is that error reporting of a faulty routine call tends to be vague (since the correct types of the parameters cannot be reported with certainty).

4 CONCLUSION

Existing programming languages have clear shortcomings for the introduction object-orientation to beginners, resulting in unnecessary difficulties in the teaching and learning process. These difficulties arise mainly from too many redundant constructs, relevant to experts but not to beginners, and the compromising of important concepts for the gain of efficiency. While this can be important for professional program development, it complicates teaching significantly. We have developed a language specifically for teaching object-oriented programming to beginners which avoids these problems.

Few new concepts have been introduced. Instead, we have concentrated on removing problems and combining positive aspects of existing languages. This reduces the risk of introducing new problems, and ensures relevance of the acquired knowledge when moving to other languages.

While few of the individual language constructs are new, their combination is, leading to a language definition that has a distinct character, differing from previously available languages. We expect this language to be a significant improvement in the teaching of object-oriented programming.

The implementation of an integrated programming environment, including an editor, a compiler and a debugger, has started. A full language definition has not yet been published, but should be available soon.

REFERENCES

1. R. Decker, St. Hirshfield: *Top-Down Teaching: Object-Oriented Programming in CS 1*, ACM, SIGCSE 1993, pp. 270-273.
2. R. Decker, St. Hirshfield: *The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught in CS 1*, ACM, SIGCSE 1994, pp. 51-55.
3. R.C. Holt: *Introducing Undergraduates to Object Orientation Using the Turing Language*, ACM, SIGCSE Bulletin, 25, 3, Sept. 1993, pp. 324-328.
4. Kölling, M., Koch, B. and Rosenberg, J. *Requirements for a First Year Object-Oriented Teaching Language*, ACM SIGCSE Bulletin, 27, 1, March 1995, pp. 173-177.
5. M. Kölling and J. Rosenberg: *An Object-Oriented Program Development Environment for the First Programming Course*, submitted to SIGCSE Technical Symposium, 1996.
6. D. Mazaitis: *The Object-Oriented Paradigm in the Undergraduate Curriculum: A Survey of Implementations and Issues*, ACM, SIGCSE Bulletin, 25, 3, Sept. 1993, pp. 58-64.
7. Dung Nguyen in *Using C++ in CS1/CS2*, ACM, SIGCSE 1994, p. 384.
8. R.J. Reid: *The Object-Oriented Paradigm in CS1*, ACM, SIGCSE 1993, pp. 265-269.
9. E. Roberts: *Loop Exits & Structured Programming: Reopening the Debate*, SIGCSE Bulletin, 27, 1, March 1995, pp. 268-272.
10. S. Skublics, P. White: *Teaching Smalltalk as a First Programming Language*, ACM, SIGCSE 1991, pp. 231-234.
11. M.C. Temte: *Let's Begin Introducing the Object-Oriented Paradigm*, ACM, SIGCSE 1991, pp. 73-77.
12. Eugene Wallingford in *Using C++ in CS1/CS2*, ACM, SIGCSE 1994, p. 384.

[3]Except for restricted change of parameter types (contra-variance).