The Blue Language[1]

Michael Kölling
School of Computer Science and Software Engineering
Monash University

mik@csse.monash.edu.au

1 INTRODUCTION

Many readers will, when reading the title of this column, react with an immediate feeling of *"Oh no – not another language!"*. Many people feel that too many languages are already out there, and that new languages add to the problem, not to the solution. We are getting closer to reaching a Babel-like scenario, where everyone speaks a different language. (It is, by the way, interesting to note that, according to the Bible, languages were introduced into the world to prevent people from getting their work done. It seems that God, when getting tired of inventing new languages, invented the computer scientist to continue His work…)

Seriously, we believe that one should have a very good reason for publishing a new language. A language designer has to be able to point out where the significant differences to other languages are. Over the last two editions of JOOP, we have discussed problems with teaching object-oriented programming to first year students. We have criticised many aspects of existing languages and environments and listed some requirements for better solutions. We will now go on to describe our attempt of building a system that comes closer to meeting our needs. The system we are about to discuss is named *Blue*. Blue is an integrated programming environment and an object-oriented language developed originally at the University of Sydney, with development now continuing at Monash University, Melbourne. It addresses all the issues discussed in this column over the past two months. In this issue, we discuss the language aspects of Blue. Next month we will continue the discussion by describing the environment.

The significant difference that distinguishes Blue from other languages is that it is *an object-oriented language specifically developed for teaching*. Pascal was such a language for procedural programming and, as such, for many years the most popular teaching language at universities. Other existing object-oriented languages try to serve different customers simultaneously: They want to be useable for real-world, industrial projects and some claim to be also suitable for teaching. With Blue, we do not try to serve both ends. By ignoring some real-world requirements, we think that we gain significant advantages in a teaching context.

This column cannot provide a complete language description. Many details are omitted, and some features are not described completely. The intention is not to provide a full language specification, but rather to introduce and explain the major design decisions that were made while developing the language. A full language specification is available [1].

---

[1] This paper has been published as: Kölling, M., "The Blue Language," *Journal of Object-Oriented Programming*, 12(1): 10–17, March/April 1999.

2 THE CLASS STRUCTURE

Blue is a simple, small and pure object-oriented language. All code is written in form of classes – there is no code outside of operations of a class. Blue is statically type safe and has a consistent, easy-to-understand object model. In the design of the language, we tried to use mechanisms that have proven successful in other languages, while avoiding those that seem problematic. We have taken ideas from many different languages. The objective was to form a language that can serve as a stepping stone to a variety of other systems to be learned afterwards. Thus, we tried to resist the temptation to invent revolutionary new constructs, but rather concentrated on mechanisms and techniques which are directly applicable to many mainstream languages. While few of the constructs in Blue are new by themselves, their combination is.

Blue uses a keyword-based syntax in the Algol/Pascal tradition. Syntactically, it looks like a relative of Eiffel, although there are many differences in numerous details. Figure 1 shows an example of a short Blue class.

The class is rigidly structured. All its parts have to appear in a predefined order. It starts with the class name, a class comment and a *uses* list – a kind of import statement. Here, other user-defined classes which are used in this class are listed. The remainder of the class consists of an internal part, an interface part and an optional class invariant (all marked with corresponding keywords). The internal part is further divided into variables and routines. (The example does not use internal routines.) The interface part is divided into the creation routine and interface routines. There are no variables in the interface.

The syntax and class structure is influenced by our requirement of readability. This rigid structure makes it easier to read and maintain classes: it is always clear where elements can be found. It also greatly helps students to start writing classes. The Blue environment creates default class skeletons for new classes, and all the keywords and placeholder variables and routines are included. Students can easily see where to add definitions. The structure serves to avoid confusion.

Variables cannot be placed in the interface. There are few occasions when public variables are a good design decision. Usually they are used for efficiency reasons or out of laziness on the programmer's part. Most of the time, variables in the interface are an example of bad design that should be avoided. This is an example where the Blue language supports the teaching of programming principles by enforcing good habits. Efficiency concerns are not an issue in Blue (since we concentrate on first year teaching). In other languages, they should be addresses via other means (such as inlining). And laziness is never an excuse for bad class design.

```
class Person is
================================================
== Author:   M. Kölling
== Version:  1.1
== Short:    an example Blue class
================================================

    uses Address
internal
    var
        name: String
        address: Address
interface
        creation (name: String, address: Address) is
                == create a new Person with name and address
            do
                this.name := name
                this.address := address
            end creation

    routines
        getName -> (name: String) is
                == return the name of this person
            do
                name := this.name
            end getName

        changeAddress  (newAdress: Address) is
                == change the address of this person to newAddress
            pre
                newAddress <> nil
            do
                address := newAddress
            end changeAddress
invariant
        name <> nil
end class
```

Figure 1: An example of a Blue class

3 THE OBJECT MODEL

Among the requirements for a teaching language which we listed earlier were those of providing a simple object model and avoiding redundancy. Both these requirements are met by the Blue object model.

3.1 Storage of objects

It is common to distinguish two different types of storage for objects: immediate storage (the address of the object is represented by a variable) or storage by reference (sometimes called *pointer variables*, the address of the object is the *content* of a variable).

Some languages, such as C++, even require different syntax to access features of objects of these different storage types. Immediate objects are accessed through dot notation, e.g.

```
myObject.feature1
```

whereas objects stored by reference are accessed through an “arrow” symbol:

```
myObject–>feature1
```

In a well designed language this distinction in syntax is not necessary. The compiler could know about the storage class of this object and automatically produce the right code (as it does, for instance, in Eiffel). But there are still important differences. Recursive or circular structures can be built only with references, not with immediate objects. Assignments and equality tests might behave differently, since they might or might not involve a copy operation. The question of identity of an object is fundamentally different. In short: it is usually necessary for a programmer to know exactly at all times when he/she is dealing with an immediate object or with a reference.

Supporting these two concepts violates the principle of avoiding redundancy. Immediate objects provide no functionality that could not be achieved with reference objects as well. (The opposite is not true, though.) The sole reason for supporting immediate objects is efficiency. Immediate objects allow faster variable and routine access by avoiding one indirection. This argument is not sufficient for justifying the inclusion of this concept in a teaching language. Blue therefore supports only reference types. All variables store references to objects.

A second advantage of this decision is that the lifetime of variables and objects is separated. The lifetime of immediate objects, commonly stored on the stack, is typically bound to the lifetime of the variable. Reference objects increase orthogonality by separating these issues. CLU [2] takes the same approach, and its authors give a good summary of the reasons [3].

Java follows a similar route. It also supports only references to objects. But it fails to achieve the same degree of uniformity because of the distinction between simple data types (which are not objects in Java and are stored as immediate data) and object types. In Blue, the uniform use of objects by reference together with the definition of simple data items as objects, leads to a very simple and clean object model as the basis of the language. Variables always store references, assignment is assignment of references, and the default equality checks equality of references. Object identity is always preserved (assignment never duplicates an object).

3.2 Object creation

Through this object model, the creation of objects is greatly simplified. The creation of objects is always explicit.

Languages with immediate objects often define different kinds of creation mechanisms: explicit creation for reference objects and implicit creation for immediate objects (the object is automatically created when the variable comes into existence). C++ defines a third creation mechanism, a “copy constructor” that is implicitly executed through assignment.

Restricting storage to reference objects allows Blue to employ only one mechanism for object creation.

3.3 Manifest classes

One of the most difficult problems in unifying the object model is to accommodate for simple data types, such as integers, boolean values and enumerations. Ideally, we would like to view them as objects, but we do not want to be forced to create them before we can use them. We would like to be able to just write

```
a := a + 7
```

without the need for creating a 7-object first in a separate statement.

There are essentially two possibilities in the interpretation of the symbol “7” in order to achieve this goal: it could be interpreted as an object constructor, or as a constant reference to an object that already exists.

The first alternative has the disadvantage of complicating the constructor rule (since it is no longer true that objects are only created through an explicit “create” expression). In addition, equality becomes more complicated. Since potentially more than one 7-object can exist, equality cannot be taken to be identity. Consider:

```
if 7 = 7 then ...
```

We would certainly expect this condition to be true, but we would have created two different 7-objects! One would have to introduce special cases for either the construction or the equality in this case.

The second approach is followed by Smalltalk. Literals are considered constant references to objects. In Smalltalk it is, however, never explained where these objects come from, when and how they are created. They just magically exist in the Smalltalk universe. This introduces a special case in the object world, since most objects must be created by the user, while some are already there. At the conceptual level, all objects are equal and nothing explains this difference.

In Blue, we deal with this issue by introducing the concept of *manifest classes*. We distinguish these from “normal” classes by referring to those as *constructor classes*.

These two kinds of class differ in the way their objects come into existence. Constructor classes are those with which we are familiar from most object-oriented languages. Initially, no objects of these classes exist. They can be created by executing a creation instruction which creates the object and executes a creation routine defined in the class. Manifest classes, on the other hand, define their objects by enumeration. All possible objects of these classes are automatically created at system startup, and the class defines identifiers by which these objects can be referenced. So instead of providing a construction method for objects, manifest classes provide the objects themselves. No additional objects can be created at runtime.

The classes “Integer” and “Boolean”, for example, are manifest classes. All their objects automatically exist, and references are provided to access them. The symbols “3” and “true” are constant references to the objects representing the integer number 3 and the boolean value *true*, respectively.

An important aspect of the manifest class model is the separation of concepts (an improvement in orthogonality): implicit creation is separated from predefined types. This leaves the possibility of the combination of implicit creation and user-defined types. Blue allows this combination for the definition of enumerations. Enumerations often do not fit in well with object-orientation and, as a result, most

languages do not include them at all. Manifest classes allow us to provide a genuine object-oriented definition of enumerations that is seamlessly integrated into the object model. Enumerations are discussed below.

Note that the distinction between manifest and constructor classes is a conceptual difference, not a technical one, and that it is on a class basis and not on a variable basis. In this respect it differs greatly from the immediate storage/reference storage distinction in other languages. Our distinction influences how objects come into existence, which is a semantic definition, and is defined in the class, which already holds semantic information about the class and objects of it. There can never be confusion over how a variable can be accessed, whether recursive structures are possible or when objects get duplicated, as there often is with the duplicate storage model. All objects are accessed in a uniform manner.

3.4 Enumeration classes

Language designers and users have long argued about whether enumerations should be included in a programming language. With the advent of structured programming and Pascal, it seemed accepted that they are a useful construct that increases code quality. Wirth, the creator of Pascal, however, seems to have changed his mind: his latest language, Oberon, does not include enumerations.

Opinions in published comments range from statements that enumerations are "useless" and that "programmers do not want to use them" [4], that they are "superfluous" [5] and "excess redundancy" [6] to strong defences that state that they result in less work and more understandable and maintainable code [7, 8].

We believe that the inclusion of enumerations enables programmers to write more readable code. A routine which might fail to complete its task, for example, can return a meaningfully named result value, instead of returning an integer number. The inclusion of enumeration types in object-oriented languages, however, was typically awkward (as in C++, where they do not fit into the object model) or non-existent (Eiffel, Java). Blue, through the introduction of manifest classes, combines enumerations with objects and provides a unified model.

An example of an enumeration class is shown in Figure 2.

```
class Colours is Enumeration
    == The colours used for our graphics display
    manifest blue, green, purple
end class
```

Figure 2: Structure of an enumeration class

"Enumeration" is a predefined abstract class. It provides (amongst others) predecessor and successor routines, which can thus be used for all enumeration classes.

4 ROUTINES AND PARAMETERS

4.1 Structure

Routine declarations follow a similarly strict structure to the class as a whole. The skeleton of a routine is shown in Figure 3. The reasons for enforcing a strict structure are the same as for the class in general: it makes it easy to find declarations and enhances readability. It has been argued that declarations of variables should be permitted anywhere in the code [9]. This would enable the programmer to declare variables at the point of the first assignment, thus avoiding the declaration of uninitialised variables. While the idea of avoiding uninitialised variables is a good one, scattering variable declarations through the code is strongly detrimental to readability. Often, when modifying a code segment, a programmer has to check the name or type of a variable. Having a position within the code for declarations that can easily be located helps program maintenance. Blue enforces a separate variable declaration section and employs a separate mechanism to deal with the issue of uninitialised variables (see below).

routine-name (*parameter-list*) –> (*result-list*) **is**
    == *routine-comment*
  pre
    precondition
  var
    variable declarations
  do
    routine body
  post
    postcondition
  end *routine-name*

Figure 3: Structure of a routine

Routine name

The first word of a routine declaration is the name of the routine. It is not preceded by a keyword or symbol. This, together with the recommended indentation, lets the routine names stand out, making it easy for a programmer to find a routine declaration. Elevated to a principle, we could call this an example of “syntax structure for human readability” (as opposed to readability for a compiler, which is so often the motivation for syntactic constructs in programming languages). Since readability improves maintainability, which in turn is a major software engineering concern, we could also claim this to be “syntax structure for maintenance support”. To see the effect of this structure, let us compare it to the C++ syntax, where the routine name can be preceded by several other keywords:

```
// C++ declarations:
virtual const int* get (int n);
static void put (int i, char c);
```

```
-- Blue declarations:
get (n: Integer) -> (res: Integer)
put (i: Integer, s: String)
```

Both examples declare the routines “get” and “put”. Comparing the two examples, it is obvious that, when searching through a class interface for the declaration of a routine, the Blue format is more convenient. In the Blue version these names are easy to locate while quickly scanning down a page. The C++ format makes it much harder to identify the routine name in the definition.

The routine name is repeated at the end of the routine body. Again, this can increase readability, especially if routines are longer than just a few lines. Many teachers have recommended this as a programming style using many different languages (usually as a comment at the end of the routine). Again, styles should be supported by the compiler if possible to emphasise their importance and to form good habits. The inconvenience of additional typing can be avoided by providing an editor which supports the automatic insertion of the name at the end.

Routine comment

The routine comment is compulsory. Blue will report an error if no comment is present. (See below for more details on comments.)

```
findElem (index: Integer) -> (found: Boolean, elem: Element) is
        == Return element at 'index'. If it exists, 'found' is true
        == and 'elem' is the element. If not, 'found' is false and
        == 'elem' is nil.
    var
        e : Element
    do
        ...             -- code left out
        found := true
        elem := e
    end findElem
```

Figure 4: Routine with result list

4.2 Parameters and result variables

Routines have an optional parameter list and an optional list of result variables. Only one kind of parameter passing mechanism exists: all parameters are passed by value. (Remember, though, that all variables hold references – the *reference* is passed by value, resulting in pass-by-reference semantics for the objects themselves.) Result lists are, as their name suggests, lists of variables. Thus a routine can return more than one value.

Consider the routine shown in Figure 4. This routine returns two values, *found* and *elem* (which may be useful if *nil* is a valid value for *elem*, and thus cannot be used to indicate failure). The result values are named (defining a *result variable*) and values are returned by assigning to those variables. The compiler reports an error if the routine body does not contain an assignment to each result variable, and the runtime

system reports an error if any of the variables are undefined at the time of the return of the routine.

This mechanism provides a substantial simplification compared to many other languages. There is only one way to pass information into a routine, and there is one mechanism to get information back.

4.3 Multi-assignments

To be able to return multiple values from a routine call, it is necessary to have a way to receive multiple values. The multi-assignment provides this mechanism.

Assignments have a list of variables on their left hand side and an expression list on their right, e.g.

```
a, b := 42, 99
```

Each of the values on the right is assigned to the corresponding variable on the left. All expressions on the right hand side are evaluated before any assignment takes place. A routine call with multiple return values evaluates to a value list and can thus be used on the right hand side of a multi-assignment. The *findElem* function discussed above, for instance, may be called in the following manner:

```
success, element := findElem (2)
```

The multi-assignment is also useful in another situation. Swapping values of two variables becomes very easy:

```
a, b := b, a
```

The parameter and return value mechanism together with the multi-assignment has several advantages:

- It reduces the number of concepts (avoids redundancy).
- It increases readability of the routine declaration.
- It clarifies semantics of the routine call.

The first point has been mentioned above: by avoiding different kinds of parameter passing mechanism, the number of (partly redundant) concepts is reduced.

The second point may be open to discussion, but we believe the Blue style to be more readable than, say, the Ada95 style of parameter declarations (Ada95 was chosen for comparison here, because it places emphasis on syntactical clarity of parameter passing modes). Consider:

```
{ Ada: }
procedure m (a: in out Integer; s: out string, b: in Boolean)

-- Blue:
m (a: Integer, b: Boolean) -> (new_a: Integer, s: String)
```

Blue forces an ordering onto the parameter list: *in* parameters are named first, results (*out parameters* in Ada terminology) are separated. Ada allows arbitrary ordering of parameters. Supplying two separate lists is more easily readable than mixing declarations in one list. Note that the Ada *inout* mode of parameter *a* is represented in Blue by two variables: *a* in the parameter list and *new_a* in the result list.

A more important detail is the third point: the routine call. Consider the Ada case:

```
m (p1, p2, p3);
```

This example shows a call of the routine declared above. At the location of the call it is not determinable which of the parameters are *in*, *out* or *inout* parameters. Compare this with the Blue alternative:

```
p1, p2 := m (p1, p3)
```

Here it is clear that *p1* and *p2* are changed by the statement, while *p1* and *p3* are passed in (*p1* thus has an *inout* behaviour).

5 COMPULSORY COMMENTS

Blue enforces the presence of some comments in a class. A comment has to be present at the beginning of each class (below the header) and in the header of each routine. The purpose of these comments is to describe the entire class and each routine, respectively. While the compiler actually enforces only the presence of a comment symbol (it can never enforce the presence of a meaningful comment), and thus this enforcement is theoretically almost meaningless, it serves in practice as a strong encouragement to write comments. Enforcing the presence of a comment sends a strong signal to the student, indicating that comments are considered to be an intrinsic part of the class. In our experience, students take language elements much more seriously if the compiler supports them, than they do if only given as style guidelines.

Writing comments is further encouraged by the environment. The automatically generated class skeleton contains comment patterns to be filled in, and some comments are displayed in an interface view which is automatically generated by the environment.

Blue supports two kinds of comments. *Interface comments* are defined by the double equals sign (==). Their purpose is to provide information about the functionality of the class or its routines. Interface comments may appear only at specified locations in the source: in the header of the class, in the header of each function and in pre and post conditions. They are visible in the interface of the class. Implementation comments are defined by a double minus sign (--). Their purpose is to provide information about the implementation of a class. They may appear anywhere in the source and are ignored by the compiler. Both kinds of comments are terminated by a new line.

Figure 5 shows the implementation view of a routine that makes use of interface and implementation comments. Figure 6 shows the interface view of that same routine.

```
printInfo (cols: Integer) is
        == Print out the information in this container formatted into 'cols'
        == columns on screen. Items are sorted alphabetically.
        – – To print the information the internal tree is traversed and each
        – – node is displayed separately. To achieve alphabetical sorting, items
        – – must be processed in infix order.
    pre
        cols > 0
    do
        ...      – – code goes here (left out for space reasons)
    post
        == Items have been displayed on screen
    end printInfo
```

Figure 5: Interface and implementation comments in implementation view

```
printInfo (cols: Integer)
        == Print out the information in this container formatted into 'cols'
        == columns on screen. Items are sorted alphabetically.
    pre
        cols > 0
    post
        == Items have been displayed on screen
```

Figure 6: Routine in interface view

6 VARIABLES

All variables, if they have a value at all, hold references to objects. In addition to that, they may be *undefined* or *nil*. Variables may or may not be initialised at declaration. For example:

```
var
  name : String
  n :    Integer := 42
  cnt :  Integer := getNumElems (1, true)
```

Variables are a combination of type, state and value. If a variable is not initialised, its state is *undefined*. An attempt to use an undefined variable is noticed at runtime and results in a runtime error. Routine results are checked at the time of return from the routine.

This catches a common error made by beginning students. Checking for uninitialised variables is a case where Blue, by concentrating on a teaching situation, can provide better support for students than other systems. Undertaking the checks costs in both time and space. In a language intended for commercial production, efficiency would take precedence over error checking, and those checks would not be justifiable. (In fact, the only other language known to us which defines such checking is CLU, but this part of the language definition was never implemented in the compiler [3]. Java defines related, but slightly different semantics: access to variables which *may* be uninitialised is considered an error. The checking is done by

using compile time flow analysis, and cases that cannot be decided at compile time are simply disallowed.)

An alternative to this technique used in some languages is automatic initialisation. We consider this not to be appropriate for a teaching language. There are three possible scenarios, all of which are negative:

- The user forgets to initialise the variable; the variable is automatically initialised to a value that is semantically *illegal* in the context (e.g. *nil* for a variable that is expected to refer to an object). In this situation, the initialisation is no better than no initialisation at all. The program will fail, as it would have without any initialisation.
- The user forgets to initialise the variable; the variable is automatically initialised to a value that is semantically *legal* in the context (e.g. 0 for an integer variable). In this case, the user will be "protected" from detecting a program error. Because the value is legal, it will not result in an (immediate) error, but it may nonetheless be wrong and cause wrong results. It is only detected later. This is the worst case for a teaching language, because programming errors may go unnoticed altogether.
- The user intentionally omits the initialisation of the variable because the default value is the intended initial value (i.e. the user exploits the automatic initialisation). The only real effect of this case is to save typing of a few characters (the explicit initialisation), thereby reducing readability (since the reader must know the default value for each particular type of variable). It is also not clear whether the initialisation was consciously or unconsciously omitted. It seems worth the effort on the writers part of including the initialisation explicitly in these cases to clarify the semantics to the reader.

In Blue all declarations are placed in a well-defined location, variables may be initialised if a sensible initial value is known (with the initial value clearly readable in the source text) and the use of uninitialised variables is always detected.

7 ALIASES

An interesting issue arises from conflicting goals of two different requirements: on the one hand we want uniformity and a small number of underlying concepts. On the other hand we want the language to be simple and intuitive. There are cases, especially when pre-existing knowledge is involved, in which these two goals can come into conflict.

An example is the treatment of simple types and mathematical operations. We have argued, on the one hand, that simple data items should be objects, and that they should be treated as such. This leads to a uniform object concept which is advantageous for understandability. Convention, on the other hand, uses infix notation for some mathematical operations. When using those operations, these two goals contradict each other. Adding two numbers *m* and *n*, for example, should, according to the first principle, be represented as:

```
m.add (n)
```

This, however, looks unfamiliar to students, although they are comfortable with the concept of addition. The following notation is more familiar:

```
m + n
```

The attempt to make a language easy and intuitive should take existing knowledge into account and exploit it to achieve its goal. A simple program, for example, such as “Hello world” or a program adding two numbers, should be easy to write without too many unnecessary explanations. Achieving that goal would be easier if the familiar infix notation were to be used.

To overcome this apparent conflict, Blue uses *aliases*. Aliases provide an alternative syntax for some constructs. In the case above, for example, both alternatives are legal. The first one (`m.add (n)`) is the “real” notation that conforms to the object model. The second notation (`m + n`) can be regarded as a shorthand notation for the first one.

Another example of an alias is a print statement to perform text output operations. A *print* command is available for this purpose:

```
print ("The answer is: ", n)
```

This print command is an alias that calls the *write* operation of the *terminal* object. (*terminal* is a predefined constant which refers to an object of class *TextTerminal*. This object controls the standard I/O terminal.) Thus the print instruction listed above is a shorthand notation for the following statement:

```
terminal.write ( str ("The answer is: ", n) )
```

The operation `str` used in this statements is itself an alias that is a shorthand for calling the *toString* and *concat* functions on its parameters. For example:

```
str (a, b, c)
```

is an alias for

```
a.toString.concat (b.toString.concat (c.toString))
```

In other words, the `str` function converts all of its arguments to strings and concatenates them to form one result string. This string may then be printed out, using the terminal routine *write*, which expects one string parameter.

The concept of aliases serves several purposes. Aliases can be initially introduced to students as statements in their own right. Adding two numbers, for example, is easy to understand:

```
add (num1:Integer, num2:Integer) -> (sum:Integer) is
     == Return the sum of num1 and num2

   do
      sum := num1 + num2
   end add
```

This example can easily be explained to beginners by initially making use of intuition and pre-existing knowledge. The underlying object model is initially hidden, avoiding the need for long explanations or hand waving.

Later, when objects and operations on objects are introduced, and students understand the principle of calling operations on objects, these aliases can be resolved. Students can then be told that those aliases are shorthand notations for normal object operations. This approach provides both an easy entrance into programming and a consistent model where everything falls into place once the student reaches the stage of examining advanced issues.

Aliases are only available for a small number of operations on the predefined classes. Users cannot define their own aliases. Their purpose is only to ease the first stage of programming.

8 INHERITANCE

Inheritance is one of the most fundamental, yet least understood constructs in object-oriented programming languages. Most authors agree that inheritance is one of the ingredients that make a language "object-oriented". A language without inheritance is, in most classifications in today's literature, not considered to be object-oriented.

The importance people place on inheritance is well deserved. This construct allows application designs which are significantly different, and usually superior, to those possible with previous imperative languages. Designs making intelligent use of inheritance often have advantages in clarity, flexibility and maintainability. Inheritance is also invaluable in supporting code reuse through libraries and frameworks. On the other hand, inheritance is often misused. One reason why inheritance is difficult to understand is that it is used for very different purposes. Evered et al. [10] for instance, have identified sixteen distinct uses of inheritance. The most important distinction in the different uses of inheritance seems to be the one between its use for subtyping and its use for code reuse. Some newer languages now provide different constructs for these different semantics.

For Blue, we did not attempt to introduce a new set of constructs to properly address these problems. This area is clearly an open research issue at this stage. Different proposals to address these questions have been made, but none has yet become really successful in practice. We do not see it as our task to develop new language constructs, but to identify and simplify mechanisms proven successful, and to provide those for teaching.

Instead, we tried to define an inheritance mechanism that provides for those uses of inheritance that are more or less uncontroversial, and provides for them in a clear and simple way. We tried to discourage the use of inheritance where it is done purely for efficiency reasons and thereby distorts the underlying model.

For us, this effectively means that we defined an inheritance mechanism that supports "is-a" relationships and subtyping, but discourages inheritance for pure code reuse (which can also be achieved through *uses* relationships).

Blue defines a simple, straightforward inheritance mechanism. Only single inheritance is supported, and the inherited interface cannot be modified. In particular, inherited routines cannot be renamed or hidden. This discourages the use of inheritance where no real "is-a" relationship (specialisation) is intended. Routine interfaces cannot be changed. Parameter types, which can be modified in many other languages, must remain the same in Blue. Covariance (as, for instance, in Eiffel) is not supported because of the associated typing problems. Contravariance (e.g. in Sather) was not seen as generally useful. No variance (insisting on unchanged parameter types) is sufficient in almost all situations and avoids a wide variety of typing problems that would have been hard for first year students to understand.

One possible criticism of this decision is that the students do not get trained in other, perfectly valid uses of inheritance. Inheriting for code reuse without inheriting the parent's interface is a powerful technique that cannot be achieved in Blue.

Our answer to this argument is that this is again, as with some other programming techniques, a question of maturity. We do not argue that those techniques are bad or should not be used at all. We do argue, though, that their application in a first course clouds the meaning of the inheritance relationship and makes it harder to understand the issues connected to the application of this flexible mechanism. We argue for a stepwise introduction of inheritance techniques: As a first step, inheritance is introduced as a clear specialisation ("is-a") relationship. Later (in a second programming course, using a language other than Blue) other, more advanced uses of this construct can be introduced. We strongly believe that by restricting the inheritance construct and the number of cases where it is sensibly applied, we clarify the issue of inheritance and its related problems.

9 GENERICITY

Blue provides a straightforward mechanism for generic classes (also referred to as *parametric polymorphism*). Genericity has been treated poorly in some recent programming languages. C++ did not include it in early version and added it (under the name *templates*) only in later language revisions. Its syntax is poor and the implementation in current compilers still often unsatisfactory. Java does not include genericity at all.

Genericity is an important and powerful concept in object-oriented programming. Maybe its most important role is in the ability to provide a good library of collection classes. Collections classes (lists, stacks, sets, hash tables, etc.) are immensely important for the teaching and development of good software and are an ideal example for teaching the idea of reuse. They can be written in a much easier and safer way with genericity than without.

Genericity, though sometimes regarded as an "advanced" concept, is very easy to understand and to use. It has two facets: the use of an existing generic class and the development of a new one. We are convinced that learning of good object-oriented software development benefits greatly from the ability to make use of generic collection classes. Lists, for example, should be used early in a first course in many cases where they are the appropriate data structure, rather than falling back on using an array, because the use of lists seems not teachable. Rather than teaching the use of an inappropriate data structure only because it seems easy (the array), we should make the use of the appropriate data structure simple. Genericity can do just that.

In practice, there is no need to teach the two sides of genericity together. We favour an approach where the use of existing generic classes is introduced very early. It is straightforward and easy to understand. The development of new generic classes can be left to much later in the course.

10 CURRENT STATUS

The Blue language is fully defined and implemented. A full language definition and an implementation of a development environment (including the compiler) are available at no cost from http://www.sd.monash.edu.au/blue. Blue has very successfully been used for first year teaching at the University of Sydney, Australia, since 1997 with over 700 students each year.

11 SUMMARY

Viewed separately, few of Blue's language characteristics are new. Nonetheless, Blue is a new language with a unique character, provided by the selection and combination of constructs included and constructs left out. Its emphasis is on the provision of clean concepts and simplicity. One of the most important characteristics is that it is a small language. With Blue, students can learn the whole language within the first year. This is psychologically very important. At some stage they know all of the language constructs. From then on they can concentrate on how to apply those constructs without constantly wondering whether a better, more applicable construct for a problem exists. The last time this was possible was when we were teaching Pascal or Lisp. Blue provides the same advantages for the object-oriented paradigm.

Acknowledgments

The work described in these columns is a cooperation of Prof. John Rosenberg, Monash University, and the author.

References

[1] Kölling, M. and J. Rosenberg, "Blue - Language Specification, Version 1.0", School of Computer Science and Software Engineering, Monash University, Technical Report TR97-13, November 1997.

[2] Liskov, B., *et al.*, "CLU Reference Manual", in *Lecture Notes in Computer Science*. 1981, Springer Verlag:

[3] Liskov, B., "A History of CLU", Laboratory for Computer Science, MIT, Technical Report , April 1992.

[4] Pohl, I. and D. Edelson, "A to Z: C Language Shortcomings," *Computer Languages*, 13(2): 51-64, 1988.

[5] Delft, A. J. E. v., "Comments on Oberon," *SIGPLAN Notices*, 24(3): 23-30, 1989.

[6] Lins, C., "Programming Without Enumerations in Oberon," *SIGPLAN Notices*, 25(7): 19-27, 1990.

[7] Cashman, M., "The Benefits of Enumerated Types in Modula-2," *SIGPLAN Notices*, 26(2): 35-39, 1991.

[8] Sakkinen, M., "Another defence of enumerated types," *SIGPLAN Notices*, 26(8): 37-41, 1991.

[9] Stroustrup, B., *The C++ Programming Language*, second edition, Addison Wesley, 1991.

[10] Evered, M., J. L. Keedy, G. Menger and A. Schmolitzky, "How Well Do Inheritance Mechanisms Support Inheritance Concepts?," in *Lecture Notes in Computer Science, 1204*, pp. 252-266, Springer-Verlag, 1997.