Kent Academic Repository

I/O Considered Harmful
(At Least for the First Few Weeks)

John Rosenberg

Basser Department of Computer Science
University of Sydney, NSW 2006
Australia
johnr@cs.usyd.edu.au

Michael Kölling

Basser Department of Computer Science
University of Sydney, NSW 2006
Australia
mik@cs.usyd.edu.au

Abstract

One of the major difficulties with teaching the first programming course is input/output. It is desirable to show students how to input data and output results early in the course in order to motivate the students and so that they can see the results of their programs. Output is also a useful tool for testing programs. However, in most programming languages input and output are esoteric and the techniques for performing input and output must be learnt by the students at an early stage, precisely when they are trying to understand the basics of programming. We argue that input/output operations need not be taught in the early stages of a course if the language environment provides appropriate tools for testing programs. This assertion is demonstrated by reference to the Blue object-oriented language and environment.

1 Introduction

There are a number of issues that arise in relation to teaching the first programming language to students. Often these conflict in terms of the order in which the concepts and constructs are taught. Of particular interest in this paper is the teaching of input and output and the decision as to when these are taught within the course.

It is often argued that it is essential to teach output, at least, very early in the course, probably in the first or second week. Output is essential in order to see a visible result from a program. Output of results allows a program to be tested and it provides motivation for students because they can see the results of their efforts. Hence the use of the seemingly popular "Hello world" program. Input may be delayed a little longer, but is required as soon as we wish students to write programs which have some controlling parameters.

While there are strong arguments for teaching about input and output early, there are equally strong arguments against this. By its very nature I/O does not fit in well with programming language design. Every language designer has encountered the problem that the task of including usable I/O facilities invariably seems to make it necessary to break some of the rules or design principles of the language. In many languages with clean and simple concepts I/O is the "odd one out".

Of particular difficulty seems to be input. Struggling with buffering, end-of-line characters and, most of all, the necessary conversion from character streams into data types in a typed language has lead to mechanisms as bizarre as the infamous "scanf" routine in the C language. Output as well, although conceptually easier, has its problems. The problems range from students forming incorrect models through the generalisation of special I/O constructs (e.g. the incorrect assumption that Pascal routines may have a variable number of parameters as allowed in the "write" procedure) to the introduction of advanced concepts too early, with the result that the students can not relate a new concept to anything else they know about the language (e.g. the use of overloaded infix operators for output in C++).

The result is typically one of two situations:
- the instructor is forced to discuss concepts that do not relate well to the rest of the language principles and advanced concepts that would not normally be discussed at that point are introduced, or
- the instructor employs some sort of hand waving, hoping the students will somehow be able to figure out which part of their work constitutes the important concepts and which part corresponds to idiosyncrasies which are better ignored for the moment.

All of these issues detract from the teaching of the basic concepts of programming and increase stress and learning difficulties for students.

In this paper we show how, with an appropriate program development environment, it is possible to delay the teaching of I/O until quite late in a first programming course. The result is that the instructor is able to teach the basic concepts of the language and develop the students' skills to a point where they can write reasonably sized programs before they have to deal with the intricacies of I/O.

The approach described utilises the Blue programming

system developed by the authors [1, 2, 3]. Blue is both an object-oriented programming language and a program development environment and was developed specifically for use as a teaching system in the first programming course. It is expected that students will drop Blue after the first course and will then learn industry standard programming languages such as C++ and Java.

Since Blue is only to be used in teaching the first course, conceptual elegance is able to take precedence over issues such as performance. Blue is a pure object-oriented language [2] with very clean syntax, a strong and statically checked type system and support for inheritance and genericity. However, it is not the Blue language which provides the support for delaying the use of I/O, but rather the environment.

The Blue environment [3] provides direct support for the creation of object-oriented applications. It provides a graphical interface for the specification of classes and their relationships, an integrated editor and compiler and a symbolic debugger. In addition, the Blue environment provides an interactive testing facility [4] which allows the interactive creation of objects of any class and calling of the methods of these objects. It is this testing facility which allows the delay in teaching I/O.

In the following section we examine the facilities provided for I/O in some common languages used in teaching the first programming course in order to demonstrate the complexities and idiosyncrasies introduced by these features. We then briefly describe the Blue language and environment. This is followed by a description of how Blue may be used to develop, test and interact with substantial applications without the use of I/O.

2. I/O in Existing Languages

In this section we examine the support for I/O in some common programming languages and describe some of the traps and complications often encountered by students learning their first language. We only consider text input and output. It is assumed that code to support graphical user interfaces would be even more complex for students.

2.1 Pascal

Until recently Pascal [5] was clearly the most popular language for teaching the first programming courses at universities. Pascal provides special constructs within the language to support input and output from a text console and files. These facilities are integrated into the language and reasonably consistent in terms of the type system. However, there are some difficulties.

The standard statements for input and output are *read* (*readln*) and *write* (*writeln*). A major conceptual problem with these is that, although they appear syntactically to be procedure calls, they break the syntax rules for normal calls in that they allow an arbitrary number of parameters of arbitrary types. This may seem to be a simple extension. However, when students are grappling with the concept of procedure calls and parameters it is a cause of considerable confusion.

There are a number of other issues which are of particular concern regarding the semantics of Pascal I/O. The first relates to input and the notion of lines. Students always have considerable difficulty understanding the difference between *read* and *readln* and the notion of an *end of line* character. This is difficult to avoid, even in simple exercises and is a source of great confusion. A second issue is the buffering of output. A common error amongst students is to leave out a *writeln*. Depending on the compiler and run-time system this may result in no output, or output mixed with the next command line prompt.

2.2 C

Surprisingly perhaps, C [6] is also a popular language for teaching the first programming course. I/O is not a part of the formal language definition for C, however, a standard I/O library is supported by virtually every implementation. The library is accessed via standard procedure calls which adhere to the syntax and semantics of the language.

There are a number of difficulties with C I/O which make it extremely cryptic and confusing for students. Considering input first, the standard input statement is *scanf*. This takes a description of the format of the input data as a string and a series of pointers to the variables in which the input data should be stored. There are several problems that arise with this approach. First, the syntax of the format string is not checked until run-time, delaying any error messages. Second, there is no type checking. The *scanf* will happily attempt to format anything as anything, often with surprising results. Third, the *addresses* of the locations in which the input data should be placed must be passed, i.e. *&location*. Forgetting the address operator is an extremely common error for students (and professionals!).

Output is only marginally better. Again, cryptic format strings must be specified and the programmer must ensure that an end-of-line character (\n) is included, otherwise no output may appear.

2.3 C++

Many computer science departments have moved to teaching an object-oriented language in their first course. Probably the most popular language at present is C++ [7]. Although C++ is essentially an extension of C, a different model of I/O is supported. This model is based on streams and in some senses is an improvement over I/O in C.

However, the stream model has a major drawback in that it introduces the concept of overloading of operators. As an example, consider the following C++ code which outputs an integer, *a*, to the console:

```
cout << a
```

The interpretation of this code is that the integer *a* is sent to the output stream *cout*. The result of this code is an output stream, which allows these operations to be concatenated. For example

```
cout << a << b
```

where *a* and *b* are integers, will output the value of *a* followed by the value of *b*. Since these are expressions, parentheses may be used and the following code is exactly equivalent to the above.

```
(cout << a) << b
```

However, the student must be extremely careful with the placement of the parentheses. For example, the

following code has a quite different meaning:

```
cout << (a << b)
```

The code outputs a single integer value formed by shifting the value *a* left *b* binary places! Explaining this to a student who is still trying to understand basic concepts of programming is extremely difficult.

2.4 Java

Java [8] is becoming an increasingly popular candidate for teaching an object-oriented language in a first year course. While still not heavily used for teaching at present, all signs indicate that it will be widely used for teaching within a very short time.

Java, being based on C++, eliminates many of the problems experienced with C++ and is in many respects nicer and cleaner than C++. In the context of I/O, however, it has its own problems. Consider the following code fragment (taken from [8], page 191):

```
int ch;
int total;
int spaces = 0;
for (total = 0; (ch = in.read(0) != -1; total++) {
  if (Character.isSpace((char)ch)
    spaces++;
}
```

In this code (which counts the number of spaces in the input stream) the variable *in* (which is declared somewhere before the code fragment we have shown) refers to an input stream. Its method *read* is used to read in characters from that stream. This is the standard method in Java to process character streams. Several problems which are potentially confusing for students can be noted:

- The variable *ch*, which is used to read characters, is declared as an integer. This is necessary because the stream's read function returns an integer type, although it logically reads characters.
- An explicit cast must be used to process *ch* as a character. (If the processing of *ch* involved access to *ch* more than once, it would have to be cast each time, or a second variable of type *char* must be used).
- The end of the stream is marked by returning -1. This makes it necessary to use *ch* first as an integer before casting it to *char* and prevents casting at the time it is read in.

It is clearly unfortunate that a concept such as type casting must be introduced early in the course to be able to use character input.

2.5 Summary

The facilities provided to support input and output in most common programming languages bring with them considerable baggage which causes complications, confusion and difficulties for students new to programming. In the next section we briefly describe the Blue language and environment which is the vehicle we use to avoid I/O in the first few weeks of a course.

3 The Blue Language and Environment

It is difficult to separate the language and the environment in Blue since it is an integrated system. Users are not directly aware of the underlying operating system, the file system, the command language, etc. All interaction with the system takes place using a graphical interface. It is not envisaged that there will ever be a version of Blue with a textual interface.

Blue is a pure object-oriented programming language. By this we mean that the only compilable construct is a class and all data items (including built-in types) are represented as objects. Syntactic sugar is provided to allow a familiar syntax for the built-in types such as integers. The notion of a program can be represented by having an initial class with a single method, perhaps called "run". All objects are manipulated using reference semantics.

Each Blue class defines some internal data (and possibly internal routines), a creation routine (constructor) and a set of methods. There are no destructors. These are unnecessary since Blue has garbage collection. Only the constructor and the methods are visible from outside of the class.

Figure 1 shows a Blue class definition for the class *Person*.

```
class Person is
 == The Person class
 internal

  var
   _firstName : String
   _lastName : String
   _yearOfBirth : Integer

 interface
   creation (firstName : String, lastName : String,
            yearOfBirth: Integer) is
    == Create a person object
    do
      _firstName := firstName
      _lastName := lastName
      _yearOfBirth := yearOfBirth
    end creation

 routines
    name -> (nm : String) is
     == Return person's first and last name
     do
       nm := str (_firstName, " ", _lastName)
     end name

    changeName (firstName : String,
               lastName : String) is
     == Change person's first and last name
     do
       _firstName := firstName
       _lastName := lastName
     end changeName
end class
```

Figure 1: A sample Blue class

There are three internal data items, *_firstName*, *_lastName* and *_yearOfBirth* (by convention all instance variable begin with the underscore character). The constructor has two parameters. Constructors are called when the *create* operation is executed..

Thus, an instance of a person may be created by the following code:

```
aPerson := create Person ("John", "Rosenberg", 1953)
```

The variable "aPerson" is a reference to an object of type *Person*. The class *Person* has two methods which may be called using the familiar "dot" notation as follows:

```
aPerson.changeName ("Michael", "Kolling")
```
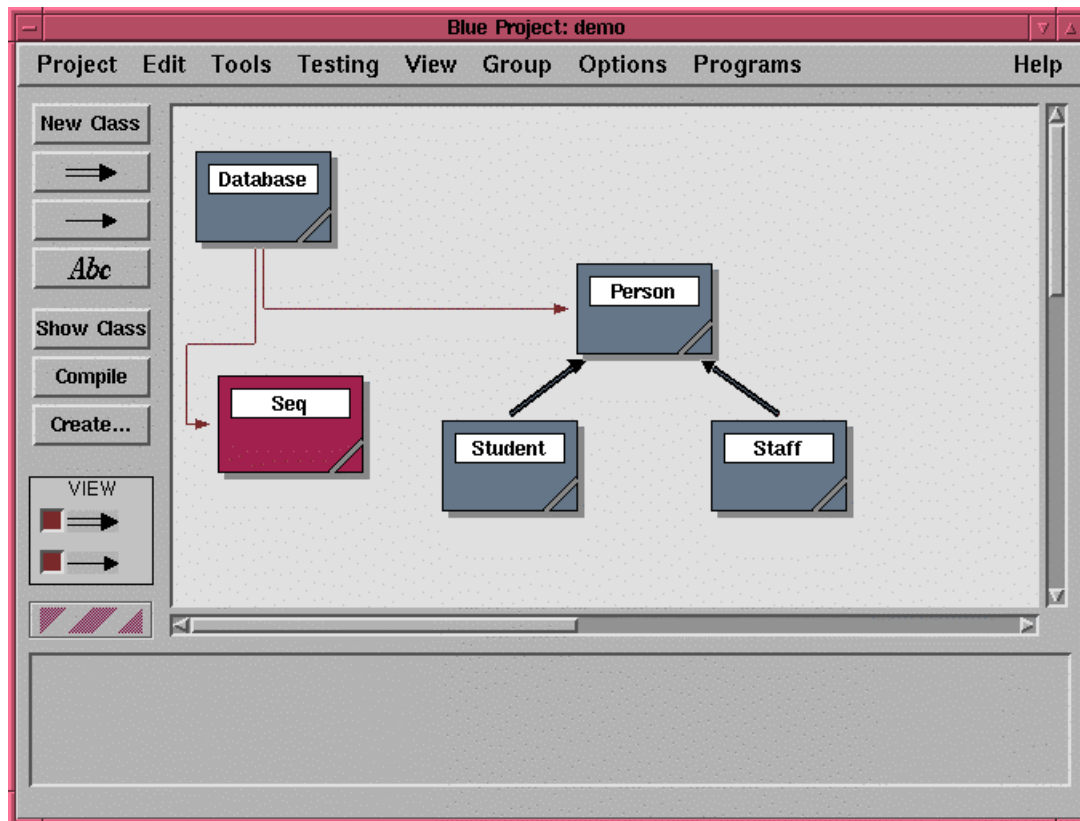
Figure 2: Blue project window

Return parameters (there may be more than one) are listed after the "->" symbol. These are accessed using assignment notation, e.g.

```
theName := aPerson.name
```

Blue routines can also optionally specify pre and post conditions and classes may have class invariants. These have been omitted here for simplicity.The Blue environment is based around the notion of projects. Each program is treated as a separate project. Students begin by defining the classes for the project.

This is done graphically with icons on the screen representing classes. Existing classes from the library may be included in the project using a graphical class browser.

A sample project is shown in figure 2. Lines between the classes indicate relationships. Double lines indicate inheritance while the single lines represent inclusion. These relationships may be defined graphically using the mouse.

The code associated with a class may be edited by double clicking on the class icon. This starts the editor (known as "Red"). There are no header files. Instead the code may either be viewed as the interface only or the implementation.

Blue automatically maintains the relationships between classes and a single button causes recompilation of those classes which have been modified and all classes which depend on these.

The region of the project window at the bottom is known as the Object Bench and is the key to avoiding I/O in the early stages of a course.

4. An Alternative to I/O

It is instructive to review the reasons for requiring I/O during the first few weeks. First, we need output so that we can examine the results of the execution of some code. We may also want to display intermediate values of execution in order to test or debug. Second, we need input so that we can provide input data or parameters to programs, usually to thoroughly test them. It should be noted that input can often be avoided by "hard coding" input values and changing these and recompiling in order to test different values. However, this is inconvenient and inefficient in terms of computing resources.

The mechanisms described in this section support both the display of results and the input of parameters and therefore avoid the need for specialised I/O facilities to be taught in the first few weeks.

4.1 The Basic Mechanisms

The Blue system provides three basic mechanisms which eliminate the need for I/O during the early stages of a course. These are the ability to interactively create an instance of a class, the ability to interactively call the methods of an object and the ability to examine the internal state of an object.

An instance of a class may be created by selecting a class in the project and clicking the *Create* button which results in a dialogue box being displayed. The dialogue includes the definition of the creation header and allows
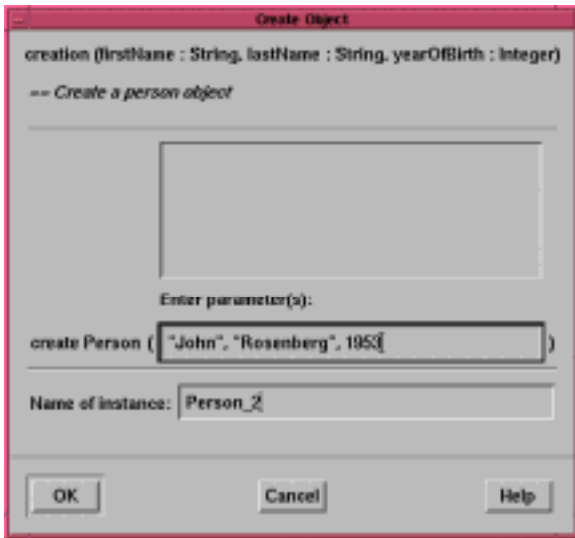
Figure 3: Object creation dialogue

entry of the creation parameters (if required) and a name for the instance (see figure 3). Previous creation parameters are available for reuse in the area above the entry field. After *OK* is clicked the object is created and is represented on the Object Bench by an icon.

Selection of the object icon with the right mouse button displays a popup menu of the methods of the corresponding class as well as inspect and remove options (see figure 4). Methods may be called by selecting them from the menu. Again, a dialogue box is displayed for parameter entry and returned results are displayed after the method has been called. The internal data of an object may be viewed by selecting *inspect* from the pop-up menu.
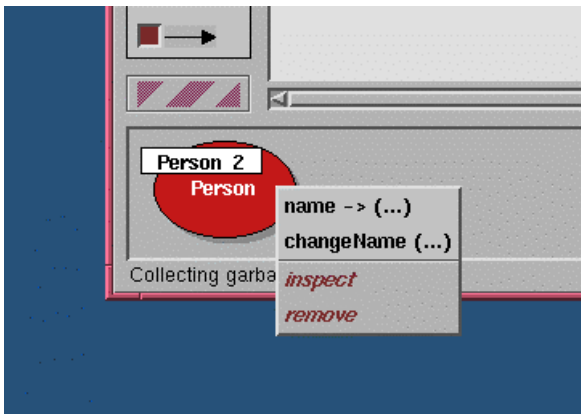

Figure 4: Method menu

There are additional facilities provided for handling object references which cannot simply be displayed as a value and for following these references. These are beyond the scope of this paper and are described in [4].

4.2 An Example

In this section we show how the mechanisms described above may be used to test a fairly complex object-oriented application without any use of input/output statements. The code has been simplified by the removal of comments and pre and post conditions.

The example is a database of people and is represented

by the project window in figure 2. The database is implemented using a library class which implements a simple sequence as a linked list. There are operations to add people to the database and to scan through all of the entries in the database.

The base class of records in the database is *Person*. There are two specialised categories of people, namely staff and students. This is implemented by defining two new classes, *Staff* and *Student*, which inherit from the class *Person*.

The code for the class *Person* was shown in figure 1. The creation routine requires that the first name, last name and date of birth are provided, and there are two routines, one which returns the complete name and another which allows the name to be changed.

The code for class Staff and class Student is shown in figures 5 and 6 respectively. Both of these classes include additional internal data, change the signature of the constructor and add new routines.

```
class Staff is Person
 == The Staff class
 internal

  var
    _roomNumber : String
    _position : String

 interface
   creation (firstName : String, lastName : String,
             yearOfBirth : Integer,
             roomNumber : String, position : String) is
     == Create a Staff object
     do
       super!creation (firstName, lastName, yearOfBirth)
       _roomNumber := roomNumber
       _position := position
     end creation

 routines
     room -> (roomNumber: String) is
       == Return the room information for this person
       do
          roomNumber := _roomNumber
       end room

     changeRoom (newRoom : String) is
       == Change the room information for this person
       do
          _roomNumber := newRoom
       end changeRoom
end class
```

Figure 5: The code for the class *Staff*

```
class Student is Person
 == The Student class
 internal

  var
    _studentId: String
    _accountName: String

 interface
   creation (firstName : String, lastName : String,
             yearOfBirth : Integer, studentId : String,
             accountName : String) is
     == Create a Student object
     do
       super!creation (firstName, lastName, yearOfBirth)
       _studentId := studentId
       _accountName := accountName
     end creation

 routines
     studentId-> (sid : String) is
       == Return student id
       do
          sid := _studentId
       end studentId
end class
```

Figure 6: The code for the class *Student*

Finally, the code of the class Database is shown in figure 7. This class creates a sequence of *Person* and

provides routines to add *Person*s and scan through the database. It uses the class *Seq* (the code for which is not shown) from the library.

```
class Database is

  uses Seq, Person

 internal
  var
    personSeq : Seq<Person>

 interface
   creation is
    == Create a Database object
    do
      personSeq := create Seq<Person>
    end creation

 routines
    add (p : Person) is
     == Add a Person to the database
     do
       personSeq.add(p)
     end add

    initScan is
     == Initialise a scan of the database
     do
       personSeq.initScan
     end initScan

    getNext -> (p : Person) is
     == Get the next Person in the database
     do
       p := personSeq.getNext
     end getNext
end class
```

Figure 7: The code for the class *Database*

Had we been developing this system in another object-oriented language and environment, say C++, we would have almost certainly included routines on the interfaces of *Person*, *Staff* and *Student* to print out the details of the objects. The class *Database* would have used these to provide a routine to print all entries in the database. Finally, we would needed some input routines to be able to enter some *Person*, *Staff* and *Student* data. This would have added considerable complexity and length to the application.

With the tools described above, which are provided as part of the Blue system, this application can be developed and fully tested without including a single line of code to support I/O. The testing could proceed as follows.

First, a *Person* object can be created as shown in figure 3. The constructor parameters are entered and the object appears as an icon on the Object Bench at the bottom of the project window as shown in figure 4. The routines of the class *Person* may then be tested using the pull-down menu also shown in figure 4. The internal data of the object may be examined using the *inspect* option on the pull-down menu. Several different *Person* objects may be created to thoroughly test the class. The *Staff* and *Student* classes may be tested in a similar way and objects of these classes may be created on the Object Bench. Note that we have created several different classes of object without the use of input statements.

An object of the class *Database* can also be created. The constructor has no parameters and so the object is immediately created and the corresponding icon appears on the Object Bench. The situation may, at this stage look like that shown in figure 8, with a number of *Person*, *Student* and *Staff* objects, plus a *Database* object on the Object Bench.

We would now like to create a database of *Person* objects. We can do this using the *add* routine of the *Database* object. The dialogue shown in figure 9 appears and we are required to pass to the routine an object of class *Person*. We have many objects of class *Person* on the Object Bench (including all objects of class *Staff* or *Student*, since, through inheritance, these are also of class *Person*). We can pass one of the objects on the Object Bench by clicking on its icon, or entering its name. By repeated use of the *add* routine we can create a database of *Person* objects.
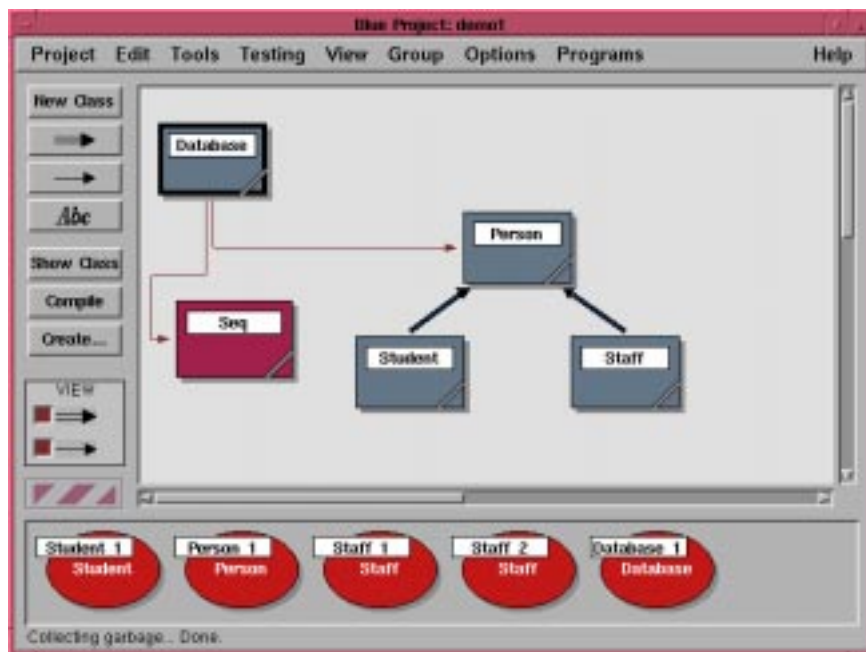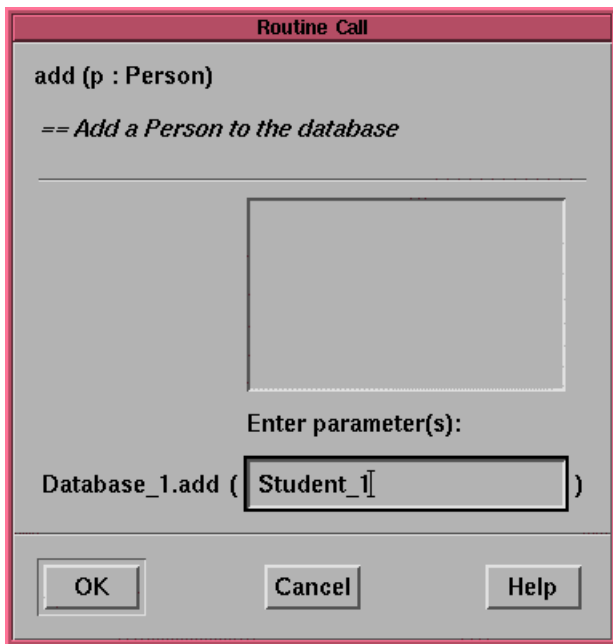

Figure 8: A number of *Person* objects

Figure 9: Add a *Person* to the database

Finally, we would like to scan through the database to ensure that it has been created correctly. This is achieved in two steps. First, we select the *initScan* routine of the *Database* object. This routine has no parameters and simply initialises a traversal of the database. Each subsequent call of the *getNext* routine returns an object of class *Person*.

Let us assume that the first object to be placed in the database was *Student_1*. Figure 10 shows the dialogue displayed as a result of the first call on *getNext*. It indicates that a reference to an object of class *Person* was returned. (If the end of the database had been reached then a *nil* reference would have been shown.) The object reference returned may be inspected by selecting it and clicking on the inspect button in the dialogue. The window shown in figure 11 is then displayed. This shows the *actual* class of the object (in this case *Student*) and the internal data. Note that we have managed to view the contents of the database without writing a single output statement.
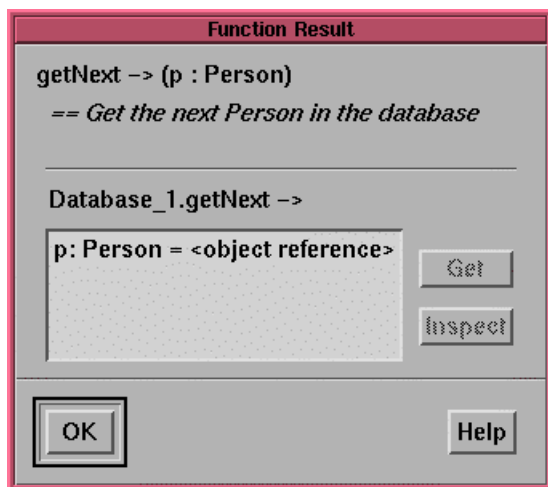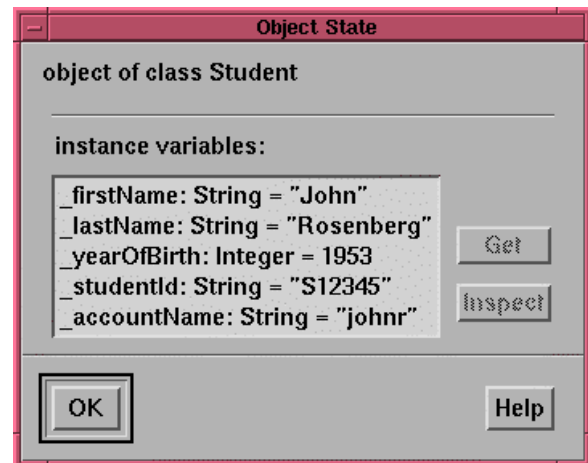

Figure 10: The first *Person* in the database

Figure 11: Inspection of the *Person* details

5 Conclusions

As educators we often underestimate the difficulties encountered by students as they try to learn their first programming language. It is important that we make this experience as less painful as possible. Clearly one of the stumbling blocks for beginners is input and output. This is evidenced by the numerous examples shown above in several different common programming languages.

In this paper we have described a relatively simple set of mechanisms, included as part of the Blue programming system, which allow complex programs to be developed and tested without using any input or output statements. This allows students to concentrate on understanding the basic concepts of programming without having to deal with the particular idiosyncrasies of I/O in the language they are learning. Of course, I/O must be introduced at some point, but this can be delayed until several weeks into the course when students have more confidence with the constructs of the language.

The Blue language and program development environment are being used for the first time starting in March 1997 to teach object-oriented programming to approximately seven hundred and fifty first year computer science students. Although at the time of preparation of this paper we are only two weeks into the course, initial indications are that the students find the Blue facilities both intuitive and valuable as an aid to understanding and testing programs.

References

[1] 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.

[2] Kölling, M.. and Rosenberg, J. "Blue - A Language for Teaching Object-Oriented Programming", *Proceedings ACM SIGCSE Symposium*, 1996, pp. 190-194.

[3] Kölling, M.. and Rosenberg, J. "An Object-Oriented Program Development Environment for the First Programming Course", *Proceedings ACM SIGCSE Symposium*, 1996, pp. 83-87.3.

[4]   Rosenberg, J. and Kölling, M. "Testing Object-Oriented Programs: Making It Simple", *Proceedings ACM SIGCSE* Symposium, San Jose, 1997, pp. 77-81.

[5]   Jensen, K. and Wirth, N. "Pascal User Manual and Report", Springer-Verlag, 1975.

[6]   Kernighan, B. and Ritchie, D. "The C Programming Language", Prentice_Hall, 1978.

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

[8]   Arnold, K., Gosling, J. "The Java Programming Language", Addison Wesley, 1996.