Testing Object-Oriented Programs: Making it Simple

John Rosenberg and Michael Kölling
Basser Department of Computer Science
University of Sydney, NSW 2006
Australia

{johnr,mik}@cs.usyd.edu.au

Abstract

One of the major difficulties facing anyone trying to teach the first programming course is how to encourage students to thoroughly test their programs. We would argue that the main reasons for this are the lack of suitable tools for testing and the need to write extra "debug" code in order to verify correct operation. We further argue that the problem is even worse with object-oriented languages because of multiple classes and encapsulation. In this paper we describe the testing tools within the Blue programming environment which allow object-oriented programs to be thoroughly tested without writing a single line of new code.

1 Introduction

The design and implementation of error-free programs is an extremely difficult task, even for experienced computing professionals. It is therefore not surprising that students have great difficulty in producing well debugged programs. This is further compounded by the fact that most students do not undertake rigorous testing of their code. The aim of this paper is to explore the reasons for this and to describe some tools which alleviate the problem.

As experienced programmers it is obvious to us that when we develop new programs we must also test them. Why is this not obvious to students? The answer may well lie in the fact that students begin by writing very small programs, so small that in most cases it is difficult to argue the case for any serious testing. The classic example of this is the infamous "Hello World" program.

One of the advantages of teaching using an object oriented paradigm is that it actively encourages decomposition of the program into classes and the re-use of existing library code. This means that students can work on larger projects earlier in the course. They need not write all of the code for the project themselves; they can write just a few of the classes and these can then be integrated with those provided.

Proceedings of 28th SIGCSE Technical Symposium on Computer Science Education, San Jose, California, U.S.A., February 1997, pp 77-81.

However, this advantage in terms of structure is itself an obstacle in terms of testing. Typical object oriented programs will have a number of classes. We would encourage students to test each of these classes individually before combining them to form a solution to the problem. In order to do this, test programs must be developed. There may well be one test program for each class. Since a class typically has several methods, these test programs can become quite lengthy and complex. In fact it would not be unusual for the test program to be longer than the class being tested! The development of such test programs can easily become more of an obstacle for the students than the development of the original classes themselves.

Clearly what is required is better tools for testing. In particular we would like to reduce the amount of code which must be written in order to test classes. Ideally no special testing code should have to be written.

This paper describes an environment which supports this ideal by allowing the interactive creation of instances of classes and interactively invocation of their methods. This, coupled with the ability to examine the internal state variables of objects allows students to interactive test their classes without writing a single line of test code.

The tools described have been developed as a part of a larger project known as Blue [1]. Blue is both an object-oriented programming language [2] and a program development environment [3] and has been specifically designed for teaching programming to first year students.

The tools described in this paper are only those used for testing. There is still a need for specialised debugging tools which in the Blue system include breakpoints, single-stepping, display of variable values, etc. It must be emphasized that we see a clear distinction between testing, which must take place for the very first program written by a student, and debugging, which can be introduced after the students have some experience.

In this paper we first describe the Blue language and environment followed by the use of the tools. We then conclude with a brief discussion of the advantages of our approach.

2 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
 internal
  var
   surname : string
   givenName : string
   numChildren : integer

 interface
   creation (Surname : string,
             GivenName : string) is
    == creates a person object
    do
      surname := Surname
      givenName := GivenName
      numChildren := 0
    end creation

   routines
     addChildren (New : integer) ->
                 (Total : integer) is
      == add some extra children
      do
        numChildren := numChildren+New
        Total := numChildren
      end addChildren

     getNames -> (Surname : string,
                  GivenName : string) is
      == get the names of the person
      do
        Surname := surname
        GivenName := givenName
      end getNames
end class
```

Figure 1: A sample Blue class

There are three internal data items, surname, givenName and numChildren. The constructor has two parameters. Constructors, when called, always return an instance of their class.

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

```
aPerson := create person ("Rosenberg", "John")
```

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:

```
n := aPerson.addChildren(2)
```

Note that methods may have multiple return parameters listed after the "->" symbol. These are accessed using a multiple assignment notation, e.g.

```
sName, gName := aPerson.getNames
```

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 (not used in the example) 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. For library classes only the interface can be accessed.

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 described in the next section.

3 Interactive Testing - The Object Bench

3.1 The Problem

One of the major advantages of object-oriented programming languages is that they actively encourage the decomposition of programs into classes where each class encapsulates the state of an object type and provides interface routines (methods) for accessing and manipulating the state. The concrete structure of the internal state is hidden from users of the class so that it can potentially be changed in the future without affecting other classes. This hiding of information improves reliability and reduces the cost of software maintenance.

Although encapsulation is clearly of value in terms of software engineering, it does create difficulties for the implementor of a class in terms of testing. It is certainly possible to write a test program which creates an instance of a class and calls the various methods, printing out results for later examination. What is of concern is that such a program will tend to be quite large since it must prompt for parameters from the user to enable testing of various combinations of parameters to the methods.

Assuming that we are able to write such a test program, it is more difficult to ascertain whether the internal data of the class being tested is correct. It may not even be possible to access some of the internal data because no appropriate methods are provided. Such hiding of data purely related to the implementation is not uncommon.

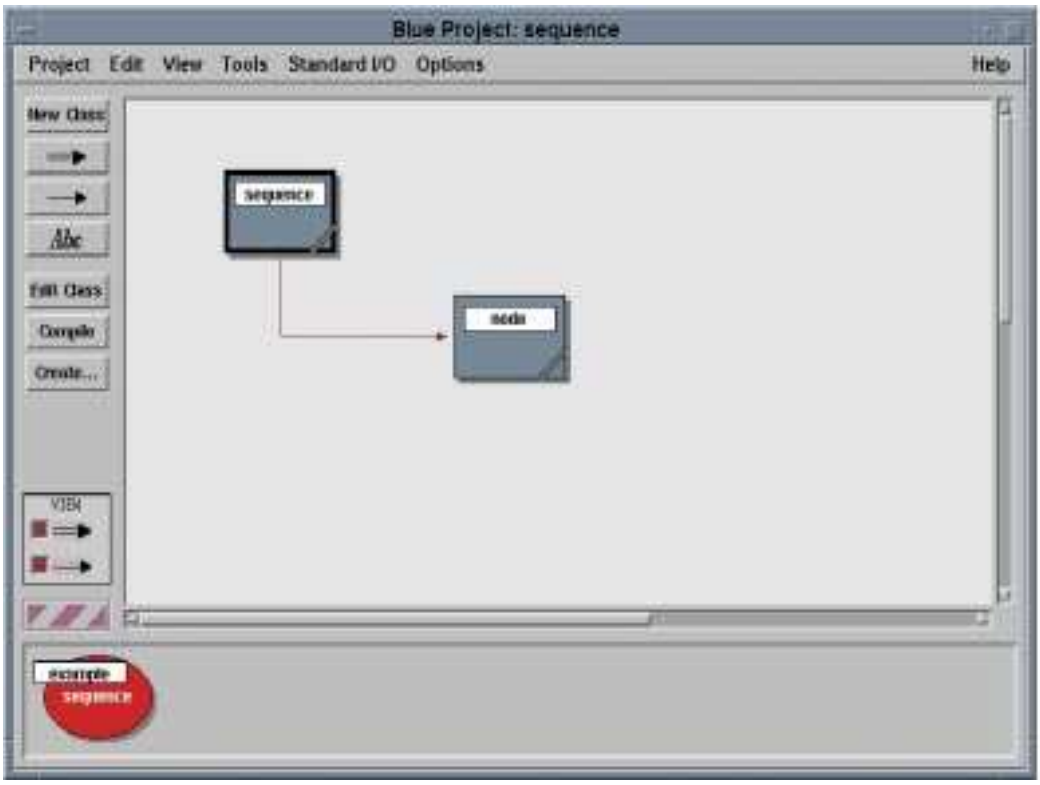

Figure 2: Blue project window

There are at least two solutions to this problem. First, "debug" print statements could be added into the class code to print out relevant internal data when methods are called. This has several associated problems. The insertion of new test code could well introduce errors in itself. How often has a student come to you with a program that has errors stating that he has not changed a thing - he just added a print statement! In addition if there are several classes, the volume of output can become difficult to interpret.

The second solution is to use a symbolic debugger to insert breakpoints and examine the data. This requires the student to become familiar with the debugger at a very early stage. Since we would like students to test the very first class they write, it may be unrealistic to expect them to learn to use the debugger at the same time.

We have above identified two major problems with testing object-oriented programs. First, a substantial volume of test code must be written and second we need to be able to examine the internal data of objects.

3.2 The Solution

The Blue system provides a solution to both of the problems identified by providing a facility for interactive testing of classes. This facility has two components. The first allows an instance of a class to be created interactively and the second allows examination of 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 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 instance is created and is represented on the Object Bench by an icon.

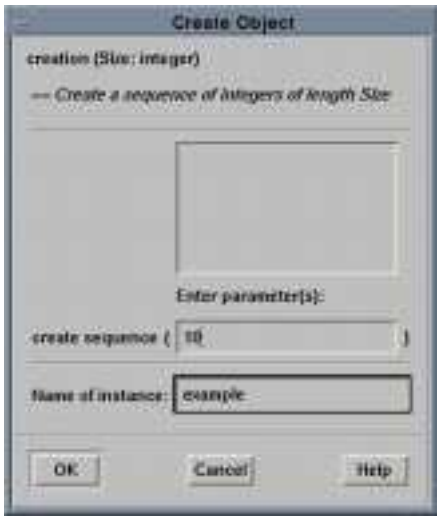

Figure 3: Object creation dialogue

Selection of the instance 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.

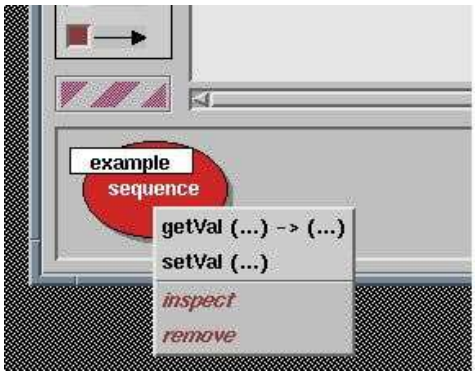

Figure 4: Method menu

There are additional facilities provided for handling object references which cannot simply be displayed as a value and for following these references. These are discussed in the next section.

4 An Example

In this section we demonstrate the interactive testing mechanisms using a simple example. The example consists of two classes which implement a sequence of integers. The code is greatly simplified to reduce its size and does not include any error handling, etc.

The sequence class is designed to support a fixed (at construction time) size sequence of integers with an integer index. There are only two operations, *getVal* and *setVal*,

```
class sequence is
  == Implements a sequence of integers
  uses node
 internal
  var
    head : node

 interface
   creation (Size : integer) is
    == Create a sequence of integers
    var
      count : integer
    do
      head := nil
      count := Size
      loop
        head := create node(head, count)
        count := count - 1
      exit on count = 0
      end loop
    end creation

   routines
     getVal(Index : integer) ->
           (Value : integer) is
      == Get the value at position Index
      var
        count : integer
        pos : node
      do
        count := 1
        pos := head
        loop
        exit on count = Index
          pos := pos.getNext
          count := count + 1
        end loop
        Value := pos.getVal
      end getVal

      setVal(Index : integer,
             Value : integer) is
       == Set the value at position Index
      …
end class
```

Figure 5a: The class *sequence*

which both have an index as a parameter and return/set the value at the index respectively. The nodes in the sequence are represented using another class called *node*. The code for these two classes is shown in figures 5a and 5b.

The creation routine of *sequence* creates a linked list of *node*s with each element having an initial value which is its index. Figure 3 shows the dialogue that would be displayed if an instance of *sequence* was created. The dialogue requires the size to be entered and a name to be given to the sequence object. Once *OK* is selected the sequence is created and is displayed on the object bench as shown in figure 2.

Selecting *example* with the right mouse button will display the popup menu shown in figure 4.

```
class node is
  == Implements the nodes for a sequence
 internal
  var
   value : integer
   next : node

 interface
   creation (Next : node,
             Value : integer) is
    == Creates a node for the sequence
    do
      next := Next
      value := Value
    end creation

   routines
     getNext -> (Next : node) is
      == Get a reference to the next node
      do
        Next := next
      end getNext

     getVal -> (Value : integer) is
      == Get the value in the current node
      do
        Value := value
      end getVal

     setVal (Value : integer) is
      == Set the value for the current node
      do
        value := Value
      end setVal
end class
```

Figure 5b: The class *node*

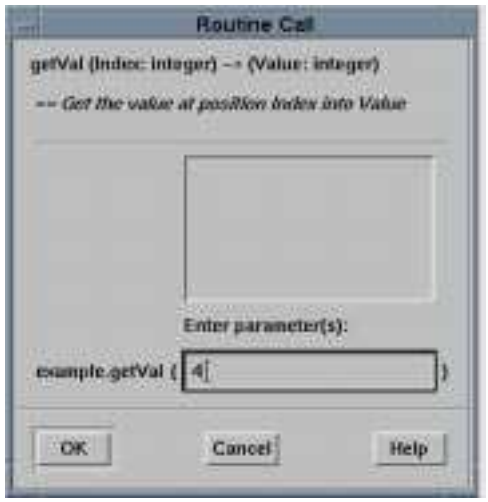

Figure 6: Method call dialogue

We may now interactively call the methods of example. Suppose we select *getVal* from the menu. The dialogue in figure 6 will be displayed. This allows the entry of an index and, after selecting *OK*, displays a new dialogue with the return value.

Selection of the *Inspect* option from the menu displays the dialogue shown in figure 7. The only internal data in an object of class *sequence* is the reference to the head of the list. The declaration of the variable is shown with its value being <object reference>. We know that the reference is to an object of class *node*. This can be inspected by selecting the object reference and clicking the inspect button in the sequence inspection dialogue.

The dialogue shown in figure 8 will be displayed. It shows the internal state of the object referenced by the *head* variable of the *sequence* object. We can see the value in the node and that it has a reference to the node in the list. This reference can again be followed using the inspect button. The procedure could be repeated until we reach the end of the list which will be marked by a *nil* object reference that cannot be followed.

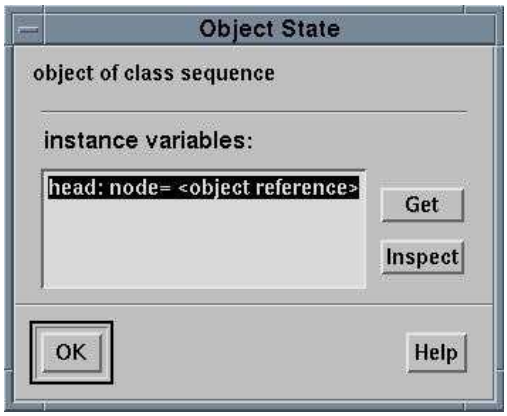

Figure 7: Inspection dialogue

With these two basic mechanisms, interactive method invocation and internal data inspection, we are now able to fully test the sequence and node classes.

Note that we are able to examine any object reachable from an object available on the Object Bench. Sometimes it can become clumsy to repeatedly navigate through object references to reach an object we wish to examine. The Get button on the inspection dialogue (figure 8) allows a reference to any existing object to be placed on the object bench so that it can be re-examined at a later time.

In more complex examples we may wish to create instances of many different classes and test these each individually. Each object created will appear on the object bench with its class name and object name. Such objects may also be used as parameters to methods of other objects and are referred to using their name.

Finally, there is a record facility which will textually record all interactive object creations, method invocations, return values, text input and text output. This may be used by students as part of an assignment submission.

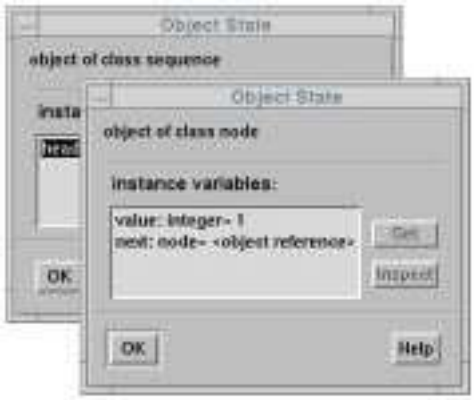

Figure 8: Following object references

6 Conclusions

Our experience with teaching the first programming course over many years is that students, in general, fail to adequately test their software. Indications are that a move to an object-oriented language may well make the problem worse because a test program must be written for each class. In addition, the encapsulation of data within classes complicates testing.

In this paper we have argued the need for testing facilities which do not require code to be written and are simple enough to be used by novices. The Blue system provides such a facility as an integral component within the programming environment. Objects may be created and tested interactively and the encapsulated data of objects may be inspected. Appropriate facilities are provided to examine arbitrary dynamic data structures.

Since these tools are a part of the standard environment and are simple enough to use to test the first programs written by a student, testing is actively encouraged and considered to be a part of the normal program development cycle. We expect this to result in an improvement in the reliability of student programs and a better appreciation by the students of the importance of thorough testing.

The environment described in this paper has been fully implemented and will be utilised as the basis for our first programming course at the start of 1997. We are also developing more advanced visualisation tools to be used later in the course. These will include the ability to examine the stack and to navigate through data structures in the heap in a graphical manner.

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.