Support for Object-Oriented Testing

Michael Kölling
School of Computer Science &
Software Engineering
Monash University
Australia

michael.kolling@csse.monash.edu.au

John Rosenberg
Faculty of
Information Technology
Monash University
Australia

johnr@fcit.monash.edu.au

Abstract

Object-orientation has rapidly become accepted as the preferred paradigm for large scale system design. There is considerable literature describing approaches to object-oriented design and implementation. However discussion of testing in an object-oriented environment has been conspicuous by its absence. At first sight it appears that decomposition of a system into a potentially large number of information-hiding classes greatly increases the cost of testing. However, in this paper we show that by taking an object-oriented approach to testing, and the inclusion of appropriate tools in the development environment, testing time can be greatly reduced and special purpose test code can be virtually eliminated.

1 Introduction

Object-orientation has rapidly become accepted as the preferred paradigm for large scale system design. The reasons for this are well known and understood. First, classes provide an excellent structuring mechanism. They allow a system to be divided into well defined units which may then be implemented separately. Second, classes support information-hiding. A class can export a purely procedural interface and the internal structure of data may be hidden. This allows the structure to be changed without affecting users of the class, thus simplifying maintenance.

Third, object-orientation encourages and supports software reuse. This may be achieved either through the simple reuse of a class in a library, or via inheritance, whereby a new class may be created as an extension of an existing one. In both cases the result is a reduction in the amount of software which must be written and, as a result, an improvement in the reliability of the resultant system since previously tested classes may be utilised.

If we are to capitalise on the potential advantages of object-orientation then it is important that the object-oriented approach is adopted and supported throughout the software development process. The design phase is now reasonably well understood (although notoriously difficult) and there are various tools to assist with the process. Similarly there are tools to support the implementation phase. Library browsers may be used to assist with locating existing classes which may be reused in the project and tools provide support for editing, compilation, etc.

However, testing is often ignored by the designers of software tools and the programmer is left to his/her own resources. Some may argue that testing should not be difficult (or even necessary at all) if a proper design and implementation process has taken place. We all know this not to be true and we must subject new software to rigorous testing before it can be used in a production environment.

Unfortunately the very advantages of object-orientation cited above become potential disadvantages when we consider testing. The structuring of the system as a set of independent classes requires that each of these must be tested and there may be a large number of them. In addition, information-hiding, which encourages designers of classes to have purely procedural interfaces, makes it difficult to determine whether the class is working correctly, since the state of internal data may not be accessible via the interface.

The result is that the programmer must effectively develop a test program for each class. Each such test program must create an instance of the class being tested and include calls to each of the methods supported by the class. The test program will need to prompt the user for the parameters for these method calls so that various combinations can be exercised. The test program must also display the results of each method call. The result is that the test program may well be more complex and larger than the class being tested.

Once we have written such a test program, we still may not be able to ascertain whether the internal data of the class being tested is correct because of the inability to access all of the internal data via the procedural interface.

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 and these can detract from the original testing process. 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 adds further complexity to the process. In addition, the symbolic debugger may not be able to adequately display complex linked structures.

Clearly what is required is tools specifically designed to assist with testing object-oriented applications. 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. These tools should be an integrated component of a complete object-oriented development environment.

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 the programmer to interactively 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 [3]. Blue is both an object-oriented programming language [4, 6] and a program development environment [5] and has been specifically designed for teaching programming to first year students. The system has been in use for nearly two years. The authors are currently working on a new version of the environment designed for Java developers.

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 is required for every program, and debugging, which is only required if testing finds a failure.

In this paper we first discuss a general approach to testing object-oriented programs. We then show how this technique has been included in an object-oriented program development environment. This is followed by a brief description of the technology employed to implement this environment.

2 Object-oriented testing

The key advantage of the object-oriented paradigm is that it provides a uniform structure for all components in the form of a procedural interface. Although as indicated above, this appears to complicate the testing process, it may be exploited to support an object-oriented approach to testing.

In order to test a class the programmer must be able undertake the following activities:

(a) create an instance of the class, i.e. an object, passing the appropriate parameters to the constructor

(b) call the methods of the object passing parameters and receiving results

(c) examine the internal data of the object

As discussed above, this can be achieved by writing a test program for each class and the inclusion of debug statements. However, it could also be achieved by the inclusion of appropriate mechanisms in the program development environment itself. This would eliminate the need for both test programs and the modification of the class being tested.

The mechanisms would work in the following manner. First, the environment provides the user with the ability to interactively create an object of any class. The class is selected and the system prompts for the constructor parameters. Once the object has been constructed, any of its methods may be interactively invoked; again the user is prompted for parameters. Results (return parameters) are displayed in a dialogue box.

Second, the environment provides an inspection facility which allows the internal data of an object to be examined. The data is displayed in a dialogue box along with the types of each field.

The mechanisms described so far are sufficient to test classes with scalars as parameters. However, it is of course common to pass other objects as parameters. The system can support this by allowing an arbitrary number of objects of arbitrary classes to be constructed. These may then be composed, i.e. one such object can be passed as a parameter to another.

A further problem is that results of method calls may include objects, as may the internal data of a class. How should such objects be displayed? Our approach is to initially display these as a typed object reference. The user may then choose to display the contents of the object referred to by such a reference. The contents may contain further object references and these may be accessed in the same manner. This facility allows arbitrary data structures to be examined and traversed.

We call this an object-oriented approach to testing because it exploits the uniform nature of classes and objects to provide a generic testing facility. The major advantage of the approach is that it virtually eliminates the need to write special purpose testing code.

Note that Smalltalk environments [2] traditionally come close to meeting some of these goals. Especially an “instance centred” variation of Smalltalk, named “Portia” supports similar techniques [1]. Smalltalk, however, has disadvantages in other areas, many of which result from the fact that it is a dynamically typed language. More recent environments for statically typed languages seem to have neglected this area.

3 Testing in the Blue environment

The Blue environment is an integrated graphical environment which supports the techniques described in the previous section. Its main window presents a graphical overview of the application structure. Each class in the application is represented by an icon, and relationships between class are displayed (figure 1).

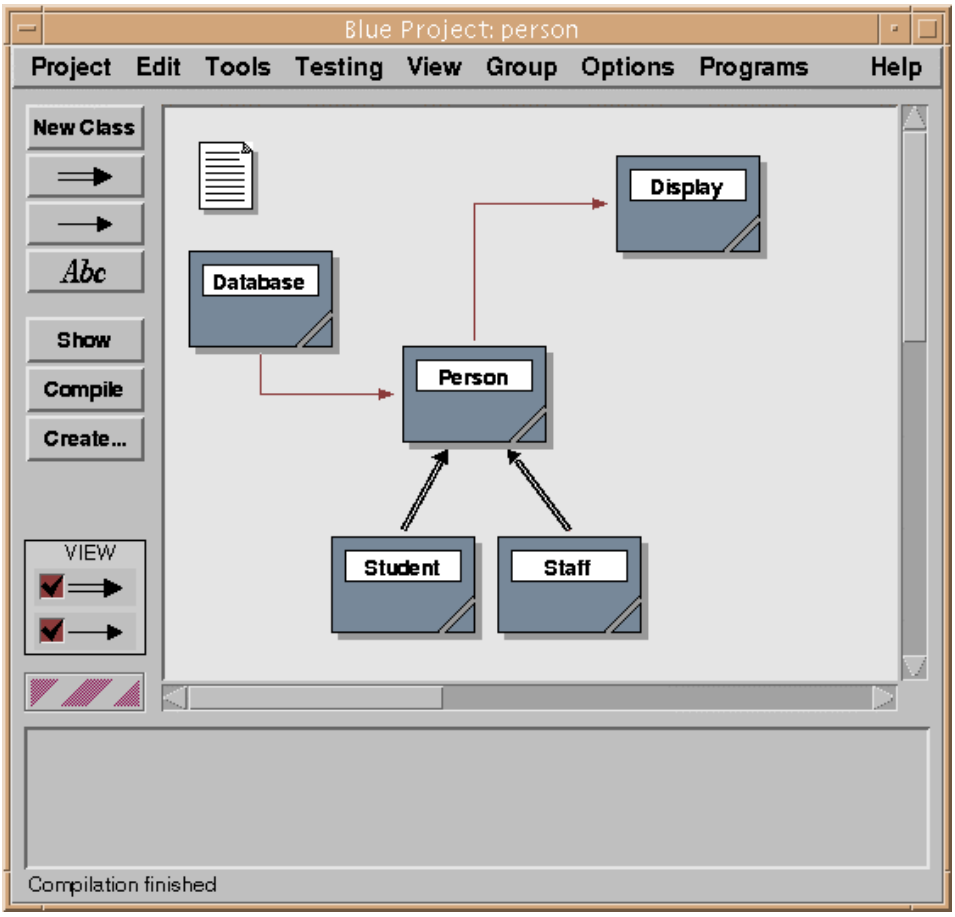

Figure 1: The Blue main window

Double-clicking a class icon opens an editor displaying the class's source code. Each class or the whole application can be compiled with a click on a "compile" button. (For a more detailed description of the Blue environment, see [5]).

The empty area at the bottom of the main window is the *object bench*. We discuss below how it is used to interact with objects.

Once a class within a project has been compiled, objects of that class may be created. Interactive creation of objects is achieved by selecting the class and clicking the "Create" button. (Clicking the right mouse button on a class provides a shortcut to the same function.)

This operation is similar to interactively sending a "new" message to a class in a Smalltalk environment. An instance is interactively created and available for operation. No equivalent of this operation is available in common environments for more recently developed, statically typed programming languages.

Invoking the creation operation on a class results in a normal object creation, including the execution of the creation routine (the "constructor" in C++/Java terminology). As an example, we will use a class "Person" which stores some information about a person and provides interface routines to change and access that information. This class is not meant to be complete or really useful in any sense – it is used here only as an example to demonstrate the Blue object interaction facilities. The interface of the class is shown in figure 2.[1]

[1] This example uses the Blue language. The language itself is not important here, and a similar environment can be constructed for other languages.

```
class interface Person is
================================================
== Author:     M. Kölling
== Version:    1.0
== Date: 8 June 1998
== Short:      Person class for university management project
==
== The class Person implements object representing a person
== in a university management project. It contains
== information common to all persons in the university...
==
================================================
  creation (firstName : String, lastName : String, age : Integer)
    == Create a new person with given name and age.
    pre
      lastName <> nil and age <> nil

 routines
  changeNames (firstName : String, lastName : String)
    == Change the names for this person
    pre
      lastName <> nil

  changeAge (newAge : Integer)
    == Set a new age for this person
    pre
      newAge <> nil

  getNames -> (firstName : String, lastName : String)
    == Return both names of this person

  getAge -> (age : Integer)
    == Return age of this person

end class
```

Figure 2: Interface of class "Person"

When the create operation is invoked a dialogue is displayed to let the user enter routine parameters (figure 3). At the top of this dialogue, the interface of the creation routine is displayed. The interface includes the routine header and the routine comment. Further down is a text field for entering parameter values. Under the parameters is another field to provide a name for the object to be created. A default name is provided and is often adequate. The name will be displayed on the object after it has been created. The large area in the middle of the dialogue is a (currently empty) list of previously used parameter lists. It is provided for convenience during testing of a class: previously made calls can be easily repeated by selecting a parameter combination from the list.

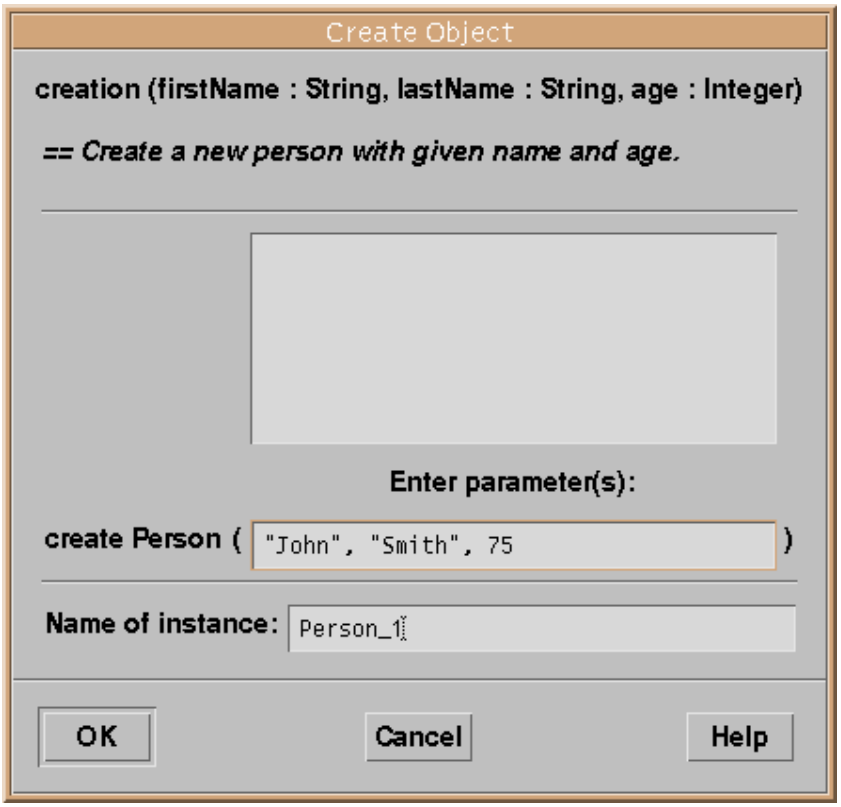

Figure 3: Object creation dialogue

Figure 4: An object on the object bench

Once the dialogue is filled in and the OK button is clicked, the object is created and displayed on the object bench (figure 4). The object is then available to the user for direct interaction. Many different objects of the same or different classes may be created and stored on the object bench at the same time.

Clicking on the object with the right mouse button displays a menu that includes all interface routines of that object (figure 5). Also included in the menu are two special operations available for all objects: *inspect* and *remove*. The remove operation removes the object from the bench when it is no longer needed. The inspect operation is discussed below.

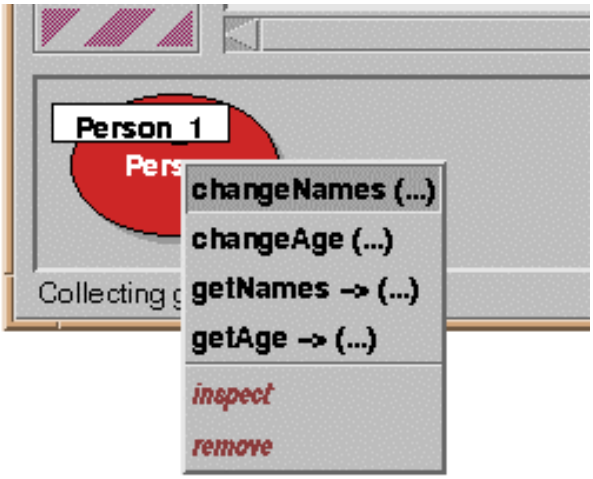

Figure 5: Calling a routine on an object

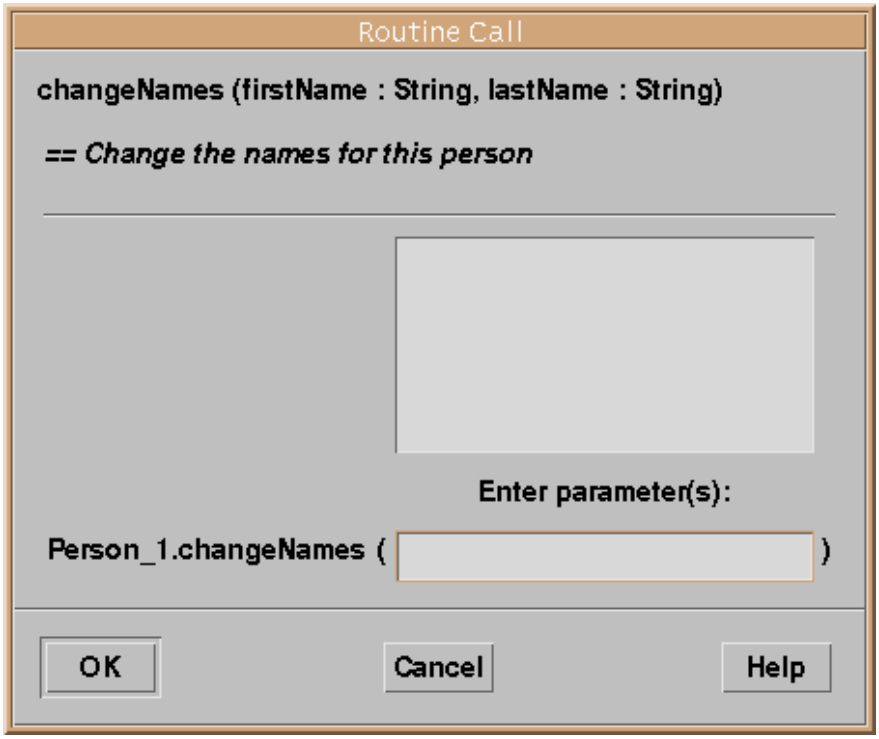

Figure 6: Routine call dialogue for "changeNames"

Symbols in the routine menu indicate whether a routine has parameters or return values. When a routine is selected from the menu, a call to that routine is executed. If the routine has parameters, a parameter dialogue similar to the one seen at the creation of the object is displayed (figure 6). On the click of the OK button the routine is executed and, if the routine returns results, the result values are displayed in another dialogue. Figure 7 shows a function result dialogue for a call to the routine "getNames".[2] Again, at the top of the dialogue window the interface of the called routine is displayed. Below, the actual call is shown in standard Blue syntax (the name of

[2] In Blue, a function can return more than one value. Return values are named, similar to parameters in a parameter list.

the called object, the routine name and – if present – actual parameters). This is followed by a list of the result values of the function. For each result its name, type and value are displayed.

The result of the facilities described so far (interactive creation of objects, interactive routine calls and result display) is that a project can be incrementally developed. There is no need to complete all classes in a project before the first tests can be performed. Instead, each class can be tested as soon as some of its routines have been completed without the need to write special purpose test code. This possibility dramatically changes the style of work available to the developer.

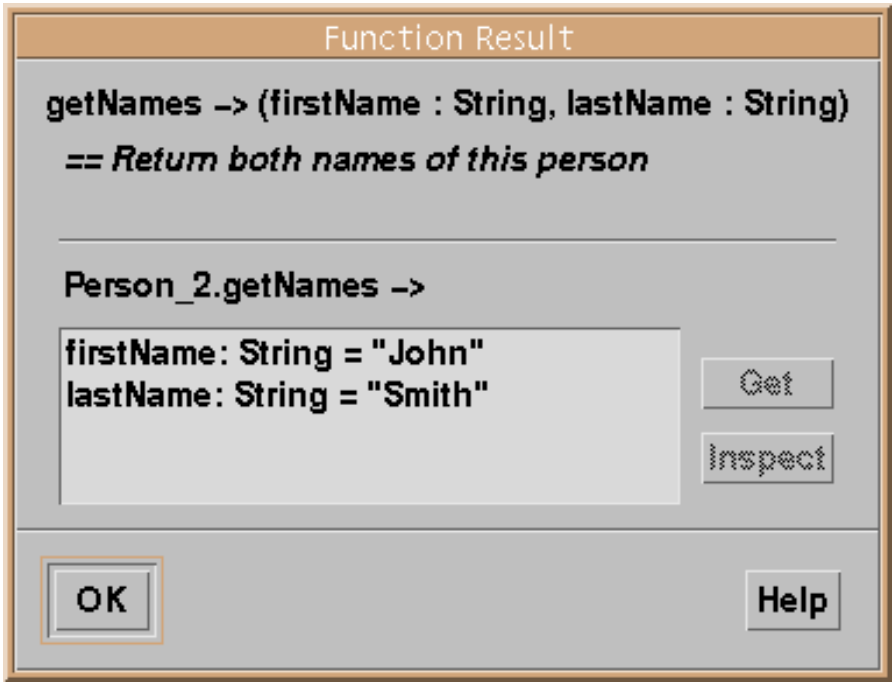

Figure 7: Result dialogue for function "getNames"

4 Composition

During the interactive testing of the system, objects accessible on the object bench may be composed, i.e. one object may be passed as a parameter to the routine of another object. If, for instance, a project includes a database class and a person class with the intention of creating a database of persons, then objects of these classes may be combined. Several person objects could be created. Then a database object is created and its "addPerson" routine is invoked. When the routine call dialogue is visible on the screen, a click on one of the person objects on the object bench will enter its name into the parameter field of the routine call dialogue. The object will be passed as a parameter. This can be done repeatedly to add all the persons from the object bench to the database.

5 Inspection of objects

As mentioned above, a mechanism is needed to examine instance data of objects for cases where an object does not provide accessor functions for that data. This functionality is provided by the *inspect* operation. Using the inspect operation (by selecting it from the object menu or, as a shortcut, double-clicking the object) opens the object and reveals its internals. Figure 8 shows the dialogue displayed as a result of inspecting a person object.

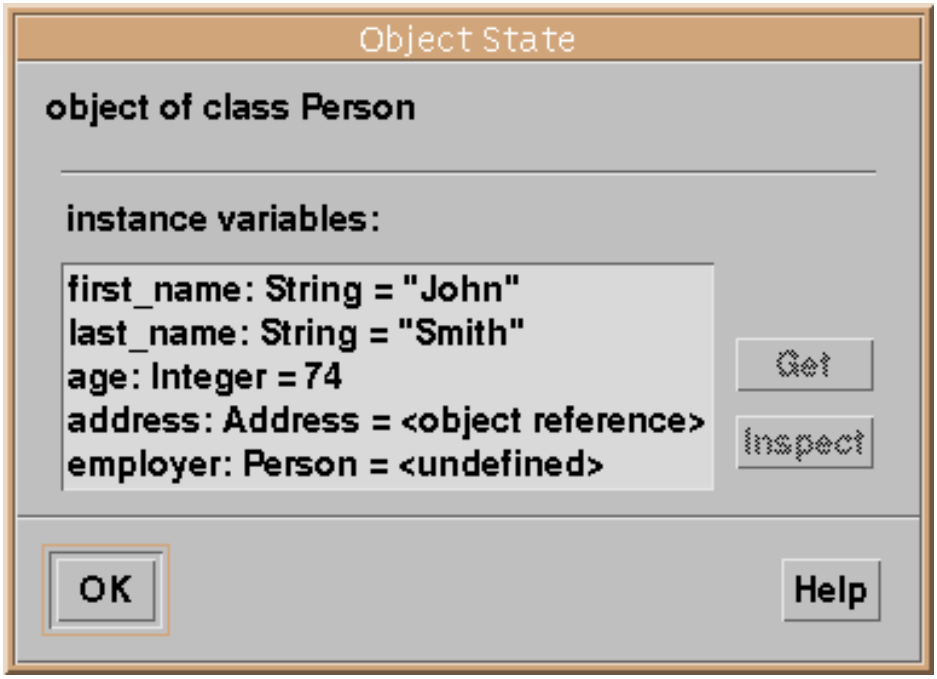

Figure 8: Object inspection dialogue

For this example, we have modified the above definition of the class "Person" to include *address* and *employer* variables, so that we can show how more complex objects can be inspected. The *address* variable holds a reference to another user-defined object of class "Address"; the *employer* variable refers to another person.

The names, types and values of all instance variables of this object are shown. For manifest objects, which have a simple textual representation, values are shown as literals. For variables holding more complex objects only the state of the variable is displayed (whether it is undefined, contains *nil* or an object reference). Those variables may then in turn be inspected by double-clicking on the variable or selecting the variable and then clicking the "Inspect" button. Another window will be opened displaying the internals of that object. An example is shown in figure 9 for the inspection of the address object.

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. This also allows the user to interactively call interface routines of objects that were created internally.

Overall, inspection of objects assists users in thoroughly testing objects of any class by allowing users to observe the effect of routine executions on internal data.

Finally, there is a record facility in Blue which will textually record all interactive object creations, method invocations, return values, text input and text output. This may be used to document the testing that was carried out.

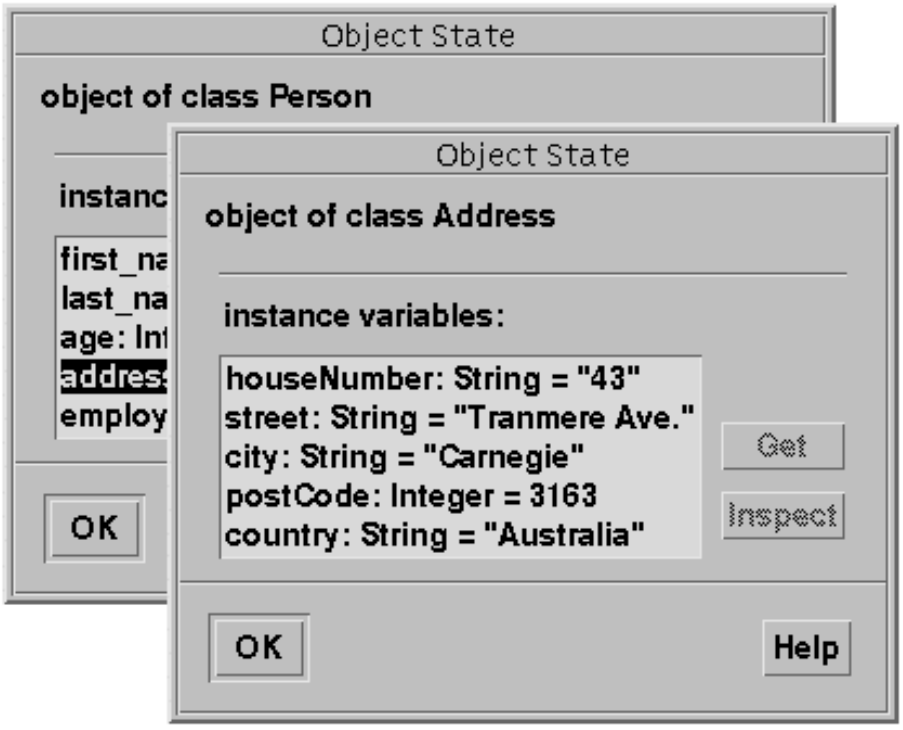

Figure 9: Inspection of "Address" object

6 Pedagogical benefits

The Blue environment was initially developed as a teaching environment for first year students. While the testing facilities described here are generally applicable to all forms of object-oriented software development (educational as well as professional) and thus a teaching environment is only one example of possible applications of these techniques, it is worthwhile summarising the educational benefits which Blue gained from these tools.

- *Incremental development*. Projects can be incrementally developed and tested. There is no need to even syntactically complete a whole application. As soon as one class (or even one routine) is completed it can be compiled and objects can be created, executed and tested. This leads to greater motivation (results are visible more quickly) and a better ability to cope with errors (since early errors can be found and removed before more errors are made, avoiding the harder to find cases of multiple interacting errors). This is clearly also an advantage professional software development situations.
- *Class/object distinction*. Students often have difficulties understanding the relationship between classes and objects. Allowing the direct creation of and interaction with objects greatly facilitates the understanding of these fundamental issues. The pure act of creating a number of objects from a class demonstrates in a powerful way the respective roles of the concepts. If a student has, for example, a class "Person" and creates three different people with different names, the role of the class and the role of each object becomes much more directly understandable.
- *Programming without I/O*. Since input/output operations are often difficult to understand initially (because they often do not conform to language rules or force the

use of advanced concepts), it might lead to a clearer understanding of the abstraction concepts if routine calls are taught before language exceptions (like I/O operations) are shown [7].

- *Interface/implementation distinction*. The distinction between the interface and the implementation of an object – itself an important concept – is clarified. Since only the interface operations are visible to a human user when directly interacting with an object the concept that this is the only part of an object visible to other objects seems a logical conclusion.
- *Testing support*. As was our initial goal, good testing, essential to all serious software development, is supported much better than in conventional systems.

Overall, the interaction and inspection facility provided by the object bench not only meets our initial goal, but offers additional benefits beyond the area of testing.

7 Implementation

To execute an interactive call, the Blue environment uses linguistic reflection. A class is constructed internally that includes the interactive call as its only statement in its creation routine. This class is then passed to the compiler to be translated. An object of the resulting class is instantiated which, as part of the creation, executes its creation routine and with it the interactive call. Result values are stored in this internal object and can then be extracted for display in the result dialogue. To illustrate this technique let us consider an interactive call to the following routine:

```
extract (line: String, o: Object) -> (word: String, valid: Boolean)
```

In Blue, return values are written in a list after a "–>" symbol. Thus, the routine shown has two parameters and two return values. The actual call we want to execute is

```
parser.extract ("input line", obj1)
```

We assume that `parser` and `obj1` are the names of objects on the object bench. To execute this call, the Blue system internally creates the source for another class, usually referred to as the *shell class*. The source code created for our example call would look like this:

```
class __SHELL__ is
  == shell class for interactive call
  uses Parser, Object
  internal var word: String
               valid: Boolean
  interface
    creation (parser: Parser, obj1: Object) is
        == execute interactive call
      do
        word, valid :=
              parser.extract ("input line", obj1)
      end creation
end class
```

The interactive call is then executed by creating an object of the shell class. Creation of the shell object automatically includes the execution of the interactive call as part of its creation routine execution.

The shell class is constructed to have one instance variable for every return value of the interactive call. The return values are stored in those variables and can, after the call, be retrieved

from the created object to be displayed to the user. The display of the return values is, in fact, nothing else than an inspection of the shell object.

Several advantages are associated with this technique. Firstly, the parameter list does not need to be parsed and evaluated by the project management part of the system. The compiler is used for this purpose, thus avoiding duplication of equivalent code. The project manager sets up only the parameter list for the shell creation routine, which contains only object references. Secondly, error messages for mistakes found in the parameter list are produced by the compiler and are thus guaranteed to be the same messages that would be produced for the same error in a non-interactive call. This increases consistency in the environment. Thirdly, the only call ever to be initiated by the object bench (the call to the shell creation routine) has a simple and known interface. Most importantly, it has only object parameters, no literals. This greatly simplifies the implementation. The interactive call, having an arbitrary parameter list, is turned into an internal call completely handled by the runtime system.

8 Conclusion

The object-oriented paradigm has brought obvious advantages to the software development process. However, these advantages have not been exploited in the testing phase. In this paper we have shown how testing itself can be achieved in an object-oriented manner. We have described generic tools which support this process and virtually eliminate the need to write special purpose test code. This approach fully supports incremental software development, allows testing to take place earlier in the project and results in a considerable reduction in overall testing time.

The system described has been implemented for a teaching language known as Blue. The authors are currently constructing a similar environment for Java. A limited version of the environment is currently being tested and should be available for general use by the end of 1998.

References

[1] E. Gold and M. B. Rosson, *Portia: An Instance-Centered Environment for Smalltalk*, in OOPSLA 91 Conference Proceedings, ACM, 62-74, 1991.

[2] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment,* Addison-Wesley, 1984.

[3] M. Kölling, B. Koch and J. Rosenberg, *Requirements for a First Year Object-Oriented Teaching Language*, in ACM SIGCSE Bulletin, ACM, Nashville, 173-177, March 1995.

[4] M. Kölling and J. Rosenberg, *Blue - A Language for Teaching Object-Oriented Programming*, in Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education, ACM, Philadelphia, Pennsylvania, 190-194, March 1996.

[5] M. Kölling and J. Rosenberg, *An Object-Oriented Program Development Environment for the First Programming Course*, in Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education, ACM, Philadelphia, Pennsylvania, 83-87, March 1996.

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

[7] J. Rosenberg and M. Kölling, *I/O Considered Harmful (At least for the first few weeks)*, in Proceedings of the Second Australasian Conference on Computer Science Education, ACM, Melbourne, 216-223, July 1997.