Teaching Object Orientation with the Blue Environment[1]

Michael Kölling
School of Computer Science and Software Engineering
Monash University

mik@csse.monash.edu.au

1 INTRODUCTION

In the last three issues of JOOP [1-3] we discussed problems with and possible solutions for teaching object-oriented programming to beginners. One area mentioned as a problem was programming environments. We claimed, in fact, that the programming environment is, in a teaching context, crucially important. After we made that claim in an earlier column, I received several e-mails from people disputing this point. Their experience was that the environment they used added to their problems and was not a help for them.

This is not a contradiction to our claims, though, but rather confirmation. The point we are trying to make is that most *existing* environments introduce numerous problems, while in *ideal* environment could be a great help. This month's column introduces the Blue environment – our attempt to get closer to an ideal environment.

The main aim of the Blue environment can be summarised in the following two goals:

- to provide an environment which encourages the students to think in terms of objects and classes, and
- to provide an environment that is so easy to use that it does not distract from the task of learning to design and implement a program.

To achieve these goals, details of the underlying operating system are hidden and a point and click world, in which classes and objects are the fundamental concepts of abstraction, is presented.

Figure 1 shows the main window presented to students when they open a Blue project. A project is a group of classes which relate to a particular application. This main window is often referred to as the *project window*. Apart from the pull-down menu bar at the top of the window, it has three components: a toolbar, a class structure overview and the *object bench*. As each class is created it is represented in the class structure overview by a box, with the name of the class at the top. The "classes" may be moved around the screen and inheritance and client relationships can be created using the arrow buttons. These relationships are represented graphically using lines and arrows. Different colours, patterns and symbols are used to mark different kinds and states of classes. This includes information as to whether the class has been compiled and its category, e.g. abstract class, library class, etc. The object bench is discussed later.

---

[1] This paper has been published as: Kölling, M., "Teaching Object Orientation with the Blue Environment," *Journal of Object-Oriented Programming*, 12(2): 14–23, 1999.

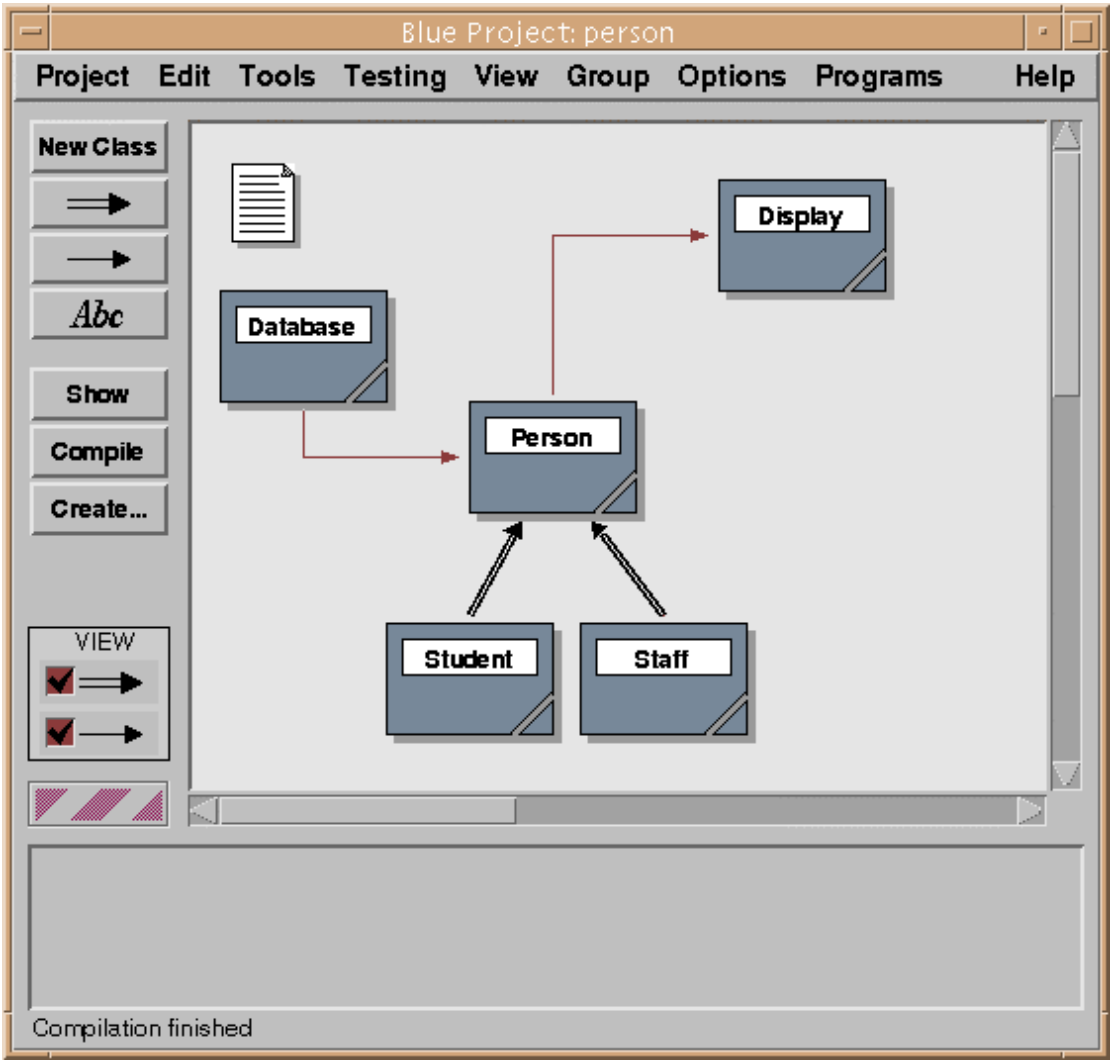

Figure 1: The Blue main window

Students are encouraged to begin their design of an application by creating the required classes. They should think about the relationships between these classes and represent them graphically. Only when the overall class structure has been determined should they start to think about the lower level code.

The text associated with a class may be viewed and edited by double clicking on the class (or selecting a class and pressing the “Show” button). The editor is language sensitive and provides specific functionality for convenient insertion of Blue code. Either the interface of the class or its implementation may be viewed. There is only one representation of the text internally, so it is impossible for these to get out of step with each other.

Compilation of classes or the whole project is achieved by a single click on a button in the project window. The environment automatically keeps track of changes to classes and dependencies. Only classes that need to be recompiled will be recompiled.

A central feature of Blue is the ability to dynamically create instances of classes. When the “Create” button is pressed, a dialogue is shown which prompts for the constructor parameters and an instance of the selected class is created. This instance is displayed in the bottom section of the main window, the object bench. An

interface routine of an instance may be called by selecting it from an associated pop-up menu. Again, Blue prompts for the parameters. Object instances may be composed and passed as parameters to each other. The results of an object invocation are displayed and if a result is an object it can be placed on the object bench. This allows interactive testing of components as they are developed.

A standard console object exists that displays a standard output window and acts as standard input. Alternatively, an application can open its own windows.

2 THE PROJECT

2.1 Working with structure

As mentioned above, the project window is the first window users see when starting up the Blue system. The project structure (classes and their relationships) is displayed on the screen. The effect of this is that the first thing users see and interact with are classes. This meets a goal stated in last month's column: that an object-oriented environment should support classes as its fundamental unit of abstraction.

The effect is twofold. Firstly, students are immediately aware of structure and classes as the basic structuring unit. When we start teaching with Blue, the first thing we show to students is an existing project including a handful of classes. Students see the project representation and are shown how to execute the application. After experimenting for a while with the application itself, they are encouraged to explore its implementation: to look at the source code, to make guesses as to the meaning of certain statements and to make small modifications (such as a change to a string literal in the program). With relatively little guidance they encounter the full edit–compile–execute cycle and very quickly get accustomed to it. All this happens in their very first lab class. An implicit effect of the visualisation of the project structure is that students understand, from the very beginning, the idea that an application is a set of cooperating classes. These classes are distinct entities, they each have their own source code, and they form relationships with each other. All of this is conveyed implicitly by the visualisation and interaction technique; not one word of instruction has to be given about this in the laboratory class.

The second aspect of the structure display takes effect in more advanced exercises: when students create a new project from scratch. (Note that we treat this as an advanced exercise. The first things students do is to read and modify existing projects. There they deal with the modification or implementation of single classes. Starting a new project from scratch involves creating a new class structure – an exercise that is done later in the course.) The Blue interface ensures that no code can be written before a class has been created. The environment structure strongly encourages the creation of the project structure before writing lower level code.

Blue distinguishes two kinds of relationships in its display: inheritance and "uses" relations. Inheritance relationships are represented by a double arrow while uses relationships are represented by a single arrow. The arrows also differ in layout and colour to be clearly distinguishable on screen.

We have for a long time, in teaching well structured programming, told students to think about program structure first, before starting to write low-level code. This is true for structured programming approaches just as well as for object-oriented ones.

Students tend to need a lot of convincing to take this design issue seriously. Since all that is done on the computer itself is the typing in of the code, this is seen as the "real" programming task. Creating a design in advance is frequently seen as a nuisance that they are forced to do *before* the programming work is done (instead of regarding it as *part of* the programming task). Consequently, students often do not take the design phase seriously enough and spend too little effort thinking about it.

The Blue system, by incorporating the class structure into the machine manipulated part of the programming task, has the effect that students take the creation of the structure more seriously. The interface almost forces them to think about class structure before they can start writing code.

The layout of the class diagram on screen is semi-automatic. The position of the classes can be manipulated manually (by dragging the icon) and the arrows are layed out automatically. The layout algorithm attempts to create an aesthetically pleasing diagram, for example by trying to avoid unnecessary crossing of arrows. This approach has proven to be very successful. Classes can usually easily be arranged in a way that provides a clear and well layed out project overview.

2.2 Design notation

As we have seen, Blue uses a very simple design notation. The structure display shows classes, their names, inheritance relationships and uses relationships. This diagram is considerably simpler than notations commonly used for professional software engineering, such as Booch Diagrams, OMT or UML Class Diagrams. Those notations record more information about both the classes and their relationships.

We consider a more detailed notation as inappropriate for introductory teaching. It is undoubtedly valuable for professional software development, but poses too much of an overhead and distraction from more fundamental issues in an introductory course.

Much simpler systems have been developed to support teaching and learning of object-oriented concepts on a more informal level. The best known is the use of *CRC cards* [4]. CRC stands for "Class, Responsibility, Collaboration". The CRC card method uses index cards (one per class) to record each class's name, its tasks (responsibilities) and other classes with which it cooperates (collaborations). The class cards can then be arranged on a table to form a class diagram.

The Blue diagram is more closely related to CRC cards than to the professional design notations. It shows (as do CRC cards) the class name and the collaborators. It allows a manual layout that provides additional cues through layout options such as order and grouping (e.g. classes can be shown close together to indicate a close relationship). It does not, however, show responsibilities at the diagram level.

The Blue diagram is not intended to replace CRC cards (or equivalent design tools), but rather to complement them and provide a link between design and the coding stage. We have had very positive experiences with the use of CRC cards for teaching object-orientation to beginners. One of the important aspects of CRC cards is that it is a method that does *not* use a computer. The direct interaction and possibilities of simultaneous manipulation by several people, which results from their real physical existence (as opposed to a representation on a computer screen), is very valuable. The Blue diagram provides a simple tool to record the work done with CRC cards and continue the development process in a seamless way.

3 EDITING

The editor is fully integrated into the Blue environment. It does not appear as a separate tool, but rather as a function of each class: the class can be *opened*. It is, in fact, more than an editor in the strict sense of the word, since it provides functionality beyond that of text editing, such as displaying the interface of a class. It can better be thought of as a viewer of class details (with the ability to edit some of the details).

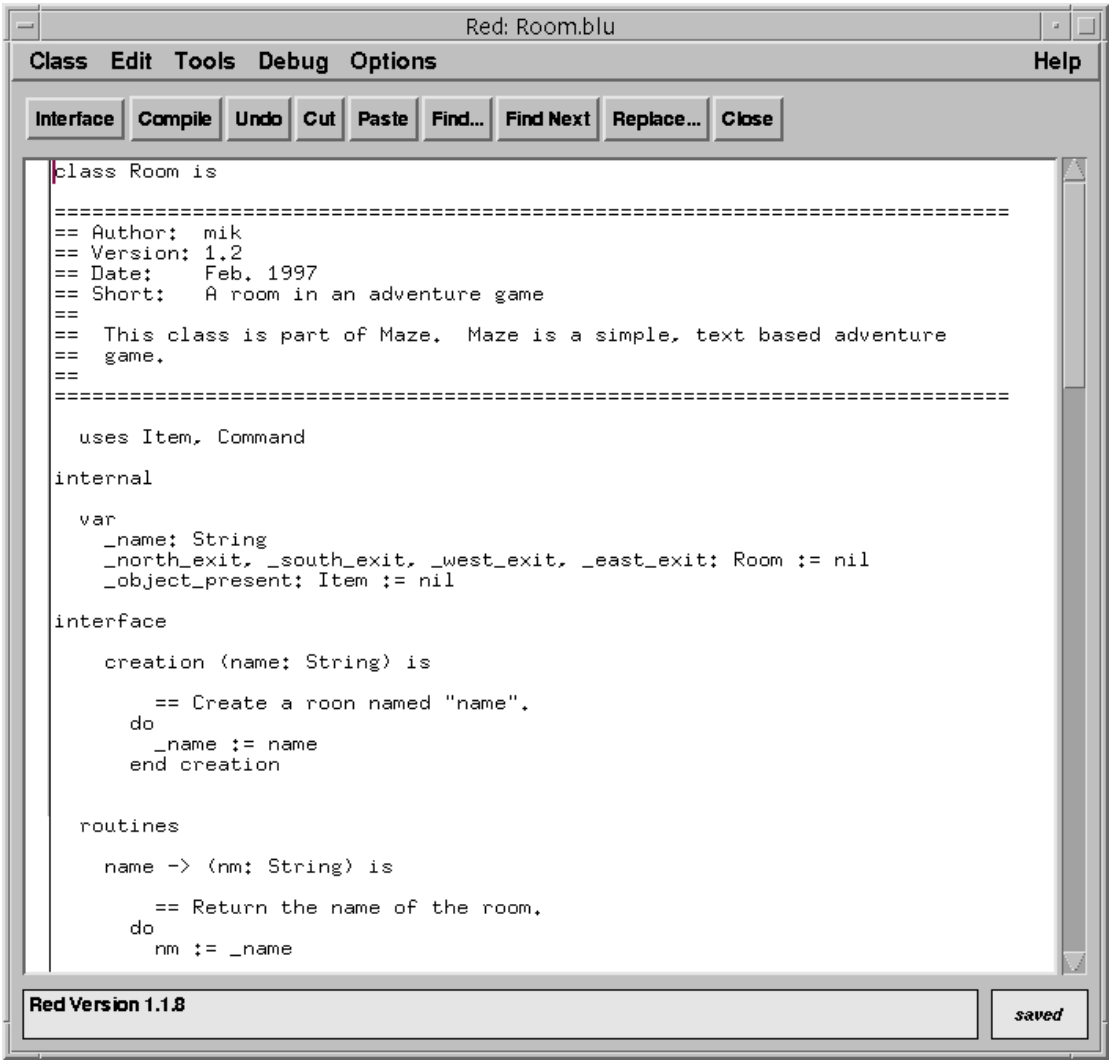

Figure 2: Implementation view of a class

From the project window, a *show* operation can be invoked on a class by double-clicking the class icon or clicking the "Show" button. This operation will open a window displaying class details – either the source or the interface of each class. The two views are usually referred to as the *implementation view* and the *interface view* (Figure 2 and Figure 3). Which view a user wants to see depends on the context: during development or maintenance of a class, the implementation view is used. After the implementation of a class is finished (e.g. while a client class is being developed) only the interface view is needed.

The interface view not only includes the routine headers, but also pre and post conditions and interface comments. Thus, the interface provided by the interface

view does not only provide syntactical information, but serves as full documentation of the class.

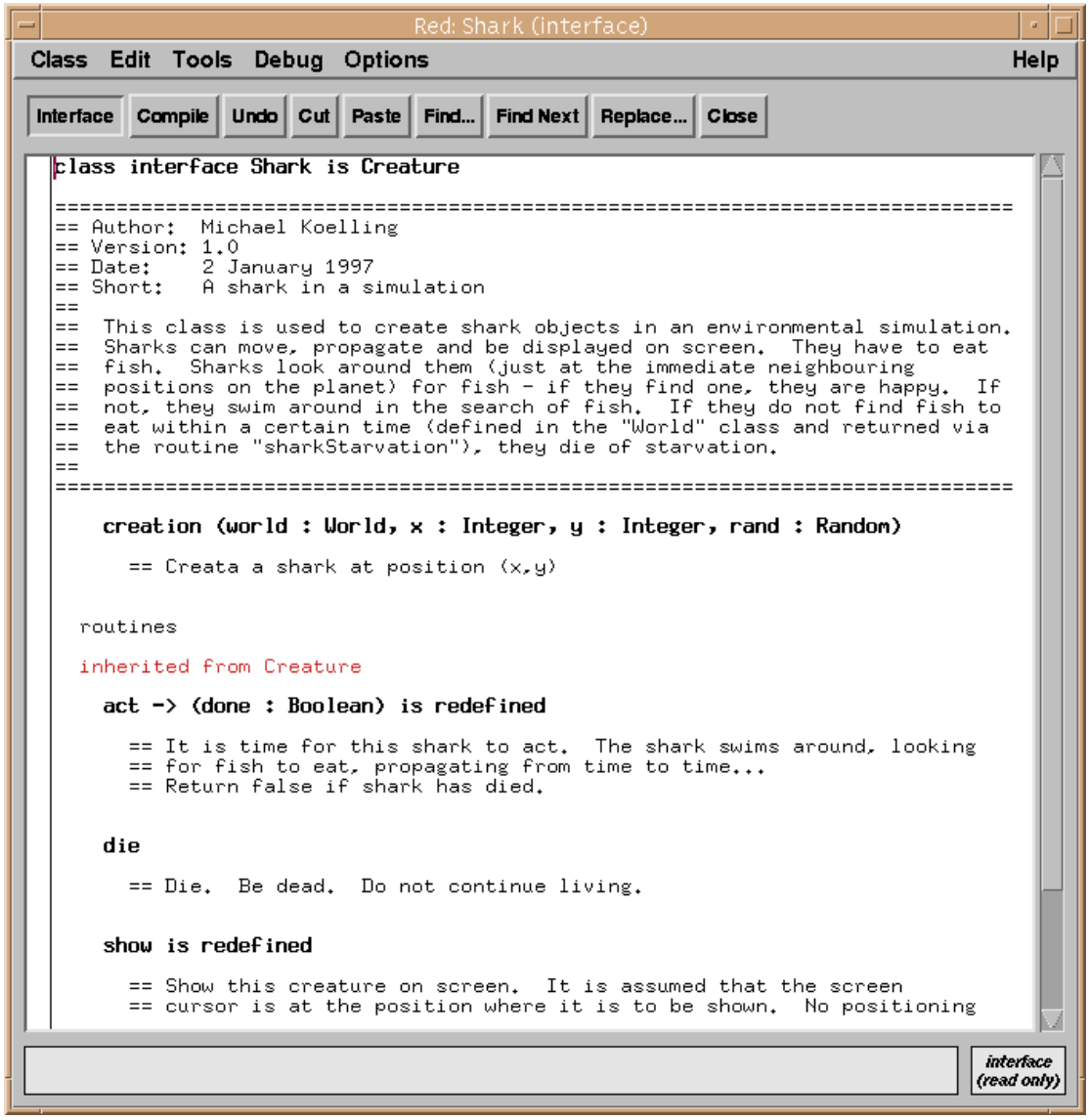

Figure 3: Interface view of a class

The editor is tightly integrated with the project manager and the compiler. When a new class is created in the project window, a code skeleton is automatically created. The initial source is a valid (but empty) Blue class. This ensures that a project can immediately be compiled after classes have been added (so that the user can work on one class at a time, without having to open all classes created in a project before the first full compilation).

The automatically created skeleton together with the strict class structure of Blue make it very easy for students to recognise where different aspects of the source, such as variables and routines, have to be inserted.

The project can be edited graphically (by inserting arrows into the class structure diagram, as discussed above) or textually in the editor. Both views are kept consistent at all times. If, for example, an inheritance arrow is inserted graphically, the source of the class is automatically updated. When the class is opened, the inheritance definition is visible in textual form. (If the class was open at the time of

the insertion of the arrow, the source code is immediately updated on the screen.) The reverse case also works: when a definition of a uses or inheritance relationship is added to the source of a class, the project diagram will be updated as soon as the class is saved. Closing a class view automatically saves the source, so the standard sequence of opening a class, modifying its definition and then closing it leaves a diagram that always reflects the current relationships. To achieve this effect, the editor cooperates with the compiler. After saving the class's source, it calls the compiler to perform a superficial parsing of the class to extract the information about its superclass and used classes. The compiler then reports the result to the project manager, which updates the graphical project representation. This analysis is, even for large classes, fast enough that it is not noticeable in normal interactive operation.

4 COMPILING

4.1 Invoking the compiler

The compiler, just as the other environment tools, does not appear as a separate tool, but is integrated into the project interface. We have already mentioned the "Compile" button in the project window: it performs a sophisticated dependency analysis and compilation sequence, possibly involving several compilation phases to deal with circular dependencies. An operation to compile individual classes is also provided in the project window.

The compiler also cooperates with the editor. It can be invoked from within the editor, and it uses the editor for the display of error messages.

Compilation is initiated from within the editor by clicking on a "Compile" button in the editor toolbar or by a keyboard shortcut to invoke the same function. The compile button in the editor has a different functionality from the "Compile" button in the project window – it compiles only the class currently being viewed. This is consistent with the object-oriented view of the world: "Compile" in the project window invokes the compile operation on the project, whereas "Compile" in the class window invokes the class compile operation. The tools themselves can be viewed as objects which provide operations to be performed on them.

Once a class has been compiled, its appearance in the project diagram changes. Compiled classes appear solid, whereas uncompiled classes are striped. (The classes in Figure 1 have all been compiled.)

4.2 Display of error messages

When the compiler detects an error in a class, it displays the class source, highlights the region of the error and displays an error message in the information area of the editor (Figure 4). Displaying the class source may involve opening the class, bringing its window to the front, de-iconifying its window or switching from interface to implementation view. No file name or line number information needs to be handled by the user – the environment automatically processes this information and finds and displays the relevant code fragment. The user can then immediately edit the class and remove the error. The error messages have been carefully worded to be comprehensible to beginners. Jargon is avoided as much as possible, and the messages attempt to give precise information as to the source of the problem.

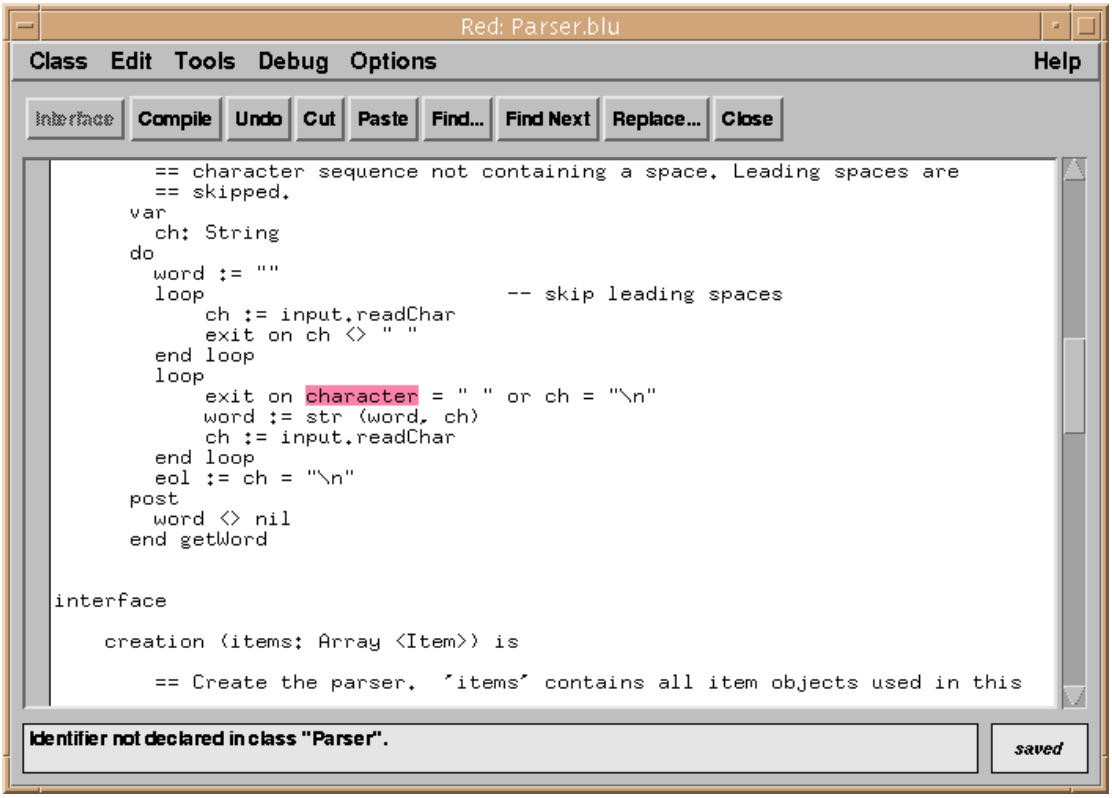

Figure 4: An error message reported by the compiler

Good error messages make a big difference in the useability of a system for beginners. Often the wording of a message alone can make all the difference between a student being unable to solve a problem without help from someone else and a student being able to quickly understand and remove a small error. The first student might be delayed for hours or days if help is not immediately available (and even in a class with a tutor it may take several minutes for the tutor to be able to provide the needed help). Enabling students to understand errors by themselves avoids a lot of frustration and frees the tutor's time for more important tasks. This can be achieved to a large degree through the use of carefully worded messages and is a goal well worth the investment of significant effort.

The quality of messages that a compiler can produce is significantly influenced by the grammar of the language itself and the compiler technology used. In languages like C++, so many ambiguities exist that it is impossible to avoid producing very general or misleading messages. Blue has been carefully designed to provide a type of grammar and a degree of redundancy within the grammar that enables the generation of good error messages in most cases. The grammar has been designed to be LL(k) (with k=2 in the case of Blue). This enabled us to use a recursive descent compiler – a compilation technique that lends itself more easily to the production of good error messages than LR compilers.

In case of an error during compilation, the Blue compiler displays only the first error message and aborts – no attempt at error recovery is made. In an earlier design, we discussed an alternative where the compiler attempts to detect all errors in a class. It would somehow mark the errors in the source, but only the first error can be initially

displayed (the others might not be on the same screen). The editor would then have a “Next Error” button that takes the user to the location of the next error.

In practice, that design has proved to be unnecessarily complex. Compilation in Blue is so fast (usually under a second for a class of 2000 lines) that the “Compile” button works as a de-facto “Next Error” button. After the first error is fixed, the user just clicks “Compile” again and the next error will be highlighted almost immediately. This has several advantages: the compiler is simpler (because no error recovery has to be attempted) and the error messages are more accurate (because incorrect messages caused by preceding errors are avoided).

5 INTERACTING WITH OBJECTS

5.1 Calling interface routines

As soon as a class within a project is compiled, objects of that class may be created. (A brief description of this interaction mechanism has been published earlier [5].) Interactively creating objects is done by selecting the class and clicking the “Create” button.

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. As an example, we assume that we have a class “Person” which stores some information about a person and provides interface routines to change and access that information.

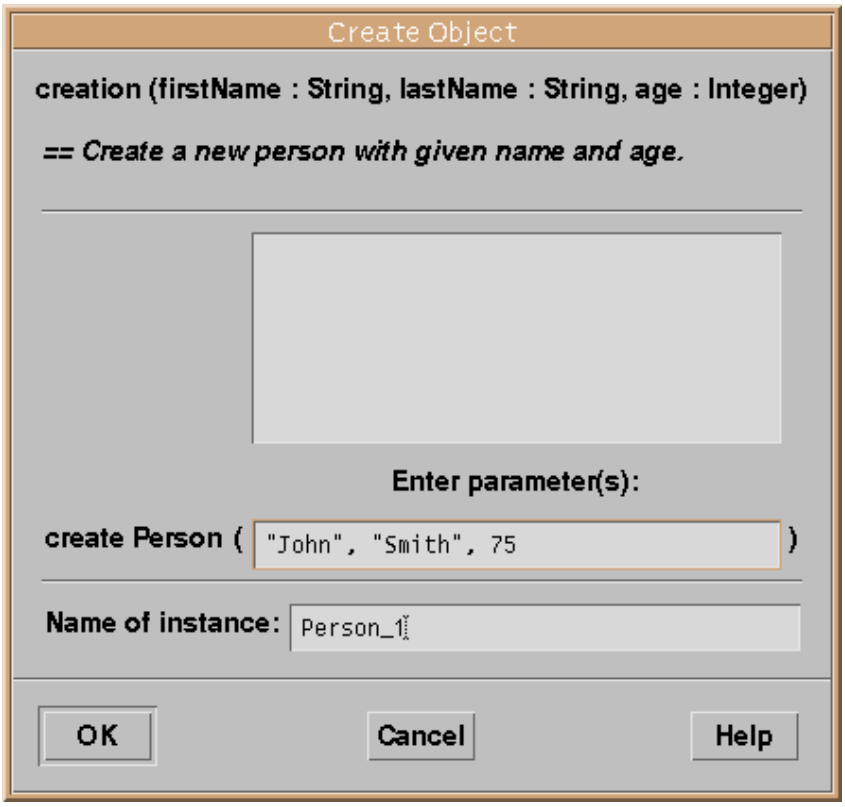

Figure 5: Object creation dialogue

When the create operation is invoked a dialogue is displayed to let the user enter routine parameters (Figure 5). 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.

Once the dialogue is filled in and the OK button is clicked, the object is created and displayed on the object bench (Figure 6). 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.

Figure 6: An object on the object bench

Clicking on the object with the right mouse button displays a menu that includes all interface routines of that object (Figure 7). 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 later.

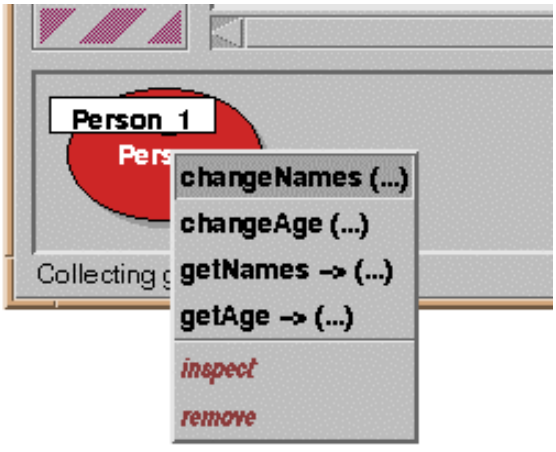

Figure 7: Calling a routine on an object

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 (without the field for entering the object name). 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 8 shows a function result dialogue for a call to the routine “getNames”. 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 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.

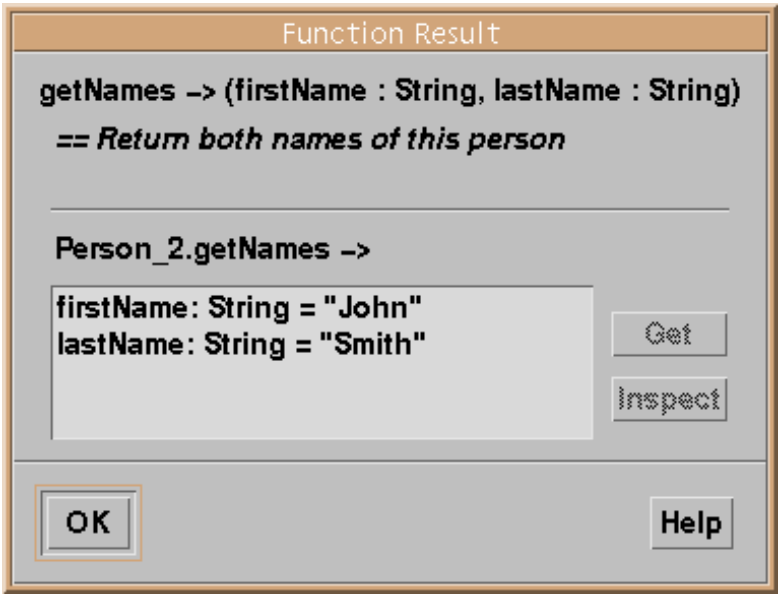

Figure 8: Result dialogue for function “getNames”

5.2 Linguistic Reflection

To execute an interactive call, the Blue environment uses linguistic reflection. A shell class is constructed internally that includes the interactive call as the 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. As part of the creation, it 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.

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. 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.

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.

5.3 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.

5.4 Inspection of objects

Sometimes objects contain data which is not directly accessible through interface routines. For this situation the *inspect* operation is provided. 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 9 shows the dialogue displayed as a result of inspecting a person object.

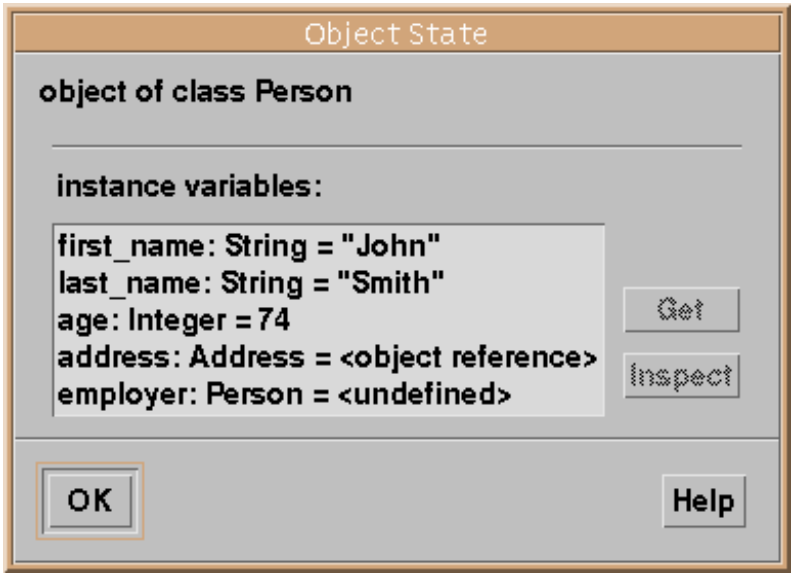

Figure 9: Object inspection dialogue

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. (The concept of a manifest object is part of the Blue language and has been described in last month's column [3]). 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.

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 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.

5.5 Execution without I/O

An interesting effect of the direct interaction and inspection mechanisms is that significant code can be written, executed and tested without the need to use conventional input/output constructs. Avoiding I/O statements for the first few weeks of instruction may well serve to convey a cleaner picture about concepts underlying the programming language [6].

5.6 Pedagogical benefits

The facilities described in this section – interactive object creation, interactive routine calls and object inspection – lead to a number of benefits for learning and teaching object-oriented programming:

- *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, 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 advantage is, in fact, not only relevant in a learning/teaching situation, but would be beneficial for software development in general.
- *Class/object distinction*. Students often have difficulty 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*. 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.
- *Testing support*. Good testing, essential to all serious software development, is supported much better than in conventional systems.
- *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.

Overall, the interaction facility provided by the object bench constitutes one of the most powerful and most valuable learning and teaching mechanisms in the Blue environment.

6 RUNTIME SUPPORT

The Blue system offers sophisticated runtime support for the detection of programming errors. Runtime errors cause the Blue application to gracefully terminate and the user is informed of the cause and location of the error. This is done in a similar fashion to the reporting of compilation errors: the source window is displayed, the region of the error is highlighted and an informative message is displayed in the information area of the source window.

Error detection and graceful termination is provided for all possible runtime errors. They include

- *pre condition or post condition violation*: A client fails to meet the pre condition or the server does not meet the post condition.
- *invariant violation*: The class invariant does not hold. This indicates an internal error in a class.
- *undefined variables*: The attempted use of an undefined variable is detected and reported.
- *undefined return values*: An error is reported if a function returns without assigning values to all of its return variables. (If the body of the function contains no assignment statement for the return variable at all, the error is detected at compile time.)
- *stack overflow*: Stack overflow, most commonly caused by infinite recursion, is detected and reported.

Some of these examples highlight one of the aspects of the advantage that explicit teaching systems can have over general purpose systems used in a teaching context: better error checking. The use of undefined variables, for example, is a very common error among first year students. Most systems, however, do not check at runtime for this error. The result is most commonly a system crash (which, depending on the operating system and environment, may even terminate the whole environment or operating system). In any case, the student does not get any information about the cause or the location of the error.

7 DEBUGGING

In keeping with the spirit of the tools we have discussed so far, the debugger is integrated with the other parts of the system. The goal, from the users' perspective, was to avoid the common student perception that the debugger is an additional, complicated tool. This would have hindered the acceptance of the debugger by users.

Today, in most first year courses, a debugger is not used. The reason usually is lack of time. Considerable time already has to be spent on becoming familiar with other tools (editor, compiler, file system, etc.) and teachers are reluctant to further reduce the time spent on programming concepts. The effect is that, although most teachers

agree that a debugger would be a useful tool in the students' effort to understand the concepts of programming, this tool is not being used.

Blue overcomes this problem by reducing the number of concepts and controls used by the debugger to keep it simple and by shifting some of the controls into tools that are already familiar. Breakpoints, for example, can be set directly from within the editor.

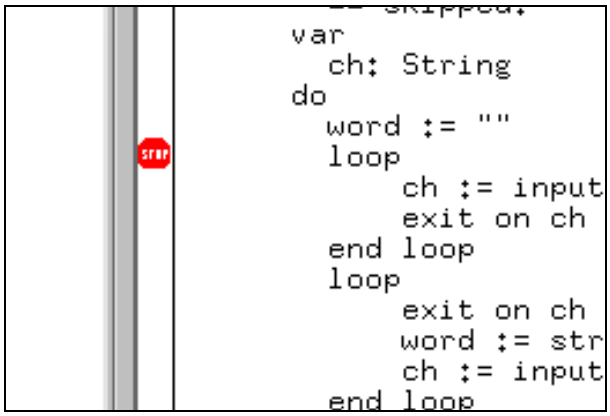

Figure 10: Setting a breakpoint

We have already seen other parts of the Blue system that support debugging: interactive object creation, direct interaction with and inspection of objects. (This is what we call "debugging without a debugger".) The result is that the debugger functionality can be reduced to three simple concepts to provide all the debugging means necessary for a teaching environment:

- *breakpoints*. The ability to set breakpoints at arbitrary places in the source code.
- *single stepping*. The ability to institute step-by-step execution to observe a program's control flow and state changes.
- *inspection*. The ability to inspect temporary variables (local variables and parameters) and the call sequence.

The first tool, a breakpoint, is easily used. A user can set a breakpoint by just clicking in the side bar next to the source line in the editor. A stop symbol appears to indicate the breakpoint (Figure 10). This is an example of using a debugging technique without the need for the introduction of a new tool.

When the breakpoint is reached during execution, it is handled in a similar fashion to runtime errors. The source window is displayed, the current line is highlighted, and a message is shown informing the user that a breakpoint has been reached. At the same time, an "Execution Control" window is opened. This window contains some simple buttons that enable the user to control the continuation of the execution, such as stepping, continuing or terminating the execution. It also contains a display of the call sequence (the stack) and the values of all variables in scope. No explicit "inspect" function is necessary to view variable values. They are automatically displayed and updated when their values change.

Overall, these three mechanisms, breakpoints, single stepping and variable inspection, provide, in conjunction with the object bench interactions, all the necessary

debugging facilities in a manner that is easy to use, easy to learn and easy to remember. Students typically have no problems, after being told or having read about these features, making effective use of them almost immediately.

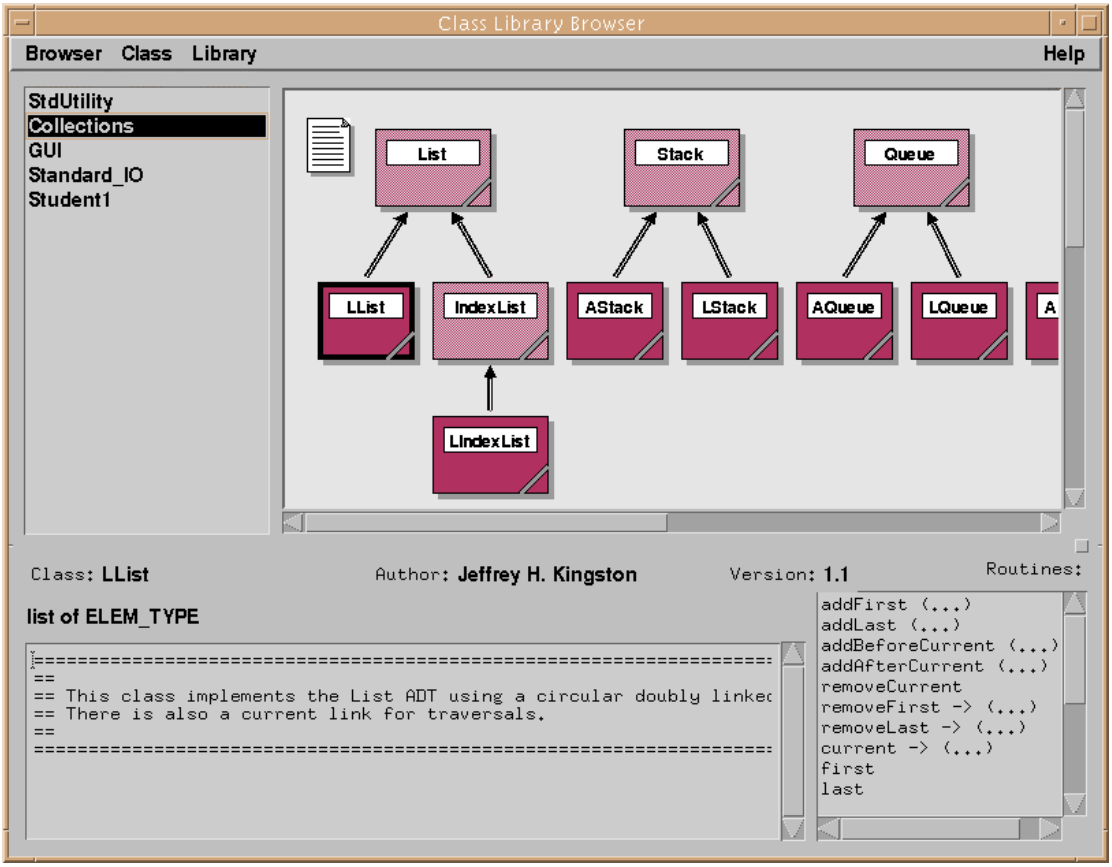

Figure 11: The Blue library browser

8 BROWSING CLASS LIBRARIES

Using class libraries is an important part of modern programming, and reuse of library classes is a fundamental software development technique. To be effective, a course based on an object-oriented language must integrate the use of libraries at an early stage in the curriculum [7]. It has frequently been observed that introducing the practice of reuse into existing organisations is very difficult. Many programmers rather reimplement classes than reuse existing ones. Three characteristics have been identified as prerequisites for the acceptance of reuse:

- the programming language must support a mechanism for reuse of existing code,
- the environment must ensure that the effort of finding an appropriate class to be reused is smaller than writing it again, and
- a culture must be established in which the programmer views the reuse of class libraries as an intrinsic part of normal programming.

The first point has been addressed by object-oriented languages. A substantial part of the hope that object-oriented languages may improve the quality of software

being developed has to do with the possibility of greater reuse. The other two points, however, were initially neglected.

The second point – finding classes – can be addressed with better tools. A good class browser is needed that allows a programmer to easily search and browse libraries of existing classes. The third point – reuse culture – is best addressed by teachers. If reuse is conveyed as normal part of programming from the very beginning, the difficulties associated with requiring an attitude shift later on are avoided.

For a teaching context this means that a good, easy to use browser is an important part of the environment.

The Blue browser can be opened by selecting the “Library Browser” command from the “Tools” menu. The main browser window is shown in Figure 11.

The browser uses a pane based design with three panes. At the top left a list of library sections is shown. Each section groups related classes together. On the top right a graphical representation of the currently selected section is shown in the same manner in which Blue projects are presented. In the bottom pane detailed information about the currently selected class is displayed. This includes the class comment and a list of interface functions. The bottom pane is updated each time the user clicks on a class icon to display the details of the selected class. The class icon may also be opened (with a double-click) in the same manner as classes in user projects are opened to display its full interface or (depending on permission) its implementation. Libraries can be explored by browsing the sections available or through a search function that lets users find classes that contain specific keywords.

When a suitable class is found, the programmer can select the class and invoke the “Use class in project” command from the “Class” menu. The class will then appear in the current project, identified as a library class by its colour. Library classes in a project can be opened, read, called and used like other classes in the project. They cannot be edited, though.

8.1 Documentation

A class library should not only provide the code of useful classes, but also its documentation. Users need to know the signatures and the semantics of library routines.

Many systems separate these issues and use two different tools to provide these two sides of the same coin. Borland C++, for instance, uses the standard C++ include mechanism for the inclusion of library classes and provides documentation for those libraries through a separate help system. This mechanism is error prone, duplicates effort and is not easily extensible. It is error prone because it can easily happen (and does regularly happen) that the implementation and the documentation get out of sync. Library classes might be changed or added, while the documentation is not updated. It duplicates effort because the creator of a library class is forced to write explanations twice: once in the class interface and once in the help system. (In the case of C++ it is, in fact, often written three times: in the implementation file as well.) And it is not extensible because it is not always equally easy to add library classes and help files. Often user classes may be added into the library collection for general inclusion, but help information cannot be added, effectively leaving the user-defined library classes without documentation.

Java tries to overcome some of these problems with a similar construct as Blue: interface comments. Certain comments in the Java classes are especially marked as interface documentation and can be extracted by a separate tool (called “Javadoc”). This creates documentation for a class. This mechanism avoids the duplication of effort: documentation is only written once (in the class source) and automatically extracted for outside documentation. It does not, however, solve the problem of outdated or missing documentation. It may still happen that a class’s code is updated while its documentation is not, leaving the user with incorrect information. It also does not guarantee the availability of documentation.

In Blue, the documentation is written only once in the class’s source code. It is then stored together with the class in its symbol table. The class’s code and its documentation cannot be separated. This guarantees the accuracy and availability of the documentation (as far as it has been written by the author). The class browser doubles as a help system that can display detailed information about interfaces and semantics of classes and routines.

8.2 The libraries

As with other parts of the programming environment, the requirements for libraries used for teaching differ from those used in professional software development. Tewari and Gitlin [7] state that “most discussions about the design of object-oriented libraries revolve around the need of professional programmers. However, we should not assume that the needs of students are identical with the needs of experienced software engineers. In fact, there are a number of areas in which the two differ.” The primary requirements for a teaching context are ease of use and clarity. This means that, as with the environment as a whole, the libraries used should be specifically designed for a teaching system. If a library is complicated and the student needs to study it for a long time before it can be effectively used, its value is greatly reduced. It will then detract from other concepts the student is investigating. Tewari and Gitlin [7] observe that many libraries are unsuitable for teaching because “many commercial vendors seem fixated on supplying every possible option imaginable. The extensive features add to the difficulty of using the library without necessarily adding to its pedagogical value.” The Blue libraries were designed to offer an interface that is easy to understand and to use.

9 CONCLUSION

While discussing many different aspects of the Blue environment, we tried to emphasise three ideas as the important principles underlying the Blue design: simplicity, visualisation and interaction.

Simplicity refers to the ease-of-use aspect of the environment: while it provides sophisticated functionality in many respects, it still presents an interface that is easy to use and understand for beginners. It avoids the cognitive overload often present in advanced programming environments. The interface is consistent so that users can, after having understood some basic principles, explore the environment on their own by experimentation and guessing. The combination of sophisticated functionality, clarity and simplicity presents a unique opportunity for use in a teaching situation and is not found in any other environment.

Visualisation is used to represent the application structure. This represents a shift from a pure coding environment to a software development environment that includes the design process. Visualising class structures and objects provide an invaluable help in teaching and learning about object-oriented principles. It helps students to understand classes, class relationships and class design. Making the structure visible encourages more conscious creation and discussion of designs.

Interaction refers to the ability to create and execute objects interactively. It is also represented in debugging tools that allow the interactive examination of object or machine state. Direct interaction is a powerful tool that helps in understanding of programming principles (e.g. the class–object relationship), debugging, testing, and many other aspects of the software development process. It helps students to more quickly obtain a deeper insight into the principles behind the software development task.

An implementation of the Blue system and more information is available at *www.csse.monash.edu.au/blue*. Currently, we are also developing a Blue-style environment for Java, named BlueJ. Information about BlueJ can be found at *www.csse.monash.edu.au/bluej*.

ACKNOWLEDGMENTS

Blue was designed and developed by Prof. John Rosenberg, Monash University, and the author.

REFERENCES

[1] M. Kölling, *The Problem of Teaching Object-Oriented Programming, Part 1: Languages*, Journal of Object-Oriented Programming, Vol. 11 No. 8, 8-15, 1999.

[2] M. Kölling, *The Problem of Teaching Object-Oriented Programming, Part 2: Environments*, Journal of Object-Oriented Programming, Vol. 11 No. 9, 6-12, 1999.

[3] M. Kölling, *The Blue Language*, Journal of Object-Oriented Programming, Vol. 12 No. 1, 10-17, 1999.

[4] K. Beck and W. Cunningham, *A Laboratory For Object-Oriented Thinking*, in OOPSLA '89 Conference Proceedings, ACM, New Orleans, Louisiana, 1-6, 1989.

[5] J. Rosenberg and M. Kölling, *Testing Object-Oriented Programs: Making it Simple*, in Proceedings of 28th SIGCSE Technical Symposium on Computer Science Education, ACM, San Jose, Calif., 77-81, February 1997.

[6] 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.

[7] R. Tewari and D. Gitlin, *On Object-Oriented Libraries in the Undergraduate Curriculum: Importance and Effectiveness*, in Proceedings of the 25th ACM SIGCSE Symposium, ACM, 319-323, 1994.