AN OBJECT-ORIENTED PROGRAM DEVELOPMENT ENVIRONMENT FOR THE FIRST PROGRAMMING COURSE

Michael Kölling and John Rosenberg
University of Sydney, Australia
{mik,johnr}@cs.usyd.edu.au

ABSTRACT

Over the last ten years there has been a major shift in programming language design from procedural languages to object-oriented languages. Most universities have adopted an object-oriented language for their first programming course. However, far less consideration has been given to the program development environment. In this paper we argue that the environment is possibly more important than the language and existing environments fail to fully support the object-oriented paradigm. We describe a new program development environment and show how it has been specifically designed to support object-oriented design and programming.

1 INTRODUCTION

Over the last ten years there has been a major shift in programming language design from procedural languages (e.g. Pascal, C), to object-oriented languages (e.g. C++, Eiffel [11], Smalltalk). The main reason for this shift is to capture the potential advantages offered by object-orientation. These include improved programmer productivity, an increased level of robustness and resilience, improved modelling of the real-world using objects and increased levels of code re-use [10]. Many tertiary institutions are now either teaching an object-oriented language in the first programming course or seriously considering a move to such a language.

Concurrent with the object-oriented development we have seen major improvements in programming development environments. Whereas early systems simply provided an editor and a compiler, modern programming environments provide facilities such as source code control, library management, support for group working, version control and integrated edit/compile/test/debug environments [3,14].

As object oriented languages have grown in popularity there have been attempts to bring together the two technologies [6]. In general the approach has been to adapt an existing software development environment to an object-oriented language. Such attempts have not managed to capture the advantages offered by object orientation.

Proceedings of 27th SIGCSE Technical Symposium on Computer Science Education, Philadelphia, Pennsylvania, U.S.A., SIGCSE Bulletin 28,1, March 1996, pp 83-87.

The most significant reason for this is that existing systems concentrate on abstractions that are appropriate for procedural languages. Consequently they provide support for development of procedural programs, management of source files, organisation of test data, etc. An object-oriented development environment should provide support for classes and objects as the fundamental abstraction. Attempting to use mechanisms designed for procedural program development to develop objects is not necessarily appropriate.

As a simple example, consider testing. A *procedural program* development environment will provide support for testing *procedural programs.* This would include setting up of input data, capture of output data and comparison of the actual output with some expected output data. An *object* development environment will provide an *object* test facility. This should allow interactive invocation of the interface routines of an object, a quite different paradigm than that required for procedural programs. Note the potential advantages of an object-oriented test environment which include support for incremental development and the removal of the need to write test programs.

It is our contention that the lack of truly object-oriented development environments has created major difficulties for teaching object-oriented technology. In particular, students have major conceptual difficulties and tend to write procedural programs in an object-oriented language. This is particularly prevalent when using languages such as C++ which have a downwards compatible procedural form (C). If we expect our students to fully embrace object-orientation then we must provide them with an appropriate program development environment.

In this paper we provide the motivation and background for the design and construction of a new object-oriented program development environment. The environment is a component of the Blue project, which includes the design of a new object-oriented programming language [8], development environment, debugging system, integrated editor, etc. All of the components of Blue are specifically designed for use in the first programming course at tertiary institutions. The effects of this orientation on the design of Blue are discussed in [7].

The paper is structured as follows. In section 2 we provide some background on programming languages and development environments. In section 3 we argue that it is essential that the program development environment is unified with the programming language paradigm. In the following section we briefly describe the Blue program development environment and report on the status of the project.

2 BACKGROUND

Since the beginnings of computer science education there has been great emphasis placed on the importance of the choice of the first programming language to teach. There is a strong feeling that this choice has long term effects on the quality of the graduates produced and their ability to produce reliable and well structured software.

Over the last ten years we have seen the almost universal adoption of the object-oriented programming paradigm. While this approach has been in use for a long time (since Simula was developed in the 1960s), its significance for the software engineering process was not generally recognised. Smalltalk, for a long time the only widely available object-oriented language, was seen by most people as applicable only to research projects without direct practical application and certainly inappropriate for introductory courses. This was mainly because of performance disadvantages of Smalltalk compared to other languages and its unfamiliar syntax. In the mid 1980s the development of other object-oriented languages and major improvements in compiler technology resulted in an increase in research and discussion about object-oriented concepts.

Object-oriented programming is viewed by many as the best answer so far to managing increasing complexity in software systems. Many of the new languages which have evolved in the last decade include object-oriented concepts. While some of these extend previously known languages such as C or Pascal to include support for objects, other languages such as Eiffel or Sather [13] have been developed with object orientation as the key goal. C++, the object based successor of C, has quickly become the most popular language in industry projects, replacing C. The sole reason for the popularity of C++ is the inclusion of object-oriented concepts in the language.

Most universities have either adopted an object-oriented programming language as the first language in their courses or are seriously considering such a move. However, we have seen reports from a wide variety of institutions which suggest that this approach is meeting with mixed success. It is our contention that one of the difficulties with teaching an object-oriented language is the lack of an appropriate program development environment [12]. Despite the large emphasis placed on the choice of a particular object-oriented language, there seems to be little discussion of the programming environment, which may well have more of an influence on the learning experience than the choice of language.

In systems in use in many universities today, such as Unix, program development has traditionally been based around a command line/textual environment, where a set of separate tools is provided to support the development process. These tools (typically an editor, a compiler, a debugger and a "make" facility) are based on concepts developed in the 1960s and have not changed much since their introduction. They are basically standalone tools and have only been slightly enhanced to cope with the increased complexity issues and new programming paradigms. "Make", for example, a tool intended to help manage the complex compilation process for large systems with source code that is spread over multiple files, often becomes a complexity problem itself.

This situation has been dramatically improved by the appearance of integrated graphical programming environments, which are most prevalent on personal computers. These environments are able to significantly reduce the management overhead in software development, integrating editors, compilers and debuggers into one coherent system, thus significantly reducing the complexity of the overall software development task. All tools are seen as part of the overall process to build an application and link smoothly together. A debugger, for instance, can use the editor to point to source lines corresponding to code currently executed.

3 UNIFYING OBJECT-ORIENTED LANGUAGES AND INTEGRATED DEVELOPMENT ENVIRONMENTS

The next logical step in improving software development tools is to unify the new language facilities with advanced development environments. Although Smalltalk was integrated into a sophisticated environment (relative to the time at which it appeared), more recent object-oriented languages seem to have initially overlooked this aspect of software development and have tended to focus on text based interfaces. Only relatively recently, when actual use of new object-oriented languages has become more widespread to a large variety of users and projects, has this deficiency been noticed. This has provided an impetus for the development of graphical, integrated environments for object-oriented languages. Several environments for the more popular languages, such as C++, Eiffel and Oberon, have now been produced.

Fundamental deficiencies exist in most current development environments for object-oriented languages. The main problem is that the particular requirements and potential of object-oriented systems have not been understood and utilised to assist the software development process.

The main shortcomings of existing environments can be devided into two groups: insufficient object support and insufficient structure visualisation and manipulation techniques. We will now explore each of these areas and give examples of improvements to object-oriented environments.

3.1 OBJECT SUPPORT

Traditional development environments facilitated the design and construction of programs. The basic entity was source code and their functionality revolved around the convenient manipulation of source code. The ultimate goal was to produce a program, an algorithm description with exactly one entry point that could only be built and executed after all its parts were (in some sense) completed. No notion of runtime objects existed within the environment, since those objects could not exist independently from an active execution of a program. All data available outside the execution was in the form of files. Consequently, the environment had only to deal with

source and data files.

When these environments were adapted to object-oriented languages, source files were replaced by class descriptions. Typically more source files exist in the object-oriented environment than in the procedural form. In addition, these files have more relationships with each other, e.g. usage and inheritance relations. Tools have been added to the environment to manage these class files and some of their relationships.

The general paradigm of the environment, however, has not changed. The environment is still used to build an application with exactly one entry point that can be compiled and executed only after all its parts have been completed. This is the program/procedure-oriented paradigm. *The object-oriented paradigm, used in the programming language, has not been adopted in the environment.*

The object-oriented paradigm is based on the idea that objects exist independently, and that operations can be executed on them. Consequently, a user in a true object-oriented development environment should be able to interactively create objects of any available class, manipulate these objects and call their interface routines. The composition of objects by the user must be possible [1,2,4].

This leads to the possibility of incremental application development. Any individual class (which would be a collection of routines in a procedural language) can be tested independently as soon as it is completed. Testing then becomes much more flexible than in procedural systems. In most current object-oriented environments, the objects have to be embedded in a non-object-oriented main program or script language to create and invoke the first object or objects. A direct call of object interfaces is not possible, since object instances are not supported by the environment.

In short: the programmer must fall back to the procedural paradigm to start and test a program.

Summarising, we can say that the two major tasks of an object-oriented environment are:

- to make classes the main mechanism of code structuring (which has been achieved in some environments), and
- to make objects the entities that are operated on (which has been neglected by most existing environments).

The only widely available programming environments fulfilling our demands are Smalltalk environments, which suffer from other problems, outlined below and in [7].

3.2 VISUALISATION SUPPORT

The second major shortcoming of object-oriented programming environments is the lack of object-oriented visualisation mechanisms. Graphical visualisation techniques (e.g. as in [9]) should be used to display relationships between classes and objects. For example, inheritance and usage relations as well as call structures could be shown. While thinking in terms of diagrams is common at the design level and some CASE tools provide some support for graphics to model program structures, this is not well represented in programming environments. Even though the relations between objects are the most important factor in the design of an object-oriented system, little support exists in development environments for their management with modern visualisation and manipulation techniques. This lack of support discourages the use of graphical representations by students.

Manipulation using the graphical or textual representation of a class should be possible interchangeably. That means, for instance, that it should be possible to edit graphically the inheritance relationships of the objects of an application and have the changes reflected in the source code of the classes. The same should work the other way around: if a class is specified as an ancestor in the source code of another class, this relationship should automatically be reflected in the graphical representation.

Most existing environments for object-oriented languages today lack all of these features. All command line based environments such as Unix obviously lack facilities for visualisation. Class and object relations, which are a fundamental part of the programming process, are not sufficiently visualised and poor modelling techniques for these are provided. Also, the integration of the tools involved in the development process is typically poor.

Graphical systems for most languages support only some of these requirements. Most graphical systems provide good tool integration, but lack support for object-oriented characteristics as described above. The most advanced professional object-oriented development environments, such as Visual C++ or Delphi, use graphical support only to build the user interface of an application, but neglect the internal structure of the program itself. The single most important facility lacking in all these environments is dynamic interaction with single objects and classes.

The only available system providing these facilities is the Smalltalk environment. Objects can be created and used interactively. (A good example is the Portia Smalltalk Environment [5].) Also, a browser exists to support use of the library of classes and objects. Smalltalk lacks facilities on another front: no visualisation facilities for class relations are available. The main problem with this lies in the Smalltalk language itself. Since it is not statically typed, it is not possible to extract usage relations from its source code. No indication exists before runtime as to call relationships between classes. Inheritance relationships as shown in the browser do not present the relationships of one application but rather the whole Smalltalk environment and so the browser is not used as an application modelling tool. It also does not use graphical representation techniques.

All recently developed object-oriented languages include strong static typing as one of their fundamental concepts. Language developers seem to agree that static typing is a valuable tool in modern software development. The lack of this makes the provision of some modelling support for Smalltalk impossible and seems to indicate that statically typed languages can provide better support for large scale development of correct software.

3.3 SUMMARY

Existing program development environments simply fail to fully grasp, support and facilitate the use of the object-oriented programming paradigm. It is essential that we provide a more appropriate environment if we wish our students to produce truly object-oriented programs and to capture the potential of object-orientation.

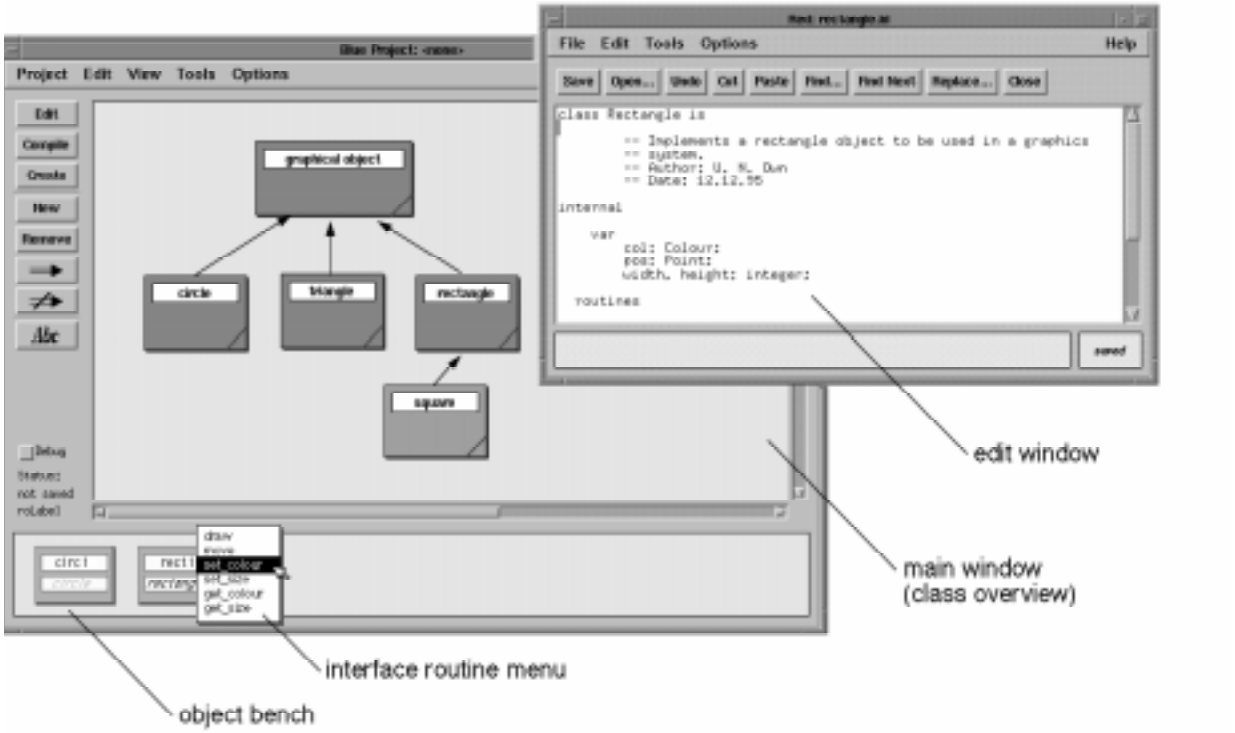

Figure 1: The Blue Programming Environment Desktop

4 THE BLUE DEVELOPMENT ENVIRONMENT

In this section we briefly describe the key features of the program development environment provided with the Blue programming system. As we have stated earlier, Blue is designed to be used in the first programming course. It is expected that students will move to a more conventional environment in later courses and to one of the popular object-oriented programming languages (e.g. C++). We are not suggesting that the ideas presented in the Blue programming environment are not appropriate for larger scale projects and languages, it is simply that this has not been our major emphasis.

The aim of Blue is to provide an environment which encourages the students to think in terms of objects. Specific details of the underlying operating system are hidden and a point and click world in which classes and objects are the fundamental concepts is presented. We assume that bit-mapped displays will be used at all times so that we can make extensive use of graphics.

Figure 1 illustrates the desktop presented to students when they log on to the system. The system has a notion of projects. A project is a group of objects which all relate to a particular application. This is simply an organisational facility somewhat analogous to directories. On entry to the system the student chooses the project on which they wish to work.

The large window at the left of the figure is initially displayed. Apart from the pull-down menu bar at the top of the window, it has three components. At the left is a set of push-buttons which activate frequently used operations. In addition there are buttons to allow the graphical definition of inheritance relationships between classes. As each class is created it is represented in the middle of the window by a box, with the name of the class at the top. The "classes" may be moved around the screen and the inheritance relationships 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 whether the class has been compiled and its category, e.g. abstract class, library class, etc.

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 object structure has been determined should they start to think about the interfaces and the 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 "edit" button). Space prohibits a full description of the editor (known as "red"), however, there

are a few significant features worth mentioning. Red is language sensitive and provides automatic formatting and indentation. Either the interface of the class, the full interface (including inherited routines) or the implementation may be viewed and edited. There is only one representation of the text internally and so it is impossible for these to get out of step with each other.

Red is not a toy editor. It may be used by students unfamiliar with editing techniques as a simple wysiwig editor using only the mouse and pull-down menus. However, it also has a powerful and configurable command set allowing more experienced users to perform more complex operations.

Compilation of classes or the whole project is achieved by a click on a single button in the project window. Compilation is based on routine units – only routines that have been changed since the last compilation are recompiled (unless instance variables were changed). This removes the need for students to become familiar with systems such as *make*, allowing them to concentrate on the concepts and techniques of object-oriented programming.

A unique feature of Blue is the ability to dynamically create instances of classes. When pressed, the "create" button prompts for the constructor parameters and creates an instance of the selected class. These instances are displayed in the bottom section of the main window, known as the *object bench.* An interface routine of an instance may be called by selecting it from a pop-up menu as shown at the bottom of figure 1. 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.

An integrated debugger is available that is presented to the user not as a separate program but rather as an alternate mode of execution. By setting the debug switch, a row of buttons is added to each edit window and this allows the specification of breakpoints, display and watch variables. It can also display current stack and heap contents (identifying objects and variables by their names and showing their links). The simple but powerful interface provided by the debugger should encourage the students to become familiar with these facilities at an early stage, so that as their programs become progressively more complex, they will be able to effectively debug them.

5 CONCLUSION

The object-oriented programming paradigm has been widely adopted and acclaimed by both industry and educational institutions. There has been considerable discussion concerning the choice of first year teaching language. However, the programming environment has received less attention. In this paper we have argued that this is equally important and that it is essential that the environment actively encourage and support the object-oriented design process.

Blue provides a programming environment specifically designed for building object-oriented applications. It provides a graphical view of the object design, an integrated editor, testing system and source level debugger. It has a simple point and click interface so that students can concentrate on understanding the object concepts and are not distracted by the underlying operating system.

The Blue system is currently being constructed. The language has been defined and the editor has been implemented. Work is continuing on the program development environment. It is expected that a working prototype will be available at the end of 1995. It will then be used with a trial group of students so that we can evaluate its effectiveness.

REFERENCES

1. Beck, K., Cunningham, W. "A Laboratory For Teaching Object-Oriented Thinking" in N. Meyrowitz (Ed.), OOPSLA '89 Conference Proceedings, ACM.
2. Booch, K. "Object-Oriented Development" in IEEE Transactions on Software Engineering, Vol SE 12, No. 2, pp. 211-221.
3. Clarke, L.A., Richardson, D.J., Zeil, S.J. "TEAM: A Support Environment for Testing, Evaluation, and Analysis" in SIGSOFT Software Engineering Notes, Vol. 13, No. 5, 1988, ACM.
4. Evered, M., Kölling, M., Schmolitzky, A. "A Flexible Object Invocation Language based on Object-Oriented Language Definition", The Computer Journal, 38,2, 1995.
5. Gold, E., Rosson, M.B. "Portia: An Instance-Centered Environment for Smalltalk" in OOPSLA Proceedings 1991, pp. 62-74, ACM.
6. Haarslev, V., Möller, R. "A Framework for Visualizing Object-Oriented Systems" in ECOOP/OOPSLA 1990, pp 237-244, ACM.
7. 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.
8. Kölling, M.. and Rosenberg, J. "Blue - A Language for Teaching Object-Oriented Programming, SIGCSE Technical Symposium, 1996.
9. McDonald, J.A., Stuetzle, W. "Painting multiple views of complex objects" in ECOOP/OOPSLA procedings 1990, ACM.
10. Meyer, B. "Object-Oriented Software Construction", Prentice-Hall, 1988.
11. Meyer, B. "Eiffel - The Language", Prentice Hall, 1992.
12. Notkin, D. "The Relationship Between Software Development Environments and the Software Process", Panel Session, in SIGSOFT Software Engineering Notes, Vol. 13, No. 5, 1988, ACM.
13. Omohundro, S.M. "The Sather Language", ICSI, 1991, part of the Sather system distribution.
14. Rodden, T. "Interacting with an active, integrated environment" in SIGSOFT Software Engineering Notes, Vol. 13, No. 5, 1988, ACM.