The BlueJ system and its pedagogy[1]

Michael Kölling[†], Bruce Quig, Andrew Patterson, John Rosenberg

[†]Mærsk Institute, University of Southern Denmark
[†]mik@mip.sdu.dk

Faculty of Information Technology, Monash University
{ajp, bquig, johnr}@infotech.monash.edu.au

Abstract

Many teachers experience serious problems when teaching object orientation to beginners or professionals. Many of these problems could be overcome or reduced through the use of more appropriate tools. In this paper, we introduce BlueJ, an integrated development environment designed for teaching object orientation, and discuss how the use of this tool can change the approach to teaching.

1 Introduction

Teaching software engineering with an object-oriented language has become commonplace in universities in the last decade or so. Most courses have moved towards teaching object-orientation with some software engineering elements in their introductory programming course in the first year of study. We agree with these moves and will not argue the benefits of this approach anymore - we rather assume that the reader agrees or leave it to other papers to pick up this argument.

In this paper, we will discuss how such a course should be taught. It is a common observation that those teaching introductory object-oriented programming courses find this more difficult than they experienced with the teaching of procedural languages. Why is this?

Our hypothesis is that teaching object orientation is not intrinsically more complex, but that it is made more complicated by a profound lack of appropriate tools and pedagogical experience with this paradigm.

This paper will introduce BlueJ, an integrated development environment (IDE) specifically developed for teaching and learning object-oriented programming, and present a pedagogical approach developed to be used with a system such as BlueJ. We will not remain at an abstract, theoretical level, but will give concrete examples by presenting a sequence of assignments designed to support and exploit the pedagogical ideas and technical possibilities of the environment.

We start, however, by summarising briefly the problems we have found in other environments for object-oriented languages.

2 Shortcomings of traditional systems

This section provides a brief summary of what we see as the key criticism of existing development environments for object-oriented teaching. For a more detailed discussion, see (Kölling, 1999a).

The fundamental problems with most existing environments can be summarised in three key points:

1. The environment is not object-oriented.
2. The environment is too complex.
3. The environment focuses on user interfaces.

We discuss each of these in some more detail.

[1] *published in the Journal of Computer Science Education, Special Issue on Learning and Teaching Object Technology, Vol 13, No 4, Dec 2003.*

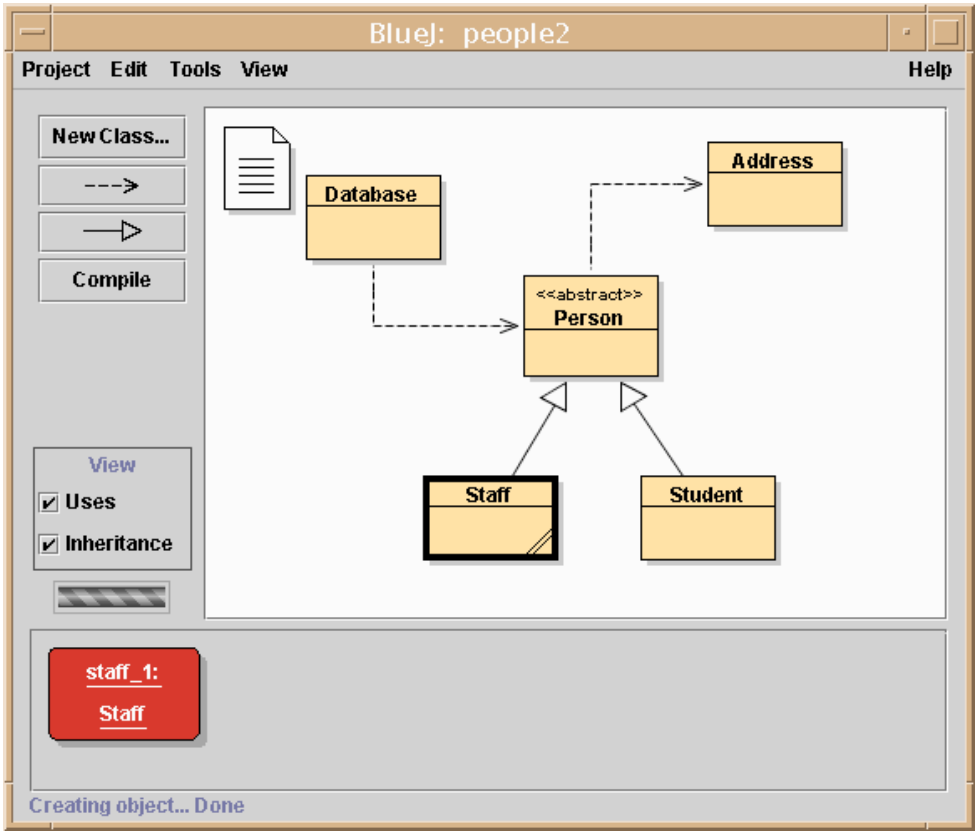

Figure 1. The BlueJ main window

2.1 The environment is not object-oriented.

An environment for an object-oriented language does not make an object-oriented environment. The environment itself should reflect the paradigm of the language. In particular, the abstractions students work with should be classes and objects. In most existing environments, students deal with files and an application instead. They are forced to think about the operating system's file system and directory structure. Setting up projects can be difficult. All this creates overheads that hinder teaching and distract from the important issues. When students work in such environments, their interaction is exclusively with lines of source code, not with objects. Consequently, they come to view programming as dealing with lines of code, rather than dealing with object structures. Objects as interaction entities are not commonly supported. Yet they are one of the most fundamental abstraction concepts.

2.2 The environment is too complex.

Many teachers do not use an integrated environment, because of problems with finding a suitable one. Students must work from a command line (using Sun's Java SDK) and spend considerable time becoming familiar with Unix or DOS instead of learning about programming. The result is a loss of valuable opportunities for improved teaching and learning through the use of better tools. The converse problem is that many other environments are developed for more professional users and present an over-whelming set of interface components and functionality. Students are lost in these environments, and the effect can be as bad as having no integrated environment at all. Other environments are really modifications of non-object-oriented (procedural) environments and offer the wrong set of tools and abstractions. Thus, the tools are either too minimalist, too complicated or inappropriate and cause considerable problems.

2.3 The environment focuses on user interfaces.

Many environments using graphics use the graphics for the wrong tasks. In particular, many environments concentrate on building graphical user interfaces (GUIs). Building GUIs from the start conveys a very distorted picture of programming and object-orientation. Students spend their time dragging buttons rather than thinking about building an application. In discussions about the value of IDEs for teaching, people often equate environments with GUI builders. This is a dangerous trap that should carefully be avoided when discussing IDEs. There are more useful tools for learning object orientation than GUI builders.

One of the most beneficial uses of graphics is often neglected: a display of class structure. Object-oriented program structure can be represented graphically in a way that makes it easier to understand and discuss design issues. Few existing environments make good use of this.

3 BlueJ

BlueJ is an integrated Java development environment specifically designed for introductory teaching. BlueJ is a full Java 2 environment: it is built on top of a standard Java SDK and thus uses a standard compiler and virtual machine. It presents, however, a unique front-end that offers a different interaction style than other environments.

BlueJ offers a unique mechanism of direct parameterised method calls. This mechanism allows teachers to delay the introduction of other interface technologies such as text based interfaces, GUIs or applets until a more appropriate point in the course.

The environment's interface facilitates the discussion of object-oriented design and aids in using a true "objects first" approach.

3.1 Overview

When a Java project is opened in BlueJ, the main window shows a Unified Modeling Language (UML) class diagram visualising the application structure (Figure 1). Users can then interact directly with classes and objects. A class icon can be used to execute a constructor, which results in an object being created and placed on the object bench at the bottom of the main window. Once the object has been created, any public method can be executed (this is discussed in more detail below).

A double-click on a class icon opens a text editor that lets users read or edit the class's source code. A simple click on a "Compile" button will recompile all changed classes and execution can start again. Compile-time errors are displayed directly in the editor by highlighting the corresponding line and showing the text of the error message.

The following sections discuss some of the most important aspects of the system in more detail.

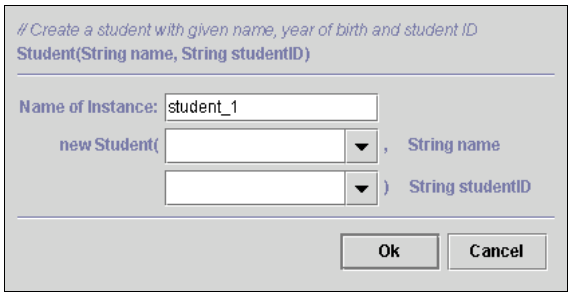

Figure 2. Object creation dialogue

3.2 Interaction

3.2.1 Creating objects

Clicking on a class icon with the right mouse button displays a class menu. This contains some environment operations (such as compiling, editing and removing the class) as well as entries to invoke the constructors of the class.

When a constructor is invoked, a dialogue is displayed prompting the user for a name for the object (a default name is supplied) and, if appropriate, the parameters. Figure 2 shows the dialogue for a constructor with two parameters.

Once the dialogue is confirmed, the constructor is executed and the resulting object is placed on the object bench.

3.2.2 Calling methods

A right-click on an object displays an object menu (Figure 3). The object menu contains two environment operations ("Inspect" and "Remove") and an entry for each public method defined in the object's class. Inherited

methods are placed in submenus. Selecting one of the methods results in that method being executed. If the method expects parameters, a dialogue similar to that shown for object creation is displayed to let the user specify the parameter values. Parameters can either be typed in (any valid Java expression is allowed) or other objects from the object bench can be chosen. Objects are specified as parameters by supplying their name (a simple click on the object is a shortcut to inserting the object's name into the parameter dialogue).

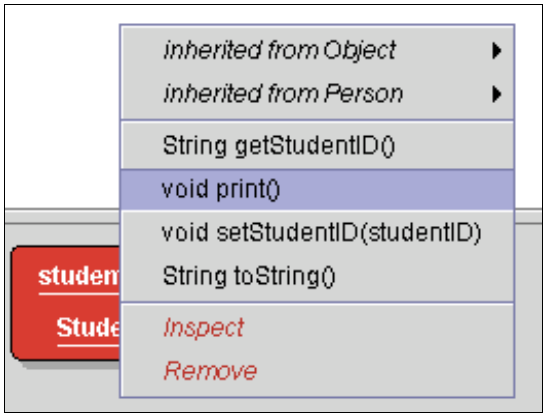

Figure 3. The object menu

If a method has a non-void return type, the result is displayed in a method result dialogue. If the result value itself is an object type, the object can be placed on the object bench for further interaction.

BlueJ also provides a mechanism to instantiate classes from the standard Java class library. Users can then interact with these objects in the same way they do with objects from their project. This allows students to explore and experiment with library classes and objects. They can, for instance, directly interact with string objects or hash tables to observe their behaviour.

3.2.3 Inspection

The interaction mechanisms allow very sophisticated and detailed testing of classes. Once a method has been written it can immediately be tested without the overhead of writing test drivers. Sometimes, however, a user wants to test a method that alters the state of the object, while no accessor methods are available to directly observe the state. For example, a constructor has just been written, and no other methods are implemented yet, but we would like to test the constructor before proceeding.

In this case, object inspection can be used to check the effect of the method. The "Inspect" operation from the object menu opens the object inspector, which displays the values of all static and instance fields of the object (Figure 4). Any fields that are themselves objects can be recursively inspected.

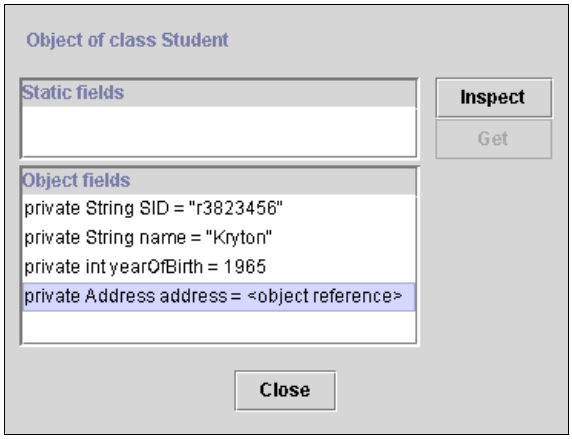

Figure 4. The object inspector

3.3 Visualisation

One of the central aspects of the BlueJ environment is the class structure display in its main window. This forces students to recognise and think about structure from the very first time they see a Java program. When showing students the very first example program, it becomes immediately clear that an application is a set of cooperating classes.

Traditionally, one of the hard-to-explain (but very important) issues is the difference between classes and objects, and their relationships. Using BlueJ, a teacher can interactively create multiple objects of a class and inspect and interact with every one of them. The relationship between classes and objects usually becomes clear very quickly. Without the need to talk much about it, students see that the class is used to create objects (as many objects as desired), and that the objects contain data. They also notice that the type of data in each object of the same class is the same, while the actual values are different.

It also becomes apparent that objects are manipulated by invoking operations on them (which they provide) that alter their state. Some operations return information about the state.

Thus, visualising the important abstraction entities of object orientation (classes and objects) and allowing direct interaction with each serves to illustrate the OO concepts in a powerful and easy-to-understand manner without the need for long, dry explanations.

3.4 Simplicity

The third cornerstone of the BlueJ architecture (besides interaction and visualisation) is simplicity. The major problem with many existing environments is their complexity. Most environments were designed primarily for professional programmers, and the complexity of their tools overwhelms beginners. Beginning students need different tools than professional software engineers. This issue has been discussed in detail in the context of the original Blue system (Kölling, 1999b) on which BlueJ is based.

BlueJ is designed specifically for beginners. The central aim is that we want to teach about OO programming, not about using a particular environment.

With BlueJ, students can start using the environment on their own almost immediately. After the first half hour of the first tutorial, we never talk about the environment again, and students are able to competently use it. We have traded some advanced functionality not needed in first year courses for ease-of-use, resulting in an environment not necessarily suitable for professional development, but much better suited to first year teaching.

3.5 Other BlueJ features

BlueJ includes a variety of other features, which we will not discuss in detail here. Some of the most important are an integrated, easy-to-use debugger, integrated support for *Javadoc* generation, sophisticated support for generating and executing applets and an *export* function that can create executable *jar* files.

The applet support includes automatic generation of an applet skeleton, automatic generation and loading of an HTML page and the ability to run the applet in web browsers and applet viewers.

Details can be found on the BlueJ web page (Kölling, 2001) and in the BlueJ documentation (available from that web page).

4 Pedagogy

BlueJ, through its unique functionality and interface, allows teachers to teach introductory courses differently than can be done without it. Standard programming examples from existing courses and textbooks can be used, and students will benefit from the greater level of interaction and the simplicity of the interface. To exploit the full potential of BlueJ, however, a course should be specifically designed for the functionality of BlueJ.

In an earlier paper (Kölling & Rosenberg, 2001) we have outlined the principal ideas behind the pedagogy for teaching with BlueJ. The main points were presented as a sequence of eight guidelines for developing programming assignments for BlueJ. They were:

Guideline 1: Objects first.

Guideline 2: Don't start with a blank screen.

Guideline 3: Read code.

Guideline 4: Use "large" projects.

Guideline 5: Don't start with "main".

Guideline 6: Don't use "Hello World".

Guideline 7: Show program structure.

Guideline 8: Be careful with the user interface.

In summary, these guidelines suggest an approach to teaching that starts by presenting to students reasonably large projects from the beginning. Students would then be expected to execute, read, modify and extend the projects (in that order). Writing completely new projects from scratch is seen as an advanced exercise. This approach contains elements described in (Linn & Clancy, 1992), which report benefits from the use of case studies in programming teaching. It is also related to the Applied Apprenticeship Approach (Astrachan & Reed, 1995), which encourages students to read, study, modify and extend existing programs. Some elements from problem based learning approaches (Barg et al., 2000) are also included.

In the following section, we present a sequence of assignment projects that implements these ideas.

5 An assignment sequence

The sequence of assignments presented here is designed to span two courses over two semesters. Source code for all assignments is available from a web site at http://www.mip.sdu.dk/~mik/code.

5.1 The first step: execution

The purpose of the first project is to convey a feeling of the basics to students. These are an impression of objects, classes and methods and of a program as a collection of interacting classes.

For this we use a project called "shapes". This project contains classes for creating circles, squares and triangles, which are represented on screen and can be moved, resized and changed in colour by interactively invoking methods on the separate shape objects.

Students interactively manipulate these objects to create a picture on screen. Note that the manipulation is done via interactive method calls, not by dragging picture objects as in some graphics programs.

In doing this, students practice creating objects, calling methods and passing parameters. They also get a first hint at types: integer and string types are used as parameters. In addition, we let students inspect objects (that is: view the values of the internal variables). This activity illustrates several important concepts:

Java applications consist of a collection of classes;

classes can be used to create objects;

many objects can be created from one class;

objects have operations (methods);

methods may have parameters and return values; and

objects have a state (fields with values that may change through method calls).

Students can experiment with objects and get a feel for these important concepts without being distracted and held back by Java syntax issues.

5.2 Writing Java: "picture"

The next step is to give students an impression of program source and Java syntax. In this activity, they start making small modifications to existing code. We use a project named "picture". "picture" is similar to the "shapes" project, but contains an additional "Picture" class that combines various shapes to draw a picture. Students look at the picture by creating a picture object and invoking its "draw" method. This draws the picture on screen.

We then get students to open the source and find and read the "draw" method. The picture, inside its draw method, creates a few rectangles, triangles and circles, changes their size, position and colours, and thus creates what looks like a simple block painting of a house with the sun in the sky.

Students immediately notice that the source code implements exactly what they have just done interactively in the "shapes" exercise. They can very quickly understand the meaning of the source and make modifications.

We start by giving them simple tasks to do, such as "Make the sun blue". We fairly quickly move to creating completely new pictures. Here, students write their first Java code, consisting of object creation, method calls and parameter passing.

Good students very quickly come up with quite sophisticated ideas. The shape objects, for example, have methods such as "slowMoveVertical(int)" and "slowMoveHorizontal(int)" to create an animation effect by moving them slowly across the screen. Students regularly start creating animated pictures in which the sun sets or a ball rolls over the screen.

5.3 Implementing methods: Calculator, Blocks and MIF

The next step is for students to extend existing classes by implementing methods or adding their own methods to an existing class. Here, the project typically consists of multiple classes, but all the work the student is expected to do is within one or two selected classes. All other classes are provided fully implemented. We use three assignments of this kind, the first one is a "calculator" project.

Similar projects have been used by other people. A good example is described in a paper by Reges (Reges, 2000).

The calculator project has a graphical user interface (completely implemented) and a "CalcEngine" class with method stubs, in which students implement the calculator logic. The CalcEngine class has fields to hold the internal values and methods that are invoked when a number or operator button is pressed. Implementations for all these methods are very simple. Loops, for instance, are not required. Students deal mainly with variables, assignments, simple operations (addition, subtraction), instance fields and return values. One of the main aims is for them to understand the interaction of different objects in order to make a program work.

There are two more assignments of this style, where students are expected to implement slightly more complex methods in an existing class. The first is called “blocks” and is a partial implementation of a Tetris-like game. Again, students are given several classes, including a graphical user interface, but are expected to modify only a single class.

The last example of this kind deals with image manipulation (named “imageviewer”). Images are represented in MIF - Monash Image Format - which is a simple two-dimensional array of bytes. Students implement a set of image operations, such as “brighter”, “darker”, “smooth”, “pixelise”, “flip” or “threshold”. This assignment is partly designed to practice loops and other control structures as well as arrays. On the other hand, this assignment also requires students not only to fill in bodies of existing methods, but also to add completely new method definitions.

This is the last assignment in semester 1. The following two projects are semester 2 assignments.

5.4 Adding classes: The World of Zuul

The next step is a project where students create complete classes (again as part of an existing project). Here, we have used a simple text based adventure game called "The World of Zuul" similar to that described in Adams (2002). A basic framework is given to students that implements different rooms, input of commands and movement through rooms.

Students are asked to invent a game scenario, add items to rooms, the ability for players to carry items (up to a certain weight), etc. The scope for challenge tasks is endless.

One crucial aspect is that some of the tasks clearly require the addition of new classes, the most obvious one being an “Item” class. Students also go through exercises reading and understanding the existing code. They have to make changes in most (but not all) of the classes, but they have to figure out themselves what they have to change and where.

This has been one of the most successful assignments, with surprisingly elaborate student submissions both in inventiveness of story telling and technical implementation.

5.5 The ultimate challenge: Do it all

The last step is a project where students work in groups and create a whole application from scratch. This time, only a brief problem description is given, and students have to go through the whole development process, including the class design (with a lot of guidance).

We have used a variety of continuous event simulations as projects. They included a supermarket checkout simulation, a traffic intersection with traffic lights, a lift simulation, emergency evacuation from buildings, a marine life simulation and others.

At this stage of the course, we don’t discuss small scale programming issues very much anymore. The low level code writing is assumed to be mastered by students, and the project serves as a practice ground for applying these skills. The really new and challenging issues at this stage are application design and group work.

Simulations are an ideal example for practicing object-oriented design, because almost all objects needed in the application have corresponding objects in the real world, and are very easy to recognise with fairly simple methods. We use the noun/verb analysis and CRC cards (Beck & Cunningham, 1989) for class discovery.

Small scale problems are usually solved by groups internally, while the lecturer and tutor concentrate on discussing analysis, design and group work issues. It is made very clear that the group work aspect is not a coincidental side issue, but one of the important study topics of this course. Well organised group work processes are expected to be set up and documented.

This is the first time students do design, but not the last. In the following year of study, there is a whole subject about analysis and design. We go through their first design with a lot of advice and attention to make sure that all groups arrive at a solution that is implementable within their given time.

This project is by far the longest of the assignment projects. Students are given eight weeks to complete the project, and the deliverables include a report and a demonstration.

6 Discussion

The projects and assignments presented here implement a sequence of activities that introduce the important concepts of object-orientation in a significantly different order than traditional programming courses. One of the unique aspects is a true "objects first" approach: students start seeing and interacting with objects as the very first thing, even before being confronted with Java syntax or source code.

"Objects early" approaches are quite popular in the introductory object-oriented teaching community, and most newer textbooks attempt to follow such an approach. The problem, however, that generally has to be overcome is that of the syntax required to arrive at the first objects. In traditional environments, Java syntax has to be dealt with first, before objects can be created. In addition to syntax, the required Java code exposes concepts such as the main method, array parameters, object creation, variable declaration and dot notation for method calls.

Beginning students typically struggle with this syntax and language constructs, with the result that they are already working hard on understanding details by the time they get to encounter the big concepts. For students, the big concepts – classes, methods, parameters, invocation – do not stand out very clearly. They are in danger of being lost among the detail.

The BlueJ environment, through its interaction facilities, allows reversal of the order of introduction. Interaction with objects can be presented first, leading to detailed discussions of the main concepts of object orientation, before the need to deal with source code. Students can interact with object as their first task.

From there on, students go through a sequence of progressively more complex activities. They are:

make small modifications to existing methods;

implement complete method bodies where method signatures are supplied;

add new methods to existing classes;

add new classes to existing projects; and

finally, create a complete project.

All of this work is done in the context of relatively "large" projects. Students get used to reading and modifying existing code from the very beginning. Many of these activities conform to an educational pattern called "Fill in the Blanks" (Bergin, 2000).

Without BlueJ, following such an approach is not as easily possible.

It would be interesting to test BlueJ and its capabilities in a real problem-based learning (PBL) course. While the above set of projects has some elements of PBL, the whole course is not run in a PBL mode. We suspect that BlueJ and PBL would complement each other in an ideal way.

7 Evaluating BlueJ

BlueJ was first used in a CS1 course at Monash University in 1999. Students taking this course were invited to participate in a series of surveys to allow us to evaluate the environment. A detailed summary of this evaluation was presented by (Hagan & Markham, 2000). Student perceptions were that the environment was helpful, particularly the object bench functionality and integration of the compiler error messages and source editor.

The majority of negative feedback related to product stability and difficulty of installation. Since that evaluation took place, the stability of both BlueJ and the underlying Java libraries upon which it depends have been greatly improved. The study also notes that the results of subject examinations and assignment interviews showed that students generally had a good grasp of object-oriented concepts. This was in contrast to their earlier experiences using C++.

There is continuing work on providing more insightful methods of evaluating development environments for teaching introductory programming (McIver, 2002). This study aims to develop a method for the empirical study of development environments, both comparatively and in isolation. The first planned stage of this work is the use of BlueJ in pilot trials. This will involve the use of an instrumented version of BlueJ with logging facilities in which user interaction is recorded.

8 Related Work

Several systems exist that provide similar functionality to parts of BlueJ. Class structure visualisation is provided by a number of object-oriented design tools, such as Rational Rose (*IBM Rational Rose,* 2003), and development environments, such as JBuilder (*JBuilder*, 2003). These systems, however, are aimed at professional developers and lack the ease-of-use needed to make them appropriate for introductory teaching. They also do no support interaction at an object level.

More recently, several systems were published that allow direct interaction with Java objects. Most notable among these are BeanShell (*BeanShell*, 2003) and DrJava (*DrJava*, 2003). Both of these are Java interpreters that allow interactive evaluation of a series of Java statements. Their interface resembles that of a Unix shell or a traditional Lisp read-evaluation loop.

The main difference between Java source interpreters and BlueJ is the level of conceptual abstraction provided by the user interface. The abstraction used for interaction in Java interpreters is lines of source code. The conceptual abstractions used in BlueJ are classes and objects, represented graphically.

We believe that the initial focus on higher level concepts benefits a deeper overall understanding of object-oriented programming. The early fixation on source code can distract from important issues and hide the bigger picture. We are, however, not aware of a formal study to confirm or reject these assumptions.

9 Potential Problem Areas

While our experience with BlueJ is overwhelmingly positive, there are several potential sources of problems that teachers should be aware of. They are occasional reluctance of students to leave BlueJ behind, organisation of the transition out of BlueJ, and the treatment of lower level language issues. We discuss each of these topics in more detail.

9.1 The Need To Leave BlueJ

BlueJ is intended as an introductory learning environment. Mastering the use of BlueJ has no value in itself – it is a tool for a purpose. A professional software engineer or computer scientist should be familiar with more professional development tools and be able to cope with minimal installations, such as command line environments and plain text editors for the purpose of developing programs. Thus, it is important that students learn to use professional tools before leaving the university.

In some students we observe a reluctance to change: students cling on to the use of BlueJ and use it at inappropriate levels. (Other students are only too happy to migrate to more powerful tools!)

The design goal for BlueJ was to support programming in the first year. The optimal exit point for BlueJ is not entirely clear, and can depend on a variety of factors. We feel, however, that BlueJ should not be used beyond the first year. We consider it essential that students mature out of BlueJ and are forced to gain experience with professional development tools afterwards. Second or third year courses should ensure, by setting appropriate requirements, that students make this step.

9.2 Transition To Other Environments

When students change to a different environment the second potential problem can arise: mastering the transition. Our experience shows that the mere fact that use of another environment is required, can leave some students with problems.

We have found it beneficial to explicitly address the transition to the next environment in discussions. Expecting students to transfer the concepts on their own, obvious as they might seem to the teacher, is not always successful. We have repeatedly observed an effect where students who were successful in the use of BlueJ have difficulty applying the same concepts in a more traditional environment.

Discussing the transition and the transfer of concepts does not take much time: a single one-hour lecture is usually enough. Such a proactive approach, however, seems to make a big difference.

9.3 Treatment Of 'Traditional' Material

We receive occasional feedback that students, while gaining a good understanding of object-oriented concepts, are weaker in traditional areas, such as data structures and algorithms, than students who learned without BlueJ.

We speculate that the reason for this is not intrinsic in the BlueJ environment, but in the way a BlueJ course may be structured. In our BlueJ-related publications (such as in this one), we often concentrate discussion on object concept issues. It is not our intention to suggest that these issues should replace more traditional programming skills, but rather that the environment facilitates a reordering of topics. There seems to be a danger, however, that teachers focus on object-oriented concepts to such an extent, that basic writing of algorithms is somewhat neglected.

It is a general problem that more and more concepts are introduced into modern introductory programming courses (such as group work, testing, GUIs, concurrency, design issues, etc.). This necessarily leads to a reordering that results in some traditional material being moved into another semester or being dropped altogether. This is an issue independent of the environment used. The point to note is that important skills – the competent use of control structures, algorithmic thinking, recursion, etc. – still need the same amount of attention they needed in previous course structures. The use of an objects-first-approach does not lead to students magically mastering these issues.

10 Recent Developments

As a result of the continued growth in BlueJ's user base we are able to gain valuable feedback from both course providers and end users. This BlueJ community has helped to shape the future directions the BlueJ development group plan to take with the further enhancement and refinement of BlueJ.

Two of the most notable recent additions to the BlueJ environment are the integrated support for regression testing using JUnit, and an extension interface.

10.1 Regression Testing With JUnit

BlueJ's object interaction facilities provide a very low cost entry to testing on an informal level, but lack support for more organised testing. There is a growing recognition of the value of the use of unit testing frameworks such as JUnit (*JUnit*, 2002).

JUnit is a small framework that allows organised regression testing through writing of test methods. It provides functionality to easily execute test banks, express assertions on the results and be notified of failing test cases.

The JUnit test framework has recently become very popular in the Java community as a tool for organising testing, partly because of its extensive use in the extreme programming (XP) process (Beck, 1999).

The latest version of BlueJ (version 1.3.0) includes an integrated implementation of JUnit to support the teaching of organised testing to students. The interaction mechanism described above supports ad-hoc testing in a convenient way, but is not suitable for more organised repetition of test cases. The integration of JUnit overcomes this problem.

The result is not only the sum of these two test mechanisms, but the creation of a new quality

of test tool through the combination of both: interactive test sessions can now be recorded and automatically saved as JUnit test methods to be replayed later for regression testing.

10.2 The Extension Mechanism

A continuous tension exists between requests for additional features being added to the BlueJ environment and our desire to keep BlueJ simple and small.

This tension has led to the development of an extension (sometimes called "plug-in") interface. Using this interface, third party developers can now write extensions to the BlueJ environment. Extensions have access to most of the BlueJ constructs and can be notified of user or environments actions via an event mechanism. The first extensions that have become available support automated project submission and code style checking.

11 Future Work

Areas which are currently being investigated are the support for group-work and scripting support. Both are likely to be created using the extension mechanism.

Group work support aims at integrating tools that allow groups of students to work on a common project. A possible model currently under investigation is an implementation based on CVS with a simplified user interface.

Scripting support would allow the creation of interactive tutorials with live links between the tutorial and a BlueJ instance. An HTML based tutorial could initiate actions in BlueJ and monitor user actions to dynamically adapt its sequence.

Both projects are now underway.

References

[1] Adams, R. (2002). *The Colossal Cave Adventure Page* [Website]. Retrieved September 2002 at http://www.rickadams.org/adventure/

[2] Astrachan, O., & Reed, D. (1995). *AAA and CS 1: The Applied Apprenticeship Approach to CS 1*. Paper presented at the 26th SIGCSE technical symposium on Computer science education, Nashville, Tennessee USA.

[3] Barg, M., Fekete, A., Greening, T., Hollands, O., Kay, J., & Kingston, J. (2000). Problem-based learning for foundation computer science courses. *Computer Science Education, 10*, 1-20.

[4] *BeanShell* (2003). *BeanShell - Lightweight Scripting for Java* [Website]. Retrieved June 2003 at http://www.beanshell.org/

[5] Beck, K., & Cunningham, W. (1989). *A Laboratory For Object-Oriented Thinking*. Paper presented at the OOPSLA, New Orleans, Louisiana USA.

[6] Beck, K. (1999) *eXtreme Programming eXplained*. Addison-Wesley.

[7] Bergin, J. (2000, July, 2000). *Fourteen Pedagogical Patterns for Teaching Computer Science.* Paper presented at the Proceedings of the Fifth European Conference on Pattern Languages of Programs (EuroPLop 2000), Irsee, Germany.

[8] *DrJava* (2003). *DrJava* [Website]. Retrieved June 2003 at http://drjava.sourceforge.net/

[9] Hagan, D., & Markham, S. (2000, December). *Teaching Java with the BlueJ Environment.* Paper presented at the Australian Society for Computers in Learning in Tertiary Education (ASCILITE 2000), Coffs Harbour, Australia.

[10] *IBM Rational Rose* (2003). *Visual Modeling With Rational Rose* [Website]. Retrieved June 2003 at http://www.rational.com/products/rose/index.jsp

[11] *JBuilder* (2003). *Borland Software Corporation - JBuilder* [Website]. Retrieved June 2003 at http://www.borland.com/jbuilder/

[12] *JUnit* (2002). *JUnit, Testing Resources for Extreme Programming* [Website]. Retrieved September 2002 at: http://www.junit.org

[13] Kölling, M. (1999a). The Problem of Teaching Object-Oriented Programming, Part 2: Environments. *Journal of Object-Oriented Programming, 11*(9), 6-12.

[14] Kölling, M. (1999b). Teaching Object Orientation with the Blue Environment. *Journal of Object-Oriented Programming, 12*(2), 14-23.

[15] Kölling, M. (2001). *BlueJ - Teaching Java* [Website]. Retrieved September 2001 at: http://www.bluej.org

[16] Kölling, M., & Rosenberg, J. (2001). *Guidelines for Teaching Object Orientation with Java*. Paper presented at the 6th conference on Innovation and Technology in

Computer Science Education (ITiCSE 2001), Canterbury, UK.

[17] Linn, M. C., & Clancy, M. J. (1992). The Case for Case Studies of Programming Problems. *Communications of the ACM, 35*(3), 121-132.

[18] McIver, L. (2002, 18-21 June). *Evaluating Languages and Environments for Novice Programmers*. Paper presented at the Fourteenth Annual Workshop of the Psychology of Programming Interest Group (PPIG 2002), Brunel University, Middlesex, UK.

[19] Reges, S. (2000, March 2000). *Conservatively Radical Java in CS1*. Paper presented at the SIGCSE 2000, Austin, Texas USA.