Using BlueJ to Introduce Programming

Michael Kölling

University of Kent, Canterbury, Kent CT2 7NF,
United Kingdom
mik@kent.ac.uk

Abstract. This chapter describes the BlueJ system. The discussion includes both the software tool aspects of BlueJ, as well as pedagogical approaches that should be considered when teaching with BlueJ. Pedagogical changes suggested go deeper than merely introducing a new software tool: they include changes to a more software-engineering-oriented course, removal of considerable chunks of traditional material, and introduction of new skills and approaches. We discuss experiences with using the system over eight years at various institutions, and discuss successes and failures as seen retrospectively today.

1 Introduction

In this chapter, we describe our experiences with the BlueJ system.

BlueJ is, at its core, an integrated development environment (IDE). In practice, it has grown to become much more. When we started our work on BlueJ, we did not realise what it would grow into. For us, BlueJ, as it stands now, is a software tool, a set of examples and resources, a framework of pedagogical principles and a distinct pedagogical approach to introductory programming.

At the start of the project we had the intention of creating a software tool, designed along some clear, explicitly formulated guidelines, which in turn were based on pedagogical principles. BlueJ, started in 1998, was a direct copy of its predecessor system, Blue (Kölling, 1999b), this time with Java as the supported language.

One goal was to design a system that allowed a different approach to teaching. Instead of wading through a sea of syntactic detail at the start to be able to make it to the solid ground of programming concepts, we wanted to create a visible exposition of those concepts upfront, to be able to deal with those first. BlueJ (and Blue before it) succeeded in doing that.

Our – retrospectively naïve – expectation was that we would publish this tool, and people would take it and start teaching their courses differently. This did not happen. We published BlueJ, and after some time it received a reasonable amount of visibility and attention. Some institutions started using it. At first, we were delighted to see our system being used, then we were horrified.

The revelation that things were not going as expected came when we looked more closely at what people were actually doing with BlueJ. What we found was that

teachers would use BlueJ, but would still introduce material in exactly the same order and in the same manner as they had always done.

BlueJ's main goals, in our mind, were three things:

- To make the environment truly object-oriented by representing objects and classes as first-class entities in the user interface. This visualisation was intended to create a mental emphasis on class and object relationships, instead of concentrating on the positioning of semicolons and parenthesis.
- To allow interaction with individual objects to encourage small-scale experimentation. We believed that frequent interaction with single objects and methods would almost automatically lead to a better understanding of the inner workings of any program in particular and Java in general.
- To simplify the user interface of the environment to a degree that presents minimal distraction from the principles of programming. While learning to deal with IDEs is a worthwhile goal for a whole curriculum, it should not be forced on students simultaneously with learning their first programming concepts. It is a question of ordering: We wanted to free the teaching from the environment overhead.

In observing teachers using the BlueJ IDE in the first years following its release, we found that many of them were using BlueJ exclusively for the third reason. They appreciated the simpler interface to the editor and compiler, but did little to exploit the potential of the visualisation or interaction mechanisms. To our mind, this was missing the majority of the benefit of the tool.

At that stage, we realised that it was not enough to give people a new tool and expect them to discover new teaching methodologies automatically and change their habits on their own. After some time, we became convinced that this was in fact the much harder part: Not designing the tool was the challenge, but figuring out how to use it to the best effect and how to change the pedagogical approach to teaching programming was where the real difficulty lay.

At that time, we got into a pattern of travelling around universities and conferences and giving workshops. We tried to tell people how we thought BlueJ should be used, and what our underlying ideas about teaching were when we designed it. The workshops were generally well received, but it was a slow process. The number of people we could reach in this way was limited. It became clear that we really needed a detailed written description of our approach to teaching – a textbook. In late 2002, this textbook was published, written with David Barnes (Barnes & Kölling, 2005).

In this chapter, we try to summarise what we have learnt from the whole BlueJ experience, until today. We describe the BlueJ IDE, as well as some of the pedagogical ideas of the project. We then discuss some experiences and later developments and end with some observations and thoughts about the future. This chapter expands on several previous papers, with different focus and differing amount of detail. Some prior papers describe the BlueJ IDE (Kölling *et al.*, 2003), and some discuss approaches to teaching with BlueJ (Kölling & Rosenberg, 2001).

2 Background

When we started designing BlueJ, many teachers expressed that they found teaching object orientation more difficult than teaching procedural programming. This is, in fact, still the case today, eight years later: many teachers still find it more difficult.

Our hypothesis has always been that teaching object orientation is not intrinsically more complex, but that it is made more complicated by a number of external factors. Two of these are a profound lack of appropriate tools and pedagogical experience with this paradigm.

To start our discussion, we summarise the problems we have found in other environments for object-oriented languages, as they were when we started working on this project. For a more detailed discussion, see (Kölling, 1999a). The observations presented here are not much different today. Some small progress has been made, and we will discuss the more recent developments towards the end of this chapter.

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 building user interfaces.

We discuss each of these in some more detail.

2.1 Object orientation

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 overhead that hinders teaching and distracts 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 Complexity

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 User interface building

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 be avoided carefully 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.

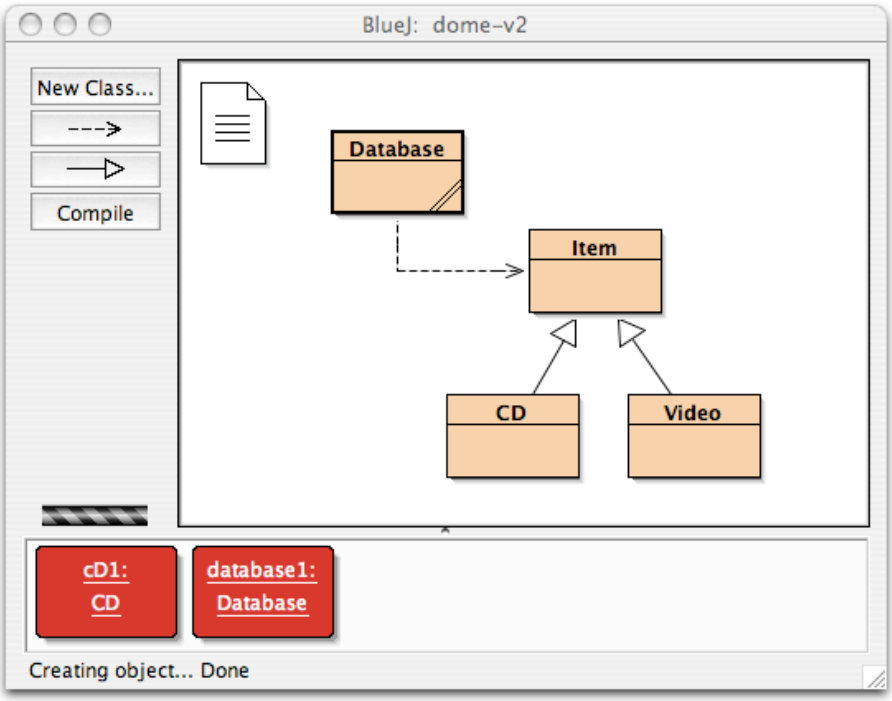

Fig. 1. The BlueJ main window

3 BlueJ

BlueJ is an integrated Java development environment specifically designed for introductory teaching. BlueJ is a full Java 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 The main window

When a Java project is opened in BlueJ, the main window shows a Unified Modelling Language (UML) class diagram visualising the application structure (Fig. 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.

Fig. 2. The class menu

3.2 Creating objects

Clicking on a class icon with the right mouse button displays a class menu (Fig. 2). 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. Fig. 3 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.

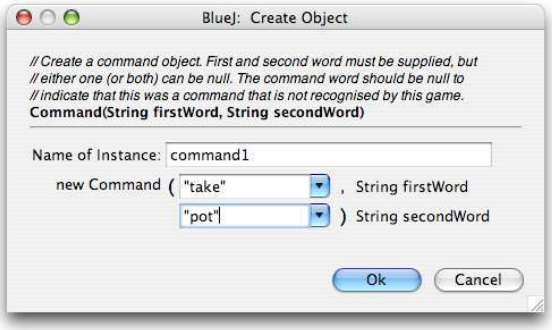

Fig. 3. A creation dialogue

3.3 Calling methods

A right-click on an object displays an object menu (Fig. 4). 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).

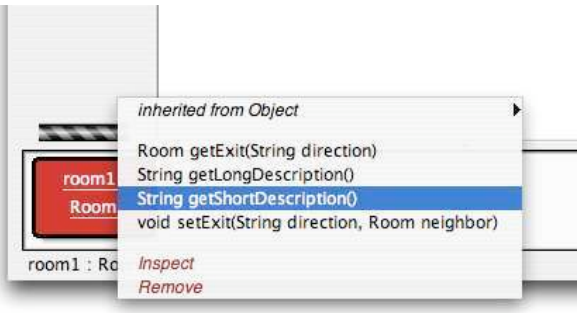

Fig. 4. The object menu

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.

Fig. 5. Object inspection

3.4 Inspection

The interaction mechanisms allow 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 instance fields of the object (Fig. 5). Any fields that are themselves objects can be recursively inspected.

3.5 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, 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.

3.6 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.

3.7 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).

BlueJ includes an integrated implementation of JUnit to support the teaching of organised testing to students (Kölling & Patterson, 2004). 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.

3.8 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 environment actions via an event mechanism. Examples of available extensions are a project submission system and a code style checker.

3.7 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 Pedagogical considerations

So far in this chapter, we have concentrated on presenting the technical details of the BlueJ environment. There is, of course, no single "right" or "wrong" way in which to use these tools. There are, however, known problems with introducing object-oriented programming to beginners. Many of these problems are related to mastering the inherent complexity of object orientation, Java and the software tools.

Maybe the greatest single problem with teaching Java is the large number of circular dependencies of language concepts and constructs (as discussed in the first chapter in this book section, *Transitioning to OOP/Java — A Never Ending Story*). It sometimes seems that to understand anything, you have to know everything. This characteristic makes especially the first few weeks of a course hard to deal with.

Over time, we have identified a set of five guidelines that help in designing a course in a way that avoids some of the most common problems. These were presented in an earlier paper (Kölling & Rosenberg, 2001). Some of the guidelines are independent of BlueJ, some assume a BlueJ-like environment. Here, we briefly summarise those guidelines.

Guideline 1: Objects first. Start talking about objects from the first day. Objects and classes are the fundamental concepts that students need to understand. Do not obscure this by talking about syntax first. Everything else can then be introduced in the context of classes and objects.

Guideline 2: Don't start with a blank screen. If students start writing projects from scratch, they have to structure the problem first. In effect, they have to do design. Design is hard and requires experience. This is not a task that students can master at the beginning.

Guideline 3: Read code. Reading code is an essential skill for any software developer. This skill can (and should!) be trained as any other programming related skill. It is the first thing students will be required to do when they enter a software development job in industry. Teach it explicitly. Students can learn a great deal from reading code.

Guideline 4: Use "large" projects. One of the major benefits of object orientation lies in the ability to structure problems into independent modules (classes). Beginners cannot understand the purpose and character of object orientation looking at single-class examples. Show multi-class examples from the start to convey the right ideas. Linn and Clancy describe a similar approach (Linn & Clancy, 1992).

Guideline 5: Show program structure. The internal structure of applications lies at the hart of understanding object-oriented programming. Spend a large share of time discussing this.

Some specific applications of these guidelines to particular areas are important enough that we formulate them as explicitly expressed corollaries to the general guidelines:

Corollary 1: Don't start with "main". The main method is the worst code example to study when trying to understand object orientation. It has nothing to do with objects – it is a quirk of history.

Corollary 2: Don't use "Hello World". Hello World teaches nothing about objects. If the problem really was to produce the words “Hello World” on the screen, an object-oriented language is the wrong tool. If the purpose is to simply illustrate the edit-compile-execute cycle, choose more reasonable examples to demonstrate this.

Corollary 3: Be careful with the user interface. Dealing with user input and output is a difficult issue. Text I/O can be complicated, GUIs are complex to program, applets are mysterious. BlueJ provides a mechanism that allows data input and output without explicitly programmed I/O code. Use this first, and then carefully introduce programmed I/O. Delay GUIs until students can fully understand them.

5 The project-driven curriculum

In this section, we present an example of a sequence of projects, which build on each other and drive the contents of the course. Other course activities, such as lectures or lab classes, are structured largely to support and enable the projects. In that sense, the introduction of class material is problem driven, and closely related to problem based learning approaches (Barg et al., 2000). The sequence of projects presented here is designed to span two courses over two semesters.

The sequence presented here 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.

True "objects early" approaches, even though popular in theory, are difficult to implement in practice. With most environments, the syntax that is required to arrive at the first objects, represents a real problem. 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.

The BlueJ environment, through its interaction facilities, allows to avoid this problem. 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. (The same idea has been discussed in the first chapter in this book section, *CS1: Getting Started*.) 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).

Another characteristic in which students are introduced to more advanced (and more realistic) concepts is the introduction of group work. Whereas the first few steps are done as individual work, the last step or two are carried out as group work projects. This adds the usual challenges of group coordination.

1 Getting your feet wet: executing code

The first project should be designed to achieve two things: to familiarise students with the environment, and to convey the basic concepts underlying object orientation. The abstractions students are expected to encounter are objects, classes and methods.

Our example for this stage is 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. 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).

Some of the most fundamental Java concepts are illustrated through this activity: that Java applications consist of a collection of classes; that classes can be used to create objects; that many objects can be created from one class; that objects have operations (methods); that methods may have parameters and return values; and that objects have a state (fields with values that may change through method calls).

Note that there is no need at any stage of this project to deal with Java syntax. We show the classes’ source code at the end of this project, but only to illustrate the existence of source code as an underlying construction mechanism. We do not expect students to understand any of it at this stage.

2 Manipulating source code

The purpose of the second project is to familiarise students with source code, Java syntax, to introduce the first few programming constructs and the edit-compile-run cycle. In our example sequence, we use a class that implements a simple ticket machine to achieve this. Students can investigate the behaviour of the class, and they quickly notice that the program has several bugs: in our example, the ticket machine sometimes collects too much money, and sometimes it prints tickets without proper payment.

When investigating the source code, students are able to map the methods they discovered through experimentation and observation with object instances to the relevant source code sections. We then analyse the existing code. Students start to understand structure and syntax, and we begin to introduce new constructs needed to improve the behaviour of the faulty class (conditional statements, in this case).

During this stage, students learn about basic Java syntax, variables, constructors and methods, assignment, the 'return' statement, simple arithmetic operations and 'if' statements.

3 Creating new behaviour – the next step

After fixing or adding code within an existing method, students are now ready for the next step: adding new methods. For this stage, we typically provide students with partly implemented projects, which students have to extend or complete in obvious ways. These extensions involve the addition of new methods, as well as modification of existing ones.

Over time, we have used a number of different projects for this stage. They include a calculator program similar to that described Reges (Reges, 2000), an image manipulation example, a simplified Tetris game, an Eliza-like dialogue system, an encryption/decryption system and an electronic auction system.

In a typical course, we do three of these projects in sequence, in increasing order of complexity. At this stage, the projects typically do not require students to write GUIs (they are either provided or the example is text-based). The number of language constructs required for the solution increases steadily.

4 Building blocks: adding more classes

The next step is a project where students create complete classes (again as part of an existing project). Here, we have often 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. A number of more interesting challenge tasks is set, such as implementing non-player characters, adding dialogue with game characters, adding random transporter rooms, various scoring systems, and many others. 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 in 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.

We have done this stage as individual tasks or as group tasks at various times. Both variants can work well, and the choice depends largely on the goals of the course.

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

5 The master test

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 (see also chapter 6).

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 Traps and pitfalls

There is no light without shadow, and all we do has to balance at the end. Thus, when we introduce something new, something old has to give way. It is no different when introducing objects-first teaching with BlueJ: we are introducing new methods and new content, and necessarily something else is lost. Our observations show that students in our course have a better grasp of object concepts and general software engineering issues, but are in danger of being weaker in other areas. Some of these weaknesses were expected and accepted, other were accidental and had to be addressed.

The planned trade-off involved treatment of data structure implementation and searching and sorting algorithms. This has traditionally been second semester material in many institutions. With the shift to a software engineering focus in the course using an object-first methodology, we have removed a significant part of this material from the first year. We now discuss competent use of data structures (as provided in the standard Java library) but not implementation of these. This topic has been moved to later courses in the curriculum.

Other problems we did not anticipate, but which we observed or were reported to us by other BlueJ teachers, include transition issues out of BlueJ, complete application development and treatment of low level coding issues. These are discussed below.

6.1 BlueJ dependency

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 rarely 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.

We include a segment towards the end of our first year that requires students to develop and run an application without BlueJ, using just an editor and command line tools. This is their first glimpse at the “other world” out there. This, we feel, is essential, but it is not sufficient for moving students on who have comfortably settled into their environment. Second or third year courses should ensure, by setting appropriate requirements, that students gain confidence with more professional tools.

6.2 Change is painful

Requiring a change to another environment does not guarantee a successful transition. Often curricula require use of different tools, but little explicit support is provided to master these. Our experience shows that merely requiring use of a professional environment 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 repeat-

edly 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.

6.3 Self sufficiency

In our course sequence, a large part of the activities consist of making modifications and extensions to existing code. The main driver classes of projects are often provided. Students experience the challenge of starting from scratch with a blank screen only towards the end. Developing a complete application is easily underrepresented in this curriculum.

The effect of this can be that students have difficulty to structure and implement a complete application from scratch. This is especially the case if the last activities in the course (where this is practiced) are done as group work.

It is possible to overcome this problem if it is consciously addressed. Regular exercises should be included early on, where students are required to develop small, but complete applications.

6.4 Control structures

Feedback from some teachers indicates that they are worried about a decrease in student ability in handling low level coding structures, such as control structures or parameter semantics.

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 lower level 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.

7 Inspiration and related systems

Inspiration for the direct interaction techniques in BlueJ came from early Smalltalk environments and the Monads system (Keedy & Rosenberg, 1990), both of which also allow direct interaction with objects. Neither, however, had a graphical representation of objects. Smalltalk's interaction usually assumed a multi-window GUI, but strangely still represented classes, objects and their relationships purely textually. Monads was a persistent object-oriented system that included its own hardware, operating system and programming language, with a purely text-based user interface.

Later Smalltalk versions, such as Squeak (Ingalls et al., 1997), have added extensive graphical visualisations, far beyond the functionality of BlueJ. Self (Ungar & Smith, 1987) is another integrated language and environment that provides much of BlueJ's functionality and much more sophisticated visualisations, with emphasis on different aspects.

A number of object-oriented design tools exist that provide class structure visualisation, such as Rational Rose (IBM Rational Rose, 2003), and 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 not 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, similar to BlueJ's code pad functionality. Their interface resembles that of a Unix shell or a traditional Lisp read-evaluate-print 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.

8 Reflections

Was the BlueJ experiment successful? Well, that depends on your criteria. We like to think so. But it's not all roses. Or, in other words: There's good news and there's bad news.

Do I think students have learned better using BlueJ? Yes. Do I think BlueJ has helped me in my teaching? Most certainly. Are our problems with teaching object orientation solved? Not at all.

Assessing the effect of BlueJ is a very difficult task. I am not aware of any serious study into the effects of BlueJ on learning that included meaningful indicators, such as a control group or properly set up evaluations. Setting this up is an inherently difficult task. If an experiment were run within an actual teaching situation with a control

group, where both groups were taught with different methodologies (one with BlueJ and one without), then difficult ethics problems arise. Even the suspicion on part of the teacher that one method may be better than the other would make it unacceptable to use the weaker method for a group of students. If, on the other hand, the test was done in voluntary, extra-curricular activities, the situation would probably be significantly different because of self selection of participant groups. The most realistic chance would probably be when teaching methods change in a single course between one year and the next, but setting up a meaningful experiment in this situation is also not easy. Firstly, one would have to run the first part of the experiment a year before the change to get the control group, which requires a level of planning that rarely exists in a typical institution. And secondly, it almost never happens that individual aspects of a course are changed separately. Usually, when a change is done, several aspects, such as programming language, paradigm, teaching approach, teacher, textbook, software tools, change simultaneously, so that meaningful results are hard to extract.

That said, there are some results that we can report. Firstly, there is adoption. The BlueJ software is now used at more than 800 universities, and the number of downloads is still rising every year. This indicates that user numbers are increasing, which in turn seems to indicate that current users are happy enough with the system to not switch away from it (at least not in larger number than new adoptions).

When BlueJ was first used at Monash University, Australia, in 1999, students taking this course were invited to participate in a series of surveys 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.

A study by Madden and Chambers in 2002 (Madden, 2002) surveying student attitudes to learning Java contained data showing that students found BlueJ significantly easier to use than a second environment that was used, JBuilder. (70% of students responded that BlueJ was easy to use, compared with 39% who found JBuilder easy to use.)

In addition to this, there is a lot of anecdotal feedback we receive from teachers, which is overwhelmingly positive.

On the other hand, there seems to be a backlash slowly developing in the computing education community of teachers in general reporting unexpected difficulties in teaching object orientation. While some are happy with their courses, others have concluded that object orientation is too complex in principle to be taught at introductory level (Astrachan, 2005).

Our conclusion at the moment is that the art of teaching object-oriented programming has a long way to go. We are firm believers in the benefits of doing this, and we believe that it is possible to succeed. But it is clear that the required change is much greater than just changing a software tool. New approaches to teaching are needed, and it will take time for those to mature in our profession.

References

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

[2] Astrachan, O., Bruce, K., Koffman, E., Kölling, M. & Reges, S. (2005). Resolved: objects early has failed. Proceedings of the 36th SIGCSE technical symposium on Computer science education, 451-452, St. Louis, Missouri, 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] Barnes, D. J., Kölling, M. (2005). Objects First with Java – A Practical Introduction Using BlueJ. 2nd ed. Pearson Education, Harlow, England.

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

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

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

[8] 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.

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

[10] 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.

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

[12] Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A. (1997). Back to the future: the story of Squeak, a practical Smalltalk written in itself. In Proceedings of the 12th ACM SIGPLAN OOPSLA conference, Atlanta, Georgia.

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

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

[15] Keedy, J.L. & Rosenberg, J. (1990). Support for Objects in the MONADS Architecture, Proc. International Workshop on Persistent Object Systems, pp. 392-405, Newcastle, Australia.

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

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

[18] 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.

[19] Kölling, M., Quig, B., Patterson, A., Rosenberg, J. (2003) The BlueJ system and its pedagogy. In Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, Vol 13, Nr 4.

[20] Kölling, M., & Patterson, A. (2004). Going interactive: Combining ad-hoc and regression testing. In The Fifth International Conference on Extreme Programming and Agile Processes in Software Engineering (XP 2004), 270-273, Garmisch-Partenkirchen, Germany.

[21] Linn, M. C., & Clancy, M. J. (1992). The Case for Case Studies of Programming Problems. Communications of the ACM, 35(3), 121-132.
[22] Madden, M. & Chambers, D. (2002). Evaluation of student attitudes to learning the Java language. Proceedings of the inaugural conference on the Principles and Practice of programming, 2002, Dublin, Ireland.
[23] 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.
[24] Reges, S. (2000, March 2000). Conservatively Radical Java in CS1. Paper presented at the SIGCSE 2000, Austin, Texas USA.
[25] Ungar, D., & Smith, R.B. (1987). Self: The Power of Simplicity. In *OOPSLA Conference Proceedings*, 227-241, Orlando, FL.