Apprentice-Based Learning

Michael Kölling and David J. Barnes

The Computing Laboratory, The University, Canterbury, Kent CT2 7NF,
United Kingdom
{m.kolling, d.j.barnes}@kent.ac.uk

Various methods have been proposed in the past to improve student learning by introducing new styles of working with assignments. These include problem-based learning, use of case studies and apprenticeship. In most courses, however, these proposals have not resulted in a widespread significant change of teaching methods. Most institutions still use a traditional lecture/lab class approach with a strong separation of tasks between them.

In this chapter we propose an approach to teaching introductory programming in Java that integrates assignments and lectures, using elements of all three approaches mentioned above. In addition, we show how the BlueJ interactive programming environment can be used to provide the type of support that has hitherto hindered the widespread take-up of these approaches. We arrive at a teaching method that is motivating, effective and relatively easy to put into practice.

1   Introduction

Most introductory computing courses follow a roughly similar organizational structure: a sequence of weekly lectures is complemented by laboratory classes. The lectures are used to introduce new material to the students, and the lab classes reinforce the material by requiring the students to work through and discuss small exercises and somewhat larger assignments. Intriguingly, despite numerous changes in programming languages, and shifts in programming paradigm over the last couple of decades, this delivery pattern has changed little and remains the predominant one in many institutions. New teachers tend to model their practices on those that they experienced as students, and textbooks typically reinforce the pattern of successive introduction of language features followed by small-scale exercises that use those features.

For teachers and students alike, appropriate practical work is the key to the success of the learning process. Small-scale exercises that focus on programming language details offer little scope for creativity on the part of students, and are unlikely to create a sense of ownership. In contrast, learner-centred tasks [14] that capture the interest of students are more likely to generate a sense of excitement and motivate further investigation. If programming projects with a real purpose and interesting

goals are offered, the results have the potential to be enlightening and rewarding for both students and teachers.

However, several problems exist in providing sufficiently motivational assignments:

In an introductory programming course, it is not easy to see how students can work on problems large enough to be truly interesting early in the course, as they have little experience with software development.

It is often hard to create an obvious connection between the lecture and the assignment. Both often exist as fairly separate activities, making it harder to create interest in and motivation for the lectures.

Programming environments are often either overly complex, incomplete in their language support, or do not provide good support for the teaching and learning processes, thus hindering active assignment work early in the course.

In this chapter, which extends the discussion in [11], we consider a technique that can be used to integrate assignments and lectures more tightly. This serves better to motivate lecture content, results in the ability to carry out more interesting assignments and allows inclusion of important software engineering concepts into an introductory course.

All of the techniques described here have been tested for several years in CS1 level university courses in Denmark and in England. Courses have ranged between 7.5 and 15 ECTS points, with student numbers up to about 200. Course durations have been between one and two semesters (12-24 weeks) with typically two one-hour lectures and a single one-hour class per week.

2   Previous Work

In the 1990s, several related tracks were followed in an attempt to find more effective ways of motivating and presenting material on introductory programming courses. Seminal among these attempts was the work of Linn and Clancy [13], who made a strong argument for the use of case studies to support program design. Particularly effective in their study was the use of expert commentary to accompany a design.

Also significant was the work of Astrachan and Reed [2, 3] whose 'applied apprenticeship approach' encouraged students to read, study, modify and extend programs written by experienced programmers. One of the principles of the apprentice-based approach is that it is particular *applications* that are the motivation for introducing new programming constructs or data structures, rather than studying language constructs as an end in themselves. This is significantly different from the typical language-feature driven approach. Providing a credible context for features, structures and algorithms makes it more likely that students learn language-independent skills in addition to a particular programming language.

A similar problem-driven motivation can be found in the use of problem-based learning environments [5, 7]. This approach also often features group work.

Modern students – and hence, necessarily, the introductory programming courses they take – are dependent upon the availability of suitable companion textbooks to

support them. Progress in this area has been slow. Despite the insights of the early 1990s offered by Astrachan et al, most modern introductory programming text books still tend to exhibit mainly traditional characteristics of language-construct driven chapters, and small example problems. Why is this?

One reason may be the arrival on the scene of Java in the late 1990s. This offered enormous benefits over many of the alternative procedural and object-oriented programming languages of the time, such as Pascal, C and C++. Increasingly, Java became the most common language chosen for introductory programming courses. A strong influence here was that the easy availability of GUI-based examples was seen as one of the main means to motivate students [16], particularly via the synergy that Java offered with the emerging World Wide Web through applets. Unfortunately, while these may well have motivational potential, there remains the question of whether – of itself – these offer enough to deliver the broader educational and software engineering requirements of an introductory programming course.

As a consequence, there is a good case for revisiting and extending this earlier work in order to support a high quality introduction to object-oriented programming.

3 Problems With Assignments

In most courses assignments are not set weekly, whereas lectures are held at least once a week. Therefore assignments tend to be somewhat removed from the lectures in both organization and content. Usually, a lecture introduces new programming constructs or techniques and, some time later, an assignment is given to practice application of these techniques. The larger (and with it, the more interesting) the assignment is, the more removed it would tend to be from the lecture content, since it would include material covering a longer period of time. Particularly in the early stages of a programming course, such a separation limits students' engagement with the practical task of programming and may be unhelpful.

Another problem with early programming assignments is that it is hard to get students to do things well. While students are struggling with getting their program to do something at all, they often have little time left for thinking about non-functional aspects, such as structural software quality.

The solution in many courses is to leave software quality aspects, such a maintainability, coupling and cohesion, to later courses, and concentrate on getting something running first.

This is unfortunate and we would like to incorporate critical assessment and evaluation of existing code into the curriculum very early on.

4 Our Goals

Our goals are twofold:

Firstly, we want to use a more problem-driven approach than the traditional style. The problem-driven approach starts by presenting a practical programming

problem, and then examines a range of possible solutions. In the course of this discussion, new programming constructs or techniques are often introduced. This approach both ties the assignment and lecture close together, and provides a motivation for the introduction of new lecture material. In fact, the role of lecture and assignment is reversed: it is not the lecture content that drives the assignment, but the assignment problems that drive the lectures. This is a significant change because the curriculum is no longer driven by the features of a particular programming language.

Secondly, we want to achieve the inclusion of modern software engineering tasks into the computing curriculum early on. Traditionally, early computing assignments often use a blank screen approach: students start with nothing more than a problem specification. They then start designing and coding a new application from scratch. The essential assignment task is to write code. This style does not reflect realities in the contemporary computing industry, where tasks such as reading and understanding of existing code, maintenance and refactoring, adaptation and extension are far more common than the development of new applications. We would like to emphasize that critical code reading and maintenance are essential skills for any programmer let loose on the world today.

Thus, this proposal affects both the form and the content of the material used in lectures and assignments. While our discussion of previous work shows that neither of these goals is new in itself, experience shows that implementation of them has been slow. In part, this is because it has often remained difficult to put these ideas into practice. Linn and Clancy [13] noted, for instance, that students often found lengthy expert commentaries difficult to read, while the familiar syntactic hurdle of Java's `main` method almost forces an early focus on syntax that may be hard to break away from.

However, although we have suggested the widespread adoption of Java had a possibly negative influence, it does actually also have a positive effect in this area. Java libraries and tools are now available that may help in supporting these approaches and make it worthwhile to revisit these issues. In the remainder of this chapter we present a very practical, easily realizable example of how these goals can be achieved.

We aim to do this on two levels: firstly, by presenting one concrete example; and secondly, by presenting abstract guidelines for the development of such teaching units in general.

5   A Concrete Example

5.1   Exploration

The example we discuss here to illustrate our approach is called *The World of Zuul* [4]. We typically use this example towards the end of the first semester of an introductory course.

When introduced to the students, the application has already reached a coherent stage of development. It compiles and displays recognisable runtime behaviour. The first student activity is to explore and describe the application.

The exploration takes place within the BlueJ environment (Fig. 1) [6, 12] and includes discussion of both its functional aspects (*What does the program do?*) and aspects of its implementation (*What is the role of each class in the application?*). BlueJ overcomes the problems usually associated with the use of many other programming environments because:

Unlike professional programming environments, it is designed for use by novice programmers.
It is specifically designed to support the teaching and learning of key object-oriented concepts.
It supports the full implementation of the Java language and not just a subset.

Interactive exploration is enhanced significantly by the features provided by BlueJ, such as the UML visualisation of the application structure, interactive object creation and method calling, object-state inspection and source-code exploration. For instance, an instance of any class shown in the UML diagram can be created independent of running the whole application. Inspection of the object's state and direct interaction with its methods are also possible (Fig. 2).

Students quickly find out that the application implements a framework for a text-based adventure game [1] that allows the player to enter text commands and move around between a small number of locations, using commands such as *go east* (Fig. 3). Exploration of the implementation is done as a group activity, in which students examine the classes' source code and explain each class in turn to other group members. The classes are well commented, so that most of the important information is easily accessible without the need to understand all details of the code.

5.2   The First Tasks

It is clear to the students from their exploration that the game framework in its current form is rather limited in its functionality, and that it needs to be extended to turn it into a real game.

The next thing the students do is to invent an alternative game scenario. This can be done using a large variety of interactive or group activities, in which students develop and discuss ideas, finishing with every student selecting and describing a game plan. The explorations do not need to be constrained by implementation considerations. Topics can be anything: blood cells travelling through the human body; "you are lost in the shopping mall" themes; or the typical dungeon and dragon style scenarios.

Next, a number of small improvements to the given application are discussed. These are the addition of new movement directions (*up* and *down*), introduction of items in rooms (initially only one item per room) and appropriate new commands, such as 'take' and 'drop'.

5.3   Discussion

These first small tasks are discussed in detail in a lecture, including interactive development of an improved solution from the original limited version.

Even without the students having written a single line of code, it is clear that a number of important topics have been explored and practiced, such as code reading, exploration of class interfaces, and abstracting from the details of a particular game to its general characteristics.

Discussion of the necessary changes to the source code to attempt the first extension fits the model of learning from experts [2, 13] but with an important difference – the quality of the artefact being presented. Discussion quickly focuses on code quality and it becomes obvious that the given code makes the proposed extensions quite hard to implement, because it is badly structured. This gives us the opportunity to discuss aspects of code that make maintenance easy or difficult.

We discover cases of code duplication, broken encapsulation and bad distribution of responsibilities, and we see how these make our life harder. Developing an ability to evaluate code critically is key here.

This gives us the further opportunity to discuss the fact that the functional view of an application does not tell us about the quality of the underlying code (the program 'worked', after all). Students often struggle with the idea that they received a low mark because their program appeared to do all that was required of them, but it was badly implemented.

From what we have, and where we want to get to, we can illustrate the process of refactoring – improving quality without increasing functionality. This is a process students rarely engage in, usually because of time constraints – a not dissimilar experience of programmers in 'the real world'! Closely associated with the task of refactoring is the need to establish that changes to the source code have not broken anything. Any course that seeks to embrace key software engineering concepts and techniques [9] must include testing as an integral part. BlueJ supports JUnit-style unit testing [10] through interactive recording and replay of test cases and establishment of test fixtures [15]. Use of these features during both the sort of refactoring activities we are discussing here, and during work on the extensions, reinforces the message that testing is an activity that is naturally concurrent with development rather than an afterthought.

The refactoring process allows us to evaluate the code in terms of concepts such as coupling and cohesion, and goals such as localization of change. We improve the underlying design first by refactoring relevant bits of code, and then we find that making our intended extensions has become much easier than it would have been previously. The first task (adding up and down movement) is solved completely via a practical demonstration in the lecture, with extensive discussion about considerations of code quality while making modifications to the given source.

5.4   Exercises

The second task (adding items to the game scenario) is done as a series of exercises. The problem and some aspects of a solution are discussed, and then students are expected to implement the detailed solution on their own.

The discussion contains a hint to the solution, and asks the relevant questions to make students consider important aspects; 'We have discussed responsibility-driven design – which class should be responsible for printing out the details of the items present in a room? Why?'

The exercises are organised in a sequence of manageable steps of increasing complexity. Adding items to rooms, for example, is initially done by supporting at most a single item being placed in each room, and then extended in a separate exercise to allow rooms to hold an arbitrary number of items. Correspondingly, the player can initially pick up and carry a single item, which is later extended to multiple items. The number of items carried can later be limited by a maximum possible weight that a player can carry, for instance, or incompatibility between items, e.g., 'The player may not carry both matches and a flammable potion at the same time.'

5.5   Assignment

The exercises then lead into a larger assignment. In this assignment phase students implement their own game scenario including their own ideas for making the game interesting. Typical elements students implement include forms of time limits, magic transporter rooms, trap doors, locks, talking characters, moving characters, and more.

At this stage, students receive less help and guidance in developing their solutions than during the exercise phase. They are expected to develop solutions on their own, with the possibility of asking a tutor for support.

During this phase, tutors frequently discuss the quality of students' solutions under maintainability and extendibility aspects with the students. It is made clear that code quality, reviewed under the aspects discussed in the lecture, will be a major component in the marking scheme for the assignment.

6   The Three Steps

The previous section described a specific example of an assignment following our approach. In this section, we discuss the ideas behind this structure in a more general form. On the first level, the assignment approach can be divided into three steps: Observation, Application and Design. We discuss each of these in detail below. It is worth nothing that the order of these activities is exactly reversed compared to the classical, clean slate style of assignments: there, students typically have to start with design, followed by application, before they observe behaviour.

6.1   Step 1: Observation

In this step, the instructor actively demonstrates a software engineering task in the lecture. This part is modelled on the apprentice approach: students observe the instructor performing a relevant task and listen to the instructor's commentary, while having the opportunity to interrupt and ask questions.

Aspects of this phase are typically the analysis of given code and the discovery of problems and ideas for solutions. It gives an opportunity to reflect on existing code and to evaluate critically before making changes.

Typically, the problems discovered during the evaluation of the code lead to a motivation for new course material, which can then be introduced and discussed. Students then observe the application of the new material in a well-chosen example, with the opportunity to discuss alternatives.

6.2   Step 2: Application

The educational goal of the second step is the application of new material under guidance.
Teachers discuss selected problems, chosen to display similar challenges to those demonstrated in step one, and give hints to solutions. Problems are chosen so that variations of the material from step one are applicable for the solution. Students are expected to mirror the critical analysis and evaluation activities of their teacher and actively reason about the given code and argue about intended solutions. This phase usually spans an arbitrary mix of lecture and lab classes, and alternates repeatedly between active coding activities and reflective discussion.

6.3   Step 3: Design

In the third step, students design their own tasks as extensions of the project at hand. It is a free programming assignment that allows students to apply all the techniques they are familiar with at that stage.

Typically, students are given a minimum of guidance on expected tasks, simply to communicate the required amount of work for marking purposes. At this stage, this only includes the description of sample tasks, not usually pointers to solutions.
One of the advantages of doing an assignment task in this context is that students are familiar with the framework they are expected to extend. Since steps one and two have served to familiarize the student with a given application, larger, more complex and more interesting applications can be used. Extension tasks proposed by students are typically reviewed and guided by a tutor to ensure their usefulness in applying interesting course material, and their suitability in workload and level of difficulty.

7   Important Aspects of this Approach

7.1   Problem Driven

The introduction of new material is driven by a concrete problem. The motivation for introduction of new concepts comes from a concrete task at hand.

This can easily be combined with additional problem-based learning approaches, such as student-controlled discovery of new material. Instead of presenting all new constructs in a lecture, students can be guided towards resources that enable them to discover new material as part of a student activity.

7.2   Apprentice Approach

Our approach is an extension of the apprentice approach. Students start by studying expert written code: both well-written code and code to be critically evaluated and improved under expert guidance. An important part of students' learning comes from experiencing an expert in action, hopefully imitating some of the activities considered good practice in their own work. This activity should be facilitated by a Java environment that supports incremental development and testing.

One of the important additions to the original apprentice approach as described in [2] is that students can also observe the *process* of the expert's work, in addition to the created artefact.

7.3   Open / Closed

It is important to have characteristics of both open and closed assignments. The task should be well enough described so that weaker or less enthusiastic students have clear guidance as to what is expected of them, and how much they have to accomplish to receive a satisfactory mark in the assessment. On the other hand, the task should be open enough so that students can incorporate their own ideas and progress much further than the minimum required pass level.

It is common in computing classes that student groups display a wide variety of skills, and making the task challenging and interesting for even the best students is an important goal. This encourages both creativity and innovation.

7.4   Ownership

Whenever possible, the problem should be set in such a way that the student can take ownership of the task. In the *Zuul* example, this is achieved by letting students invent and design their own game scenarios and individual extension tasks. This, in turn, is the result of their abstracting from a particular example to the general principles it embodies, and on to a further instantiation. From that moment on, when students work on the implementation of the task, they don't view it so much as work on a problem outside their control, but as implementing *their* game.

7.5   Student Controlled

Another related (but distinct) issue is the ability of a student to take control over significant parts of the task. Game-based assignments have been discussed in the past in the context of gender bias [8]. Studies have found that female students are often interested in different kinds of computer games than male students, and that games without any social component or relevance are less likely to engage female students.
While *The World of Zuul* is clearly a game-based example, we have not observed the described gender effect in its use. (While we have not carried out a formal investigation, we have consciously monitored this aspect and held informal talks with students about it.) We speculate that the reason for this is that students can individually decide the context of their tasks by inventing their own scenarios. Giving students this degree of control might lead to higher acceptance of the relevance of the task.

8   Conclusion

Despite the evident educational and motivational value of problem-based approaches to introductory computer science, the traditional delivery style involving separation of lecture material and lab material continues to dominate in many places.

In this chapter we have described a concrete example of a more integrated approach that can be used in introductory programming courses. It includes problem-driven aspects, but is easier to realise than a complete problem-based learning model.

In addition, we have provided guidelines to assist others to apply this approach to their own material. This approach fosters a concept-driven approach to delivery of new material, and encourages ownership of these concepts by the students. It has been used by the authors for several years at the University of Southern Denmark in Odense, and at the University of Kent in Canterbury, England. Experiences are overwhelmingly positive, including student feedback, pass rates and teacher satisfaction.

Further ideas for application of this approach within the context of an object-first programming course can be found in [4] where, for instance, study of a multimedia database motivates the introduction of inheritance features, and a predator-prey simulation leads to discussion of abstract classes and interfaces.

It is worth pointing out that the approach is general enough to be applicable in an even broader context than simply introductory object-oriented Java teaching

9   References

1. Adams, R.: The Colossal Cave Adventure. http://www.rickadams.org/adventure/, accessed 17 August 2005
2. Astrachan, O., Reed, D.: AAA and CS 1: The Applied Apprenticeship Approach to CS 1. *Proceedings of SIGCSE 27* (1995) 1-5

3. Astrachan, O., Smith, R., Wilkes, J.: Application-based Modules using Apprentice Learning for CS 2. *Proceedings of SIGCSE 1997* (1997) 233-237
4. Barnes, D. J., Kölling, M.: Objects First with Java – A Practical Introduction Using BlueJ. 2nd ed. Pearson Education, Harlow, England (2005)
5. Barg, M., Fekete, A., Greening, T., Hollands, O., Kay, J., Kingston, J.H.: Problem-based learning for foundation computer science courses. *Computer Science Education 10* (2000) 1-20
6. BlueJ: BlueJ – The interactive Java environment. http://www.bluej.org/, accessed 17 August 2005
7. Ellis, A., Carswell, L., Bernat, A., Deveaux, D., Frison, P., Meisalo, V., Meyer, J., Nulden, U., Rugelj, J., Tarhio, J.: Resources, Tools, and Techniques for Problem Based Learning in Computing. *Proceedings of ITICSE '98* (1998) 46-50
8. Inkpen, K., Upitis, R., Klawe, M., Lawry, J., Anderson, A., Ndunda, M., Sedighian, K., Leroux, S., Hsu, D., "We Have Never Forgetful Flowers in Our Garden:" Girls' Responses to Electronic Games. *Journal of Computers in Math and Science Teaching 13* (1994) 383-403
9. Jackson, U., Manaris, B., McCauley, R.: Strategies for Effective Integration of Software Engineering Concepts and Techniques into the Undergraduate Computer Science Curriculum. Proceedings of SIGCSE '97 (1997) 360-364
10. JUnit: Testing Resources for Extreme Programming. http://www.junit.org/, accessed 17 August 2005
11. Kölling, M., Barnes, D.J.: Enhancing Apprentice-Based Learning of Java. Proceedings of SIGCSE 2004 (2004) 286-290
12. Kölling, M., Quig, B., Patterson, A., Rosenberg, J.: The BlueJ system and its pedagogy. In Journal of Computer Science Education, Special issue on Learning and Teaching Object Technology, Vol 13, Nr 4, (2003)
13. Linn, M.C., and Clancy, M.J.: The Case for Case Studies of Programming Problems. *Communications of the ACM* (1992) 121-132
14. Norman, D.A., Spohrer, J.C.: Learner-Centered Education. *Communications of the ACM 39* (1996) 24-27
15. Patterson, A., Kölling, M., Rosenberg, J.: Introducing Unit Testing with BlueJ. Proceedings of ITiCSE 2003 (2003), 11-15
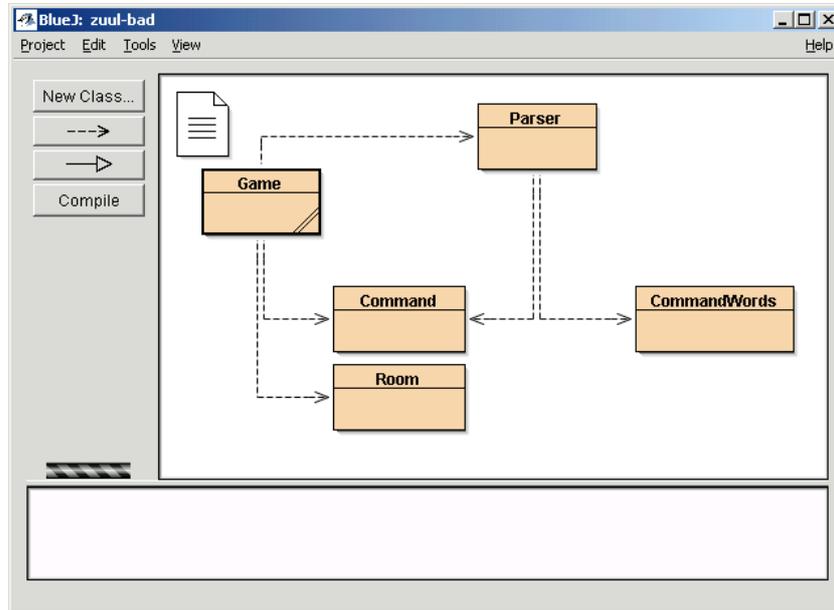16. Reges, S.: Conservatively Radical Java in CS1. Proceedings of SIGCSE 2000 (2000) 85-89

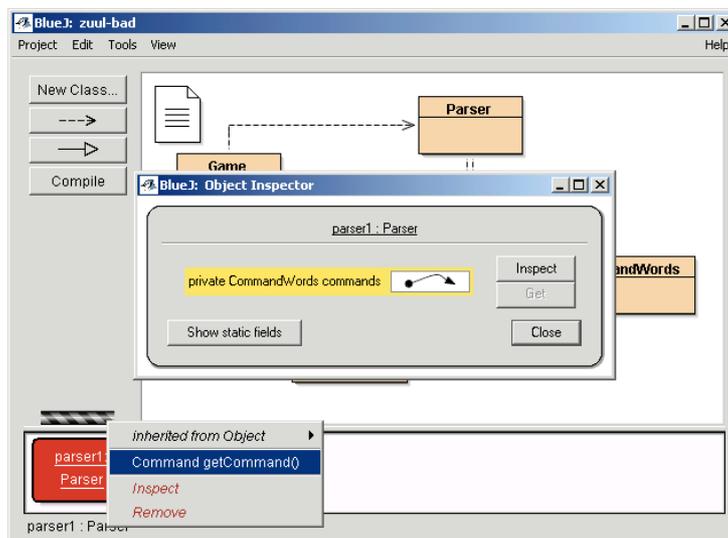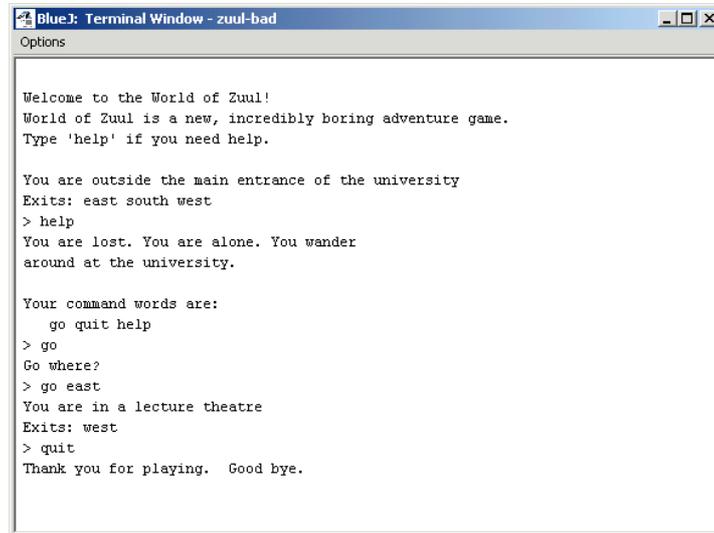**Fig. 1.** The World of Zuul project within BlueJ.

Fig. 2. Direct object manipulation and inspection.

Fig. 3. The user interface and exploration of the Zuul game.