IWAVE: Interactive Web-based Algorithm Visualization Environment

Internal Visual Learning Lab Report
September 2007

Ben Moss

School of Computer Science and Information Technology
University of Nottingham

bxm@cs.nott.ac.uk

Abstract

This report discusses one of the challenges faced in the teaching and learning of introductory computer programming. The demographic of students has changed considerably in recent years, and teaching styles must adapt accordingly to suit the change in learning styles. Some of the issues involved in making these changes are discussed, before introducing a method for calculating the relative difficultly of a concept based on the submission rate and average mark of its exercises. This method was applied to the results of students' programming exercises throughout a semester to identify one concept area that is particularly problematic - Arrays.

A customized visual learning environment for interactive animation of programming code was developed, allowing students to visualize code and the affects of any changes they make. In addition, a deployment wizard was developed to allow a practitioner to integrate the learning environment with their existing learning material with minimal effort. These tools were then used to create a demonstration learning resource targeted towards the concept of Arrays.

1 Introduction

The *Programming* module *G51PRG* is core to all undergraduate degree programmes in the School of Computer Science and Information Technology (approximately 150 students per year). Running over two semesters, the first semester provides an introduction to programming concepts and the tools used, whereas the second semester focusses more on application of these concepts in combination with other advanced topics. The module is based around the *Java* programming language.

The first semester is divided into eight distinct units that roughly equate to different core concepts, and previously-completed units are typically prerequisites for their successors. A major part of the learning and assessment process is a set of weekly programming exercises, which are automatically assessed via the *CourseMarker* system, based on several criteria (e.g. functionality, quality, methodology, completeness, etc.). Between one and three exercises are set each week, and students are encouraged to work on them in their own time, but assistance is provided by postgraduate demonstrators in weekly lab sessions.

Over the past ten years the demographic of students who take the *Programming* module has changed considerably. Classes are no-longer predominantly single-honours computer science students, but come from a wide range of academic departments, including engineering, humanities and business. The differing backgrounds and expectations in this diverse student group require alternative methods from the classical teaching and learning styles used for introductory programming. This "new breed" of programming student tends to struggle with many of the concepts which are critical for learning programming, particularly when skills such as spatial awareness and abstraction are involved. Students are often confused by the syntax of the code, which can inhibit and remove them from the semantics of the exercise's underlying problem.

In a response to the changes in the demographics of programming students, numerous research has investigated problems concerning the visualization of program execution [2]. For example, code syntax can be encapsulated in visual objects to allow students to concentrate on what the code means (its semantics), rather than how it is represented (its syntax). Many institutions use passive visualization in course notes to cover concepts such as arrays, methods and objects by use of single diagrams (statically) or by use of animation (dynamically). However, creating passive teaching resources is time consuming and can be difficult to maintain. Further research has addressed these problems by developing interactive methods that can automatically generate diagrams and animations from samples of program code [1]. This interactive method is typically seen as a time-saving mechanism for the lecturer, but if such mechanisms were presented to the student, then they could be encouraged to develop a deeper understanding of topics through experimentation with different aspects of their program code (i.e. they can actually *see* the affects of adding, deleting, moving and substituting code fragments).

Despite the time-saving capabilities of existing code animation technologies, it is generally under utilized in introductory programming modules. Two suggested reasons for the narrow adoption of such visualization software are:

- A lack of dissemination amongst practitioners (i.e. teachers are not aware that such teaching aids exist);
- The typical steep learning curve for practitioners to embed it into their current teaching practices, and for students to use it in their learning process.

2 Aims and Objectives

The overall aim of this project is to automatically visualize the execution of program code to complement the teaching experience for practitioners of introductory programming and the learning experience of their students. This aim will be achieved through the following objectives:

- To determine concepts students find most difficult, using evidence from our automated-assessment system (*CourseMarker*);
- To select one of these difficult concepts where teaching and learning can be enhanced by visualizing the execution of program code;
- To develop a system for creating and deploying visual teaching and learning resources;
- To demonstrate use of the system through a range of examples within the selected difficult concept, exploiting the interactive visual learning aspect of the system, allowing students to experiment freely by modifying the example code.

Additionally, the system should have the following practitioner-oriented goals:

- **Usable**: To minimize the extra knowledge required by the practitioner to create and deploy program code visualizations as visual learning resources.
- **Malleable**: To minimize the effort required to make small changes to an existing visual learning resource that was created with the same system (e.g. correcting small mistakes, or making regular yearly changes).
- **Embeddable**: To use widely-adopted and/or standards-compliant technologies to aid integration with existing teaching resources and encourage wider adoption/dissemination.

The system also aims to achieve the following student-oriented goals:

- **Usable**: To provide a system that students can use effectively, with little or no training.
- **Interactive**: To allow students the ability to view animations of both code examples from the notes and their own coursework code at any stage during its development.

3 Methodology

The project was conducted in three phases. The first stage involved analysis of the extant data from *CourseMarker*. This was followed by customization of the *Jeliot* system for closer integration with the module's requirements. The third stage involved development of a software wizard, that enables a practitioner to create instances of the customized *Jeliot* system for specific code examples without requiring any additional technical knowledge. The final stage ties the previous stages together as a demonstration in the form of an on-line lesson. This lesson addresses one of the problems found by analysis of the extant data using visual examples provided by the customized *Jeliot* system, and was developed using the software wizard.

3.1 Analysis

To determine the introductory programming concepts that students find difficult, the *CourseMarker* results for semester 1 of the 2005-06 academic year were analysed. These results are comprised of an integer percentage representing the overall exercise score, for each exercise, and for every student. Students have the option of whether to submit each exercise, but non-submission results in an exercise score of zero. However, it is possible to determine between a submitted and non-submitted exercise score of zero, the former case being a rare occurrence. Each exercise belongs to a unit, which may contain one or more exercises designed to test the core programming concept of that unit. In an effort to remove any bias, students who did not complete semester one due to transfer, withdrawal, or suspension of the course were not included in the analysis.

For each of the eight units, two values were determined from the data. The *average submission rate* is the percentage of a unit's exercises for which a student solution was submitted. It was assumed that this value would provide an indication of a unit's difficulty, since more difficult exercises typically result in less students attempting and submitting their solutions (see Figure 1).

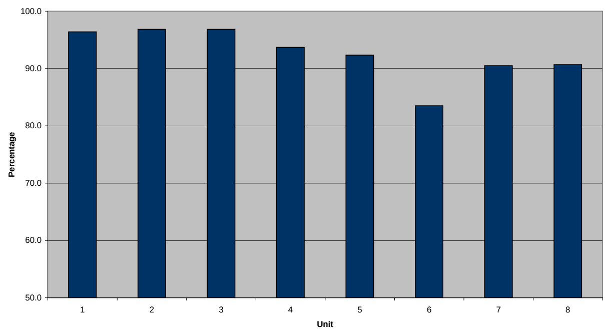

Figure 1. Average Submission Rate

The *average submitted mark* is the average score of a unit's submitted exercises. It was assumed that this value would provide an indication of a unit's difficulty, since more difficult exercises typically result in lower marks for submitted solutions (see Figure 2).

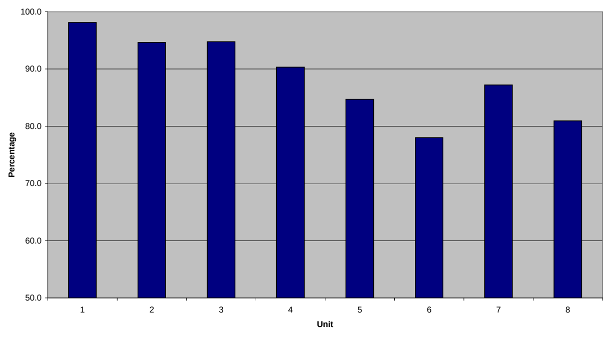

Figure 2. Average Submitted Mark

The average submission rate and average submitted mark values were combined to form an *average difficulty metric*, which is a single comparative value for each unit (see Equation 1).

$$difficulty = submissionrate \times submittedmark \quad (1)$$

The average difficulty metric is intended to measure the comparative difficulty of each introductory programming concept (see Figure 3). For overall comparison, all three averages are included in Figure 4.

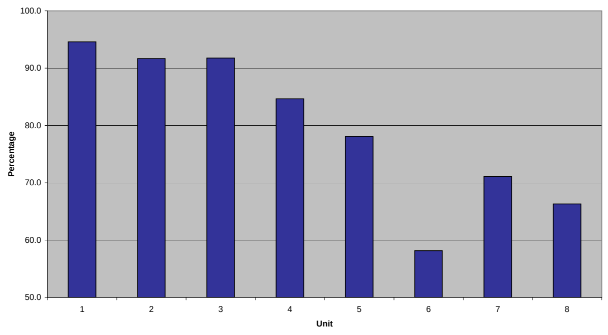

Figure 3. Average Difficulty Metric

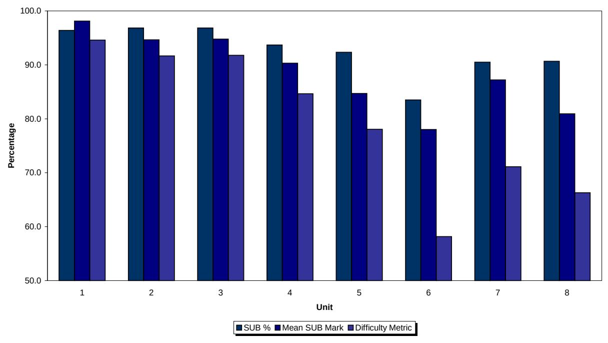

Figure 4. All Averages

The results indicate a downward trend in both submission rate and average mark as the semester progressed, and this trend is amplified in the average difficulty metric. In all three values, Unit 6 (Arrays) is anomalous to the gradual downward trend, being markedly lower than other units, and the only unit with an average difficulty metric below 60%.

In programming, arrays are abstract container-like structures used for storing a list of values. For example, an array might be used to store a list of each item's price when paying for shopping at a supermarket checkout, or a list of lap times in an athletics event. Arrays are typically taught using drawings or diagrams that depict an array as a series of connected boxes, in which their values can be written. However, visualizing arrays in this manner can be extremely time consuming, particularly when the contained values change.

3.2 Jeliot Customization

To develop a system for creating and deploying visual teaching and learning resources a brief survey of code animation tools was conducted. The survey indicated that the *Jeliot* system would be highly suitable for this project for the following reasons:

- **Java code animation**: The system animates code written in the Java programming language, which is the language used throughout the *G51PRG* module.

- **Low-level visualization**: The system is well-suited for visualizing low-level programming constructs such as variables and control blocks, as opposed to high-level constructs such as data structures and object-oriented structures. Low-level constructs are covered in introductory programming, whereas high-level constructs tend to be taught in more advanced modules.

- **Interactive**: The system allows students to modify examples and instantly visualize the effects of their modifications.

- **Web deployable**: The system incorporates *Java Web Start* technology, which allows it to be deployed via the Web.

- **Open source**: The system is developed as open-source software, and as such users are permitted to change, improve, and redistribute it in modified or unmodified form. This allows the software to be customized for local integration with existing teaching and learning resources.

- **Written in Java**: The system itself has been developed using the Java programming language, allowing better integration with other Java-based software used as part of *G51PRG* for teaching and learning.

The first stage of *Jeliot*'s customization concerned integrating the existing *G51PRG* IO (input/output) system into *Jeliot*. Student's often find Java's IO too complex at the introductory level, and therefore *G51PRG* students are given an alternative method that simplifies the processes. Similarly, the *Jeliot* system also provides its own simplified alternative, but this is not directly compatible with the *G51PRG* simplification. Therefore it was necessary to integrate the *G51PRG* IO method into the *Jeliot* system to allow *G51PRG* students to visualize their own exercise solutions in *Jeliot* without them having to make significant changes to their solutions.

The second stage of customization involved several changes to the *Jeliot* GUI. The layout was modified to maximize the size of the visualizations, and the controls were organized in a more logical manner. To better integrate the system with existing teaching and learning practice, a new colour theme and icon set was created.

3.3 Wizard Development

The *Jeliot* system is primarily designed as a learning tool, whereby students create and edit their code using *Jeliot* as desktop software to visualize the program execution. Despite being deployable via Java's *Web Start*, the *Jeliot* system is not well-suited for teaching practitioners to deploy a visualization for a specific code example. In other words, for a practitioner to allow their students to visualize a code example via the Web, the students would need to launch *Jeliot* via *Web Start*, download/copy the code example from the Web, then open/paste the code example into *Jeliot*. It would be far simpler, if *Jeliot* could be launched with the code example pre-loaded.

For a code example to be pre-loaded when *Jeliot* is launched, several steps are needed that require complex technical knowledge of the *Jeliot* system and Java's *Web Start* technology. A software *wizard* was developed to simplify this process, by leading the user through a sequence of simple steps, and gathering the required information to perform this complex task. When *Jeliot* is launched, it loads a default template Java code example. This default template is archived into the single *Web Start* file, which is built through an *ant* build script that incorporates compilation and code signing. The overall process is comprised of four stages:

1. Incorporating a user-specified code example into the source material, which is used to build *Jeliot*. The code example is translated into an internal format that *Jeliot* can interpret and built into a resource file using a default skeleton resource file.

2. Creating a build file from a user-specified *JDK* installation path and a default skeleton build file.

3. Creating a Java *Web Start JNLP* file from two user-specified values (a name and a URL) and a default skeleton *JNLP* file.

4. Executing the generated build file to build an executable *Jeliot jar* file from the source material generated in the first stage. The *jar* file is then digitally signed using a predefined cryptographic *key store*, and the deployable *Web Start* files are generated and packaged into a release folder, ready for copying to the Web server.

3.4 Demonstration

The wizard has been used to visualize a range of examples to complement the existing teaching resources for Unit 6 (Arrays). These visualizations have been embedded into a Web-based lesson on the subject of arrays, covering the basic and intermediate concepts. The Web-based system for delivery of the lesson is based around standardized Web technologies, such as XML, XSLT, CSS, and Java, for maximum compatibility and longevity. The code examples have been carefully chosen to demonstrate the key concepts of arrays, whilst fully exploiting the use of visualization for greater impact. Students can also experiment freely by modifying the example code as part of their self study. The demonstration has been integrated with the existing Web-based teaching materials for the *G51PRG* module.

4 Conclusions

The analysis stage of this project showed a gradual downward trend in both the submission rate and average mark of students throughout the semester. This pattern was amplified in the average difficulty metric, which combines the students' submission rate and average mark. The downward trend suggests that the module's units become more difficult throughout the module, which is both intended and expected. The result for Unit 6 (Arrays) was anomalous to the gradual downward trend, being markedly lower than other units for all measurements, and the only unit with an average difficulty metric below 60%. This anomalous result suggests that Unit 6 covers a concept that the students find more diffult than any of the others covered in the semester. From previous experience of the teaching staff involved, it was expected that the concept of Arrays would be the most difficult, but this is not intentional, and is a problem that needs to be addressed.

In order to address the problem, the *Jeliot* system was customized for integration with the *G51PRG* module. The customizations were focussed around ease of use for the students; providing seamless integration with existing teaching materials and software. A software wizard was developed for practitioners to easily deploy instances of the customized *Jeliot* system. The wizard devolves all the complexities of deploying an example visualization into existing teaching material by guiding the user through a simplified decision-making process.

Having developed a visualization environment that could be integrated into existing teaching materials using the deployment wizard, a demonstration project was created. The demonstration was based around extending existing lecture notes and example code to produce two lessons (*Basic* and *Intermediate*) on the concept of Arrays. The lessons incorporate "one-click" visualizations of the code that can be viewed by students. The visualizations are interactive, allowing students to modify the code and instantly see the affects of their changes in the visualizations.

The scope and depth of this project have been limited by its relatively short duration (8 weeks). The main impacts of this limitation were in the analysis and demonstration stages. Other factors might have an affect on trends in the average mark and submission rate of a unit's exercises. For example, the weighting of exercises, the time taken to complete a unit, and clashes with deadlines for other coursework or social events occurring at the time of study/assessment. Future work should consider student results over several years to minimise the impact of external factors.

Throughout this research, priority was placed on developing a generic system that could be useful in any of the concepts, rather than identifying and developing materials for the most problematic concept. Moreover, it was deemed that user trials would not be effective in such a short time period, and could not be scheduled to coincide with the Unit 6 learning. Future work should consider running user trials to coincide with student learning for Unit 6, and comparing the resulting difficulty metrics with previous years' results.

The deliverables of this project have been disseminated via the Web, for maximum exposure.

References

[1] E. S. J. T. Mordechai Ben-Ari, Niko Myller. Perspectives on program animation with jeliot. *Springer: Software Visualization: International Seminar*, 2269:618–621, 2002.

[2] U. Z. T. Crews. The flowchart interpreter for introductory programming courses. In *28th Annual Frontiers in Education Conference*, volume 1, pages 307–312, November 1998.