Befriending computer programming: a proposed approach to teaching introductory programming

Issues in Informing Science and Information Technology **Volume 4, 2007**

Befriending Computer Programming: A Proposed Approach to Teaching Introductory Programming

Iwona Miliszewska and Grace Tan
Victoria University, Melbourne, Australia

Iwona.Miliszewska@vu.edu.edu Grace.Tan@vu.edu.edu

Abstract

The problems encountered by students in first year computer programming units are a common concern in many universities including Victoria University. A fundamental component of a computer science curriculum, computer programming is a mandatory unit in a computing course. It is also one of the most feared and hated units by many novice computing students who, having failed or performed poorly in a programming unit, often drop out from a course. This article discusses some of the difficulties experienced by first year programming students, and reviews some of the initiatives undertaken to counter the problems. The article also reports on the first stage of a current research project at Victoria University that aims to develop a balanced approach to teaching first year programming units; its goal is to 'befriend' computer programming to help promote success among new programming students.

Keywords: automated assessment, introductory computer programming, programming support, student mentors

Introduction

Computer programming is an integral part of a computer science curriculum and a major stumbling block for many computing students, particularly in the first year of study; many of those students find programming difficult to grasp, let alone master (Dunican, 2002; Jenkins, 2002; McCracken et al., 2001; Proulx, 2000). Difficult to learn, programming skills are difficult to teach too (Allison, Orton & Powell, 2002), not least because "traditional teaching methods do not adapt well to the domains of coding and problem solving, as it is a skill best learned through experience" (Traynor & Gibson, 2004, p. 2). According to Kölling and Rosenberg (2001), the situation is even more challenging when it comes to teaching object-oriented programming to beginning students as "software tools, teaching support material and teachers' experience all are less mature than the equivalent for structured programming" (p. 1).

The issue of computer programming is no different at Victoria University where, since 1999, object-oriented programming using Java has been taught to the introductory programming students. Here, too, students struggle with programming, and programming has continued to be a major factor contributing to the attrition of first year students from the computing courses. Various restructurings of the programming unit and changes to teaching methods implemented over the years, for example the use of different textbooks, or the introduction of an electronic assignment assessment system, have done little to improve the situation; a new approach was called for. To this end, a new research project, supported by a Teaching and Learning Support grant, was launched in July 2006 to investigate the nature of the difficulties encountered by programming students and develop a 'friendly' framework for teaching programming to novices; the framework was intended to make computer programming welcoming and accessible to novice programmers and, at the same time, achieve pedagogical objectives.

To address the difficulties associated with computer programming, first it is necessary to understand them well. Accordingly, this article first looks in detail at the reasons why first year students find programming such a daunting prospect, and discusses the impact that poor performance or failure in an introductory programming unit can have on computing students. An overview of the various interventions created to alleviate the programming problem is also presented. Then, the article outlines the features of a proposed approach to teaching introductory programming currently being developed at Victoria University.

Difficulties Encountered by First Year Programming Students

Undergraduate students enrolling in computing courses are not expected to have prior programming experience; computing experience is not a prerequisite. While some students study some computing units in secondary schools, many do not. The lack of prior computing experience does not seem to be a problem however, the lack of problem-solving skills is. Dunican (2002) indicated that subjects offered in secondary schools do not include any logic/problem-solving modules, which puts students in a difficult position when they enrol in computing courses at university. Stamouli, Doyle, & Huggard (2004) also pointed out at the lack continuity between subjects studied in secondary schools and those encountered in the first year of university studies; they went on to say that several of the first year units including computer programming were "beyond the students' previous experience".

Even though computer literacy may be high among some of the commencing computing students, most of them tend to lack experience with programming. This includes not only program design and construction, but also routine tasks such as compiling or running a program, or, indeed, a basic understanding of a computer model with its hardware and software components. This lack of understanding of a mental model of a computer often results in much frustration when students are expected to not only construct a program but make it work, too (Ben-Ari, 1998).

Another difficulty faced by programming students is the need to imagine and comprehend many abstract terms that do not have equivalents in real life: how does a variable, a data type, or a memory address relate to a real life object? Programming concepts tend to be difficult to grasp (Dunican, 2002). Consequently, many computing students claim to 'hate programming' as they struggle to comprehend even the most basic of programming concepts (Stamouli et al., 2004; Thomas et al., 2002).

One more difficulty is the task to meet the requirements of programming syntax. Even students, who have adequate problem-solving skills and manage to phrase a solution to a programming problem in terms of a pseudocode, can find it difficult to turn the pseudocode into a syntactically correct computer program (Dunican, 2002; Kölling & Rosenberg, 2001; Sheard & Hagan, 1998).

Impact of Failure/Poor Performance on Students

While some programming students experience only one of the types of problems outlined above, others encounter several of them (Dunican, 2002). The effect of such experiences can be devastating. As Dunican (2002) has pointed out, "it takes very few negative experiences at the early stages to disillusion the student". Consequently, a student's initial enthusiasm for programming wanes as rapidly as difficulties emerge (Sheard & Hagan, 1998). Jenkins (2001) agreed, pointing out that the challenging aspects of learning how to program may de-motivate students; he concluded that if students are not motivated, they will not learn and, subsequently, they will not succeed. Some of the students who struggle with programming will drop out of the computing course altogether, while others will continue but will "assiduously avoid future programming projects and ultimately choose a career path that does not involve programming" (Stamouli et al., 2004).

High dropout and failure rates of first year programming units have been a growing concern at many universities. Tavares et al. (2001) investigated the problem, and identified two main factors that, according to students, precipitate the failure at introductory programming units: the curriculum organisation and the teaching methods; students pointed out complexity of the material covered in class as one of the reasons for dropping out. A different study found that, discouraged by a difficult curriculum, only less than half of the programming students attended practical classes and participated in assessment (Huet et al., 2003). Meisalo et al. (2002) reported that nearly 30% of their introductory programming students had dropped out from the course because they had found programming exercises too difficult, or had failed a re-take examination.

Interventions Employed to Help Novice Programming Students

Various types of interventions have been created over the years to help students develop programming skills. The interventions ranged from changes to the curriculum, pedagogy, and assessment, to the provision of additional support to new programming students.

Curriculum

Van Roy et al. (2003) successfully based programming units on concepts rather than on single paradigms (object oriented programming, logic programming, or functional programming) or languages. Having taught with this approach for two years in four universities, they found that it enabled students to "reason in a broad and deep way about their program's design, its correctness, and its complexity" (p. 270).

In environments where a programming unit was based on a single, object-oriented paradigm, one of the major issues is the way in which object orientation is introduced to students (Blumenstein, 2004; Lister & Leaney, 2003). Two contradictory approaches to curriculum design have been used and tested in various institutions: the *objects-first* approach and the *structured programming- first* approach; both these approaches have been reported as successful. Sheard & Hagan (1998) reported on changes to the curriculum of an introductory programming unit following the findings of a research project investigating the teaching and learning of introductory programming. They observed that students "started to feel lost … about the same time when object-oriented paradigm was introduced" to the unit (Sheard & Hagan, 1998, p. 315). Consequently, it was decided to use the more traditional bottom-up (structured programming- first) approach first, and introduce the object-oriented concepts after the students have had gained an understanding of expressions, statements, parameters, etc. This was one of the changes introduced to the unit but, as a result, "a significant increase in student performance has been noted since these changes were implemented" (Sheard & Hagan, 1998, p. 319).

Boris Magnusson (Van Roy et al., 2003) on the other hand, swore by the object-first approach. He opined that an early introduction of structuring mechanisms, classes, methods, and inheritance, helped students understand the mechanisms of problem analysis and solution development. He reported to have had over ten years of positive experience using this approach.

Pedagogy

One pedagogical technique employed to teach programming concepts to students is based on analogy. This technique is particularly useful when teaching programming fundamentals such as input/output, data types, sorting, searching, etc.; it uses illustrative examples of concepts that students have seen before, and relates the familiar concepts to new ones. In an analogy, the familiar concept is identified as the source and the new one as the target and, when an analogy is made, the source is mapped onto the target (Blanchette & Dunbar, 2000). Dunican (2002) describes several analogies for example: the use of children's toys to teach assignment statements; the use of boxes to determine the smallest and largest number in a list; and, the use of a leaflet distributor to explain the concept of array manipulation.

Another important pedagogical facet is relevance: students should see a purpose to what they are learning. Sheard & Hagan (1998) report on positive feedback from students after games with attractive graphical interfaces, including Solitaire and Minesweeper, were used to illustrate the benefits of the object-oriented paradigm. This illustration provided an opportunity to explain the advantages of object-oriented programming and design over other styles of programming for complex applications such as the presented games.

Iterative approach to learning and continuous reinforcement of concepts was yet another well-received technique introduced by Sheard & Hagan (1998) to their first year programming unit. In addition to an ongoing reinforcement effort, two entire lectures were devoted to consolidation and revision of the object-oriented concepts covered earlier.

Another approach relies on the use of technology for teaching. Clancy et al. (2003) described their efforts to develop a laboratory-based model for computer science instruction. Their model included three components: an online course builder for the instructor, a Web-based learning environment for the delivery of all student activities, and a course portal that served as a learning management system. The evaluation of the system showed that student performance in the course had improved and that the students found the course enjoyable. However, the new model had no impact on the attrition rate from the course.

Assessment

Frequent assessment is favoured in an introductory programming unit (Blumenstein, 2004) and the two types of assessment most commonly used include objective testing and performance-based assessment. Objective testing such as multiple choice questions is said to be useful in providing instant feedback to students in their understanding of language syntax or program behaviour; performance-based assessment such as laboratory exercises, programming assignments and examinations help to test students' ability to write working computer programs (McCracken et al., 2001).

While the most common assessment methodology requires all students to work on the same assessment tasks, Lister & Leaney (2003) advocated the use of criterion-referenced grading scheme in assessing their students. They suggested that such a technique was likely to maximise the potential of every student in a disparate class of different capabilities.

There is also considerable empirical evidence to support the view that student learning is enhanced when students are aware of their own learning (Boud, Keogh & Walker, 1985). Hence, educational theory indicates the benefits of promoting learner reflection in the learning process. In their study of learning styles and performance in an introductory programming sequence, Thomas et al. (2002) found that reflective learners, who learned by thinking things through and working on their own, scored higher than active learners who learned by trying things out and working with others. To enhance student reflection in an introductory programming unit, Fekete et al. (2000) incorporated different assessment strategies, and reported that, in their view, reflection enhanced the technical mastery of their students.

Support for Programming Students

One successful form of support provided to programming students was the introduction of discussion classes reported by Sheard & Hagan (1998). The classes, used to consolidate material introduced in lectures, were a success, particularly when object-oriented programming was introduced. Special exercises were developed to stimulate discussion among students and, when students reported difficulties with a particular aspect of an assignment for example, that aspect became the subject of the discussion class to assist students with their work.

Web pages for programming units have proved to be a useful support feature. Typically they contain unit details, staff timetables, lecture slides and laboratory exercises. In addition, most of the unit Web sites give students an opportunity to provide feedback to the staff. In the Web support system reported by Sheard & Hagan (1998), anonymous student feedback was used. It was a useful source of student comment about various aspects of the unit and, it was found that, “many students were willing to comment anonymously but not to contribute to a newsgroup” (p. 318).

An “emergency hotline”, or help desk, operating outside class time is yet another form of support well received by the students, proving particularly popular before assignment submission time. This type of service is often manned by tutors, and the problems referred to it usually concern programming syntax or logic errors, although questions concerning design strategy have been recorded too, as reported by Sheard & Hagan (1998).

A different service, a Programming Support Centre, has been launched by the Department of Computer Science at Trinity College (Stamouli et al., 2004). Its distinct feature is the provision of structured one-to-one support to students with programming difficulties. Like in other centres of this type, attendance is voluntary, but students are encouraged to take advantage of the service. The service operates for sixteen hours a week, in addition to regular classes; it is manned by professional programmers; and, it is housed in a dedicated well-equipped computing room. The results of a quantitative and qualitative evaluation of the centre indicate that it has had a positive impact on student learning.

A Proposed Approach to Teaching an Introductory Programming Unit

Computer programming is a core unit in six undergraduate degree courses offered by the School of Computer Science and Mathematics at Victoria University. As a mandatory first year unit in all courses, computer programming is a prerequisite to a number of second year units.

Issue to be Addressed

Over the years, the programming unit, RCM1311, has proven to be a stumbling block for many first year students and the poor pass rate for the unit, as reported in Table 1, has been an ongoing concern.

Table 1: RCM1311 Programming 1 - Percentage of students who failed the unit

Year	2001	2002	2003	2004	2005	2006
% of failures	47	33	36	42	58	39

This first year programming unit is considered crucial to students' success in the computing courses. This statement is supported by findings of a recent Higher Education Equity Program (HEEP) funded equity project reported in (Miliszewska et al., 2004; Miliszewska et al., 2006). The project revealed a number of transition related problems that seem to impact negatively on commencing students; it identified the first semester of the course as the 'make or break' period, and the introductory programming unit as the biggest 'break' factor. The first semester was not only an important period with respect to adjustment to the course, but also it was a period most likely to influence most students' decisions about quitting the course. Hence, the first year programming unit shapes students' perceptions about the entire course and, if taught well, it can help sustain students' interest, and ensure their success in, and completion of, the course. Conversely, a number of students have withdrawn from the course after the first semester because of their difficulties with the programming unit.

Information about the particular difficulties faced by new programming students at Victoria University was obtained from three sources: the 2006 cohort of first year programming students; comments on the Student Evaluation of Unit forms (spanning three years); and, interviews with lecturers involved in the teaching of the introductory programming unit. Classes and methods, graphical user interfaces (GUIs), and event handling were considered to be the most difficult topics to master, followed by iteration, selection, and input/output. In addition, students found it difficult to understand the mechanics of programming. These difficulties are common to many new programming students, as reported in the literature (Carbone et al., 2001; McCracken et al., 2001; Meisalo et al., 2002; Thomas et al., 2002).

Since the issue of first year computer programming has been recognised as important to students, staff, and the University (it affects the future of computing courses), a research project was launched in July 2006 to address the issue. The project team includes academics with experience in computer programming and education experts; the project is being funded by Teaching and Learning Support grant.

The Unit

The introductory programming unit, RCM1311 Programming 1, is the first programming unit that all computing students encounter in the School of Computer Science and Mathematics at Victoria University. It is the first of two mandatory units (RCM1312 Programming 2 is the second one) taught in the first year of a computing degree; a pass in RCM1311 is required to proceed with RCM1312. A pass in RCM1312 on the other hand, is required to enroll in three core units and three elective units in the second year of the degree. Hence, a pass in the very first programming unit is a virtual prerequisite to fulfilling the requirements of the degree.

The unit is based on the object-oriented paradigm, and it is taught according to the *structured programming – first* approach. The structured programming- first teaching methodology has been adopted as it was agreed that, "if students find it difficult to construct a viable model of variables and parameters, why should we believe that they can construct a viable model of an object such as a radio button?" (Ben-Ari, 1998, p. 260) Accordingly, it was decided that students should become familiar and comfortable with basic programming fundamentals, before getting acquainted with the concepts of the object-oriented paradigm; the syllabus of the unit is presented in the following order:

- Introduction: Course overview, editing, compiling and executing programs. Basic elements of Java programs: class, method, identifier, white space and comments. Basic data types, arithmetic operations, type conversion.
- Program Development: Control structures – selection and iteration.

- Objects and Classes: Class definition (instance variable, constructor, method), instantiation of objects, UML diagrams, access modifiers, static variables and methods.
- Using Selected Classes: Java's Class Library (e.g. Math, String, DecimalFormat, StringTokenizer, etc.) Applets and GUI components – label, text field, button, event handling, the graphics class, color, drawing shapes and displaying text.

The Java programming language, Standard Edition (Java 2 Platform), has been used as the development environment for the unit. Students can easily download the latest version of Java and the associated documentation from two alternative sources: the Web site of Sun MicroSystems, or a CD included in the prescribed textbook.

The delivery of the unit comprises of two one-hour lectures, a one-hour tutorial, and a one-hour laboratory per week in a twelve-week semester. Assessment of the unit includes summative assessment (weekly practical tasks, an assignment, and a test), which accounts for 30% of the final mark, and a three-hour final examination, which contributes 70% of the final mark.

The Approach

The proposed approach aims to change the negative view that computer programming is difficult and unfriendly for novice programmers; it aims to create a climate where students embrace programming. To this end, the approach builds on a variety of strategies that have been reported as 'successful' in the literature (as described in the previous section of this article). The approach incorporates the individual strategies with a view to achieving a better overall outcome.

Structure

The first year programming unit has been always taught in two one-hour lectures presented on separate days, and this structure will be retained; research shows that students find it difficult to concentrate for a two-hour span (Sheard & Hagan, 1998). Students will also attend two hours of laboratory/tutorial sessions a week. To facilitate active learning and hands-on practice, laboratory sessions will be merged with tutorial sessions; this will afford students more and better opportunity to interact with each other and the tutors; this increased interaction will assist in early identification of students 'at risk'. In addition, five hours of mentoring classes a week will be offered so students can voluntarily seek one-to-one assistance with their programming difficulties.

Pedagogy

The teaching methods of the proposed approach will include a careful study of examples of well-written code. This follows the recommendations of Kölling & Rosenberg (2001) who believe that students should read code before attempting to write it, as they "can learn a lot from studying well written programs and copying styles and idioms" (p. 2).

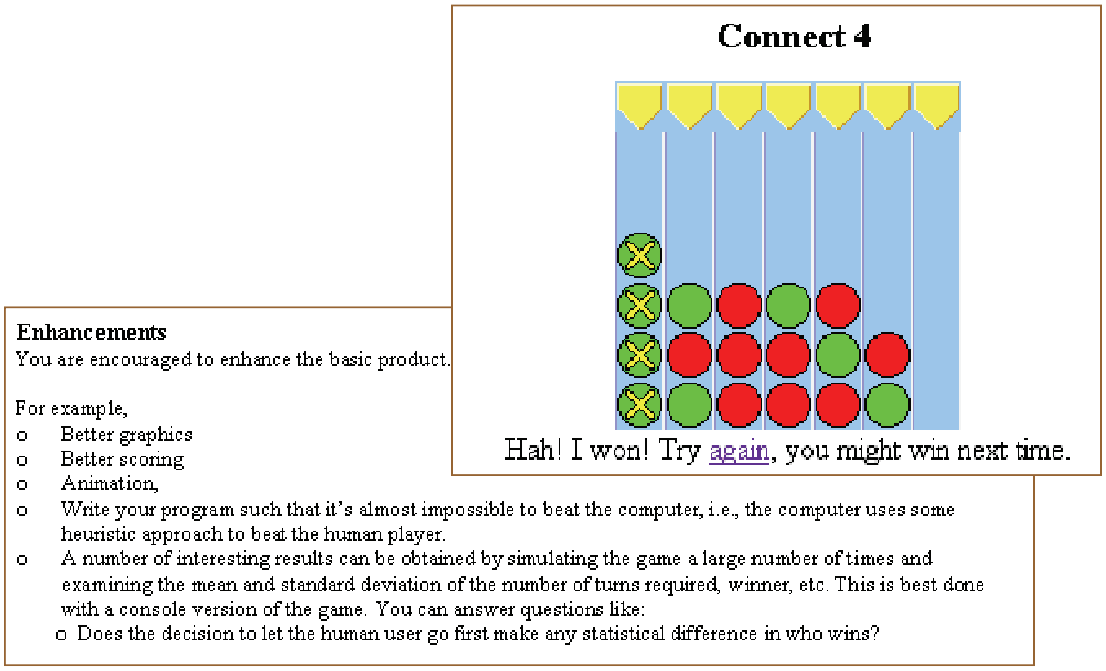
As students tend to respond well to analogy as a method of illustrating unknown concepts (Dunican, 2002), the proposed approach for teaching introductory programming will use analogy as tool for teaching abstract concepts. An analogy involving a classic children's shape toy for example, will be used to teach the concept of data types, assignment statements and type mismatches. In this analogy, adapted from Dunican (2002), an 'integer' shape can be stored in an 'integer' hole, or in a 'real' hole; on the other hand a 'char' shape cannot fit in either an 'integer' or a 'real' hole. The use of such simple analogies has met with student approval in previous offerings of the programming unit. In particular, a wooden box divided in a number of small pigeonholes was used to illustrate the concept of memory allocation; each pigeonhole had its own unique label (a memory address), and it could also store content.

It is intended to: design illustrative examples for most concepts; involve the programming students in providing alternative examples of their own; and, compile a data bank of analogies for future use.

Assessment

Students in the unit will be assessed through: laboratory exercises on a fortnightly basis, a major group assignment set half way through the unit, a mid-semester test, and a final examination. Summative assessment will be used in the laboratory exercises to assist students in developing their programming skills. The exercises will be short and simple, and designed primarily as learning experiences; for instance students will be required to make minor modifications to existing code. This should address concerns raised by Buck & Stucki (2001) who found that students, who were required to write a complete program on their own, often did not know how to begin and, instead of thinking a problem through, experimented by randomly throwing statements together hoping to achieve a desired outcome.

The nature of the assessment will also address the development in students of several skills including: collaborative skills, problem solving skills, and initiative; the open-ended scaffolding assignment involving a team of students will support the development of these skills in students. For instance, the assignment may require the students to write a program to play a game. The assignment will be divided into two parts: part one calling for a typical solution to the problem, and to be attempted by all groups; and, part two seeking possible extensions to the standard solution, as illustrated in Figure 1. This will motivate students to enhance their game programs and gain additional skills (and marks).



Hah! I won! Try again, you might win next time.

Enhancements
You are encouraged to enhance the basic product.

For example,
- Better graphics
- Better scoring
- Animation,
- Write your program such that it's almost impossible to beat the computer, i.e., the computer uses some heuristic approach to beat the human player.
- A number of interesting results can be obtained by simulating the game a large number of times and examining the mean and standard deviation of the number of turns required, winner, etc. This is best done with a console version of the game. You can answer questions like:
  - Does the decision to let the human user go first make any statistical difference in who wins?

Figure 1: Scaffolding assignment.

Game playing problems, such as the one illustrated in Figure 2, have been selected as assignment tasks to make the assessment task relevant and 'friendlier' to students. Students have been found to be keener to learn programming when they can easily produce attractive graphical interfaces.

To this end, tasks involving the implementation of computer games that manipulate graphical elements have been found particularly useful (Lorenzen & Heilman, 2002).

Mentoring classes

The biggest innovation with respect to support will be the introduction of mentoring classes. Similar in concept to the Programming Support Centre (PSC) described by Stamouli et al. (2004), the mentoring classes will be offered in addition to lectures, tutorials and laboratory classes, in a designated laboratory, every day of the week, at the same hour every day.

Unlike the PSC, which was manned by professional programmers, the mentoring classes will be manned by mentors recruited from among second year students. The choice of student mentors was dictated by research findings suggesting that programming students prefer to seek help from fellow students and lecturers. Research reported by Stamouli et al. (2004) suggested that "80% of students prefer to ask their lecturer or a friend when they encounter programming problems"; Pascarella & Terenzini (2005) found that structured peer assistance for "historically difficult" units, improved student progress and retention; and, an earlier study conducted among computing students at Victoria University found that for female students,

> the most preferred source of academic help was their fellow female students followed closely by female lecturers … for males … fellow male students were the first choice, followed by male lecturers. (Miliszewska et al., 2006, p. 16)

Accordingly, it was decided that to maximise the impact of mentoring classes, they would be attended by a pair of mentors at a time; both female and male mentors will be available.

The mentoring classes are intended to fulfil a dual purpose: one, they will serve as a source of 'friendly' professional feedback and support to new programming students; two, they will serve as an early 'detector' of students 'at risk'. According to Cuseo (2004), poor academic progress is a reliable indicator of potential attrition; hence, the importance of prompt feedback to students on their academic progress. Prompt feedback, combined with appropriate interventions to assist students with difficulties has been shown to improve student retention (Pascarella & Terenzini, 2005).

Attendance of the mentoring classes will be voluntary; however, students will be encouraged to make use of the service – an introduction of some incentives is being considered. The mentors will provide individual assistance to programming students. It is expected that, having gone through the introductory programming experience themselves, the mentors will be well equipped to assist new programming students.

The mentors will undergo a specialist training developed by computer science lecturers and education experts to learn the necessary mentoring skills. The training will be conducted in February 2007, prior to the commencement of classes, and will involve role-playing exercises. The exercises will emulate cases that the mentors will be likely to encounter with first year programming students. A sample mentor training scenario is presented in Figure 2.

The mentor training will aim to instil in mentors the need to encourage reflection in students (as illustrated in the sample scenario in Figure 2), and discourage the unproductive *try-it-and-see-what-happens* attitudes amongst students (Ben-Ari, 1998, p. 260). During the semester the mentors will participate in weekly meetings with the lecturer in charge of the unit to review study material, seek advice, and report possible problems.

A student writes a program that asks the user for two integer numbers. The program stores the numbers and displays them on the screen. The program then swaps the numbers and displays the two numbers again after the swap has taken place; this second display shows that the first number has been swapped but the second one seems to be lost.

- the mentor can lead the student in a discussion perhaps by showing him two glasses of different coloured liquids and asking the student how they would exchange the contents
- usually the student arrives at the need for a third glass or temporary container (recalling an analogy example presented in the lecture prior to the practical class)
- the student is then asked to code a temporary variable for the swap problem and use it as he would for the liquid swapping
- the mentor can assist in helping the student decide how to move integers between containers/variables

Figure 2: A sample scenario for mentor training

Web-based support

While students receive regular feedback on their work during scheduled laboratory sessions, the tutor's attention has to be divided among all the students in the classroom, and tutor's assistance is limited to the duration of the class. To enhance the provision of feedback, and to boost students' confidence in their programming skills, an on-line assignment submission system will be used. While the system has been in operation since 2002, plans are afoot to develop the system further and use it more extensively. The system enables students to test their programming assignments iteratively, while providing instant automatic feedback to each submission. The system serves as a 'supplementary' automatic tutor and gives the students an additional opportunity to perfect their programming skills. Providing students with self-assessment tests is one way of making them more aware of their own learning and enabling them to monitor their own competence (Carbone, Schendzielorz & Zakis, 1997).

In addition, a unit Web site will be maintained as a complementary communication and material delivery tool. All students enrolled in the unit will have access to material such as unit outline, lecture notes, tutorial exercises, laboratory tasks, assessment specifications and hints, announcements, and staff contact details; links to other useful resources and the online submission system will also be provided. In addition, a Wiki might be included as a documentation system for group assignments.

Expected Benefits and Deliverables

It is expected that the research project will yield a combination of immediate and long-lasting benefits. On the immediate end of the scale, the teaching methods employed in the project will develop and boost current students' confidence in their programming skills. In addition, the increased interaction between students, and students and tutors will assist in early identification of students 'at risk'.

The long-lasting benefits will stem from the teaching manual – The Guidelines – and the Web based assignment submission system. The Guidelines for teaching first year computer programming unit that will be compiled during the project (the framework will include a databank of analogy examples and assessment tasks) will continue to be a source of reference for staff in the School even after the project is finished. While it is expected that the Guidelines will also be of particular interest to staff in other sections of the University offering Engineering, Information Systems, and Information Technology courses, some general recommendations may be applicable to other scientific disciplines as well. The improved Internet-based assignment submission and processing system will benefit future programming students, as it will continue to operate beyond the duration of the project.

In addition to expected benefits, a number of project outcomes have been identified; the outcomes include:

- At least a 10% improvement in the unit's pass rate compared to 2006.
- An improvement in grade average for the unit.
- An improvement in student satisfaction with the unit as compared to 2006.

While the expected improvement in student pass rate and grade average may vary, as it will depend on the characteristics of the student cohort, the outcomes are still likely to improve student progression through the six undergraduate computing courses in the School of Computer Science and Mathematics.

Conclusions

This article reports on a current research project that aims to improve the negative perception that computer programming is difficult and unfriendly. Consequently, students will work individually and in groups on programming tasks throughout the semester; they will be supported and mentored by lecturers, and second-year computing students.

To facilitate active learning and hands-on practice, laboratory sessions will be merged with tutorial sessions; this will afford students more and better opportunity to interact with each other and the tutors. The mentoring classes will enhance the opportunities for interaction, provision of feedback and friendly peer support even further. In addition, the on-line assignment submission system will enable students to develop and test their programming skills in their own time.

The proposed approach will provide positive supportive atmosphere in which students can learn the intricacies of object-oriented programming; the goal is to trigger the students' interest, and show the magic of the discipline to students. While the approach aims to befriend programming, it also aims to realise the educational objectives of an introductory programming unit – those will not be compromised at the expense of 'popularity'.

References

Allison, I., Orton, P., & Powell, H. (2002). A virtual learning environment for introductory programming. *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, 48-52.

Ben-Ari, M. (1998). Constructivism in computer science education. *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education*, 257-261.

Blanchette, I., & Dunbar, K. (2000). How analogies are generated: The roles of structural and superficial-similarity. *Memory and Cognition*, *28,* 108-124.

Blumenstein, M. (2004). Experience in teaching object-oriented concepts to first year students with diverse backgrounds. *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC'04)* [electronic proceedings].

Boud, D., Keogh, R., & Walker, D. (1985). Promoting reflection in learning: a model. In D. Boud, R. Keogh & D. Walker (Eds.), *Reflection: Turning experience into learning* (pp. 18-40). London: Kogan Page.

Buck, D., & Stucki, D. (2001). JkarelRobot: A case study in supporting levels of cognitive development in the computer science curriculum. *Proceedings of the SIGSCE Technical Symposium on Computer Science Education,* 16-20.

Carbone, A., Schendzielorz, P., & Zakis, J. D. (1997). A web-based quiz generator for use in tutorials and assessment. *Global Journal of Engineering Education*, *1*(3), 341-346.

Carbone, A., Hurst, J., Mitchell, I., & Gunstone, D. (2001). Characteristics of programming exercises that lead to poor learning tendencies: Part II. *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, 93-96.

Clancy, M., Titteron, N., Ryan, C., Slotta, J., & Linn, M. (2003). New roles for students, instructors, and computers in a lab-based introductory programming course. *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, 132-136.

Cuseo, J. (2003). Academic advisement and student retention: empirical connections and systematic interventions. Retrieved November, 2006 from http://www.ulster.ac.uk/star/resources/academic_advisement.pdf

Dunican, E. (2002). Making the analogy: Alternative delivery techniques for first year programming courses. In J. Kuljis, L. Baldwin & R. Scoble (Eds), *Proceedings from the 14th Workshop of the Psychology of Programming Interest Group, Brunel University, June 2002*, 89-99.

Fekete, A., Kay, J., Kingston, J., & Wimalarante, K. (2000). Supporting reflection in introductory computer science. *ACM SIGCSE Bulletin*, *32*(1), 144-148.

Huet, I., Tavares, J., Weir, G., Ferguson, J., & Wilson, J. (2003). Co-operation in education: the teaching and learning of programming at the Universities of Aveiro and Strathclyde. Paper presented at the *ICHED Conference*, Aveiro, Portugal.

Jenkins T. (2001). The motivation of students of programming. *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education*, 53-56.

Jenkins, T. (2002). On the difficulty of learning to program. *Proceedings of the 3rd Annual Conference of the LTSN Centre for Information and Computer Sciences*, 53-58. Retrieved November, 2006 from http://www.psy.gla.ac.uk/~steve/localed/jenkins.html

Kölling, M., & Rosenberg, J. (2001). Guidelines for teaching object orientation with Java. *ACM SIGCSE Bulletin, Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education, 33*(3), 33-36.

Lister, R., & Leaney, J. (2003). First year programming: Let all the flowers bloom. *Proceedings of the 5th Australasian Computer Education Conference (ACE2003),* Adelaide, Australia, 221-230.

Lorenzen, T., & Heilman, W. (2002). CS1 and CS2: Write computer games in Java! *SIGCSE Bulletin*, *34*(4), 99-100.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.-D., Laxer, C., Thomas, L., Utting, I., & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, *33*(4), 125-140.

Meisalo, V., Suhonen, J., Sutinen, E. & Torvinen, S. (2002). Formative evaluation scheme for a web-based course design. *Proceedings of the 7th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2002), University of Aarhus, Denmark*, 130-134.

Miliszewska, I., Barker, G., Henderson, F., & Sztendur, E. (2006). The issue of gender equity in computer science: What students say. *Journal of Information Technology Education*, *5*, 107-120. Available at http://jite.org/documents/Vol5/v5p107-120Miliszewska136.pdf

Miliszewska, I., Horwood, J., Tan, G., & Venables, A. (2004). Gender bias in computing? – Student perspectives. *Proceedings of the Joint International Conference on Informatics and Research on Women in ICT (RWICT),* Kuala Lumpur, Malaysia, 1135-1146.

Pascarella, E.T., & Terenzini, P.T. (2005). *How college affects students: a third decade of research*. San Francisco: Jossey-Bass.

Proulx, V. (2000). Programming patterns and design patterns in the introductory computer science course. *SIGCSE Bulletin*, *32*(1), 80-84.

Sheard, J., & Hagan, D. (1998). Experiences with teaching object-oriented concepts to introductory programming students using C++. *Technology of Object-Oriented Languages and Systems-TOOLS 24, IEEE Technology*, 310-319.

Stamouli, I., Doyle, E., & Huggard, M. (2004). Establishing structured support for programming students. *Proceedings of the 34th ASEE/IEEE Frontiers in Education Conference,* Savannah, GA, October 2004, [electronic proceedings].

Tavares, J., Brzezinski, I., Huet, I., Cabral, A., & Neri, D. (2001). "Having coffee" with professors and students to talk about higher education pedagogy and academic success. Paper presented at the *24th International HERDSA Conference,* Newcastle, Australia.

Thomas, L., Ratcliffe, M., Woodbury, J., & Jarman, E. (2002). Learning styles and performance in the introductory programming sequence. *Proceedings of 33rd SIGCSE Technical Symposium*, *34*, 33-37.

Traynor, D., & Gibson, P. (2004). Towards the development of a cognitive model of programming; A software engineering approach. *16th PPIG Workshop,* Carlow, Ireland, April 2004. Retrieved November, 2006 from http://www.cs.nuim.ie/~pgibson/Research/Publications/E-Copies/PPIG04.pdf

Van Roy, P., Armstrong, J., Flatt, M., & Magnusson, B. (2003). The role of language paradigms in teaching programming. *Proceedings of the 34th SIGCSE Technical Symposium on Computer science Education*, 269-270.

Biographies

Dr Iwona Miliszewska is a senior lecturer in computer science at Victoria University, Melbourne, Australia. She has led and participated in research projects involving transnational education, effective teaching methods, lifelong learning and women in computer science, and has published in these areas. Currently, Iwona leads a grant-funded research project aimed at addressing the difficulties faced by first year computing students in a core introductory programming unit.

Grace Tan is a senior lecturer in Computer Science at Victoria University, Melbourne, Australia. Her research interests include investigations of innovative teaching methods, the development of graduate attributes, and issues related to female students in computing courses. Grace has experience in teaching programming to first year computing students and, at present, she is part of a research team investigating problems encountered by novice programmers.