Improving Progression and Satisfaction Rates of Novice Computer Programming Students through ACME – Analogy, Collaboration, Mentoring, and Electronic Support

Iwona Miliszewska, Anne Venables, and Grace Tan
Victoria University, Melbourne, Australia

Iwona.Miliszewska@vu.edu.au
Anne.Venables@vu.edu.au Grace.Tan@vu.edu.au

Abstract

The problems encountered by students in first year computer programming units are a common concern in many universities, including Victoria University. As a fundamental component of a computer science curriculum computer programming is a mandatory unit. It is also one of the most challenging units for many commencing students who often drop out from a computing course as a consequence of having failed, or performed poorly, in an introductory programming unit. This paper reports on a research project undertaken to develop and implement a strategy to improve the learning outcomes of novice programming students. Aimed at 'befriending' computer programming to help promote success among new programming students, the strategy incorporates the use of analogy, collaboration, mentoring sessions, and electronic support. The paper describes the elements of the strategy and discusses the results of its implementation in semester 1, 2007.

Keywords: analogy, automated assessment, collaboration, 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 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 also difficult to teach (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). Lister et al. (2004) emphasises the need for novices to be able to read code first before attempting to write programs. 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).

Material published as part of this publication, either on-line or in print, is copyrighted by the Informing Science Institute. Permission to make digital or paper copy of part or all of these works for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage AND that copies 1) bear this notice in full and 2) give the full citation on the first page. It is permissible to abstract these works so long as credit is given. To copy in all other cases or to republish or to post on a server or to redistribute to lists requires specific permission and payment of a fee. Contact Publisher@InformingScience.org to request redistribution permission.

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, such as the use of different textbooks or the introduction of an electronic assignment assessment system, have done little to improve the situation (Miliszewska & Tan, 2007). A new approach was needed. A research project, supported by a Teaching and Learning Support grant, was launched in July 2006. The project investigated the nature of the difficulties encountered by programming students and developed a 'friendly' framework for teaching programming to novices; the framework aimed at making computer programming welcoming and more accessible to novice programmers and, at the same time, achieving pedagogical objectives.

The first stage of the study examined the reasons why first year students find programming such a daunting prospect, and identified the various interventions reported in the literature that had been created over the years to alleviate the programming problem. The outcomes of the first stage of the research study were reported in 2007 (Miliszewska & Tan, 2007). The next stage of the research focused on the development of a new strategy to address the introductory programming problem. This paper presents in detail the features of this new strategy to teaching introductory programming and discusses the outcomes following the deployment of the strategy in semester 1, 2007.

Introductory Programming – Overview of Problems and Strategies

Introductory programming has been widely recognized as a major stumbling block for many computing students. Although computer literacy is high among some of the commencing computing students, most of them tend to lack programming experience. And it is the students' lack of problem-solving skills rather than the lack of prior computing experience that appears to be a problem (Dunican, 2002). In addition to the absence of problem-solving modules from Australian secondary school curricula, the lack of continuity between secondary and first year university studies exacerbates the problem; computer programming in particular appears to be "beyond the students' previous experience" (Stamouli, Doyle, & Huggard, 2004).

Lack of prior experience with programming includes lack of familiarity with complex tasks such as program design and construction, but also routine tasks such as compiling or running a program; sometimes, students even lack 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 among novice programming students (Ben-Ari, 1998; Dunican, 2002). The complexity of the relationship between the mental models held by students and their overall programming aptitude is under investigation by Bornat, Dehnadi, and Simon (2008).

In addition, many students have problems in relating the use of abstract terminologies in programming to real life objects. Consequently, these students claim to 'hate programming' as they struggle to comprehend even the most basic of programming concepts (Stamouli et al., 2004; Thomas, Ratcliffe, Woodbury, & Jarman, 2002). Last but not least, meeting the requirements of programming syntax may prove a challenge even to students equipped with adequate problem-solving skills (Dunican, 2002; Kölling & Rosenberg, 2001; Sheard & Hagan, 1998).

The problems encountered by students in first year computer programming units are a common concern in many universities. Various interventions have been introduced over the years to address this concern; they included changes to the curriculum, pedagogy, and assessment, and the provision of additional support to new programming students. Different arguments have been put

forward on what should be included in the curriculum of an introductory programming unit. Van Roy, Armstrong, Flatt, and Magnusson (2003) suggested the teaching of programming concepts rather than paradigms. Others advocated the teaching of programming based on a single paradigm, such as the object-oriented paradigm for example. Proponents of the object-oriented approach were divided into two "camps" that favoured a particular way of how object orientation should be introduced: *objects-first* (Blumenstein, 2004; Lister & Leaney, 2003), or *structured programming- first* (Sheard & Hagan, 1998): both these approaches have been reported as successful.

Various pedagogical techniques have been trialled over the years to help students develop programming skills. For instance, analogy has been used to help students learn programming fundamentals including input/output, data types, sorting, and searching; this approach relies on illustrative examples of concepts that students have seen before, and relates the familiar concepts to new ones (Blanchette & Dunbar, 2000; Dunican, 2002). Relevance is another important pedagogical facet: students should see a purpose to what they are learning. Sheard and Hagan (1998) reported on the successful use of games to illustrate the benefits of the object-oriented paradigm. They also successfully employed an iterative approach to learning and continuous reinforcement of concepts. Finally, another approach relied on the use of technology for teaching. Clancy, Titteron, Ryan, Slotta, & Linn (2003) described how the use of a laboratory-based model for computer science instruction improved student performance and satisfaction with the programming unit.

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 provides students with useful instant feedback and helps their understanding of language syntax or program behaviour; performance-based assessment helps to test students' ability to write working computer programs (McCracken et al., 2001). As well, criterion-referenced grading has been recommended as a technique likely to maximise the potential of every student in a disparate class of different capabilities (Lister & Leaney, 2003).

In addition to various pedagogical and assessment techniques, a range of supplementary support measures have been used to assist novice programming students. Successful forms of auxiliary support included: discussion classes, as reported by Sheard & Hagan (1998); Web pages for programming units (Sheard & Hagan, 1998); and, provision of structured one-to-one support to students with programming difficulties (Stamouli et al., 2004).

The ACME Strategy: Analogy, Collaboration, Mentoring & Electronic Support

The strategy developed at Victoria University to teach introductory programming aimed to create a climate where students embrace programming. It built on a variety of approaches that had been reported as 'successful' in the literature (as described in the previous section of this paper). The strategy incorporated four individual approaches with a view to achieving a better overall outcome; the key elements included: analogy, collaboration, mentoring, and electronic support. Analogy was used extensively in the teaching examples and tutorial discussions; collaboration amongst students was encouraged in laboratory sessions and required in the major assignment task; mentoring assistance was provided to students throughout the semester; and, an electronic assignment submission system provided students with an instant automatic feedback to submitted programming tasks. The details of the key elements of the ACME strategy are described below.

Analogy

Undergraduate students enrolling in computing courses are not expected to have prior programming experience. Thus, it comes as no surprise that they experience difficulties when facing pro-

gramming tasks for the first time (Dunican, 2002); program design and construction, compiling and running a program, as well as intricacies of hardware components might be difficult to understand. Further difficulties arise when there 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?

The ACME strategy incorporated the use of analogical models to bridge some of these difficulties. These models are said to help "people visualize the objects and processes which they are trying to understand" (Harrison, 2001); they use "a familiar object or experience to inform the learner about new and poorly understood objects, processes or concepts" (Harrison, 2001). For instance, an analogy involving a classic children's shape toy was used by Dunican (2002) to teach the concept of data types, assignment statements and type mismatches.

Analogies were used deliberately and extensively throughout the programming unit. Wherever possible, illustrative examples of familiar concepts were used to introduce students to new ones; an analogy was made between the familiar concept (source) and the new concept (target) by mapping the source onto the target (Blanchette & Dunbar, 2000). For instance, the concept of memory allocation (target) was illustrated with a wooden box divided into small pigeonholes (source), as depicted in Figure 1. Individual pigeonholes were identified by their own unique labels (memory addresses), and the labels referenced to assign, store and retrieve content. Here, two objects of *class Phone*, namely *myPhone* and *yourPhone*, have been instantiated and stored. A method to set the price of each object has been called, and the objects have been individually retrieved, manipulated and then stored again.

Figure 1: Analogy example- wooden box illustrating memory allocation.

To explain the need of a temporary variable in swap methods, another analogy was used. Students were presented with two wine glasses, one filled with water and the other with coffee. They were told that each wine glass represented a memory location and its content. Students were asked to move the contents of the first wine glass into the second and vice versa without mixing any of the contents. This analogy proved to be particularly useful; students immediately instructed the lecturer to supply another glass, a temporary holding vessel, to solve the problem.

In a discussion on graphical user interfaces (GUIs), an analogy was used to introduce the *JPanel* component; *JPanel* needs to be displayed by another component, such as a *JFrame* or *JApplet*. As

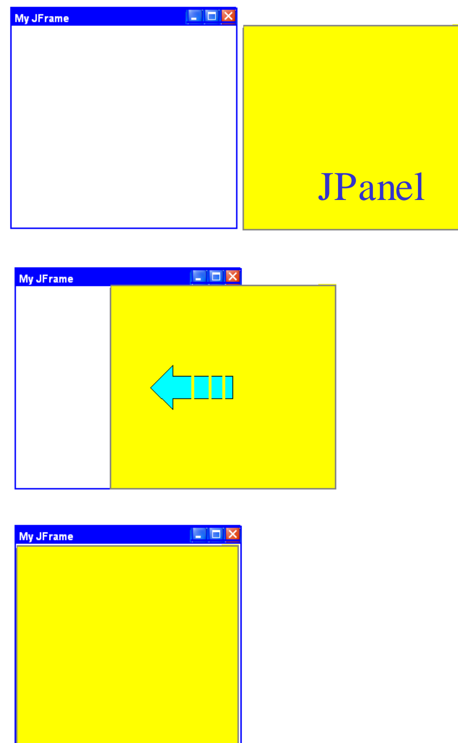

Figure 2: Analogy example- paper models of *JFrame* and *JPanel*

students were already familiar with the Windows environment, a piece of white paper with a blue border and header was used to represent the displaying window (such as a Windows screen), or *JFrame*; a discussion about the mechanics of making and displaying the window on a computer followed. Then, a separate piece of yellow paper was introduced to represent a *JPanel*; it was easy to see that the *JPanel* needed to be placed upon the *JFrame* before it could be displayed. This analogy is shown in Figure 2.

Analogy was also used to introduce the concept of iterators, such as the Java classes *StringTokenizer* and *Scanner*, and their methods. In this instance, a 'pacMan' character was used to represent the iterator class, and iterator methods *hasNext( )* and *next( )* were illustrated. The *hasNext( )* method, which returns either the value of 'true' or 'false' was likened to the 'pacMan' finding food or not. If the 'pacMan' found food, then the *next( )* method was called; the 'pacMan' used the *next( )* method to chew through the food, a *String*, breaking on the white space. In the example illustrated in Figure 3, the 'pacMan' *next()* method would eat, or return, the word 'text', followed by 'to', then 'be' and, finally 'eaten'.

Figure 3: Analogy example- pacMan model of iterators and their methods.

Collaboration

To further reduce the 'fear' of programming amongst students, particularly in regard to their assessment tasks, students were encouraged to work throughout the semester, summative assessment was used in the laboratory exercises to assist students in developing their programming skills. The set exercises were short and simple, and they were designed primarily as learning experiences; for instance, students were required to make minor modifications to existing code. This approach served to 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.

Students were instructed to work in pairs on a programming assignment; the pairs were self-selected but, as all students in the cohort were new students, the self-selection was not much different from a random assignment of students into pairs. The specification of the assignment was based on a popular television game named 'Deal or No Deal'. The topic of the assignment was chosen deliberately, as game playing problems have been reported to motivate students, especially when there is the opportunity to produce attractive graphical interfaces (Lorenzen & Heilman, 2002). The assignment was divided into two parts and scaffolded to encourage the devel-

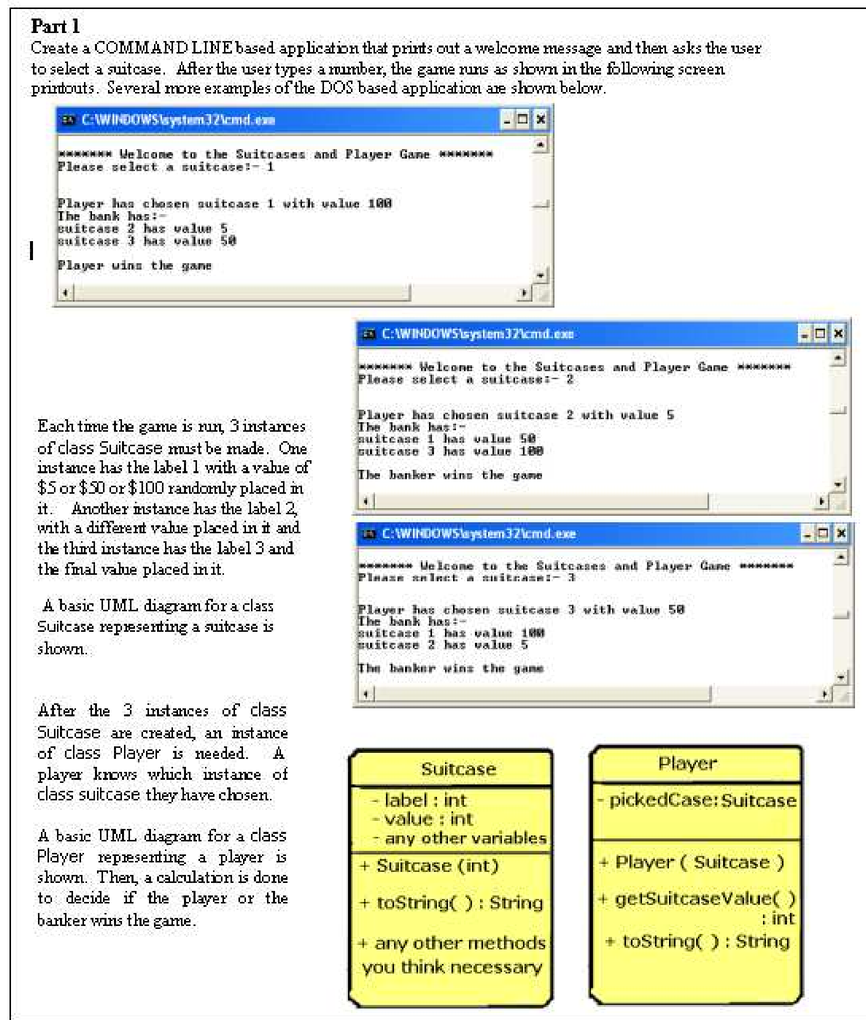

Figure 4: Partial specification of the 'Deal or No Deal' assignment.

opment of problem solving skills and collaborative learning. The first part of the specification gave a detailed description along with a suggested approach to solving the problem. Relevant UML class diagrams for the design of a typical solution were provided, together with screen dumps of typical outputs, as illustrated in Figure 4. The second part of the assignment was more open-ended and it required students to design, discuss and enhance their game programs by creating suitable GUIs for their solutions. This gave students the opportunity to extend the basic solution and explore creative ways of designing the GUIs; the assignment marking scheme provided rewards for initiative and effort. As well, by working in pairs it was expected that individual students would contribute roughly equal effort since both partners would be fully aware of the amount of work done by any one partner. In fact, the specification instructed groups that both students were expected to contribute equally.

Mentoring

A mentoring program was introduced to further support students who faced programming difficulties. Throughout the semester, mentoring sessions of one-hour duration were offered three days a week in a dedicated computing laboratory. Two student mentors (second year students) provided assistance during each session. The mentors were volunteers selected on the basis of their programming skills and communication skills; they underwent training prior to the commencement of their mentoring duties; and, they provided the mentoring assistance for free.

Participation in the mentoring sessions was entirely voluntary, with some students availing themselves of the service more than others. While the daily attendance numbers varied throughout the semester (often determined by assessment task deadlines), there were a number of ‘regulars’ who relied heavily upon the service. The problems reported by those students, and conveyed through the mentors, served as an invaluable source of informal feedback for lecturer in charge of the unit; the reported problems usually reflected the difficulties experienced by the wider student cohort. Thus, the particular problems that had arisen in the mentor sessions were subsequently addressed in the classroom, improving learning outcomes for all students.

Feedback from the mentors was actively sought throughout the program. Each week, the mentors submitted individual written reports detailing their efforts, experiences and reflections. In addition, weekly meetings were held between the mentors and the lecturer; the mentors reported back to the group upon their experiences; and the lecturer collected feedback on past week’s activities, informed the mentors of the current academic progress of the programming class, and briefed the mentors on the upcoming laboratory work and assignments.

Electronic Support

An on-line assignment submission system was used to enhance the provision of feedback, and to boost students’ confidence in their programming skills. As the service was web-based, students were able to upload their programs from a computer laboratory at the University, or from home, and receive feedback. First, the feedback stated whether a program compiled or not; if it did, the program was run against several sets of test data. Then, the students received a screen printout of how their program performed against the test data, along with a copy of the expected result for a correct ideal solution, as shown in Figure 5. A more detailed description of the development and the initial implementation of the automated feedback system has been reported by Venables and Haywood (2003).

It often happened that a student’s program worked correctly for most inputs, but the student failed to consider all the boundary cases or specific problem inputs; the use of test runs against sample data helped to highlight such anomalies. Importantly, the student was then able to modify and correct the submission as necessary, before deadline. Through this mechanism students were able

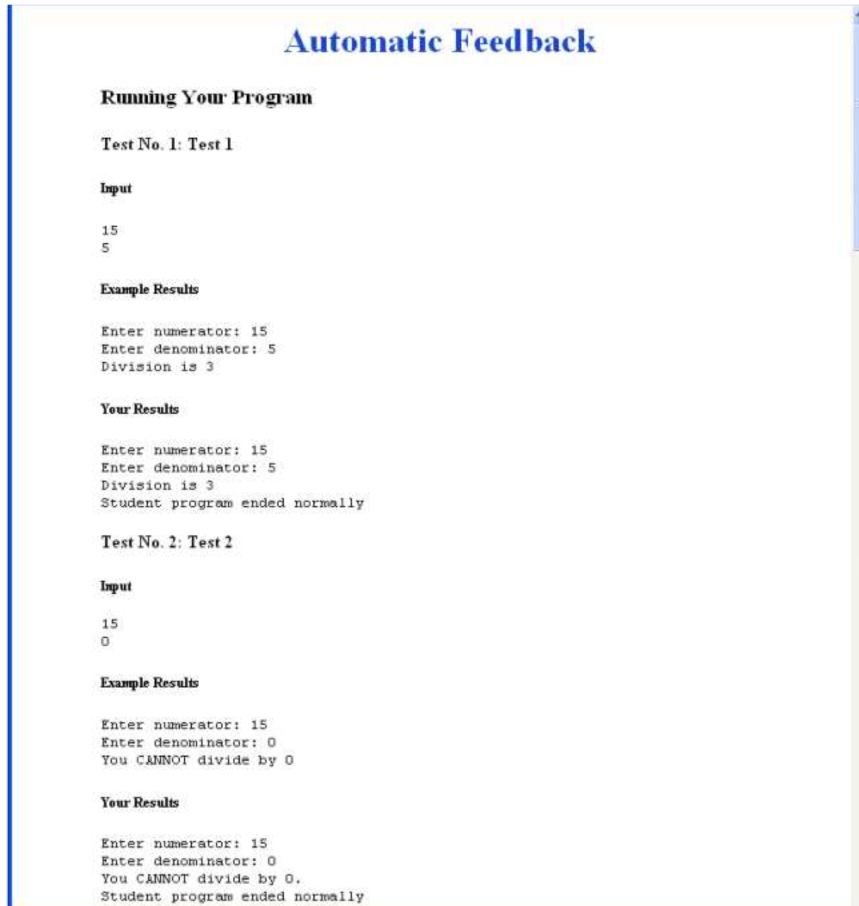

Figure 5: An electronic assignment submission system.

to learn by their mistakes and correct them without penalty. Finally, after the submission deadline, tutors entered additional descriptive feedback into the submission program; this feedback was provided to students in addition to grades.

Outcomes and Discussion

The application of the ACME approach to an introductory programming unit in semester 1, 2007 resulted in an improvement of student progression and satisfaction rates. It should be noted, that the unit was delivered in 2007 by the same lecturers and in the same learning environment as in 2006 and, while it would have been ideal to evaluate the learning outcomes of two parallel cohorts – one with, and one without the application of the ACME approach, it was not possible to implement it in 2007. Compared to 2006, the pass rate improved by 8% (from 56% to 64%) and the percentage of credits grew by 3%, as depicted in Figures 6 and 7 (Victoria University, Course Analysis Report, 2007). Although the improvement in student pass rate depended also on the characteristics of the 2007 student cohort, it was recorded across all the six undergraduate computing courses at Victoria University. While the improvement in progression rates was moderate, it seems to have particularly benefited the weaker students. On the other hand, the percentage of Distinctions and High Distinctions was slightly lower compared with the previous year. Further investigation is needed with future cohorts to determine if this pattern continues, indicating that the ACME approach might impact negatively on stronger students.

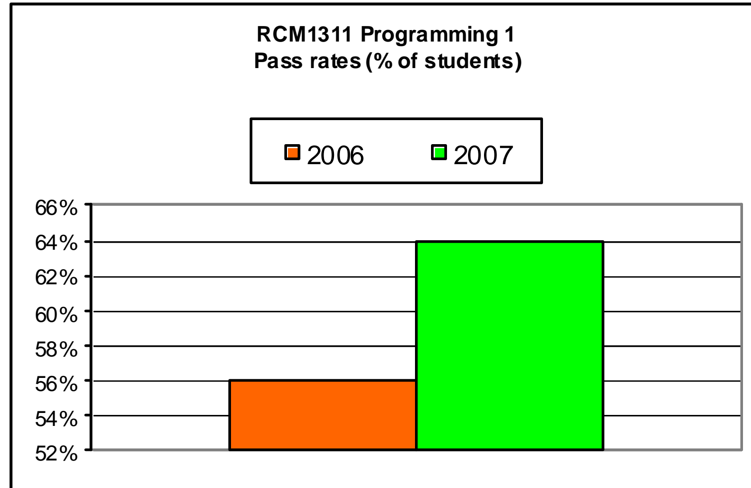

Figure 6: Pass rates of students in RCM1311 Programming 1 (% of students).

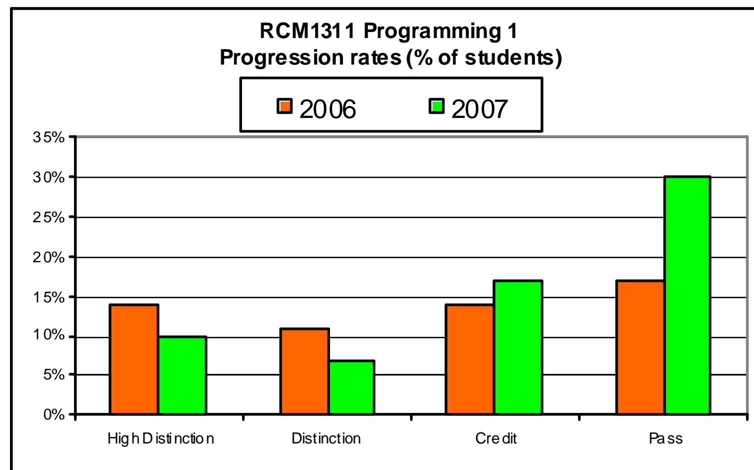

Figure 7: Progression rates of students in RCM1311 Programming 1 (% of students).

The approach had a bigger impact on student satisfaction rates with the unit. Data was collected through formal University surveys – Student Evaluation of Unit (SEU) – where a 5-point Likert scale (from 1 – least satisfied, to 5 – most satisfied) was applied to a set of ten questions related to student satisfaction with various aspects of the unit. A comparison between satisfaction rates in 2006 and 2007 shows a considerable improvement in student satisfaction in 2007, as illustrated in Figure 8.

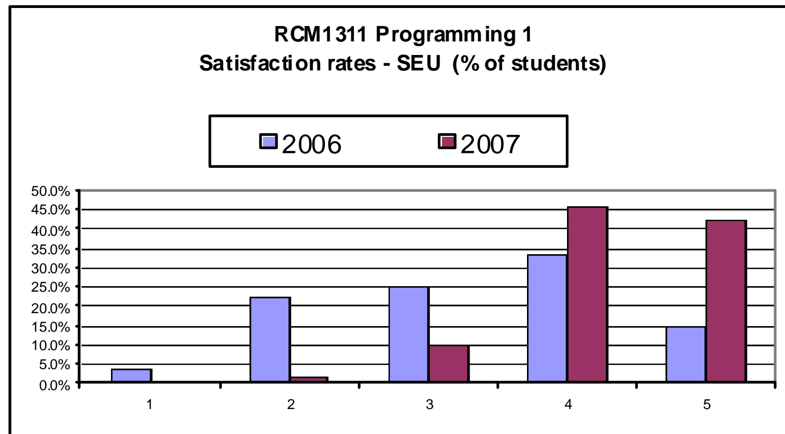

Figure 8: Satisfaction rates of students in RCM1311 Programming 1 (% of students).

Of the ten questions comprising the SEU survey, students gave their highest scores to two of the questions: *"I understand most of the content of the subject"* and *"I find the subject interesting"*. In addition, students provided written comments about the aspects of the unit that they found particularly helpful or stimulating:

> *"Detailed explanations and group discussions."*
> *"Lab questions were helpful as they enabled us to apply what we had learnt."*
> *"The online submit."*
> *"I find an example, like calling a variable a pigeon hole, much easier to understand."*

These comments seem to indicate that the various elements of the ACME approach appealed to various students. It must be noted that the use of simple analogies has met with wide student approval. In particular, they found very helpful the use of a wooden box that was used to illustrate the concept of memory allocation (as described earlier in the paper). It should be noted, that the SEU survey was used as evaluation tool in this study, as it is the mandatory evaluation tool provided by the University to measure student satisfaction rates. While it would have been useful to examine the satisfaction rates in relation the progression rates, the SEU tool does not have provisions for extraction of individual student responses.

The mentoring sessions, aimed at accommodating the specific needs of individual students and enabling early identification of students with programming difficulties, also produced the desired outcomes. The lecturer in charge of the unit commented on how at least five students would not have passed the unit if it had not been for the additional help that they received from the mentors. The mentors helped develop and boost the students' confidence in their programming skills and, the increased interaction between students and mentors assisted in early identification of students 'at risk'. Those students were subsequently offered additional support in tutorial and laboratory classes. Commenting on the usefulness of the mentoring sessions in the SEU surveys, the students wrote:

> *"Without the mentors, my labs would have been difficult to understand. They explained every possible outcome and helped me explore the meaning and use of the subject or the application of the subject; it was very good and clear. I wish we had mentors for other subjects."*
> *"Mentoring sessions were helpful because the mentors had faced the same problems we are facing now."*
> *"The mentors helped me learn things that the teacher did not have time to answer."*

In addition to the immediate benefits described above, the application of the approach produced several long-lasting benefits including a collection of teaching resources and an improved Web-based assignment submission system. The resources for teaching first year computer programming unit that have been compiled during the project (including a databank of analogy examples and assessment tasks) will continue to be a source of reference for academic staff. In addition, the improved Web-based assignment submission and processing system will continue to benefit future programming students.

Conclusions

This paper reports on the outcome of a research project that aimed to lift the "image" of computer programming among novice programming students; its goal was to improve the negative perception that computer programming is difficult and unfriendly. A multi-pronged approach to teaching introductory programming was developed to achieve the goal; the key ingredients of the approach included the use of analogy, collaboration, mentoring, and automated assessment.

Analogy, used extensively in the teaching examples and tutorial discussions, helped students comprehend some of the fundamental programming concepts. Collaborative assignments, and collaborative efforts in laboratory sessions further alleviated students' apprehension towards programming. The mentoring classes enhanced the opportunities for interaction, provision of feedback and friendly peer support even further. In addition, the Web-based assignment submission system enabled students to develop and test their programming skills in their own time.

The deployed approach provided positive supportive atmosphere in which students could learn the intricacies of object-oriented programming; it successfully triggered the students' interest, and showed them the magic of programming. While the approach aimed to befriend programming, it also aimed to realise the educational objectives of an introductory programming unit – it seems to have achieved some improvement in both respects.

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].

Bornat, R., Dehnadi, S., & Simon. (2008). Mental models, consistency and programming aptitude. *Proceedings of the Australasian Computing Education Conference (ACE 2008), Wollongong, N.S.W, Australia*, 53-62.

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.

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.

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.

Harrison, A.G. (2001). Thinking and working scientifically: The role of analogical and mental models. *Australian Association for Research in Education*, Fremantle, W.A., 2-6 December 2001. Retrieved August 2006 from http://www.aare.edu.au/01pap/har01126.htm

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., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, E., Sanders, K., Seppälä, O., Simon, B., & Thomas, L. (2004). A multinational study of reading and tracing skills in novice programmers. *SIGCSE Bulletin, 36*(4), 119-150.

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.

Miliszewska, I., & Tan, G. (2007). Befriending computer programming: A proposed approach to teaching introductory programming. *The Journal of Issues in Informing Science and Information Technology, 4*, 277-289. Retrieved from http://proceedings.informingscience.org/InSITE2007/IISITv4p277-289Mili310.pdf

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].

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.

Venables, A. & Haywood, e. (2003). Programming students NEED instant feedback! *Fifth Australasian Computing Education Conference (ACE2003) 4 - 7 February 2003, Adelaide, Australia,* 20, 267–272.

Victoria University, Course Analysis Report. (2007). *Student: VUSIS enrolment data universe*. Melbourne, Australia: Victoria University

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, life-long learning and women in computer science, and has published in these areas. Recently, Iwona lead a grant-funded research project aimed at addressing the difficulties faced by first year computing students in a core introductory programming unit.

Anne Venables lectures in Computer Science at Victoria University, Melbourne, Australia. She has research and teaching interests in artificial intelligence and intelligence systems. Anne spent several years as a secondary Science and Mathematics teacher before migrating into tertiary education. Anne is interested in innovations in education and has previously published in this field.

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, recently, she was part of a research team investigating problems encountered by novice programmers.