Learning to Program: *Going Pair-Shaped*

Donna Teague (d.teague@qut.edu.au)
Paul Roe (p.roe@qut.edu.au, http://sky.fit.qut.edu.au/~roe/)
Queensland University of Technology
Brisbane, Australia

Abstract: Students continue to struggle with learning to program. Not only has there been a significant drop in the number of students enrolling in IT courses, but the attrition rate for these courses continues to be significant. Introductory programming subjects in IT courses seem to be a stumbling block for many students. How do we best engage students in the learning of a programming language? How can our current teaching and learning methods be improved to provide a better experience for them?

Issues that have a detrimental effect on students' learning outcomes include more than simply the cognitive. Although programming really is complex and difficult to learn, there are also cultural and social influences on students presenting to introductory computer science courses. This paper highlights the advantages of intensive collaboration between students by exploiting the students' own ability and desire to interact with their peers. Peer interaction can lead to very strong learning experiences.

This paper reflects on the current approaches to teaching programming by the Queensland University of Technology (QUT), Australia, with a short summary of the current focus of QUT's first programming subject and the methods used to teach it. An overview is then given of the web-based Environment for Learning to Program (ELP) which provides scaffolding for students while learning to program.

The authors propose the introduction of tools to present a collaborative environment for students to actively engage in the course material through interaction with each other.

Keywords: Novice programmers, learning to program, problem solving, collaboration, collaborative learning, collaborative programming, collaborative problem-solving, e-learning, programming tools, pair-programming

1. Learning to Program Issues

Programming is acknowledged by tertiary educators to be a complex and difficult intellectual activity, with students struggling through their first programming subject and educators struggling to teach it (Lahtinen E, Ala-Mutka K, & Järvinen H, 2005). The high attrition rate of first year programming students has for many years been a headache and controversial topic for learning institutions (Robins A, Rountree J, & Rountree N, 2003; Sheard J & Hagan D, 1998; Truong N, Bancroft P, & Roe P, 2003); one for which further insight would benefit both students and universities alike.

1.1 Cognitive: Programming is difficult

Many cognitive theories have been offered to answer the question: *Why do many students fail to learn programming?* These include: difficulty understanding the purpose of programs and their relationship with the computer; difficulty grasping the syntax and semantics of a particular programming language (Robins A et al., 2003); misconceptions of programming constructs (Soloway E & Spohrer J, 1989); inability to problem-solve (McCracken M et al., 2001); and inability to read and understand program code (Lister R et al., 2004; Mannila L, 2006).

However, the cause of poor performance in the programming classroom is not purely cognitive.

1.2 Social Development

Introductory programming subjects are normally offered in the first semester of the first year of a bachelor degree, where the majority of the student cohort is new to university. The young university students are distracted and excited about entering the adult world, and enjoy many new freedoms not afforded them at school – where they ultimately assume the responsibility of the day to day management of their education. No dress codes, few rules, and they come and go virtually as they please. Unfortunately, there lies the trap. Settling into a productive university life seems to be a huge hurdle young students continue to face, and one that is exacerbated when one of the first subjects encountered in a computer science course at university is programming. Introductory programming is a subject which has

traditionally been a pre-requisite for other subjects that are dependant on many of the concepts introduced, and as a result, one for which there has been little option but to offer it in the very early stages of the course.

1.3 External Commitments

Many students today find it necessary to take on part-time or even full-time work (Taylor HG & Mounfield LC, 1989). In Australia in 1999, 57% of all students in higher education institutions also had a job, with 67% of part-time students working full-time and 42% of full-time students working part-time (Australian Bureau of Statistics, 2000). With an expected full-time study commitment at QUT and other similar universities of 48 hours or more per week, it becomes difficult to maintain employment, as well as balance family commitments, time to socialise, exercise and sleep. Increasingly it seems, the corners that are cut are the hours dedicated to study, with attendance numbers dropping dramatically after the early weeks of semester, and students tending to leave assessment obligations to the last minute.

Combine the social, community and employment pressures with a demanding subject like learning to program and the realisation that it takes a serious amount of time and effort to be successful often comes too late.

1.4 Cultural Perceptions

Programming is often perceived as a solitary occupation, one which is conducted in a competitive, rather than collaborative environment. This is often reinforced at university where the introductory programming subjects' assessment consists of individual assignments. Programming courses attract only a small proportion of female students and they have generally had less exposure to IT than their male counterparts (Cohoon JM, 2002b; Katz S, Allbritton D, Aronis J, Wilston C, & Sofa ML, 2006). The perception of programming being a competitive occupation coupled with the dominant image of a model student being the stereotypical 'geeky' young male can lead initially to alienation, diminution of confidence and subsequent lack of interest for women (Fisher A & Margolis J, 2002).

Academic ability would seem to have little influence on women's attraction to and retention in

programming courses. Women often perform well academically, yet perceive the programming environment as inhospitable, lacking social meaning and interaction which is incongruent with the real world. Women in IT are poorly represented at university which means there is less opportunity for support from female peers and possibly from role models in academic positions (Cohoon JM, 2002a; Vilner T & Zur E, 2006). The attrition rates for women from computer science courses is generally higher than for men (Cohoon JM, 1999; Fisher A & Margolis J, 2002) with the dropout rate of domestic female students at 36% in 2004 at QUT compared to 23.5% for domestic male students.

For other minority groups historically underrepresented in computer science courses, there seems to be a positive correlation between how well they can communicate openly with their peers and their learning outcomes. These students' social identities are developed based on how well they fit into relationships with other students (Varma R, 2006).

So for female students and other minority groups in particular, learning to program may present further issues of a social and cultural nature. This may also be true for male students, although they tend to receive a higher level of support for entering and persisting in the field of computer science (Cohoon JM, 2002a).

1.5 Generation-M Culture

It may be true that school leavers and young adults are resisting the way programming subjects are presented: unwilling to acclimate to an environment that demands individual achievement and is devoid of the continuously interactive and social multi-media-rich world to which they have become so accustomed.

A 2005 US national survey of 8-18 year olds found that the amount of time each week young people spend on media is equivalent to more than a full time job (Rideout V, Roberts DF, & Foehr UG, 2005). Rideout et al report that today's school children have become "masters of multitasking", often using several forms of media for example listening to music, having multiple MSN conversations and doing their homework simultaneously.

The constant interaction with peers during much of their leisure time via chats, email, SMS

etc. has become a significant part of the young university students' support structure for both their schooling and social life. Generation M students may not be able to adjust to a university environment that demands single focused, non-interactive study.

2. Collaborative Learning

Taking the constructivist approach (Bruner J, 1990; Huitt W, 2003) which is the predominant teaching and learning paradigm in education today, and endeavouring to engage students in regular hands-on activities from which to build their programming knowledge is only part of the solution for addressing learning to program issues. Collaborative learning addresses other issues and in particular provides benefits to first year programming students including greater achievement and interest, developing skills expected by industry, and having fun (McKinney D & Denton LF, 2006).

Collaborative learning is generally advocated as a exemplary pedagogical practice (McKinney D & Denton LF, 2006; Williams L & Kessler RR, 2000; Yerion KA & Rinehart JA, 1995) where grouped students become responsible for one another's learning as well as their own (Gokhale AA, 1995). Learning in a collaborative environment becomes a social process where students learn by working with others. Students are interactively engaged in the subject material, observing each others' approaches to problem solving, keeping each other focused on the task, and being encouraged to verbalise issues and decisions along the way.

Pair programming, which forms part of the Agile eXtreme Programming concept (Layman L, Cornwell T, & Williams L, 2006; Williams L & Kessler R, 2003; Williams L, Wiebe E, Yang K, Ferzli M, & Miller C, 2002), is an example of collaboration being used successfully in professional software development environments. It is based on the collaboration of two software developers. One takes up the role of 'driver', types the code, and addresses problems from a tactical point of view. The other becomes the 'navigator' and thinks strategically, asks questions and watches for coding errors (Aiken J, 2004). The developers in the pair frequently swap roles to benefit from both experiences.

Numerous studies (including (Wilson JD, Hoskin N, & Nosek JT, 1993), (Yerion KA & Rinehart JA, 1995), (Williams L & Kessler RR, 2000), (McKinney D & Denton LF, 2006)) have found benefits to students learning collaboratively including:

- synergistic nature of brainstorming and sharing of intellectual resources;
- monitoring the problem solving process by peer interaction is conducive to successful performance;
- positive affect on cognitive growth and skill acquisition and transfer;
- greater interest and sense of belonging;
- helps students apply algorithmic problem-solving techniques;
- deeper learning and higher retention;
- higher achievement and course success rates;
- developing skills wanted by industry;
- enhanced confidence in the solution and enjoyment of the process.

Like many universities, QUT's Faculty of Information Technology endeavours to provide the framework for a collaborative learning environment for its students. QUT uses a learning management system, Blackboard ("Blackboard Academic Suite,"), which offers students on-line access to course materials together with an extensive range of supporting communication tools including notice boards, virtual classrooms, chat rooms and discussions boards. These powerful learning resources afford students access to virtually everything they need, whenever they need it, as well as the opportunity for timely collaboration with each other and the teaching staff. The nature of these online resources not only makes it possible for students to fit study in around their other commitments, but enables them to collaborate in their learning when meeting face to face with their peers proves impossible or inconvenient.

It has been found that on-line interactions normally result in students being more open and talkative than they otherwise would have been face-to-face (Chen Z & Marx D, 2007). Given the difficulties that some students face with the perceived male-oriented, competitive, non-interactive environment in which they find themselves trying to learn to program, the opportunity to connect with other students online may fulfil their need for social interaction and provide the support required to maintain their confidence and interest.

3. Learning To Program @ QUT

3.1 Introductory Programming Subject - itb001

The itb001 subject is core to QUT's Bachelor of Information Technology and provides an introduction to the skills involved in solving computational problems using computer programs (www.qut.edu.au). There is no expectation of prior programming experience, but many school leavers already have had some exposure to programming during high school. Students are expected to attend four hours of classes per week, including a one hour lecture, a two hour active learning workshop and a one hour practical in a computer laboratory. Students are also offered optional drop-in Peer Assisted Study Sessions (also known as "Supplemental Instruction" (UMKC SI)) run by students who have already successfully completed the subject and who are employed by the university as peer facilitators.

The subject emphasises problem solving strategies and generic programming concepts rather than placing too much weight on the programming language of choice, which in this case is Scheme, a dialect of the Lisp programming language. Subsequent implementations of solutions are completed using DrScheme, an interactive, graphical programming environment (http://www.plt-scheme.org/software/drscheme/). During the weekly workshops, students are actively encouraged to work in small groups to work through a prescribed basic problem solving process towards a hand-written solution. Computers are not provided and students are specifically asked not to use their laptops during these workshops.

A template of problem solving process is provided which guides the students through the steps normally involved in the analysis of a problem, then progressive design, implementation and testing of a solution. By using the template students are provided with a model for documenting each step taken in developing their final solution. Steps include:

- restating key aspects of the problem to be solved;
- describing strategies to be used to solve the problem;
- identifying required procedures and the sequence of actions for each procedure;
- action refinement;
- test plan;

- implementation of code;
- desk check;
- application of tests and documentation of results.

Although not always readily accepted by the more 'experienced' students who habitually jump straight to the code without much preparation, the basic problem solving process template has been instrumental in highlighting the advantages of good documentation in the early stages of program design, especially for more challenging exercises by novice programmers.

The practical sessions which follow workshops are designed for students to individually implement their code in a computer laboratory following the rigorous preparation carried out in the workshops.

3.2 **Environment for Learning to Program (ELP)**

QUT subjects exist that teach about working in groups and collaborating with peers and colleagues towards a common goal, and some progress has been made towards exposing students to a similar collaborative environment in which to participate and learn some of the more technical subjects, in particular, introductory programming.

The hands-on, "learning by doing" approach continues in QUT's subsequent programming subjects which use C# and Java and involve students regularly attempting many small programming exercises through an online programming environment called "ELP". This tool was developed to help students learn to program in a simplified, non-threatening environment in contrast to the professional development tools which are designed to maximise programmers' productivity.

ELP is a web-based interactive development environment for programming which is available to students anywhere they have access to the internet and an up-to-date browser. Access is restricted to current students of the 'ELP-enabled' programming subjects, and each student can at this stage only access their own exercises.

Students complete the code for an exercise by reading the overall problem description then filling in gaps within a well scaffolded environment of skeleton code and extensive 'hints'. The code is then able to be compiled and run within ELP, providing a complete development environment while learning. The interface to ELP (Figure 1) includes a toolbar, code editor, tree view and various tabbed feedback windows.

ELP provides students with the opportunity to attempt increasingly difficult programming tasks each week in a well-scaffolded, self-paced environment. As there is no complicated software installation issue to contend with, ELP provides an immediate learn-to-program environment with multiple exercises to complement the teaching subject's course material. The ELP system provides useful, plain English compiler feedback and the opportunity for teaching staff to include exercise hints and complete sample gap solutions if and when appropriate.

Students can also annotate their code within the development context, for the attention of teaching staff or for their own information and reference. The various annotations are rendered in ELP in a feedback window in chronological order, much like providing the history of an on-line chat or discussion.

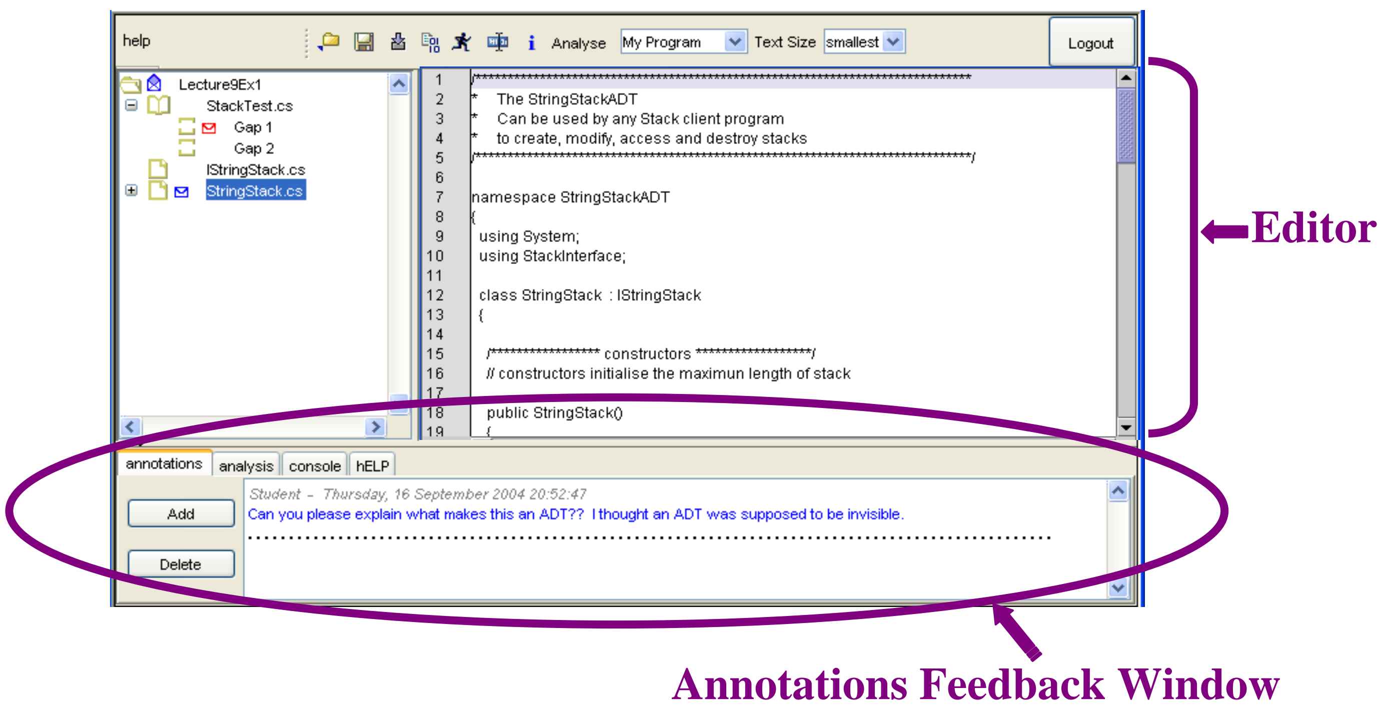


Figure 1 - ELP Interface

An annotation is depicted in the tree view by an envelope icon, the appearance of which will dynamically change depending on the author of the last annotation for that component (blue for a student and red a tutor) and whether or not that annotation has been read by its recipient: (open for read; closed for unread).

ELP has proven to be a valuable teaching and learning resource for QUT's programming subjects as well as for several Brisbane high schools, and its features have been well received by students and well utilised by teachers of programming subjects. ELP addresses the need for cognitive and technical programming support by novice programmers.

ELP also provides the means for and encourages tutor/student interaction, and has potential for further development and refinement in terms of peer to peer interactions.

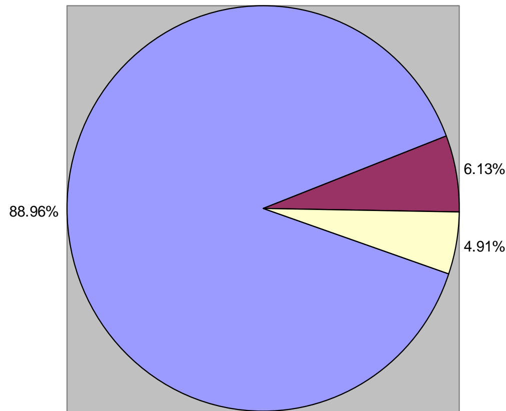
3.3 Collaborative Problem-Solving

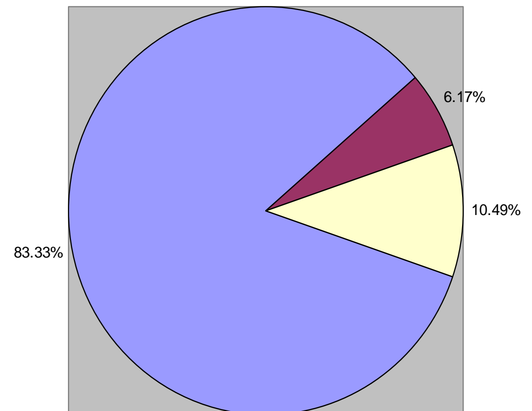
Collaborative learning has been embraced in QUT's first programming subject to the extent of encouraging students to discuss the problem domain and brainstorm solutions along a prescribed problem solving process similar to that prescribed by Polya (1957). The process involves verbalising the problem, devising an approach and developing a plan, carrying out that plan and reflecting back on the solution. Small groups of students work together during the problem solving process and are regularly called on to present their solutions to the entire class. Students who get the most out of these workshops tend to be those that are actively involved in discussions during the problem solving process. They bounce ideas around, and continually remind each other of concepts and ideas that have been introduced during lectures and in previous workshops. Workshops have become a non-threatening forum for thinking out loud – where no question is considered 'silly' and where active participation is always encouraged and rewarded. Incorrect, defective or incomplete solutions are embraced as part of the group's problem solving strategy in which the entire class benefits from ensuing discussions towards a more appropriate solution.

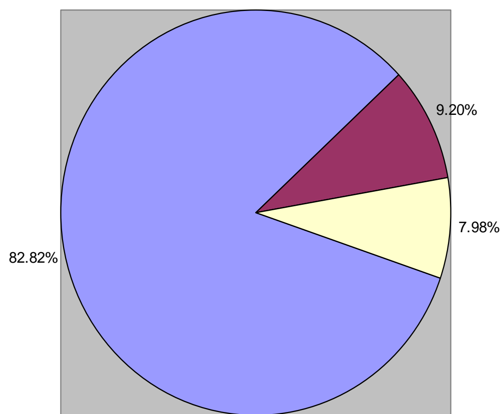
Capable students benefit from the opportunity to verbalise their reasoning and working knowledge of the problem domain while less capable students benefit from the multiple perspectives and the interactive peer support. There is always a small majority of capable students that have no interest in and see little benefit from helping and working with their peers. These students resist collaboration in favour of competitiveness, with a palpable desire for individual recognition.

A recent study (Benaya T & Zur E, 2007) has found that given a choice, a large percentage of students chose to work collaboratively rather than alone on a programming project. A survey of 170 first year programming students at QUT in semester 1, 2007 reflected a similar preference for collaboration with over 80% of students identifying the benefits of collaboration being:

- provision of a sounding board for discussing ideas and developing a plan of attack;
- that others would pick up on their mistakes and vice versa
- development of sound programming skills and keeping each other on track
- that a socially interactive environment is more fun







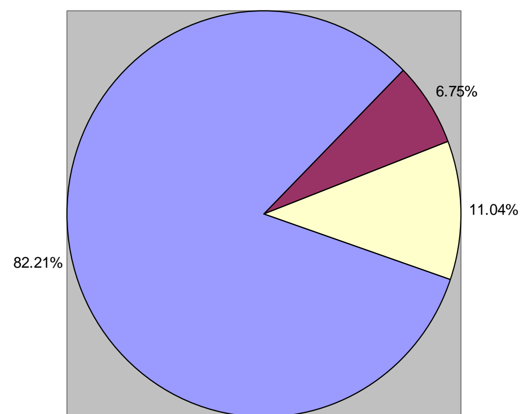


Figure 2 - Student Perspectives of Collaborative Learning

These graphs which summarise the results of that survey, show resounding student support for the concept of collaborative learning in a subject that was offering it in a limited way.

Benaya and Zur (2007) noted that students tended to choose other students of the same level to work with. The same study concluded that paired mediocre students achieved better grades in a programming project than mediocre students working solo and there was not a significant difference in the results of capable students working in pairs as opposed to solo workers. Furthermore, when there was a significant gap in the capabilities of students in a group, the grades of the stronger students are not harmed significantly.

Optional collaboration, at least for the more capable students, may therefore accommodate the interests of more of the student cohort, leaving those with a preference for individual work at no disadvantage.

4. Collaborative Programming

At the implementation stage of small programming exercise development, QUT's programming students often work by themselves to implement their code, test and refine it. This expectation of individual work is reinforced in the laboratory sessions where the layout of work stations discourages collaboration, and by the formal assessment items that demand individual submissions. Students are encouraged to discuss general approaches to solving programming assignment problems, but must not share code. Collaboration at code level for assignments is prohibited and the penalties for plagiarism regularly applied. Nonetheless, with the use of plagiarism detection software each semester a significant number of students are identified as having most likely plagiarised at least part of their submitted code.

One of the most significant issues that QUT's programming students currently have with learning to program is the transition from program design to implementation. That is, converting what is referred to as the 'action refinement', or finite set of steps used to solve the problem, into code. Unfortunately, it is at this stage of both their weekly exercises and assessment preparation that they find themselves working individually, without the many benefits of collaborative support. Sometimes it is even simple syntax issues that halt their progress. Novice programmers can be blinded by what appear to be one or more significant errors, but which are in fact simple but unrecognised syntax errors. Their immediate remedy is to redesign the action refinement and try a different approach, which often leads back to

exactly the same problem. Without appropriate intervention, which may be offered by a clear-headed second party, the student may lose confidence with their ability to problem solve and resort to a 'hack it and see' coding approach. This of course could result in extra-ordinary amounts of time wasted in trial and error and the acceptance of less than appropriate solutions in order to resolve issues that were in fact non-existent. The knowledge constructed by a student from this type of experience is likely to be neither accurate nor clear.

5. Conclusions/Recommendations/Further Work?

In the unlikely event of ever being presented with an unlimited amount of resources and funding, the perfect approach to teaching and learning programming at university may well be one-on-one tuition: with the student assuming the role of apprentice under the intimate guidance and supervision of a master, gradually absorbing the skills of the craft through observation and imitation (Dijkstra EW, 1975). Sadly, this is both unrealistic and impractical. However, encapsulating collaboration into learning to program can effectively utilise the resources already available, encourage more vigorous and active engagement by students; encourage them to think aloud and verbalise every step of their problem solving process, as well as satisfy their intense need for interaction and support.

For QUT's Faculty of Information Technology, extending the collaborative model into the coding stages should reap similar benefits to those already experienced in collaborative problem-solving, and pair programming in industry. This may involve a paradigm shift to fully embrace collaborative learning, overcome the stigma often associated with 'group work' at university, address issues of fair assessment for pairs or small groups, as well as providing students with the appropriate resources and support with which to collaborate effectively in order to improve their learning outcomes.

ELP has the potential to be developed as a more powerful collaboration tool by accommodating peer interaction, where students might have access to and review each other's work, record comments and apply assessment criteria via the annotations tool. After all, this is the approach many students already embrace in their social lives: 'Googling' a

particular issue or topic, reviewing someone else's work and leaving their appraisal of the content in a blog.

Using the internet as a free and ubiquitous source of problem solutions to any range of problems is a tempting quick fix option for some students who struggle learning to program. If developing their own problem solving skills was not only achievable, but recognised as readily transferable to many other parts of their education and careers, students may feel more inclined to persist.

Students could also benefit from e-learning tools and activities to help them during the problem solving process, in line with Polya's heuristic reasoning (1957) and clearly demonstrated in a programming context by Thomson (1996). One such tool might prompt students with a set of well formulated and progressively more leading open questions designed to set them thinking in the right direction. Figure 3 below contains examples of some of Polya's heuristics.

Heuristic	**Description**
Analogy	identifying a similar problem
Auxiliary Elements	introducing a new element to the problem in the hope it will further the solution
Auxiliary Problem	identifying another problem whose solution may help solve the original problem
Decomposing and Recombining	decomposing important parts of the problem and recombining them in some new manner
Draw a Figure	drawing a picture or figure of the problem
Generalization	identifying a more general problem
Here is a problem related to yours and solved before	identifying a known solution to a similar problem that may help solve the original problem
Induction	making a generalisation by observation and combination of particular instances
Specialization	identifying a problem more specialized
Variation of the Problem	changing the problem to a new problem that you may be able to solve and which may help with the original problem
Working backward	starting with the goal and working backwards to the initial situation

Figure 3 - Polya's Set of Heuristics (Polya G, 1957)

Support may also be incorporated for the problematic transition from action refinement to code with prompts for code template design, constructs, and automatic construction of skeleton code from action refinement. Students may simply be reminded of similar problem domains and their coding patterns leading to a design and testing recipe and culminating in a solution template.

Providing such tools in a collaborative environment mimicking the concepts of pair programming would be the key benefit as this would enable students to share mentoring roles, reflect on their own and each other's work and share the learning experience together during problem solving and program development.

Novice programmers require hands-on experience, and lots of it (Hassinen M & Mäyrä H, 2006), because their knowledge of programming is not passively absorbed through texts and lectures, but rather actively constructed via their own practical experiences (Ben-Ari M, 1998). Students should be given a supportive environment in which to experiment, and get the practical experience they need. Providing a totally collaborative learning environment may provide the support that students need to develop sound problem solving and programming skills and stop them resorting, in desperation, to plagiarism.

With collaboration a significant benefit to both already capable and less than capable students in introductory programming subjects, its infiltration into both problem solving and program development has the potential to positively affect course success rates, reduce attrition and in turn attract more interest from potential students, especially women and minority groups. A collaborative learning environment is warranted that addresses both the need students have for intensive technical support as well as encourages them to verbalise their thought processes and listen to and observe their peers while learning to program. The more confident, higher achieving students who enjoy learning to program in a collaborative environment are able to identify with the real world skills learning collaboratively has afforded them.

6. References

Aiken J. (2004). Technical and Human Perspectives on Pair Programming. *ACM SIGSOFT Software Engineering Notes, 29*(5).

Australian Bureau of Statistics. (2000). Participation in Education: Beyond compulsory schooling. *4102.0 Australian Social Trends* Retrieved 30 May, 2007, from http://www.abs.gov.au/ausstats/abs@.nsf/2f762f95845417aeca25706c00834efa/6ca07444d2673d02ca2570ec000e3632!OpenDocument

Ben-Ari M. (1998). Constructivism in Computer Science Education. *Twenty-ninth SIGCSE technical symposium on Computer science education, 30*(1).

Benaya T, & Zur E. (2007). Collaborative Programming Projects in an Advanced CS Course. *Journal of Computing Sciences in Colleges, 22*(6).

Blackboard Academic Suite. Retrieved 6 June, 2007, from http://www.blackboard.com/products/academic_suite/index.Bb

Bruner J. (1990). Constructivist Theory. Retrieved 19 July, 2007, from http://tip.psychology.org/bruner.html

Chen Z, & Marx D. (2007). ITEAM integrated teamwork enablement and management. *Journal of Computing Sciences in Colleges, 22*(6).

Cohoon JM. (1999). Department Differences and Point the Way to Improving Female Retention in Computer Science. *ACM SIGCSE Bulletin, Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education SIGCSE '99, 31*(1).

Cohoon JM. (2002a). Recruiting and Retaining Women in Undergraduate Computing Majors. *ACM SIGCSE Bulletin, 34*(2).

Cohoon JM. (2002b). Women in CS and Biology. *ACM SIGCSE Bulletin, Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education SIGCSE '02, 34*(1).

Dijkstra EW. (1975). *Craftsman or Scientist?* Paper presented at the ACM Pacific 75 - Luncheon Speech. from http://www.cs.utexas.edu/users/EWD/transcriptions/EWD04xx/EWD480.html.

Fisher A, & Margolis J. (2002). Unlocking the clubhouse: the Carnegie Mellon experience *ACM SIGCSE Bulletin, 34*(2).

Gokhale AA. (1995). Collaborative Learning Enhances Critical Thinking. *Journal of Technology Education, 7*(1), 22-30.

Hassinen M, & Mäyrä H. (2006). *Learning Programming by Programming.* Paper presented at the 6th Baltic Sea Conference on Computing Education Research, Koli Calling.

Huitt W. (2003). Constructivism. Educational Psychology Interactive. Retrieved 19 July 2007, 2007, from http://chiron.valdosta.edu/whuitt/col/cogsys/construct.html

Katz S, Allbritton D, Aronis J, Wilston C, & Sofa ML. (2006). Gender, Achievement, and Persistence in an Undergraduate Computer Science Program. *ACM SIGMIS Database, 37*(4).

Lahtinen E, Ala-Mutka K, & Järvinen H. (2005). *A Study of the Difficulties of Novice Programmers.* Paper presented at the 10th annual SIGCSE conference on Innovation and technology in computer science education ITiCSE '05.

Layman L, Cornwell T, & Williams L. (2006). *Personality Types, Learning Styles, and an Agile Approach to Software Engineering Education*. Paper presented at the SIGCSE 2006 Technical Symposium on Computer Science Education.

Lister R, Adams E, Fitzgerald S, Fone W, Hamer J, Lindholm M, et al. (2004). A Multi-National Study of Reading and Tracing Skills in Novice Programmers. *SIGSCE Bulletin, 36*(4), 119-150.

Mannila L. (2006). *Progress Reports and Novices' Understanding of Program Code.* Paper presented at the 6th Baltic Sea Conference on Computing Education Research, Koli Calling.

McCracken M, Almstrum V, Diaz D, Guzdial M, Hagan D, Kolikant Y, et al. (2001). ITiCSE 2001 working group reports: A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin, 33*(4).

McKinney D, & Denton LF. (2006). *Developing Collaborative Skills Early in the CS Curriculum in a Laboratory Environment*. Paper presented at the SIGCSE 2006 Technical Symposium on Computer Science Education.

Polya G. (1957). *How to Solve It: a new aspect of mathematical method* (2nd ed.): Princeton University Press.

Rideout V, Roberts DF, & Foehr UG. (2005). *Generation M: Media in the Lives of 8-18 Year-olds - A Kaiser Family Foundation Study*: The Henry J Kaiser Family Foundation.

Robins A, Rountree J, & Rountree N. (2003). Learning and Teaching Programming: A Review and Discussion. *Journal of Computer Science Education, 13*(2), 137-172.

Sheard J, & Hagan D. (1998). *Our failing students: a study of a repeat group.* Paper presented at the Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education: Changing the delivery of computer science education ITiCSE '98.

Soloway E, & Spohrer J. (1989). Studying the Novice Programmer. Hillsdale, NJ, : Lawrence Erlbaum Associates.
Taylor HG, & Mounfield LC. (1989, 1989). *The effect of high school computer science, gender, and work on success in college computer science.* Paper presented at the 20th

SIGCSE Technical Symposium on Computer Science Education SIGCSE 1989, Louisville, Kentucky, US.

Thompson S. (1996). Problem Solving in Haskell. Retrieved 22 May 2007, 2007, from http://www.cs.kent.ac.uk/people/staff/sjt/Haskell_craft/probSolving.html

Truong N, Bancroft P, & Roe P. (2003). *A Web Based Environment for Learning to Program.* Paper presented at the 26th Australasian Computer Science Conference - Volume 16, Adelaide, Australia.

UMKC SI. International Center for Supplemental Instruction Center for Academic Development University of Missouri - Kansas City,. Retrieved May 29, 2007, from http://www.umkc.edu/cad/si/

Varma R. (2006). Making Computer Science Minority-Friendly. *Communications of the ACM, 49*(2).

Vilner T, & Zur E. (2006). *Once She Makes it, She is There: Gender Differences in Computer Science Study.* Paper presented at the ITiCSE 06: Proceedings of the 11th annual conference on Innovation and technology in computer science education, Bologna, Italy.

Williams L, & Kessler R. (2003). *Pair Programming Illuminated*. Boston: Addison-Wesley.

Williams L, & Kessler RR. (2000). *The Effects of "Pair-Pressure" and "Pair-Learning" on Software Engineering Education.* Paper presented at the Proceedings of 13th Conference on Software Engineering Education & Training, 2000.

Williams L, Wiebe E, Yang K, Ferzli M, & Miller C. (2002). In Support of Pair Programming in the Introductory Computer Science Course. *Computer Science Education, 12*(3), 197-212.

Wilson JD, Hoskin N, & Nosek JT. (1993). *The Benefits of Collaboration for Student Programmers.* Paper presented at the 24th SIGCSE Technical Symposium on Computer Science Education SIGCSE 1993, Indianapolis, Indiana US.

Yerion KA, & Rinehart JA. (1995). Guidelines for Collaborative Learning in Computer Science. *ACM SIGSCSE Bulletin, 27*(4).