School of IT
Technical Report

The University of Sydney

STUDENT SELF-ASSESSMENT AND LEARNING TO THINK LIKE A COMPUTER SCIENTIST

TECHNICAL REPORT 577

LICHAO LI AND JUDY KAY

DECEMBER 2005

Student Self-assessment and Learning to Think like a Computer Scientist

This paper describes Assess, a system which helps students learn to self-assess their knowledge of computer science tasks which require synthesis of solutions to programming problems. The paper describes the underlying philosophy of our approach, which aims to help programming students learn to see their code as their teachers see it. At the same time, we tackle the common problems learners face when determining to appreciate exactly what they need to learn and to determine whether they have learnt what is required of them. We overview the system in operation and report our experience of using it over three semesters. We report student responses to the approaches of the system.

Programming has been long recognized as cognitively demanding, even at the most introductory level (Bonar & Soloway, 1983; Soloway, Ehrlich, Bonar, & Greenspan, 1982). This has made it an attractive and important domain for intelligent tutoring systems, for example, starting with early works reviewed in (du Boulay & Sothcott, 1987), continuing through systems such as the LISP Tutor (Corbett & Anderson, 1992) and ELM-ART (Weber & Specht, 1997).

We have chosen to take a novel approach to supporting students learning to program by building a system that helps students to learn to assess the quality of their own programs. Student self-assessment refers to *the involvement of students in identifying standards and/or criteria to apply to their work, and making judgments about the extent to which they have met these criteria and standards* (Boud, 1991). It can provide several benefits by helping remove the student/teacher barrier, lead to greater motivation and deeper learning (Boud, 1989; Falchikov, 1986). Moreover, it helps to develop metacognition, which is *knowledge and cognition about cognitive phenomena* or cognition of cognition (Flavell, 1971), in that it can help learners develop the capacity to identify their strengths and weaknesses and direct their study to areas that require improvement. These are characteristics of effective learners (Boud, Keogh, & Walker, 1985; Schön, 1983, 1987). It has been found that such metacognitive activities can facilitate problem solving (Davidson, Deuser, & Sternberg, 1994) and researchers argue that students can enhance their learning by becoming aware of their own thinking as they read, write and solve problems (Paris & Winograd, 1990).

Many studies have been undertaken to examine the procedures and effects of student self-assessment. A quantitative literature review by (Boud & Falchikov, 1989) found that

relatively able students can assess themselves in a way which is identical to the way in which they would be assessed by their teachers, if they are asked to rate themselves on a suitable marking scale. Another study by (Boud & McDonald, 2003) found that the introduction of self-assessment practices was well accepted by teachers and by students. Of particular interest and importance is that they found self-assessment training had a significant impact on the performance of those who had been exposed to it. On average, students with self-assessment training outperformed their peers who had been exposed to teaching without such training in all curriculum areas. This points to the potential benefits of explicitly developing students' ability to self-assess. To do this, one promising approach is to provide opportunities for self-assessment to be openly practiced among students in order to make them better self-assessors and, in turn, more efficient learners.

Given the potential benefits of explicitly teaching self-assessment, we have created a student self-evaluation tool, Assess. It has been used to teach programming in C in a UNIX environment for three teaching sessions, including two normal semesters and a summer school session. It has been used by over five hundred students improve their programming skills and also develop their metacognition, especially the ability to self-assess.

Promoting learner reflection is another important designing goal of Assess. Learner reflection is *a generic term for those intellectual and effective activities in which individuals engage to explore their experiences in order to lead to a new understanding and appreciation* (Boud et al., 1985). There is strong evidence suggesting that more efficient and effective learners pay more attention to their own learning experiences by reflecting on the state of their knowledge and the learning process (Boud et al., 1985; Pirolli & Recker, 1993). Learner reflection is especially important for cognitively demanding learning topics, such as programming. Reflection can only occur consciously. Therefore, it is important for the learner to be explicitly aware of the role of reflection in learning. Schön identified three types of learner reflection, namely *reflection in action*, *reflection on action* and *reflection on reflection* (Schön, 1983, 1987) that may occur in any learning experience. We explore ways to support the first two of these in Assess.

Our system is unique in several aspects. In Assess, students need to self-assess themselves with criteria supplied by teachers. Although it is common practice in classroom teaching to encourage students to review grading criteria (Fekete, Kay, Kingston, & Wimalartne, 2000), a system like Assess is novel. Another distinctive feature of Assess is that it lets students study supplied examples and assess them with teacher's criteria.

Student View of Assess

Although Assess could be used for a range of teaching areas, our experience has been in the context of a programming subject. We describe it in terms of that context. We begin with a high level overview of the student view of Assess and then explain each element in detail, with discussion of the philosophy and educational underpinnings.

Assess has a collection of tasks for students to view and self-assess. There are three steps in the student self-evaluation process, illustrated in Figure 1:

1. Students provide their own solutions to the programming problem of a task;
2. They self-assess the solution using criteria the teacher has defined for the task and;
3. They read and assess example solutions provided by the teacher.

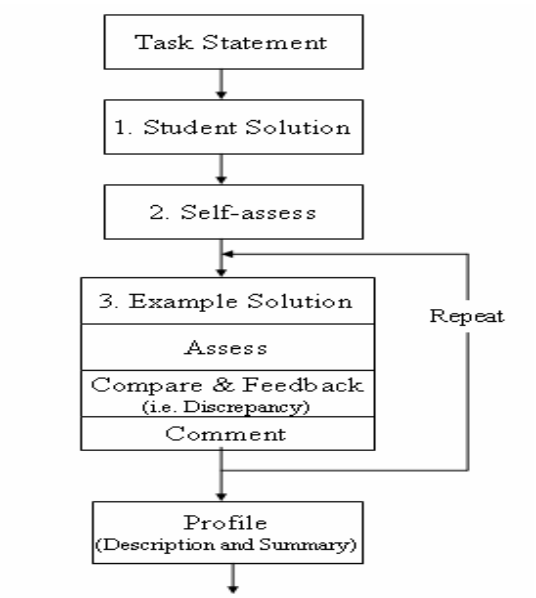

Figure 1 Workflow of Students Self-assessment

Figure 1 shows that an Assess session starts with a problem statement, an example of a task is provided in Figure 2. The student should then attempt to solve this; as indicated in Step 1 in Figure 1 and Figure 2. Once this is complete, the student saves it. Next, the self-assessment phase begins, shown as Step 2 in Figure 1 and Figure 3. The student sees a set of marking criteria, which are provided by teachers, as in Figure 3. Each marking criterion asks the student to rate one aspect of their solution. For example, in Figure 3, it asks about the coding style of their solution. The student can choose from a range of options for each marking criteria, for example, excellent, good, ok, poor or rotten. The students' solutions and assessments are saved, so that they can later reassess the solution.

Before describing the rest of the system, we briefly discuss how self-assessment in Assess supports deep learning. As described by Boud and Falchikov (Boud & Falchikov, 1989), there are two key elements in student self-assessment:

1. The learner's identification of criteria or standards to be applied to their own work.
2. The making of judgments about the extent to which work meets these criteria.

However, to make the student's self-rating more consistent with the teacher's rating of the student's work, explicit criteria for satisfactory and unsatisfactory performance need to be established (Boud, 1989). Teachers should design learning tasks to meet learning goals. In Assess, we ask the teacher to make these explicit by defining evaluation criteria for each task. For example, in Figure 3 we see that this task involved learning about good

coding style; accordingly, the teacher provided self-assessment criteria associated with style. In another situation, a teacher may want students to focus just on the code design. In that case, the self-assessment criteria would not include coding style.

Importantly, when students self-assess their solutions to a problem, it encourages reflection-in-action (Schön, 1983, 1987). They will read a marking criterion and reflect, as they rate their solution with respect to the criterion. They may also realise that they missed something in their solution and, since the criteria are shown in a new pop-up window, they can fix their solution while evaluating it. For instance, when students see the second last element of the last marking criteria in Figure 3, they may realize that they had forgotten to validate the input values. Because these criteria are created by teachers to be consistent with the teaching goals, when students assess their solutions with the set of marking criteria, they will be helped to reflect and learn, focusing on the relevant learning objectives.

At this stage, we need to use human tutors to grade the student solutions, because it is technically infeasible to comprehensively assess an arbitrary solution to a non trivial synthesis task. In a classroom setting, we can schedule time for a tutor to review a student's solution, aided by the Assess criteria so that they can serve as a foundation for the conversation between the tutor and the student.

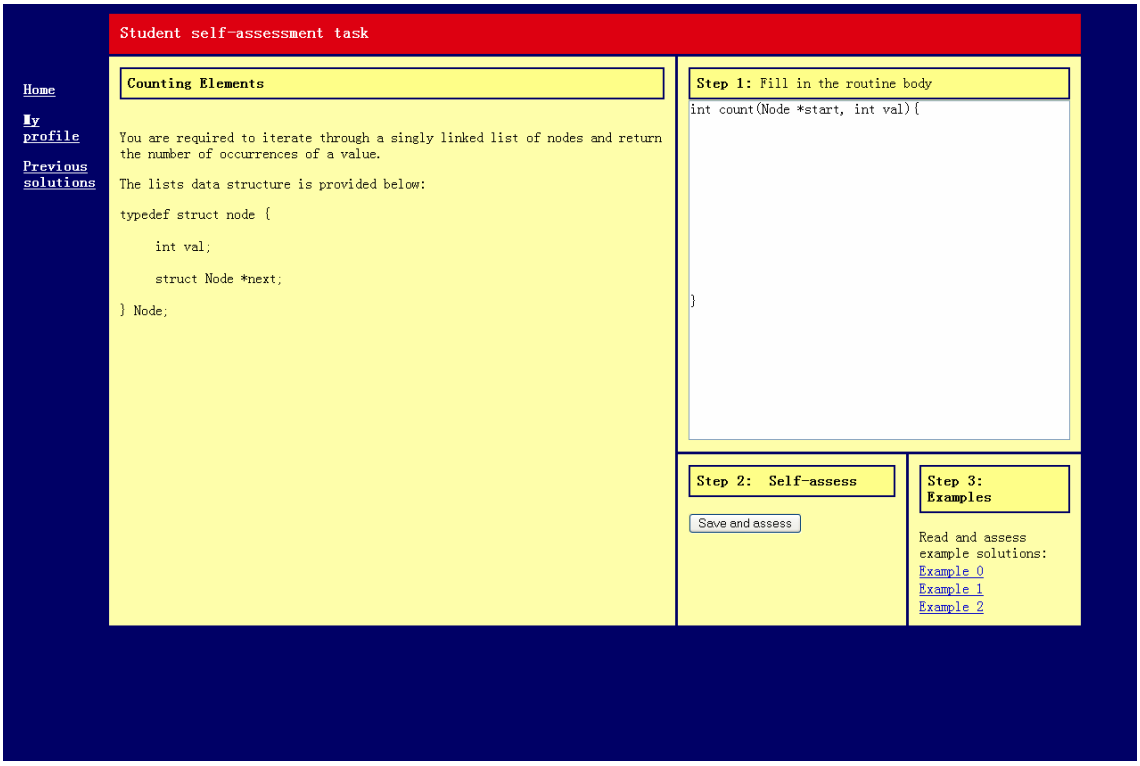

Figure 2 Problem Statement

We now discuss the remaining stages of in Figure 1. In Assess, we supplement the synthesis stage of Assess with a code reading stage. Far more tractable than automatically

assessing a student solution is the creation of example solutions by the teacher, each with its assessment according to the teacher. Essentially, once the student has created their own solution and assessed it, we ask them to review a series of example solutions. Examples clearly play a critical role in learning in general (Sweller, 1988). Researchers have found that learning from examples is of major importance for the initial acquisition of cognitive skills in well-structured domains, such as programming. For an overview, please see (VanLehn, 1996). Moreover, students prefer learning from examples (LeFevre & Dixon, 1986; Pirolli & Anderson, 1985).

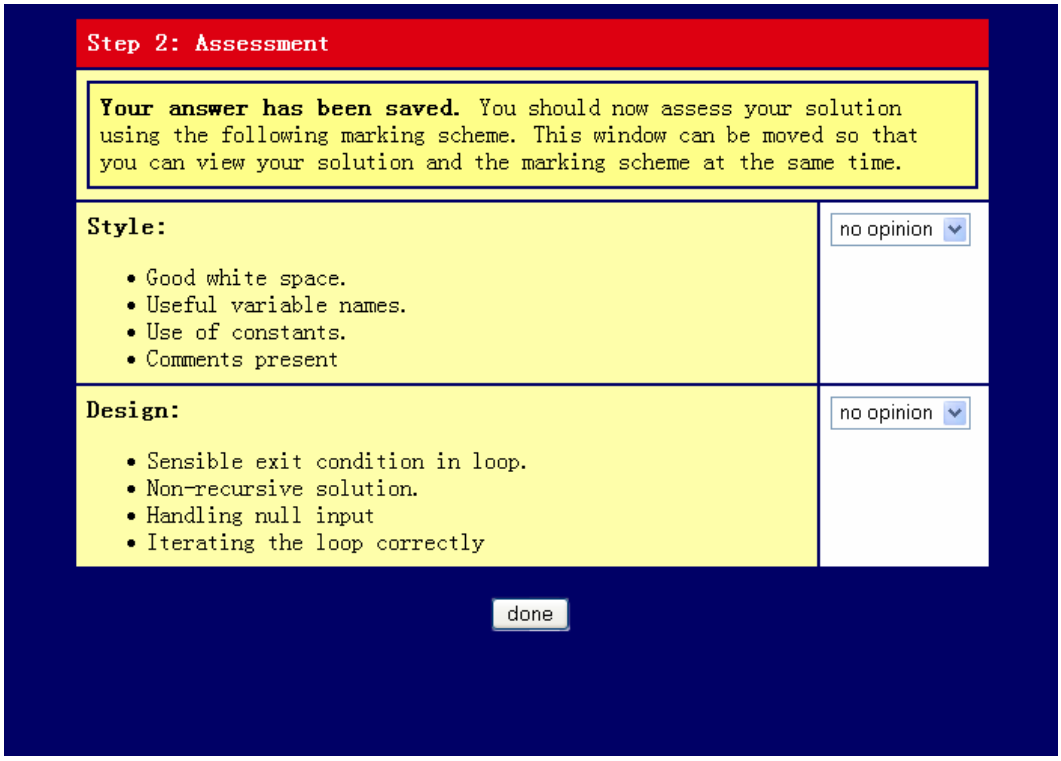

Figure 3 Marking Scheme

One of our example solutions is shown in Figure 4. Our *solutions* are normally *not* 'perfect'. Rather they provide opportunities to explore interesting and important ideas associated with the learning goals. We usually create these examples with a sprinkling of common misconceptions, poor coding style and similar elements. A particular fruitful source of these is in the students' answers to past exam questions. It has been found that showing learners some incorrect examples can enhance learning outcomes if the learners have favourable prior domain knowledge (Grosse & Renkl, 2004). We also strive to provide multiple examples, so we can illustrate different ways to do one task. Importantly, these example solutions have been pre-assessed by teachers. The students assess these solutions just as they did for their own solutions. As shown in Step 3 of Figure 1, their assessments are then compared to the teacher's assessment of the same example solution. The comparison gives the student three forms of feedback:

- How the teacher marked the example. For example Figure 5 shows the student's ratings of the example solution's style and design (in the *Yours* column) and the teacher's ratings (in the *Ours* column);
- The difference between the teacher's and the students' assessment, quantified by discrepancy values. (The smaller the discrepancy value, the better the students understand certain aspects in programming) and;
- Why the teacher assessed it that way. This is produced in comments at the bottom of Figure 5.

Generating explanations to oneself (self-explaining) has been shown to improve the acquisition of problem solving skills when studying examples. It has been found that more competent students generate many self-explanations regarding the novel parts of the example solutions and relate these to principles; by contrast less competent students do not generate sufficient self-explanations, monitor their learning inaccurately and subsequently they rely heavily on examples (Chi, Bassok, Lewis, Reimann, & Glaser, 1989). Another study by Chi and colleagues has shown that when students are encouraged to self-explain, they will do so and increase their learning (Chi, Leeuw, Chiu, & LaVancher, 1994). This stage of self-assessment in Assess aims to encourage students to self-explain of what they see in the examples. When the student reads an example and sees its marking criteria, they are encouraged to generate their own explanations of the code illustrated in the example and more importantly to relate this to the marking criteria, which represent the learning objectives of the task.

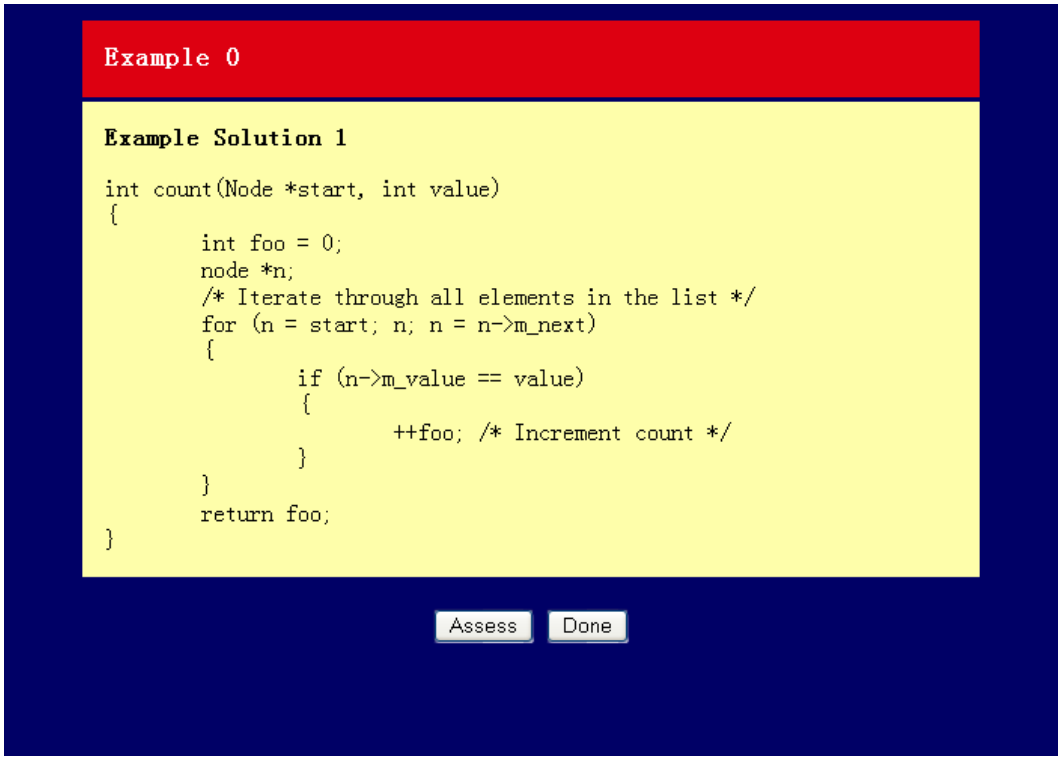

Figure 4 Example Solution

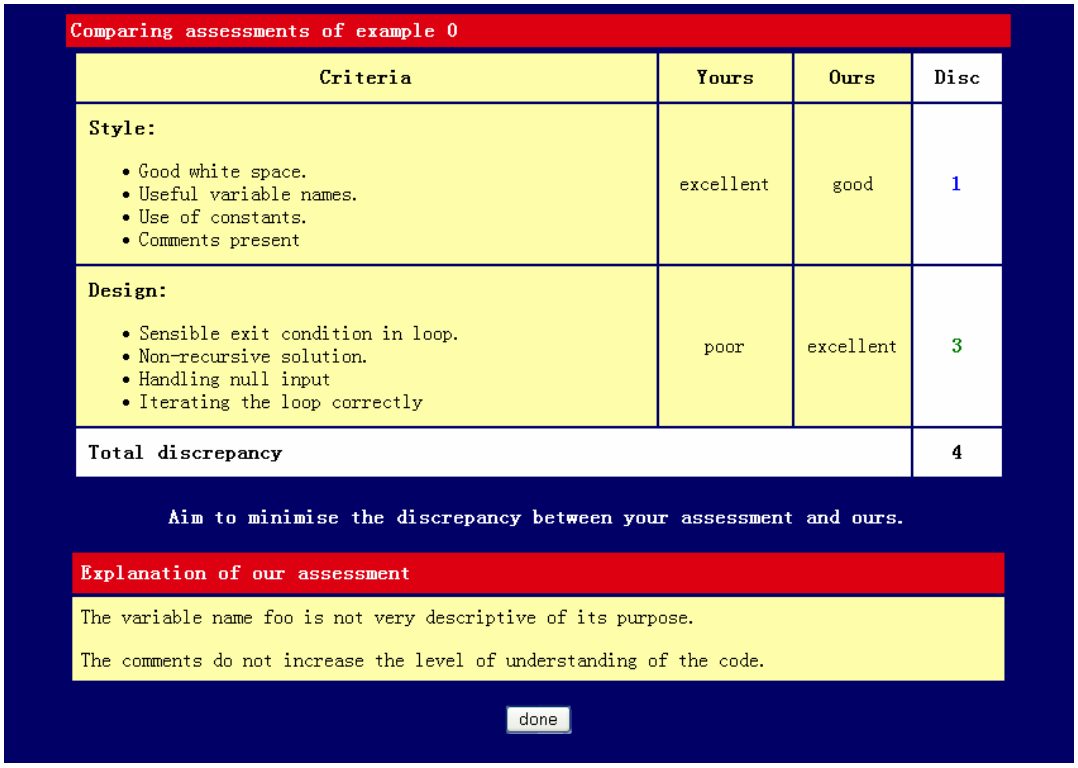

Figure 5 Assessment Comparison

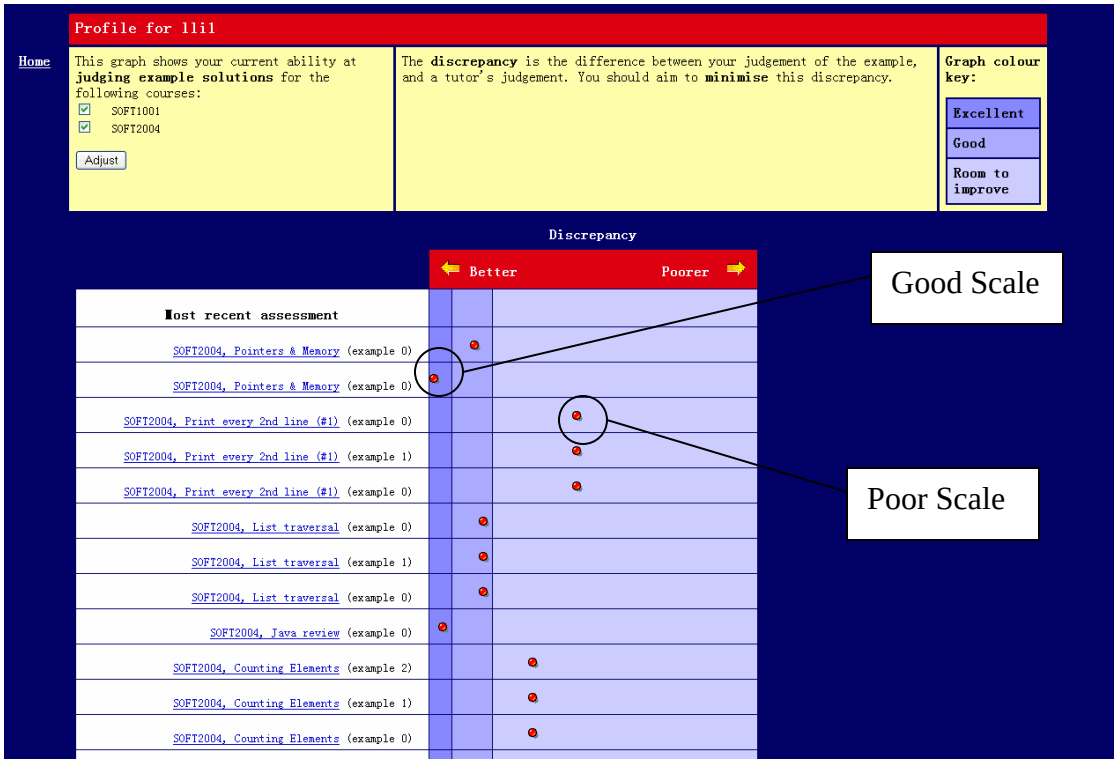

Figure 6 User Profile

Each time the student makes an assessment of an example solution, the total discrepancy between the student's assessment and the teacher's assessment is saved in the student's individual profile, which is used to generate a graph that illustrates the student's progress over time (See Figure 6). The graph is intended to help students identify what they need to improve and what they have thoroughly understood. It encourages learners to pay attention to their learning outcomes as well as experiences. The graph is also intended to promote learner reflection-on-action (Schön, 1983, 1987). As Figure 6 illustrates, it shows students their past learning experiences by listing all the tasks they have attempted. In addition, it illustrates how well they have completed a task by positioning the red dot on the marking scale of the task. If the dot is towards the left end of the scale, the student has done well and knows the concepts taught in that task well. In contrast, if the red dot is near the right end of the scale, the student needs more practice to learn the concepts in the task. From this graphical information, students are encouraged to think about what they have learnt.

Teacher View of Assess

Teachers create the tasks using web-based interfaces. Figure 7 gives a high level overview of the stages to create an Assess task. Unsurprisingly, this parallels the student view of Assess as outlined in Figure 1.

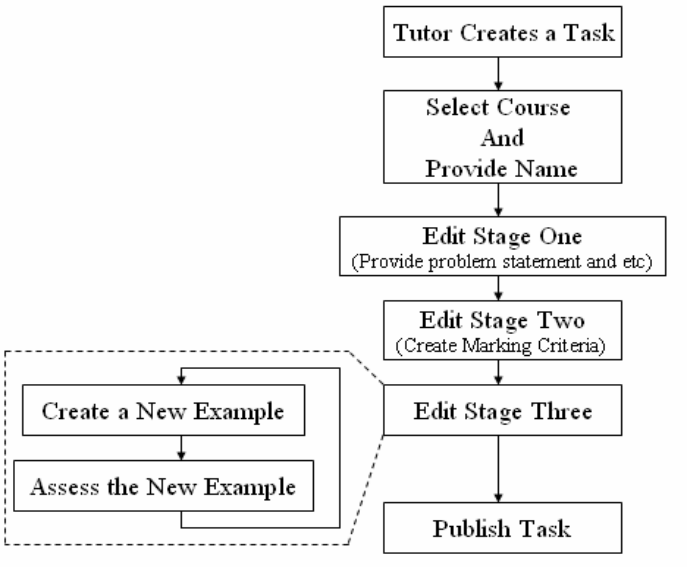

Figure 7 Task Creation Process

As shown in the figure above, there are three stages in authoring an Assess task:

Stage One. Create task statement;
Stage Two. Edit marking criteria and;
Stage Three. Create and assess example solutions.

An example of the teacher's homepage is shown in Figure 8. It allows teachers to create a new task and to manage existing tasks. The published tasks in the left column are available to students. The next column shows the unpublished tasks: a task must be unpublished while the teacher develops or alters it.

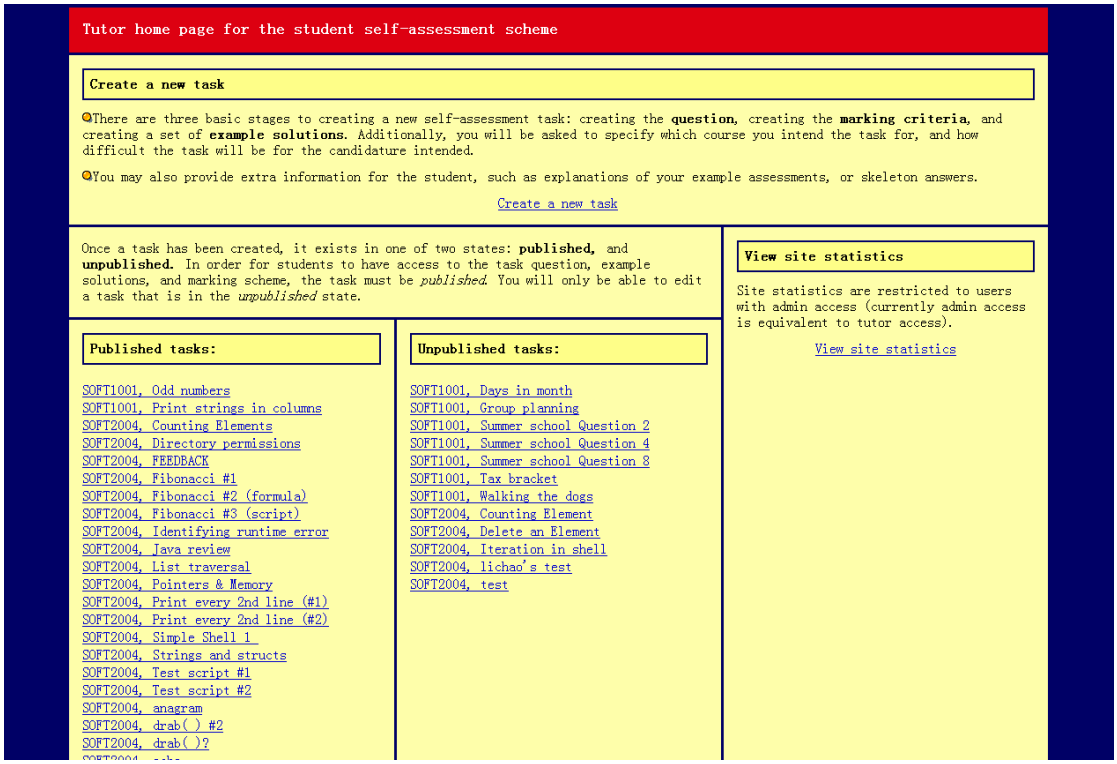

Figure 8 Teacher's Homepage

In Stage One, the teacher provides the problem statement (i.e. the task that student needs to solve) and, optionally, a skeleton answer. See Figure 9. The teacher also needs to indicate the difficulty of the problem and specify whether to force students to save and assess their own answer before viewing the example solutions. As can be seen in Figure 9, the task can be any arbitrary HTML.

The next stage is the formation of marking schemes. See Figure 10. Each marking criterion examines the students' comprehension of one of the expected teaching/learning outcomes of the task. It is accompanied by a set of grading options. For example, one criterion shown at the bottom Figure 10 is: *rate the answer according to code style. Desirable elements include: sensible block information, sensible naming conventions and good use of constants to avoid "magic numbers"*; and the marking options are: *excellent, good, ok, poor* and *rotten*. The teacher can put any HTML in the criterion window. Teachers can alter the default values for the criteria and they can choose any number of such answers, up to five. The lower part of the figure gives a preview of the criteria as the student will see them.

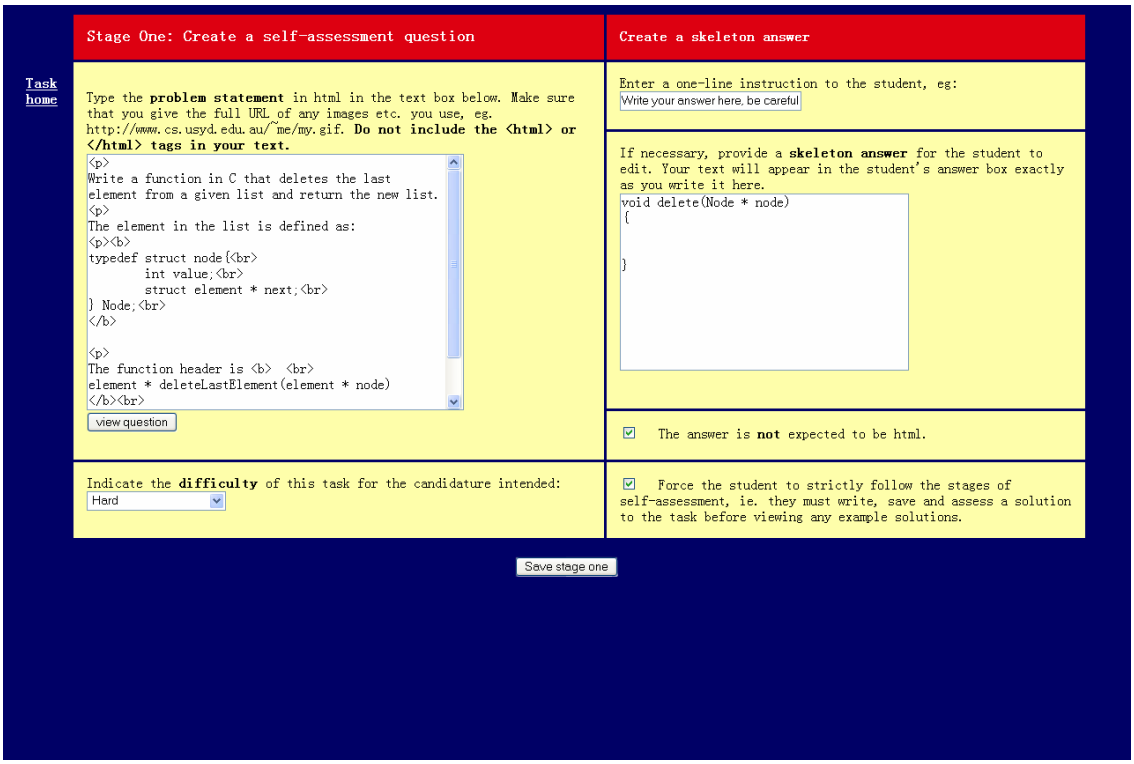

Figure 9 Task Creation Stage One

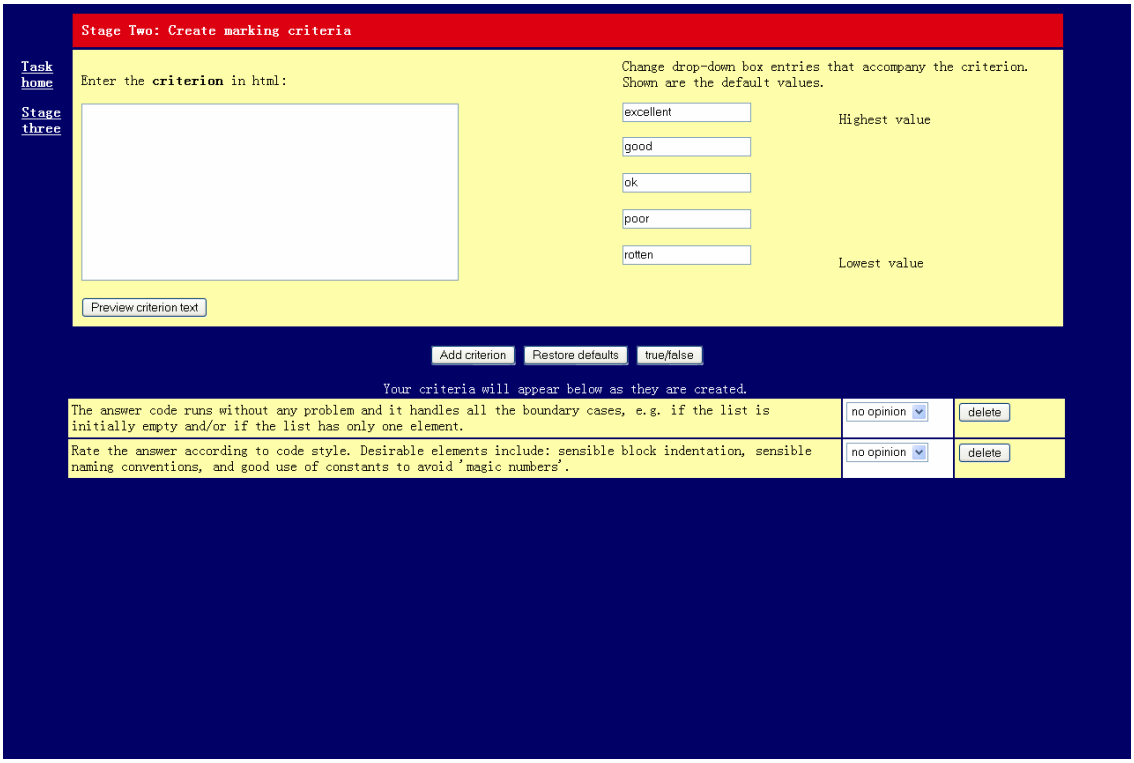

Figure 10 Task Creation Stage Two

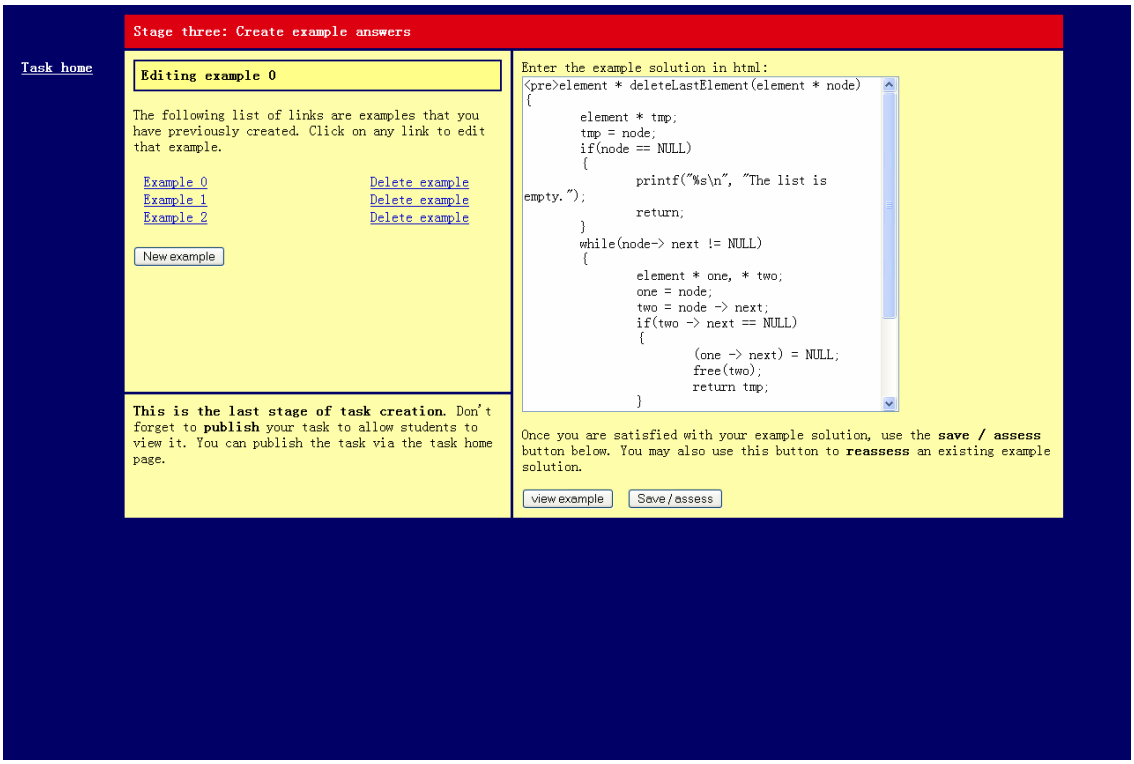

Figure 11 Task Creation Stage Three

Stage Three involves creating, editing and assessing the example solutions. Here the teacher can create an example solution as shown in Figure 11. Recall that these are not necessarily correct solutions. In fact, we believe that it would be uninteresting and tedious for students to be given only perfect answers to assess. They should provide the student with some ideas of right and wrong, interesting or useful aspects. The teacher can then assess the example solution and provide an explanation for the assessment. See Figure 12. When a student assesses this example solution, their assessment is compared with the teacher's assessment. The discrepancy indicates how well the student understands the learning concepts associated with the marking schemes, and is recorded in their user profile to provide learning feedback as was shown in Figure 6. We can illustrate this with an example. Based on the student's assessment of coding style as in Figure 3, suppose the student assesses the coding style of the example solution as excellent but the author of the task thinks it is poor. This shows the student cannot recognise the elements that make up good coding style and so does not understand this concept yet.

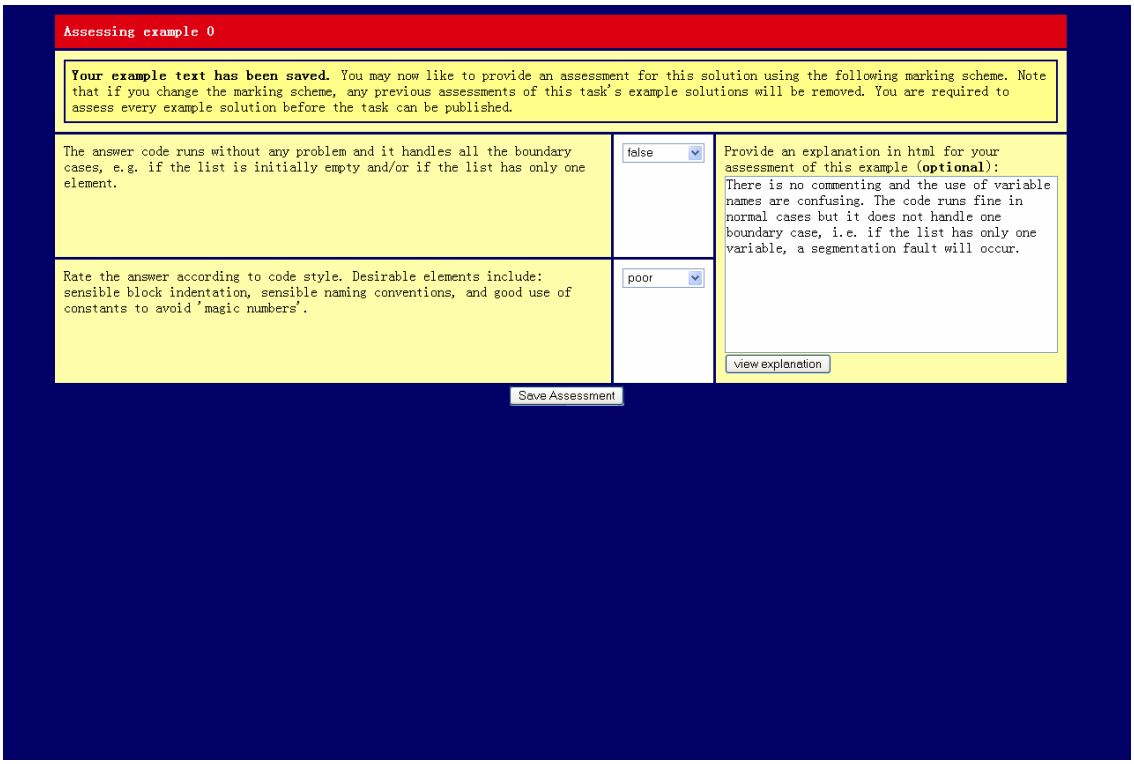

Figure 12 Task Creation Stage Three - Assess Example

Our Experiences

We have used Assess as a part of a programming subject (Programming in C in a UNIX environment) for over three semesters with more than five hundred students. In the first semester we formally introduced Assess to our students in the class, they achieved better learning outcomes (i.e. better exam marks) and the pass rate of the course increased. It is very difficult to quantify the contribution to this increase by Assess, because we improved many aspects of the subject in that semester and there are many factors that affect students' results. Nevertheless, we believe that Assess made a positive contribution.

Assess is integrated into the subject's assessment scheme. Students need to complete Assess tasks before or during weekly laboratories for course credit. Students' motivation is increased when they are aware that the Assess tasks are based on past exam questions. Typically, each week, students are required to do one or two tasks. We have found that when Assess tasks are done for homework, students tend to focus more on the code writing phase than the code reading phase, as according to the Assess records, many students make several attempts to write their own solution but most of the time they appear to read the example solutions but do not assess them. In contrast, when students are asked to complete a task during the lab, the tutor is able to encourage them to assess the examples we provide.

Recently, we swapped the code writing and code reading stages of the system to explore the effect of asking students to do the reading tasks first. Students were required to read and assess all examples before writing their own solution. We found that, generally, students read and assessed the examples as expected, but their own answers are often very similar to our example solutions. The more able students' answers were usually influenced to some degree by our examples, while the less competent students often copy and paste our solution. We still need to explore the mode of Assess operation.

The current system relies on human tutors to evaluate the students' code. We propose to add a facility for automated testing of student solutions. Even so, informal tutor impressions are that the current system is appreciated by many students. They regard it as a good place to learn programming, learn how to avoid common mistakes in programming, find out what teachers expect them to know and determine if they have learned what is required of them.

Conclusion

We have presented an overview of Assess, which is a programming teaching system that incorporates several novel educational approaches, such as supporting student self-assessment, promoting learner reflection and providing learning by example. The goals of Assess were to:

- Enable students to self-assess their knowledge;
- Enable students to monitor their learning progress and;
- Judge if they have achieved the required learning outcomes.

Assess achieves its objectives by

- Providing students with self-evaluation tasks;
- Providing students with feedback on their learning progress (i.e. in the student's profile) and;
- Providing students with feedback on the difference between the teacher's assessments and the student's assessments (i.e. the discrepancy values and the profile).

Assess is a light weight system in that it is easy for teachers to create tasks based on exam questions after seeing students' solutions. Our experience with Assess has been in the context of teaching programming but it can be used as a generic (i.e. subject independent) teaching system. It can teach a wide range of subjects, such as algebra, physics, chemistry or even English, any subject that can apply the Assess paradigm, which starts with a task statement, then asks the student to attempt a solution, self-assess that solution against set criteria, and finally requires students to read and assess a few example solutions.

References

Bonar, J., & Soloway, E. (1983). Uncovering principles of novice programming, *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. Austin, Texas: ACM Press.

Boud, D. (1989). The role of self-assessment in student grading. *Assessment and Evaluation in Higher Education, 14*(1), 20-30.

Boud, D. (1991). Implementing Student Self-Assessment. *Campbelltown: Higher Education Research and Development Society of Australasia. HERDSA Green Guide, 5*.

Boud, D., & Falchikov, N. (1989). Quantitative studies of student self-assessment in higher education: A critical analysis of findings. *Higher Education, 18*, 529-549.

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.

Boud, D., & McDonald, B. (2003). The impact of self-assessment on achievement: the effects of self-assessment training on performance in external examinations. *Assessment in Education, 10*(2), 209-220.

Chi, M. T. H., Bassok, M., Lewis, M. W., Reimann, P., & Glaser, R. (1989). Self-explanation: How students study and use examples in learning to solve problems. *Cognitive Science, 13*, 145-182.

Chi, M. T. H., Leeuw, N. D., Chiu, M.-H., & LaVancher, C. (1994). Eliciting self-explanations improves understanding. *Cognitive Science, 18*, 439-477.

Corbett, A. T., & Anderson, J. R. (1992). The LISP intelligent tutoring system: Research in skill acquisition. In J. Larkin & R. Chabay & C. Scheftic (Eds.), *Computer Assisted Instruction and Intelligent Tutoring Systems: Establishing Communication and Collaboration*. Hillsdale, NJ: Erlbaum.

Davidson, J. E., Deuser, R., & Sternberg, R. J. (1994). The role of metacognition in problem solving. In J. Metcalfe & A. P. Shimamura (Eds.), *Metacognition: Knowing about knowing* (pp. 207-266). Cambridge, MA: MIT.

du Boulay, B., & Sothcott, C. (1987). Computers teaching programming: An introductory survey of the field. In R. W. Lawler & M. Yazdani (Eds.), *Artificial Intelligence and Education* (Vol. Vol 1, Learning Environments and Tutoring Systems, pp. 345-372). Norwood, New Jersey: Ablex Publishing.

Falchikov, N. (1986). Product comparison and process benefits of collaborative peer group and self assessment. *Assessment and Evaluation in Higher Education, 11*(2), 146-166.

Fekete, A., Kay, J., Kingston, J., & Wimalartne, K. (2000). *Supporting Reflection in Introductory Computer Science*. Paper presented at the SIGCSE.

Flavell, J. H. (1971). First discussant's comments: What is memory development the development of? *Human Development, 14*, 272-278.

Grosse, C. S., & Renkl, A. (2004). *Learning from worked examples: What happens if errors are included?* Paper presented at the Proceedings of the first joint meeting of the EARLI SIGs "Instructional Design" and "Learning and Instruction with Computers" (CD-ROM), Tuebingen: Knowledge Media Research Center.

LeFevre, J., & Dixon, P. (1986). Do written instructions need examples? *Cognition and Instruction, 3*, 1-30.

Paris, S. G., & Winograd, P. (1990). How metacognition can promote academic learning and instruction. In B. F. Jones & L. Idol (Eds.), *Dimensions of thinking and cognitive instruction* (pp. 15-51). Hillsdale, NJ: Erlbaum.

Pirolli, P., & Anderson, J. R. (1985). The role of learning from examples in the acquisition of recursive programming skills. *Canadian Journal of Psychology, 39*, 240-272.

Pirolli, P., & Recker, M. (1993). Learning strategies and transfer in the domain of programming. *Cognition and Instruction, 12*, 235-275.

Schön, D. A. (1983). *The reflective practitioner*. New York: Basic Books.

Schön, D. A. (1987). *Educating the reflective practitioner*. San Francisco: Jossey-Bass Publishers.

Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1982). What do novices know about programming. In B. Shneiderman & A. Badre (Eds.), *Directions in Human-Computer Interactions*: Ablex.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science, 12*, 257-285.

VanLehn, K. (1996). Cognitive skill acquisition. *Annual Review of Psychology, 47*, 513-539.

Weber, G., & Specht, M. (1997). *User modeling and adaptive navigation support in WWW-based tutoring systems.* Paper presented at the UM-97.

School of Information Technologies
Madsen Building F09
University of Sydney NSW 2006 AUSTRALIA
T: +61 2 9351 4917 F: +61 2 9351 3838
W: www.alumni.it.usyd.edu.au

ISBN 1 86487 793 6